SOLUTION of (0-1.a):

You have to add **#include** `<Eigen/Dense>` in the beginning of every file that uses EIGEN tools. Of course the compiler must be told the directory, where EIGEN's header files reside.

HINT 1 for (0-1.b): The test case implemented in `main.cpp` is `smallTriangular(1,2,3)` and returns

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

SOLUTION of (0-1.b):

Matrix initialization with the << operator [Lecture → § 1.2.1.3] is convenient. However also the entry access operator **double** &**operator** (**int**, **int**) can be used directly.

C++ code 0.1.1: Initialization of fixed size 2×2-matrix

```
2  Eigen::Matrix<double, 2, 2> smallTriangular(double a, double b, double c) {
3    // We know in advance that we want to create a matrix of doubles with
       a fixed
4    // size of 2 by 2.  Therefore, we pass the parameters <double, 2, 2>
       to the
5    // template class Eigen::Matrix.
6    Eigen::Matrix<double, 2, 2> A;
7
8    // TODO: (0-1.b) Initialize the matrix A.
9    // START
10   // We have declared the variable A of the type
       Eigen::Matrix<double,2,2>,
11   // but we have not initialized its entries.
12   // We can do this using comma-initialization:
13   A << a, b, 0, c;
14   // END
15
16   // Question:  Is A now an upper triangular matrix, or a lower
       triangular
17   // matrix?
18
19   return A;
20 }
```

Get it on GitLab (MatrixClass.hpp).

HINT 1 for (0-1.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 (0-1.c):

Of course, one can simply set the matrix entries using the direct access to matrix entries and two nested loops.

C++ code 0.1.2: Initialization of upper triangular part of a matrix

```
2  Eigen::MatrixXd constantTriangular(unsigned int n, double val) {
3    // Now we do not know the size of our matrix at compile time.
4    // Hence, we use the special value Eigen::Dynamic to set the size of A.
5    Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A;
6
7    // Eigen::Matrix has a method Zero(n_rows,n_cols) which returns the n_rows by
8    // n_cols matrix whose entries are all equal to zero.
9    A = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>::Zero(n, n);
10
11   // To get a triangular matrix, we must set the values above the diagonal to
12   // val. This we can do by a nested for-loop.
13   for (unsigned int row = 0; row < n; ++row) {
14     // TODO: (0-1.c) Write the inner for-loop to initialize A.
15     // Hint: We can access and change the entries of A using parentheses, i.e.
16     // A(row,col) = val; Note that indexing starts at the value 0 (as usual),
17     // and not 1 (as in Matlab).
18
19     // START
20     for (unsigned int col = row; col < n; ++col) {
21       A(row, col) = val;
22     }
23     // END
24   }
25
26   // instead of the for-loop initialization, triangularView can be used
27   // A.triangularView<Eigen::Upper>() = Eigen::MatrixXd::Constant(n, n, val);
28
29   return A;
30 }
```

Get it on GitLab (MatrixClass.hpp).

Alternative: One can also use the `triangularView()` member function for **Eigen::Matrix**, see [Lecture → Code 1.2.1.6], EIGEN documentation.

SOLUTION of (0-1.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 0.1.3: Casting the scalar type of a matrix

```cpp
double casting() {
  // 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(n,x) creates a vector of size n and initializes the entries with
  // the value x.
  Eigen::VectorXi u = Eigen::VectorXi::Constant(2, 2);

  // 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 = 0.0;
  // TODO: (0-1.d) 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;
}
```

Get it on GitLab (MatrixClass.hpp).

HINT 1 for (0-1.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 (0-1.e):

Refer to the comments in the code and notice the use of the `cast()` method.

C++ code 0.1.4: Evaluating a simple matrix-vector expression

```
Eigen::VectorXcd arithmetics(unsigned int n) {
  // 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);

  // TODO: (0-1.e) Fill in the matrix B such that B(k-1,l-1) = (k+l*i)/(k-l*i),
  // where i is the imaginary unity, i^2 = -1 and k,l=1,...,n.
  // START
  for (unsigned int k = 0; k < n; ++k) {
    for (unsigned 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;

  // TODO: (0-1.e) 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;
}
```

Get it on GitLab (MatrixClass.hpp).

HINT 1 for (0-2.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 (0-2.a):

Use the write access methods `row()` and `col()` for a matrix.

C++ code 0.2.1: Initialization of fixed size 2×2-matrix

```
2  Eigen::MatrixXd zero_row_col(const Eigen::MatrixXd& A, unsigned int p,
3                               unsigned int q) {
4    // assert that we do not run into Eigen assertions.
5    assert(A.rows() > p && A.cols() > q && "p or q out of bounds.");
6
7    // Make a copy of A.
8    Eigen::MatrixXd Anew(A);
9
10   // TODO: (0-2.a) Set the entries of row number p and column number q
       to zero.
11   // Hint:  We can access rows and columns of Anew by Anew.row() and
       Anew.col().
12   // The method setZero() is useful here.
13   // START
14   Anew.row(p).setZero();
15   Anew.col(q).setZero();
16   // END
17
18   return Anew;
19 }
```

Get it on GitLab (MatrixBlocks.hpp).

HINT 1 for (0-2.b): The test call issued in `main.cpp` is swap_left_right_blocks(MatrixXd::Identity(4,3),
and yields the matrix

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

SOLUTION of (0-2.b):

We can use the write-access version of the `block()` method of matrix data types.

C++ code 0.2.2: Swapping left and right parts of a matrix in EIGEN.

```
2  Eigen::MatrixXd swap_left_right_blocks(const Eigen::MatrixXd& A,
3                                         unsigned int p) {
4    assert(A.cols() >= p && "Can only swap at most A.cols() columns.");
5    Eigen::MatrixXd B, C;
6
7    // We can use .rows() and .cols() to get the number of rows and
       columns in A.
8    unsigned int q = A.cols() - p;
9
10   // Make a copy of A.
11   Eigen::MatrixXd Anew(A);
12
13   // TODO: (0-2.b) Swap the first p columns of A with the rightmost q
       columns
14   // and write them in Anew.  The methods block(),
       leftCols()/rightCols() may
15   // come in handy.
16   // Hint:  A.block( i, j, m, n ) returns the m by n block that has its
       top-left
17   // corner at the index (i,j) in A.
18   // START
19   Anew.leftCols(q) = A.block(0, p, A.rows(), q);
20   Anew.rightCols(p) = A.block(0, 0, A.rows(), p);
21   // END
22
23   // Tip:  Many more methods exist that are special cases of block(),
24   // e.g.  topRows(), bottomRows(), topLeftCorner(), bottomLeftCorner(),
       ...
25   // For vectors we have head(), tail(), and segment().
26
27   return Anew;
28 }
```

Get it on GitLab (MatrixBlocks.hpp).

HINT 1 for (0-2.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 (0-2.c):

The code below use a special submatrix access function of EIGEN: `diagonal()`. However, a simple loop-based implementation based on entry access via **operator** `()` is hardly less efficient.

C++ code 0.2.3: Initialization of a triangular matrix in EIGEN.

```
2  Eigen::MatrixXd tridiagonal(unsigned int n, double a, double b, double c) {
3    Eigen::MatrixXd A;
4
5    // TODO: (0-2.c) Fill the matrix A with zeros.  Then fill the
       subdiagonal with
6    // the value a, the diagonal with b and the superdiagonal with c.
7    // Hint:  You can get the diagonal of A by A.diagonal().  Moreover,
       you can get
8    // the super- and subdiagonals by passing +1 or -1 as arguments to
9    // A.diagonal().  Here, however, we can use templated access because
       the
10   // distance to the diagonal is known at compile time.  This method is
       usually
11   // faster.
12   // START
13   A = Eigen::MatrixXd::Zero(n, n);
14   A.diagonal<-1>() = Eigen::VectorXd::Constant(n - 1, a);
15   A.diagonal() = Eigen::VectorXd::Constant(n, b);
16   A.diagonal<1>() = Eigen::VectorXd::Constant(n - 1, c);
17   // END
18
19   return A;
20 }
```

Get it on GitLab (MatrixBlocks.hpp).

HINT 1 for (0-3.a): Read about the `sum()` method available for matrices in EIGEN, see  EIGEN docum

HINT 2 for (0-3.a): From `main.cpp` we call `average(Matrix3d::Identity())` and should get 0.333333. ⌟

SOLUTION of (0-3.a):

C++ code 0.3.1: Computing the average of matrix entries

```cpp
double average(const Eigen::MatrixXd& A) {
  double m = 0;

  // TODO: (0-3.a) Calculate the average of the entries of A.
  // START
  m = A.sum() / A.size();
  // END

  return m;
}
```

Get it on GitLab (MatrixReduce.hpp).

HINT 1 for (0-3.b): Gather information about Boolean reductions from  EIGEN documentation. ⌟

HINT 2 for (0-3.b): The test call `percent_zero(Matrix3d::Identity())` in `main.cpp` should result in 66.6667. ⌟

SOLUTION of (0-3.b):

Note the expression `(Arr == 0).count()` and that the `size()` method returns the total number of entries for a dense matrix.

C++ code 0.3.2: Determines the percentage of (exactly) zero entries

```cpp
double percent_zero(const Eigen::MatrixXd& A) {
  // 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.
  Eigen::ArrayXXd Arr = A.array();

  double ratio = 0.;
  // TODO: (0-3.b) Calculate the ratio of zero entries in A.
  // START
  const unsigned int zeros = (Arr == 0).count();
  ratio = (static_cast<double>(zeros)) / A.size();
  // END

  return ratio * 100;
}
```

Get it on GitLab (MatrixReduce.hpp).

HINT 1 for (0-3.c): You can use partial reductions for your code, see  EIGEN documentation. ⌟

HINT 2 for (0-3.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 (0-3.c):

EIGEN's `any()` method checks whether an expression evaluates to **true** for a single entry of a matrix.

C++ code 0.3.3: Checking for an exactly vanishing column

```
2  bool has_zero_column(const Eigen::MatrixXd& A) {
3    bool result;
4
5    // A vector is the zero vector if and only if it has norm 0.
6    // The following vector contains the squared norms of the columns of
       A.
7    Eigen::VectorXd norms = A.colwise().squaredNorm();
8    // The norm is 0 if and only if the norm squared is zero.
9    // We use squaredNorm() instead of norm(), because norm() =
10   // sqrt(squaredNorm()) calculates a square root that is not necessary
       for our
11   // purposes.
12
13   // TODO: (0-3.c) Check if any one of the norms is equal to zero.
14   // Hint:  Use an array to perform entry-wise comparison.
15   // START
16   result = (norms.array() == 0).any();
17   // END
18
19   return result;
20 }
```

Get it on GitLab (MatrixReduce.hpp).

HINT 1 for (0-3.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

$$\begin{bmatrix} -2.07845 & 0.160061 & 0.561133 \\ 0.0929355 & -0.31935 & 0.896911 \\ 1.98551 & 0.159289 & -1.45804 \end{bmatrix}$$

SOLUTION of (0-3.d):

The solution gives a very compact implementation using suub-martrix access via `diagonal()` and the `rowwise` modifier for an ensuing reduction operation.

C++ code 0.3.4: Setting diagonals to make row sums vanish

```
2  Eigen::MatrixXd columns_sum_to_zero(const Eigen::MatrixXd& A) {
3    Eigen::MatrixXd B(A);
4
5    // TODO: (0-3.d) Replace the diagonal of B with values such that the
       columns
6    // of B sum up to zero.  Hint:  Use diagonal(), colwise(), and sum().
7    // START
8    assert((B.rows() == B.cols()) && "Matrix must be square");
9    unsigned int p = B.rows();
10   // Set diagonal elements to zero to avoid including them in sum
11   B.diagonal() = Eigen::VectorXd::Zero(p);
12   // Set diagonal entries to negative column sums all at once
13   B.diagonal() = -B.colwise().sum();
14   // END
15   return B;
16 }
```

Get it on GitLab (MatrixReduce.hpp).

HINT 1 for (11-1.a): Remember from [Lecture → Def. 6.3.1.2] 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 (11-1.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 $\rightarrow$ Eq. (8.5.1.17)] 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 (11-1.a):

Use the product rule: For $F : D \subset V \mapsto W$, $G : D \subset V \mapsto U$ sufficiently smooth, $b : W \times U \mapsto Z$ bilinear, *ie.*, linear in each argument:

$$T(\mathbf{x}) := b(F(\mathbf{x}), G(\mathbf{x})) \quad \Rightarrow \quad \mathrm{D}\,T(\mathbf{x})\mathbf{h} = b(\mathrm{D}\,F(\mathbf{x})\mathbf{h}, G(\mathbf{x})) + b(F(\mathbf{x}), \mathrm{D}\,G(\mathbf{x})\mathbf{h}) \,, \quad \mathbf{h} \in V, \mathbf{x} \in D \,,$$

[Lecture → Eq. (8.5.1.17)]

with the following choices: $V = \mathbb{R}$, $W = U = \mathbb{R}^{n,n}$, $F(t) := \mathbf{X}(t) \in \mathbb{R}^{n,n}$, $G(t) := \mathbf{Y}(t) \in \mathbb{R}^{n,n}$, and b as the matrix product, $b(\mathbf{X}, \mathbf{Y}) := \mathbf{X} \cdot \mathbf{Y}$.

$$\frac{d}{dt}(\mathbf{X}(t)\mathbf{Y}(t))h = \dot{\mathbf{X}}(t)h \cdot \mathbf{Y}(t) + \mathbf{X}(t)\dot{\mathbf{Y}}(t)h \,, \quad h \in \mathbb{R} \,.$$

Since h is a mere number, it can be dropped.

Let us apply this to 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.

HINT 1 for (11-1.b): Since (11.1.1) 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 (11-1.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 11.1.2: Explicit Euler method.

```
2  Eigen::MatrixXd eeulstep(const Eigen::MatrixXd &A, const Eigen::MatrixXd &Y0,
3                           double h) {
4    Eigen::MatrixXd Y1 = Y0;
5    // TODO: (11-1.b) Implement ONE step of explicit euler applied to Y0,
6    // for the ODE Y' = A*Y
7    // START
8    Y1 = Y0 + h * A * Y0;
9    // END
10   return Y1;
11 }
```

Get it on GitLab (matode.hpp).

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 11.1.3: Implicit Euler method.

```
2  Eigen::MatrixXd ieulstep(const Eigen::MatrixXd &A, const Eigen::MatrixXd &Y0,
3                           double h) {
4    const unsigned int n = A.rows();
5    Eigen::MatrixXd Y1 = Y0;
6    // TODO: (11-1.b) Implement ONE step of implicit euler applied to Y0,
7    // for the ODE Y' = A*Y
8    // START
9    Y1 = (Eigen::MatrixXd::Identity(n, n) - h * A).partialPivLu().solve(Y0);
10   // END
11   return Y1;
12 }
```

Get it on GitLab (matode.hpp).

Finally, the implicit mid-point method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}\left(\tfrac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{Y}_k + \mathbf{Y}_{k+1})\right),$$

hence

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}(\mathbf{Y}_k + \mathbf{Y}_{k+1}) \quad \Longrightarrow \quad \mathbf{Y}_{k+1} = \left(\mathbf{I} - \tfrac{h}{2}\mathbf{A}\right)^{-1}\left(\mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}\mathbf{Y}_k\right).$$

C++11-code 11.1.4: Implicit midpoint method.

```
2  Eigen::MatrixXd impstep(const Eigen::MatrixXd &A, const Eigen::MatrixXd &Y0,
3                          double h) {
4    const unsigned int n = A.rows();
5    Eigen::MatrixXd Y1 = Y0;
6    // TODO: (11-1.b) Implement ONE step of implicit midpoint rule applied to Y0,
7    // for the ODE Y' = A*Y
8    // START
9    Y1 = (Eigen::MatrixXd::Identity(n, n) - h * 0.5 * A)
10            .partialPivLu()
11            .solve(Y0 + h * 0.5 * A * Y0);
12   // END
13   return Y1;
14 }
```

Get it on GitLab (matode.hpp).

!

The method `PartialPivLu()` can also be replaced with the generic call `lu()`.

SOLUTION of (11-1.c):

C++11-code 11.1.5: Orthogonality testing code.

```
std::tuple<double, double, double> checkOrthogonality() {
  constexpr unsigned int n = 3;
  constexpr double h = 0.01;
  Eigen::MatrixXd M(n, n);
  M << 8, 1, 6, 3, 5, 7, 9, 9, 2;

  // Build A
  Eigen::MatrixXd A(n, n);
  A << 0, 1, 1, -1, 0, 1, -1, -1, 0;

  std::vector<double> norms(3, -1.);  //< the output

  // TODO: (11-1.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

  // Build Q
  Eigen::HouseholderQR<Eigen::MatrixXd> qr(n, n);
  qr.compute(M);
  Eigen::MatrixXd Q = qr.householderQ();

  Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);

  Eigen::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;

  // 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;
  }
  // END
  return std::make_tuple(norms[0], norms[1], norms[2]);
```

```
54 }
```

Get it on GitLab (matode.hpp).

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.

SOLUTION of (11-10.a):

In this case we have

$$\mathbf{f}(\mathbf{q},t) = q \quad , \quad \mathbf{g}(\mathbf{p}) = -p \quad \Rightarrow \quad f(q) := q \quad , \quad g(p) := -p \, .$$

Also note that the timestepping scheme (11.10.2) is *explicit* and, hence, the implementation is straightforward.

C++ code 11.10.5: Sub-problem (11-10.a): A single timestep of the symplectic single-step method

```
2  void sympTimestep(double tau, Eigen::Vector2d &pq_j) {
3    // Coefficients of the method
4    const Eigen::Vector3d a{2. / 3., -2. / 3., 1.};
5    const Eigen::Vector3d b{7. / 24., 3. / 4., -1. / 24.};
6    // TODO: (11-10.a) Implement a symplectic timestep on the harmonic
       oscillator
7    // ODE.
8    // START
9    // one step of the method
10   for (unsigned int i = 0; i < 3; ++i) {
11     pq_j(0) += tau * b(i) * pq_j(1);
12     pq_j(1) -= tau * a(i) * pq_j(0);
13   }
14   // END
15 }
```

Get it on GitLab (symplectic.hpp).

HINT 1 for (11-10.b): The exact solution of the initial value problem for (11.10.4) is $p(t) = \sin(t)$ and $q(t) = \cos(t)$, hence $p(2\pi) = 0$ and $q(2\pi) = 1$. ⌟

SOLUTION of (11-10.b):

C++ code 11.10.6: Sub-problem (11-10.a): Monitoring convergence of symplectic timestepping method for oscillator equation

```
void sympTimesteppingODETest() {
  constexpr unsigned int nIter = 7;   // total number of iterations
  unsigned int m;                     // number of equidistant steps
  std::vector<double> errors(nIter);  // errors vector for all approx. sols
  Eigen::Vector2d approx_sol;
  approx_sol << 0.0, 1.0;  // initial conditions
  for (int k = 0; k < nIter; k++) {
    m = 10 * std::pow(2, k);
    // TODO: (11-10.b) Evaluate the error at the final step between the approx
    // solutions as given by the symplectic method and the exact solution
    // computed from the analytic formula.
    // START
    const double tau = 2.0 * M_PI / m;
    for (unsigned int i = 0; i < m; i++) {
      sympTimestep(tau, approx_sol);
    }
    // Computing the error in the maximum norm
    errors[k] = std::abs(std::sin(2.0 * M_PI) - approx_sol[0]) +
                std::abs(std::cos(2.0 * M_PI) - approx_sol[1]);
    // END
  }
  // Printing results
  std::cout << "Convergence of Symplectic Time Stepping Method:\n";
  std::cout << "\tM\t\terr(M)\n";
  for (unsigned int k = 0; k < nIter; k++) {
    std::cout << "\t" << 10 * std::pow(2, k) << "\t\t" << errors[k]
              << std::endl;
  }
}
```

Get it on GitLab (symplectic.hpp).

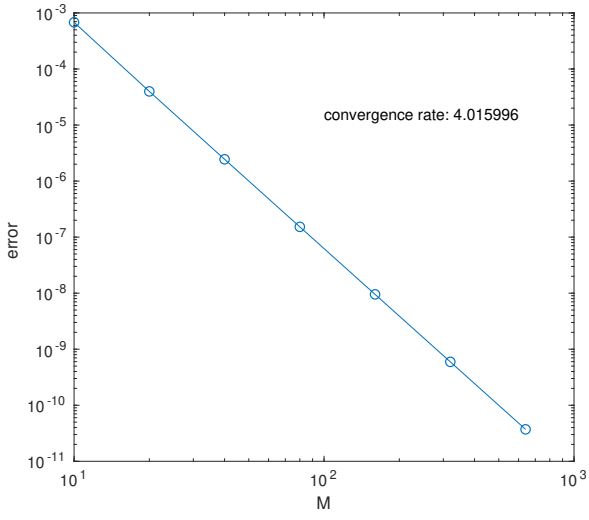


Table of errors:

M	$\mathrm{err}(M)$
10	0.000680607
$10 \cdot 2^1$	3.9793e-05
$10 \cdot 2^2$	2.4477e-06
$10 \cdot 2^3$	1.52386e-07
$10 \cdot 2^4$	9.51496e-09
$10 \cdot 2^5$	5.94542e-10
$10 \cdot 2^6$	3.71564e-11

◁ Log-log plot of `err` vs. M. The error curve is almost a line and from its slope we can read off a convergence rate of ~ 4.

A doubling of M leads to a decrease of the error by roughly a factor of 16: $\mathrm{err}(M) = O(M^{-4})$, algebraic

convergence with rate 4.

SOLUTION of (11-10.c):

In order to show that $t \mapsto E(t)$ is constant we prove that its derivative vanishes: Using the *chain rule*, we get

$$\begin{aligned}
\dot{E}(t) &= \left(\mathbf{grad} \left\{ \begin{bmatrix} \mathbf{p} \\ \mathbf{q} \end{bmatrix} \mapsto H(\mathbf{p},\mathbf{q}) \right\} (\mathbf{p}(t),\mathbf{q}(t)) \right)^\top \frac{d}{dt} \begin{bmatrix} \mathbf{p} \\ \mathbf{q} \end{bmatrix} (t) \\
&= \frac{\partial H}{\partial \mathbf{p}} (\mathbf{p}(t),\mathbf{q}(t))^\top \dot{\mathbf{p}}(t) + \frac{\partial H}{\partial \mathbf{q}} (\mathbf{p}(t),\mathbf{q}(t))^\top \dot{\mathbf{q}}(t) \\
&\overset{(11.10.7)}{=} -\frac{\partial H}{\partial \mathbf{p}} (\mathbf{p}(t),\mathbf{q}(t))^\top \frac{\partial H}{\partial \mathbf{q}} (\mathbf{p}(t),\mathbf{q}(t)) + \frac{\partial H}{\partial \mathbf{q}} (\mathbf{p}(t),\mathbf{q}(t))^\top \frac{\partial H}{\partial \mathbf{p}} (\mathbf{p}(t),\mathbf{q}(t)) \\
&= 0 \, .
\end{aligned}$$

SOLUTION of (11-10.d):

We calculate the partial derivatives using the chain rule for $h(\mathbf{x}) = (f \circ g)(\mathbf{x})$, $f(\xi) := \xi^2$, $g(\mathbf{x}) := \|\mathbf{x}\|^2$, in the case of the first one:

$$\frac{\partial H}{\partial \mathbf{q}}(\mathbf{p},\mathbf{q}) = \mathbf{grad}\left\{\mathbf{x} \mapsto \|\mathbf{x}\|_2^4\right\}(\mathbf{q}) = 4\|\mathbf{q}\|_2^2\mathbf{q}\,,$$
$$\frac{\partial H}{\partial \mathbf{p}}(\mathbf{p},\mathbf{q}) = \mathbf{grad}\left\{\mathbf{x} \mapsto \tfrac{1}{2}\|\mathbf{x}\|_2^2\right\}(\mathbf{p}) = \mathbf{p}\,.$$

These derivatives can also be computed “in the pedestrian way” by componentwise differentiation.

Hence, the Hamiltonian ODE for the special Hamiltonian from (11.10.9) is of the form (11.10.1) with

$$\mathbf{f}(\mathbf{q},t) := -4\|\mathbf{q}\|^2\mathbf{q} \quad , \quad \mathbf{g}(\mathbf{p}) := \mathbf{p}\,. \tag{11.10.10}$$

SOLUTION of (11-10.e):

The implementation of (11.10.2) is straightforward. The inner loop can be replaced with a sequence of assignments.

C++ code 11.10.11: Sub-problem (11-10.e): Symplectic timestepping for (11.10.10)

```
Eigen::MatrixXd simulateHamiltonianDynamics(const Eigen::VectorXd &p0,
                                            const Eigen::VectorXd &q0, double T,
                                            unsigned int M) {
  const unsigned int n = p0.size();
  Eigen::MatrixXd PQ = Eigen::MatrixXd::Zero(2 * n, M + 1);
  // Coefficients of the method
  const Eigen::Vector3d a{2. / 3., -2. / 3., 1.};
  const Eigen::Vector3d b{7. / 24., 3. / 4., -1. / 24.};
  const double tau = T / M;
  Eigen::VectorXd pj(p0), qj(q0);
  PQ.col(0) << pj, qj;
  // TODO: (11-10.e) Solve the Hamiltonian ODE using symplectic timestepping.
  // Store the states in the columns of PQ.
  // START
  for (unsigned int j = 1; j <= M; j++) {   // integrate
    for (unsigned int k = 0; k < 3; k++) {  // one step
      // f(q) = - 4 * |q|^2 * q
      pj -= tau * b(k) * 4. * qj.squaredNorm() * qj;
      // g(p) = p
      qj += tau * a(k) * pj;
      PQ.col(j) << pj, qj;
    }
  }
  // END
  return PQ;
}
```

Get it on GitLab (symplectic.hpp).

SOLUTION of (11-10.f):

Owing to the result of Sub-problem (11-10.c), for the exact solution $t \mapsto (\mathbf{p}(t), \mathbf{q}(t))$, $t \in \mathbb{R}$, we have

$$E(t) = H(\mathbf{p}_0, \mathbf{q}_0) = 1 \quad \forall t \in \mathbb{R} \, .$$

Thus, it is natural to study the difference $\mathcal{E}(M) - 1$. We observe that this difference approximately shrinks by a factor of 8 whenever the number of timesteps is doubled. This is a clear indication of *algebraic convergence* $\mathcal{E}(M) \to 1$ for $M \to \infty$ with *rate* 3.

HINT 1 for (11-11.a): Remember that a single-step method defined by the discrete evolution operator $\boldsymbol{\Psi} = \boldsymbol{\Psi}(h, \mathbf{y})$ is **consistent** with the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on the state space $D \subset \mathbb{R}^N$, if there exists a continuous function $\psi : I \times D \to \mathbb{R}^N$, $I \subset \mathbb{R}$ an interval containing 0, such that

$$\boldsymbol{\Psi}(h, \mathbf{y}) = \mathbf{y} + h\psi(h, \mathbf{y}) \quad \text{and} \quad \psi(0, \mathbf{y}) = \mathbf{f}(\mathbf{y}) \,. \tag{11.11.1}$$

⌟

SOLUTION of (11-11.a):

When applied to an autonomous ODE $\dot{\mathbf{y}} = \mathbf{F}(\mathbf{y})$ the discrete evolution operator Ψ for the explicit Runge-Kutta single step method according to [Lecture → Def. 11.4.0.11] is given by

$$\Psi(h,\mathbf{y}) = \mathbf{y} + h\sum_{i=1}^{s} b_i \mathbf{k}_i(h,\mathbf{y}) \,, \tag{11.11.2}$$

$$\mathbf{k}_i(h,\mathbf{y}) := \mathbf{f}(\mathbf{y} + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j(h,\mathbf{y})) \,, \quad i = 1,\ldots,s \,. \tag{11.11.3}$$

Matching with (11.11.1), we can identify

$$\psi(h,\mathbf{y}) = \sum_{i=1}^{s} b_i \mathbf{k}_i(h,\mathbf{y}) \,, \quad \mathbf{k}_i(h,\mathbf{y}) := \mathbf{f}(\mathbf{y} + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j(h,\mathbf{y})) \,, \quad i = 1,\ldots,s \,.$$

Setting $h = 0$ we end up with

$$\psi(0,\mathbf{y}) = \sum_{i=1}^{s} b_i \mathbf{k}_i(0,\mathbf{y}) \,, \quad \mathbf{k}_i(0,\mathbf{y}) = \mathbf{f}(\mathbf{y}) \,, \quad i = 1,\ldots,s \,.$$

Thus, the condition on $\psi(0,\mathbf{y})$ from (11.11.1) is satisfied for any $\mathbf{f}$, if

$$\sum_{i=1}^{s} b_i = 1 \,.$$

This is already stated in [Lecture → Cor. 11.4.0.15]. No conditions have to be met by the other coefficients.

HINT 1 for (11-11.b): The following C++ code snippet

```
unsigned int n; TYPE x; std::vector<TYPE> v(n,x);
```

initializes a vector of length `n` with all its entries set to the value passed in `x`. ⌟

SOLUTION of (11-11.b):

The implementation is straightforward from [Lecture → Def. 11.4.0.11]. In the code below the argument of the right-hand-side function $\mathbf{f}$ in the increment equations is first assembled in $\mathbf{k}_i$ and then we set `k[i] = f_(k[i])`.

C++ code 11.11.5: Implementation of evaluation operator for ExplRKSSMEvolOp
Get it on GitLab.

```
template <typename RHSFunctor>
template <typename State>
State ExplRKSSMEvolOp<RHSFunctor>::operator()(double h, const State& y) const {
  unsigned int s = A_.cols();
  assert(s == A_.rows());
  assert(s == b_.size());

  State y1{y};
  std::vector<State> k{s, y};
  for (int i = 0; i < (int)s; ++i) {
    for (int j = 0; j < i; ++j) {
      k[i] += h * A_(i, j) * k[j];
    }
    k[i] = f_(k[i]);
    y1 += h * b_[i] * k[i];
  }
  return y1;
}
```

SOLUTION of (11-2.a):

There are many ways to implement the formulas of [Lecture → Def. 11.4.0.11]. Thanks to the restriction to explicit RK-SSM all merely involve $\mathbf{f}$-evaluations.

C++11-code 11.2.1: Solution of (11-2.a).

```
2  template <class State>
3  class RKIntegrator {
4   public:
5    /**
6     * \brief Constructor for the RK method.
7     * Performs size checks and copies A and b into internal storage.
8     *
9     * \param A Matrix containing coefficents of the Butcher tableau,
10    * must be (strictly) lower triangular (no check is done).
11    * \param b Vector containing coefficients of lower
12    * part of Butcher tableau.
13    */
14   RKIntegrator(const Eigen::MatrixXd &A, const Eigen::VectorXd &b)
15       // TODO: (11-2.a) Use an initializer list to Initialize the private
16       // variables A, b, s.
17       // START
18       : A(A),
19         b(b),
20         s(b.size())
21   // END
22   {
23     // TODO: (11-2.a) Use assert() to make sure A is a square matrix of
24     // shape s*s with s=b.size().
25     // START
26     assert(A.cols() == A.rows() && "Matrix must be square.");
27     assert(A.cols() == b.size() && "Incompatible matrix/vector size.");
28     // END
29   }
30
31   /**
32    * \brief Perform the solution of the ODE.
33    * Solve an autonomous ODE y' = f(y), y(0) = y0, using a
34    * RK scheme given in the Butcher tableau provided in the
35    * constructor. Performs N equidistant steps up to time
36    * T with initial data y_0.
37    *
38    * \tparam Function type for function implementing the rhs function.
39    * Must have State operator()(const State & x).
40    * \param f function handle for rhs in y' = f(y), e.g.
41    * implemented using lambda funciton.
42    * \param T The final time T.
43    * \param y0 Initial data y(0) = y_0 for y' = f(y).
44    * \param N  Number of steps to perform. Step size is h = T/N.
45    * Steps are equidistant.
46    * \return std::vector<State> containing all steps y^n (for each n)
47    * including initial and final value.
48    */
49   template <class Function>
50   std::vector<State> solve(Function &&f, double T, const State &y0,
```

```
51                                  unsigned int N) const {
52      std::vector<State> res;
53      // Will contain all steps, reserve memory for efficiency
54      res.reserve(N + 1);
55      // TODO: (11-2.a) Compute the solution of y'=f(y), y(0)=y0 up to time T
56      // using N Runge-Kutta steps, and store the solution at step n in res[n].
57      // Use the private method step() for the Runge-Kutta step.
58      // START
59      // Step size
60      const double h = T / N;
61      // Will contain all steps, reserve memory for efficiency
62      res.reserve(N + 1);
63      // Store initial data
64      res.push_back(y0);
65
66      // Initialize some memory to store temporary values
67      State ytemp1 = y0;
68      State ytemp2 = y0;
69      // Pointers to swap previous value
70      State *yold = &ytemp1;
71      State *ynew = &ytemp2;
72
73      // Loop over all fixed steps
74      for (unsigned int k = 0; k < N; ++k) {
75        // Compute, save and swap next step
76        step(f, h, *yold, *ynew);
77        res.push_back(*ynew);
78        std::swap(yold, ynew);
79      }
80      // END
81      return res;
82    }
83
84  private:
85    /**
86     * \brief Perform a single step of the RK method.
87     * Solve an autonomous ODE using an explicit Runge Kutta Method.
88     * Compute a single explicit RK step y^{n+1} = y_n + \sum...
89     * starting from value y_0 and storing next value in y_1.
90     *
91     * \tparam Function  type for function implementing the rhs.
92     * Must have State operator()(State x)
93     * \param f function handle for rhs f, s.t. y' = f(y)
94     * \param h step size
95     * \param y0 initial state
96     * \param y1 next step y^{n+1} = y^n + ...
97     */
98    template <class Function>
99    void step(const Function &f, double h, const State &y0, State &y1) const {
100     // TODO: (11-2.a) Compute a single step of the explicit Runge-Kutta method
101     // (defined by A, b, s) for y'=f(y) starting at State y0 and using
102     // step size h. Store the result in the State y1.
103     // START
104     // create vector holding next value
105     y1 = y0;
106     // Reserve space for increments
107     std::vector<State> k;
```

```
108       k.reserve(s);
109
110       // Loop over the size of RK
111       for (unsigned int i = 0; i < s; ++i) {
112         // Compute increments and save them to k
113         State incr = y0;
114         for (unsigned int j = 0; j < i; ++j) {
115           incr += h * A(i, j) * k.at(j);
116         }
117         k.push_back(f(incr));
118         y1 += h * b(i) * k.back();
119       }
120       // END
121     }
122
123     const Eigen::MatrixXd A;  //!< Matrix A in Butcher scheme
124     const Eigen::VectorXd b;  //!< Vector b in Butcher scheme
125     unsigned int s;  //!< Size of Butcher matrix and vector A and b
126   };
```

Get it on GitLab (rkintegrator.hpp).

There is no real reason to introduce the method `step()`; everything can be implemented in `solve()`.

SOLUTION of (11-2.b):

The main task is to prepare the inputs for the application of **RKIntegrator**. The right-hand-side function **f** can be passed as lambda function, see [Lecture → Section 0.3.3].

C++11-code 11.2.6: Solution of (11-2.b)

```
2  double RK3prey() {
3    double conv_rate = 0;
4    // Dimension of state space
5    constexpr unsigned int d = 2;
6    // Exact value y(10) at final time T = 10 (approximated)
7    Eigen::VectorXd yex(d);
8    yex << 0.319465882659820, 9.730809352326228;
9
10   // Implementation of butcher scheme
11   constexpr unsigned int s = 3;
12   Eigen::MatrixXd A(s, s);
13   Eigen::VectorXd b(s);
14   A << 0, 0, 0, 1. / 3., 0, 0, 0, 2. / 3., 0;
15   b << 1. / 4., 0, 3. / 4.;
16
17   // TODO: (11-2.b) Use an RKIntegrator to solve the predator/prey IVP up to
18   // T=10, as described in the task description. Tabulate the error as the
19   // number of steps increases N = 2^7, 2^8, ..., 2^14, and estimate the
20   // convergence rate.
21   // HINT: You may use polyfit() to calculate the convergence
22   // rate.
23   // START
24
25   // Initialize RK with Butcher scheme
26   RKIntegrator<Eigen::VectorXd> RK(A, b);
27
28   // Coefficients and handle for prey/predator model
29   constexpr double alpha1 = 3.;
30   constexpr double alpha2 = 2.;
31   constexpr double beta1 = 0.1;
32   constexpr double beta2 = 0.1;
33   auto f = [](const Eigen::VectorXd& y) {
34     auto temp = y;
35     temp(0) *= alpha1 - beta1 * y(1);
36     temp(1) *= -alpha2 + beta2 * y(0);
37     return temp;
38   };
39
40   // Initial value for model
41   Eigen::VectorXd y0(d);
42   y0 << 100, 5;
43
44   // Final time for model
45   constexpr double T = 10.;
46
47   // Array of number of steps (for convergence study)
48   Eigen::ArrayXd N(8);
49   N << 128, 256, 512, 1024, 2048, 4096, 8192, 16384;
50   Eigen::ArrayXd Error(N.size());
```

```
51
52    // Start convergence study
53    std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::endl;
54    for (unsigned int i = 0; i < N.size(); ++i) {
55      auto res = RK.solve(f, T, y0, N[i]);
56      const double err = (res.back() - yex).norm();
57      std::cout << std::setw(15) << N[i] << std::setw(15) << err << std::endl;
58      Error[i] = err;
59    }
60
61    Eigen::VectorXd coeffs = polyfit(N.log(), Error.log(), 1);
62    conv_rate = -coeffs(0);
63    // END
64    return conv_rate;
65  }
```

Get it on GitLab (rk3prey.hpp).

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:

$$\operatorname{err}(\tfrac{1}{2}h) : \operatorname{err}(h) \approx 2^{-\alpha} \quad \Leftrightarrow \quad \alpha \approx \frac{\log \operatorname{err}(\tfrac{1}{2}h) - \log \operatorname{err}(h)}{\log 2} .$$

Obviously, the 3-stage RK-SSM given in (11.2.5) is of order 3.

SOLUTION of (11-3.a):

Writing the explicit/forward Euler discrete evolution as $\boldsymbol{\Psi}^h(\mathbf{y}_0) := \mathbf{y}_0 + hf(\mathbf{y}_0)$ (and $p = 1$), we obtain:

$$\begin{aligned}\widetilde{\boldsymbol{\Psi}}^h(\mathbf{y}_0) &= \frac{1}{1-2^p}\left(\mathbf{y}_0 + hf(\mathbf{y}_0) - 2^p \cdot (\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0) + \frac{h}{2}f(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right)\\ &= \mathbf{y}_0 + \frac{1}{1-2^p}\left(h(1-2^{p-1})f(\mathbf{y}_0) - 2^{p-1}hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right)\\ &= \mathbf{y}_0 + \left(hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right) .\end{aligned}$$

This is the evolution operator of the **explicit midpoint method**, a 2-stage explicit Runge-Kutta method, see [Lecture → Eq. (11.4.0.8)].

SOLUTION of (11-3.b):

The formula (11.3.1) can be implemented directly:

C++ code 11.3.2: Solution of (11-3.b).

```
template <class DiscEvlOp>
Eigen::VectorXd psitilde(DiscEvlOp &&Psi, unsigned int p, double h,
                         const Eigen::VectorXd &y0) {
  Eigen::VectorXd y1 = y0;  // overwrite this
  // TODO: (11-3.b) apply the evolution operator \tilde{\Psi}
  // with step-size h to the value y0
  // START
  y1 = (Psi(h, y0) - (2 << (p - 1)) * Psi(h / 2., Psi(h / 2., y0))) /
       (1. - (2 << (p - 1)));
  // END
  return y1;
}
```

Get it on GitLab (odesolve.hpp).

HINT 1 for (11-3.c): You can use **std::vector**<Eigen::VectorXd>::push_back() to add elements to the end of a **std::vector**. ⌟

SOLUTION of (11-3.c):

Requested is a straightforward implementation of [Lecture → Eq. (11.3.1.6)].

C++ code 11.3.3: Solution of Sub-problem (11-3.c).

```
2  template <class DiscEvlOp>
3  std::vector<Eigen::VectorXd> odeintequi(DiscEvlOp &&Psi, double T,
4                                          const Eigen::VectorXd &y0,
5                                          unsigned int N) {
6    std::vector<Eigen::VectorXd> Y;
7    // TODO: (11-3.c) Compute y from time 0 to T using N equidistant time steps
8    // return a std::vector containing all steps y_0,...,y_N
9    // START
10   const double h = T / N;
11   double t = 0.;
12   Y.reserve(N + 1);
13   Y.push_back(y0);
14   Eigen::VectorXd y = y0;
15
16   while (t < T) {
17     // append new value
18     y = Psi(h, Y.back());
19     Y.push_back(y);
20     // time stepping
21     t += std::min(T - t, h);
22   }
23   // END
24   return Y;
25 }
```

Get it on GitLab (odesolve.hpp).

HINT 1 for (11-3.d): The solution of the IVP Eq. (11.3.4) is $\tan(t)$. ⌟

SOLUTION of (11-3.d):

C++ code 11.3.5: Sub-problem (11-3.d): Implementation of `testcvpExtrapolatedEuler()`

```
double testcvpExtrapolatedEuler() {
  double conv_rate = 0.;
  constexpr double T = 1.;
  Eigen::VectorXd y0 = Eigen::VectorXd::Zero(1);
  auto f = [](const Eigen::VectorXd &y) -> Eigen::VectorXd {
    return Eigen::VectorXd::Ones(1) + y * y;
  };
  // TODO: (11-3.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 Eigen::VectorXd &y0) -> Eigen::VectorXd {
    return y0 + h * f(y0);
  };
  constexpr unsigned int p = 1;
  // lambda corresponding to \tilde{\psi}
  auto PsiTilde = [&Psi](double h,
                         const Eigen::VectorXd &y0) -> Eigen::VectorXd {
    return psitilde(Psi, p, h, y0);
  };

  // exact value
  Eigen::VectorXd 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 (unsigned int i = 0; i < 11; ++i) {
    N(i) = std::pow(2, i + 2);
    Eigen::VectorXd 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
  Eigen::VectorXd coeffs = polyfit(N.log(), err.log(), 1);
  conv_rate = -coeffs(0);
  // END
  return conv_rate;
}
```

Get it on GitLab (odesolve.hpp).

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 $\widetilde{\mathbf{\Psi}}^h$ from (11.3.1) enjoys order 2.
Conversely, the order of the underlying forward Euler method is $p = 1$.
Hence, the construction from (11.3.1) manages to raise the order of the method by 1.

HINT 1 for (11-3.e): Just adapt [Lecture → Code 11.5.2.18] using the discrete evolutions $\boldsymbol{\Psi}$ and $\widetilde{\boldsymbol{\Psi}}$ from (11.3.1) instead of the generic **Psilow** and **Psihigh**. ⌟

SOLUTION of (11-3.e):

C++ code 11.3.6: Solution of Sub-problem (11-3.e).

```
template <class DiscEvlOp>
std::pair<std::vector<double>, std::vector<Eigen::VectorXd>> odeintssctrl(
    DiscEvlOp &&Psi, double T, const Eigen::VectorXd &y0, double h0,
    unsigned int p, double reltol, double abstol, double hmin) {
  std::vector<double> t;
  std::vector<Eigen::VectorXd> Y;
  // TODO: (11-3.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);
  Eigen::VectorXd y = y0;

  double h = h0;
  while (t.back() < T && h > hmin) {
    // psitilde as higher order step
    Eigen::VectorXd y_high = psitilde(Psi, p, std::min(T - t.back(), h), y);
    Eigen::VectorXd y_low = Psi(std::min(T - t.back(), h), y);
    // estimated error of current step
    const double est = (y_high - y_low).norm();
    // effective tolerance
    const 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);
}
```

Get it on GitLab (odesolve.hpp).

SOLUTION of (11-3.f):

C++ code 11.3.7: Implementation of `solveTangentIVP()`

```
void solveTangentIVP() {
  auto f = [](const Eigen::VectorXd &y) -> Eigen::VectorXd {
    return Eigen::VectorXd::Ones(1) + y * y;
  };
  Eigen::VectorXd y0 = Eigen::VectorXd::Zero(1);
  plt::figure();
  // TODO: (11-3.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<Eigen::VectorXd> into a
  // std::vector<double>, since each Eigen::VectorXd has size 1

  // START
  constexpr double T = 1.5;
  constexpr unsigned int p = 1;
  constexpr double h0 = 1. / 100.;

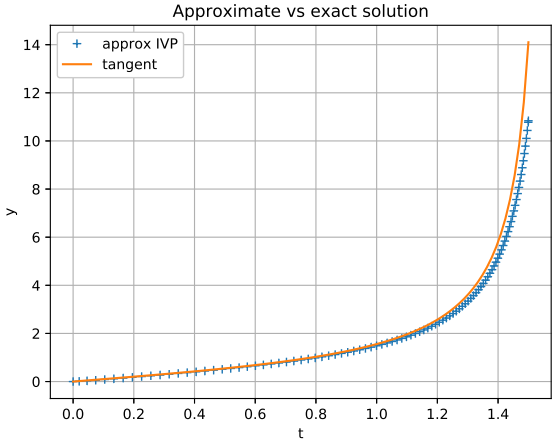
  auto Psi = [&f](double h, const Eigen::VectorXd &y0) -> Eigen::VectorXd {
    return y0 + h * f(y0);
  };
  // run adaptive algoritm
  std::pair<std::vector<double>, std::vector<Eigen::VectorXd>> vec_pair =
      odeintssctrl(Psi, T, y0, h0, p, 10e-4, 10e-6, 10e-5);
  std::vector<double> t = vec_pair.first;
  std::vector<Eigen::VectorXd> 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::plot(t, y, "+", {{"label", "approx IVP"}});
  // exact solution
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(100, 0.0, T);
  Eigen::VectorXd 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");

  // END
  plt::savefig("./cx_out/tangent.png");
}
```

Get it on GitLab (odesolve.hpp).



◁ ◁ As in [Lecture → Ex. 11.5.1.4] the adaptive timestep control manages to detect the blow-up of the solution and reduces the timestep when approach the time $t_* = \pi/2$ from which the solution ceases to exist.

HINT 1 for (11-4.a): What we face in (11.4.1) 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 (11-4.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 (11-4.a): See [Lecture → Rem. 11.1.3.6] for a recipe on how to convert a second-order ODE into a first-order system. ⌟

SOLUTION of (11-4.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 $\mathbf{v}(t) := \dot{\mathbf{u}}(t)$

$$\begin{aligned}
\dot{u}_i &= v_i \qquad i = 1, \ldots, n \,, \\
2\dot{v}_1 - \dot{v}_2 &= u_1(u_2 + u_1) \,, \\
-\dot{v}_{i-1} + 2\dot{v}_i - \dot{v}_{i+1} &= u_i(u_{i-1} + u_{i+1}) \qquad i = 2, \ldots, n-1 \,, \\
2\dot{v}_n - \dot{v}_{n-1} &= u_n(u_n + u_{n-1}) \,, \\
u_i(0) &= u_{0,i} \qquad i = 1, \ldots, n \,, \\
v_i(0) &= v_{0,i} \qquad i = 1, \ldots, n \,.
\end{aligned}$$

The ODE system can then be written in the vector form

$$\dot{\mathbf{u}} = \mathbf{v} \,, \qquad \mathbf{C}\dot{\mathbf{v}} = \mathbf{r}(u_1, \ldots, u_n) := \begin{bmatrix} u_1(u_2 + u_1) \\ \vdots \\ u_i(u_{i-1} + u_{i+1}) \\ \vdots \\ u_n(u_n + u_{n-1}) \end{bmatrix} \,,$$

where $\mathbf{C} \in \mathbb{R}^{n,n}$ is:

$$\mathbf{C} := \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \,.$$

In order to be able to use standard interfaces to RK-SSM for first-order ODEs, collect $\mathbf{u}$ and $\mathbf{v}$ in a $(2n)$-dimensional vector $\mathbf{y} := \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}$. Then the final system of first-order ODEs reads

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_{n+1} \\ \vdots \\ y_{2n} \\ \mathbf{C}^{-1}\mathbf{r}(y_1, \ldots, y_n) \end{bmatrix} \,.$$

HINT 1 for (11-4.b): The general formulas are given in [Lecture → Def. 11.4.0.11] and their connection with Butcher schemes is made explicit in [Lecture → Eq. (11.4.0.13)].

SOLUTION of (11-4.b):

The task boils down to specializing the formulas of [Lecture → Def. 11.4.0.11] for $\mathbf{f}(\mathbf{y}) = \mathbf{M}\mathbf{y}$:

$$\begin{aligned}
\mathbf{k}_1 &:= \mathbf{M}\mathbf{y}_n \,, \\
\mathbf{k}_2 &:= \mathbf{M}\left(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_1\right) \,, \\
\mathbf{k}_3 &:= \mathbf{M}\left(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_2\right) \,, \\
\mathbf{k}_4 &:= \mathbf{M}(\mathbf{y}_n + h\mathbf{k}_3) \,, \\
\mathbf{y}_{n+1} &:= \mathbf{y}_n + \frac{h}{6}\mathbf{k}_1 + \frac{h}{3}\mathbf{k}_2 + \frac{h}{3}\mathbf{k}_3 + \frac{h}{6}\mathbf{k}_4 \,, \\
\blacktriangleright \quad \mathbf{y}_{n+1} = \boldsymbol{\Psi}^h\mathbf{y}_n &:= \mathbf{y}_n + h\mathbf{M}\mathbf{y}_n + \frac{h^2}{2}\mathbf{M}^2\mathbf{y}_n + \frac{h^3}{6}\mathbf{M}^3\mathbf{y}_n + \frac{h^4}{24}\mathbf{M}^4\mathbf{y}_n \,,
\end{aligned} \tag{11.4.2}$$

where h is the size of the timestep and $\boldsymbol{\Psi}^h$ stands for the discrete evolution operator underlying the single-step method, see [Lecture → Def. 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**

$$\exp(\mathbf{X}) := \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{X}^k \,, \quad \mathbf{X} \in \mathbb{R}^{d,d} \,. \tag{11.4.3}$$

This means that $\mathbf{y}(t_{k+1}) = \exp(\mathbf{M}h)\mathbf{y}(t_k)$ and the evolution operator according to [Lecture → Def. 11.1.4. for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ has the form $\boldsymbol{\Phi}^h\mathbf{y} = \exp(\mathbf{M}h)\mathbf{y}$. Comparing with the formula for $\boldsymbol{\Psi}^h$ we notice that $\boldsymbol{\Psi}^h$ can be obtained by truncating the power series for $\boldsymbol{\Phi}^h$ after the fifth term.

SOLUTION of (11-4.c):

Using the Butcher scheme from [Lecture → Ex. 11.4.0.17], the implementation is straightforward from [Lecture → Def. 11.4.0.11] and [Lecture → Eq. (11.4.0.13)].

C++ code 11.4.4: Solution of Sub-problem (11-4.c)

```
2  template <class Function, class State>
3  void rk4step(Function &&odefun, double h, const State &y0, State &y1) {
4    // TODO: (11-4.c) Implement a single step of RK4 for
5    // the ODE y' = odefun(y), starting from y0 and using
6    // the step size h. Save the result in y1.
7    // START
8    auto k1 = odefun(y0);
9    auto k2 = odefun(y0 + h / 2 * k1);
10   auto k3 = odefun(y0 + h / 2 * k2);
11   auto k4 = odefun(y0 + h * k3);
12
13   y1 = y0 + h / 6 * k1 + h / 3 * k2 + h / 3 * k3 + h / 6 * k4;
14   // END
15 }
```

Get it on GitLab (system.hpp).

SOLUTION of (11-4.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 11.4.5: Solution of Sub-problem (11-4.d)

```
2  double testcvgRK4() {
3    // Parameters and initial values:
4    constexpr double T = 1;
5    constexpr unsigned int n = 5;
6    Eigen::VectorXd y0(2 * n);
7    for (unsigned int i = 0; i < n; ++i) {
8      y0(i) = (i + 1.) / n;
9      y0(i + n) = -1;
10   }
11
12   // TODO: (11-4.d) Implement the function f from (12-4.a) as a lambda
13   // function, i.e. the right hand side of the system of first order ODEs.
14   // START
15   // Define the relevant tridiagonal matrix.
16   Eigen::SparseMatrix<double> C(n, n);
17   // Each column has at most 3 non-zero entries.
18   C.reserve(Eigen::VectorXi::Constant(n, 3));
19   C.insert(0, 0) = 2;
20   for (unsigned int i = 1; i < n; ++i) {
21     C.insert(i, i) = 2;
22     C.insert(i, i - 1) = -1;
23     C.insert(i - 1, i) = -1;
24   }
25   C.makeCompressed();
26
27   auto f = [n, C](Eigen::VectorXd y) {
28     // The system of ODEs is y' = f(y) with y = [u;v],
29     // and f(y) = f([u;v]) = [v;C^{-1}g(u)]
30     Eigen::VectorXd fy(2 * n);
31     fy.head(n) = y.tail(n);
32
33     Eigen::VectorXd g(n);
```

```
    g(0) = y(0) * (y(1) + y(0));
    g(n - 1) = y(n - 1) * (y(n - 1) + y(n - 2));
    for (unsigned 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;

  // TODO: (11-4.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.
  const unsigned int N_exact = std::pow(2, 12);
  const double h = T / N_exact;  // step size
  Eigen::VectorXd yT_exact = y0;
  for (unsigned int step = 0; step < N_exact; step++) {
    // yT_exact is the solution at time t=h*step
    Eigen::VectorXd y_tmp;
    rk4step<std::function<Eigen::VectorXd(Eigen::VectorXd)>, Eigen::VectorXd>(
        f, h, yT_exact, y_tmp);
    yT_exact = y_tmp;
  }

  constexpr unsigned int kmax = 10;
  Eigen::VectorXd Error(kmax);
  for (unsigned int k = 0; k < kmax; k++) {
    const unsigned int N = std::pow(2, k + 1);  // number of steps
    const double h = T / N;                     // step size
    Eigen::VectorXd yT = y0;
    // Take N RK4 steps:
    for (unsigned int step = 0; step < N; step++) {
      // yT is the solution at time t=h*step
      Eigen::VectorXd y_tmp;
      rk4step<std::function<Eigen::VectorXd(Eigen::VectorXd)>, Eigen::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).
  Eigen::VectorXd logN =
```

```
91          std::log(2) * Eigen::VectorXd::LinSpaced(kmax, 1, kmax);
92      Eigen::VectorXd coeffs = polyfit(logN, Error.array().log(), 1);
93      conv_rate = -coeffs(0);
94      // END
95
96      return conv_rate;
97  }
```

Get it on GitLab (system.hpp).

SOLUTION of (11-5.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 11.5.2: Implementation of `matode()`.

```
2  Eigen::MatrixXd matode(const Eigen::MatrixXd &Y0, double T) {
3    Eigen::MatrixXd YT = Y0;  // overwrite this
4    // TODO: (11-5.a): use the ode45 class to find an approximation
5    // of the matrix IVP Y' = -(Y - Y')*Y at time T
6    // START
7    // Define the RHS
8    auto F = [](const Eigen::MatrixXd &M) { return -(M - M.transpose()) * M; };
9    ode45<Eigen::MatrixXd> O(F);
10
11   // Set tolerances
12   O.options.atol = 1e-10;
13   O.options.rtol = 1e-8;
14
15   // Return only matrix at T, (solution is vector
16   // of pairs (y(t_k),t_k) for each step k
17   YT = O.solve(Y0, T).back().first;
18   // END
19   return YT;
20 }
```

Get it on GitLab (NLmatode.hpp).

HINT 1 for (11-5.b): A time-dependent differentiable function is constant if and only if its derivative vanishes. ⌟

HINT 2 for (11-5.b): Use the product rule for differentiable matrix-valued functions $t \mapsto \mathbf{A}(t), \mathbf{B}(t)$

$$\frac{d}{dt}(\mathbf{A}(t)\mathbf{B}(t)) = \dot{\mathbf{A}}(t)\mathbf{B}(t) + \mathbf{A}(t)\dot{\mathbf{B}}(t) \,, \tag{11.5.3}$$

which is a special version of the general product rule [Lecture → Eq. (8.5.1.17)]. ⌟

SOLUTION of (11-5.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.

SOLUTION of (11-5.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 11.5.4: Implementation checking invariant.

```
2  bool checkinvariant(const Eigen::MatrixXd &M, double T) {
3    bool inv_holds = false;
4    // TODO: (11-5.c) check if Y'*Y is preserved at the time T by matode.
5    // START
6    Eigen::MatrixXd N = matode(M, T);
7
8    if ((N.transpose() * N - M.transpose() * M).norm() <
9        10 * std::numeric_limits<double>::epsilon() * M.norm()) {
10     inv_holds = true;
11   } else {
12     inv_holds = false;
13   }
14   // END
15   return inv_holds;
16 }
```

Get it on GitLab (NLmatode.hpp).

SOLUTION of (11-5.d):

Apart from realizing that the formal inverse matrix in (11.5.5) must be implemented as the solution of a linear system of equations, the implementation is straightforward.

C++ code 11.5.6: Implementation of `cvgDiscreteGradientMethod()`

```
double cvgDiscreteGradientMethod() {
  double conv_rate = 0;
  // TODO: (11-5.d) compute the fitted convergence rate of the Discrete
  // gradient method. Also tabulate the values M and the errors.
  // START
  constexpr double T = 1.0;
  // initial value
  Eigen::MatrixXd Y0 = Eigen::MatrixXd::Zero(5, 5);
  Y0(4, 0) = 1;
  for (unsigned int i = 0; i < 4; ++i) {
    Y0(i, i + 1) = 1;
  }
  // reference solution
  Eigen::MatrixXd Y_ex = matode(Y0, T);

  // define the rhs
  auto F = [](const Eigen::MatrixXd &M) { return -(M - M.transpose()) * M; };

  Eigen::MatrixXd I = Eigen::MatrixXd::Identity(5, 5);
  Eigen::ArrayXd MM(8);
  Eigen::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) {
    const unsigned int M = 10 * std::pow(2, i);
    const double h = T / M;
    MM(i) = M;
    Eigen::MatrixXd Y = Y0;
    for (unsigned int j = 0; j < M; ++j) {
      Eigen::MatrixXd Ystar = Y + 0.5 * h * F(Y);

      Eigen::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
  Eigen::VectorXd coeffs = polyfit(MM.log(), err.log(), 1);
  conv_rate = -coeffs(0);
  // END
  return conv_rate;
}
```

Get it on GitLab (NLmatode.hpp).

M	$\|\mathbf{Y}_M - \mathbf{Y}(T)\|_F$
10	0.00260457
20	0.000645272
40	0.000160726
80	4.00747e-05
160	9.9619e-06
320	2.44364e-06
640	5.84909e-07
1280	1.98872e-07

◁ Doubling the number of timesteps leads to a reduction of the error by roughly a factor of four.

This hints at *algebraic convergence* in the number of timesteps with rate 2: $\mathrm{err} = \mathrm{O}(\mathrm{M}^{-2}) = \mathrm{O}(\mathrm{h}^2)$ for $M \to \infty$, where $h > 0$ is the size of the uniform timestep.

We conclude that the discrete gradient method is a single-step method *of order* 2.

SOLUTION of (11-6.a):

Let us find an expression for the order of convergence. Set $M_k = 2^k$. We readily derive

$$\frac{E_{M_{k-1}}}{E_{M_k}} \approx \frac{C2^{-(k-1)r_k}}{C2^{-kr_k}} = 2^{r_k} \quad , \quad r_k \approx \log\left(\frac{E_{M_{k-1}}}{E_{M_k}}\right) / \log(2) \, .$$

A reasonable approximation of the order of convergence is given by

$$r \approx \frac{1}{\#K} \sum_{k \in K} r_k, \qquad K = \{k = 1, \ldots, 12 : E_{M_k} > 10^{-14}\}. \tag{11.6.2}$$

C++ code 11.6.3: Implementation of `testCvgRKSSM`

```
template <class Function>
double testCvgRKSSM(const Function &f, double T, double y0,
                    const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
  // Helper object carrying out the actual explicit RK-SSM
  RKIntegrator<double> rk(A, b);
  double conv_rate = 0;
  // TODO: (11-6.a) Print the errors at each final time point and
     estimate the
  // convergence rate.
  // START

  // Vector for collecting errors
  Eigen::VectorXd error(12);
  // Vector for storing No.  of timesteps
  Eigen::VectorXd timesteps(12);

  double sum = 0;
  unsigned int count = 0;
  bool test = true;

  // Reference numerical solution obtained with 2^15 timesteps
  std::vector<double> y_exact = rk.solve(f, T, y0, std::pow(2, 15));

  for (unsigned int k = 0; k < 12; k++) {
    // Number of timesteps
    const unsigned int M = std::pow(2, k + 1);
    timesteps(k) = M;
    // Solve IVP
    std::vector<double> y1 = rk.solve(f, T, y0, M);
    // Error at final time
    error(k) = std::abs(y1[M] - y_exact[std::pow(2, 15)]);

    std::cout << std::left << std::setfill(' ') << std::setw(3)
              << "M = " << std::setw(7) << M << std::setw(8)
              << "Error = " << std::setw(13) << error[k];

    if (error[k] < 1e-14) {
      test = false;
    }
    if (k > 0 && test) {
      const double rk = std::log2(error(k - 1) / error(k));
      std::cout << std::left << std::setfill(' ') << std::setw(10)
```

```
43                  << "Approximated order = " << rk << std::endl;
44          sum += rk;
45          ++count;
46        } else
47          std::cout << std::endl;
48      }
49      std::cout << "Average approximated order = " << sum / count << std::endl
50                << std::endl;
51      // Compute convergence rate by linear regression
52      Eigen::VectorXd fit_coeffs =
53          polyfit(timesteps.segment(0, count + 1).array().log(),
54                  error.segment(0, count + 1).array().log(), 1);
55      conv_rate = -fit_coeffs(0);
56      // END
57      return conv_rate;
58  }
```

Get it on GitLab (order_not_all.hpp).

HINT 1 for (11-6.b): For a scalar autonomous ODE the family of solutions can be determined by direct integration

$$\dot{y} = f(y) \quad \Rightarrow \quad \int_0^t \frac{\dot{y}(\tau)}{f(y(\tau))}\,\mathrm{d}\tau = t \quad \Rightarrow \quad \int_{y(0)}^{y(t)} \frac{1}{f(\sigma)}\,\mathrm{d}\sigma = t\,. \tag{11.6.6}$$

Thus, if $t \mapsto G(t)$ is a principal of f^{-1}, that is $\dot{G} = f^{-1}$, then

$$G(y(t)) = G(y(0)) + t \quad \Rightarrow \quad y(t) = G^{-1}(G(y(0)) + t)\,, \tag{11.6.7}$$

provided that G is monotone in the possible range of values of y. ⌟

HINT 2 for (11-6.b): For the logistic ODE the solution is given in [Lecture → Eq. (11.1.2.3)]. ⌟

HINT 3 for (11-6.b): For (11.6.5) you have to distinguish cases depending on the sign of $1.1 - y(t)$. ⌟

SOLUTION of (11-6.b):

As far as (11.6.4) is concerned, the solution is $y(t) = (1 + e^{-t})^{-1}$ (see [Lecture $\rightarrow$ Eq. (11.1.2.3)]).

Let us now consider (11.6.5). Because of the absolute value on the right hand side of the differential equation, we have to distinguish two cases $y(t) < 1.1$ and $y(t) > 1.1$. Since the initial condition is given by $y(0) = 1 < 1.1$, we start with the case $y(t) < 1.1$. For $y(t) < 1.1$, the differential equation is $\dot{y} = 2.1 - y$. Separation of variables

$$\int_1^{y(t)} \frac{1}{2.1 - \tilde{y}} d\,\tilde{y} = \int_0^t d\,\tilde{t}$$

yields the solution

$$y(t) = 2.1 - 1.1e^{-t}, \quad \text{for } y(t) < 1.1.$$

For $y(t) > 1.1$, the differential equation is given by $\dot{y} = y - 0.1$ with initial condition $y(\ln(\frac{11}{10})) = 1.1$, where the initial time t^* was derived from the condition $y(t^*) = 2.1 - 1.1e^{-t^*} \stackrel{!}{=} 1.1$. Separation of variables yields the solution for this IVP

$$y(t) = \tfrac{10}{11}e^t + 0.1.$$

Together, the solution of the initial IVP is given by

$$y(t) = \begin{cases} 2.1 - 1.1e^{-t}, & \text{for } t \leq \ln(1.1) \\ \frac{10}{11}e^t + 0.1, & \text{for } t > \ln(1.1). \end{cases}$$

SOLUTION of (11-6.c):

The following code is self-explanatory. We encode the various RK-SSMs through their Butcher schemes.

C++ code 11.6.8: Implementation of `cmpCvgRKSSM`

```
void cmpCvgRKSSM() {
  // TODO: (11-6.c) Print the convergence rates of four RK single step
     methods
  // on the two given ODEs.
  // START
  // Construct data for Butcher schemes
  Eigen::MatrixXd A1 = Eigen::MatrixXd::Zero(1, 1);
  Eigen::VectorXd b1(1);
  b1 << 1;

  Eigen::MatrixXd A2 = Eigen::MatrixXd::Zero(2, 2);
  A2(1, 0) = 1;
  Eigen::VectorXd b2(2);
  b2 << .5, .5;

  Eigen::MatrixXd A3 = Eigen::MatrixXd::Zero(3, 3);
  A3(1, 0) = .5;
  A3(2, 0) = -1;
  A3(2, 1) = 2;
  Eigen::VectorXd b3(3);
  b3 << 1. / 6, 2. / 3, 1. / 6;

  Eigen::MatrixXd A4 = Eigen::MatrixXd::Zero(4, 4);
  A4(1, 0) = .5;
  A4(2, 1) = .5;
  A4(3, 2) = 1;
  Eigen::VectorXd b4(4);
  b4 << 1. / 6, 1. / 3, 1. / 3, 1. / 6;

  // First ODE
  std::cout << std::endl
            << "1. ODE y' = (1-y)y, y(0)=.5" << std::endl
            << std::endl;
  constexpr double T = 1;
  auto f = [](double y) { return (1. - y) * y; };
  double y0 = .5;
  double rate;

  std::cout << "Explicit Euler" << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f, T, y0, A1, b1);
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;
  std::cout << "Trapezoidal rule" << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f, T, y0, A2, b2);
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;
  std::cout << "RK order 3" << std::endl << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f, T, y0, A3, b3);
```

```
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;
  std::cout << "Classical RK order 4" << std::endl << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f, T, y0, A4, b4);
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;

  // Second ODE
  std::cout << std::endl
            << "2. ODE y' = |1.1 - y| + 1, y(0)=1" << std::endl
            << std::endl;
  auto f2 = [](double y) { return std::abs(1.1 - y) + 1.; };
  y0 = 1;

  std::cout << "Explicit Euler" << std::endl << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f2, T, y0, A1, b1);
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;
  std::cout << "Trapezoidal rule" << std::endl << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f2, T, y0, A2, b2);
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;
  std::cout << "RK order 3" << std::endl << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f2, T, y0, A3, b3);
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;
  std::cout << "Classical RK order 4" << std::endl << std::endl;
  std::cout << std::string(50, '_') << std::endl;
  rate = testCvgRKSSM(f2, T, y0, A4, b4);
  std::cout << "Estimated rate using linear regression = " << rate << std::endl
            << std::endl;
  // END
}
```

Get it on GitLab (order_not_all.hpp).

The code produces the following output:

```
1. ODE y' = (1-y)y, y(0)=.5

Explicit Euler

__________________________________________________
M = 2       Error = 0.0111282
M = 4       Error = 0.00577587    Approximated order = 0.946109
M = 8       Error = 0.00292368    Approximated order = 0.982252
M = 16      Error = 0.00146886    Approximated order = 0.993093
M = 32      Error = 0.00073573    Approximated order = 0.997446
M = 64      Error = 0.000367898   Approximated order = 0.999868
M = 128     Error = 0.000183685   Approximated order = 1.00208
M = 256     Error = 9.15057e-05   Approximated order = 1.0053
M = 512     Error = 4.53983e-05   Approximated order = 1.01122
M = 1024    Error = 2.23403e-05   Approximated order = 1.02299
```

```
M = 2048    Error = 1.08102e-05  Approximated order = 1.04726
M = 4096    Error = 5.04482e-06  Approximated order = 1.09951
Estimated rate using linear regression = 1.0072

Trapezoidal rule

-------------------------------------------------------------
M = 2       Error = 0.00288256
M = 4       Error = 0.000677593  Approximated order = 2.08886
M = 8       Error = 0.000163389  Approximated order = 2.05211
M = 16      Error = 4.00683e-05  Approximated order = 2.02778
M = 32      Error = 9.91824e-06  Approximated order = 2.0143
M = 64      Error = 2.46712e-06  Approximated order = 2.00726
M = 128     Error = 6.15214e-07  Approximated order = 2.00367
M = 256     Error = 1.53601e-07  Approximated order = 2.0019
M = 512     Error = 3.83689e-08  Approximated order = 2.00118
M = 1024    Error = 9.58214e-09  Approximated order = 2.00152
M = 2048    Error = 2.38813e-09  Approximated order = 2.00447
M = 4096    Error = 5.89957e-10  Approximated order = 2.0172
Estimated rate using linear regression = 2.01412

RK order 3


-------------------------------------------------------------
M = 2       Error = 0.000107796
M = 4       Error = 1.5402e-05   Approximated order = 2.80712
M = 8       Error = 2.03327e-06  Approximated order = 2.92124
M = 16      Error = 2.60614e-07  Approximated order = 2.96381
M = 32      Error = 3.29725e-08  Approximated order = 2.98258
M = 64      Error = 4.14608e-09  Approximated order = 2.99144
M = 128     Error = 5.19776e-10  Approximated order = 2.99579
M = 256     Error = 6.50568e-11  Approximated order = 2.99812
M = 512     Error = 8.12717e-12  Approximated order = 3.00088
M = 1024    Error = 1.00275e-12  Approximated order = 3.01879
M = 2048    Error = 1.13132e-13  Approximated order = 3.14789
M = 4096    Error = 6.55032e-15  Approximated order = 4.1103
Estimated rate using linear regression = 3.03482

Classical RK order 4


-------------------------------------------------------------
M = 2       Error = 1.11709e-05
M = 4       Error = 7.18023e-07  Approximated order = 3.95957
M = 8       Error = 4.51194e-08  Approximated order = 3.99221
M = 16      Error = 2.82329e-09  Approximated order = 3.9983
M = 32      Error = 1.76508e-10  Approximated order = 3.99957
M = 64      Error = 1.10368e-11  Approximated order = 3.99934
M = 128     Error = 6.95333e-13  Approximated order = 3.98848
M = 256     Error = 4.74065e-14  Approximated order = 3.87455
M = 512     Error = 3.9968e-15   Approximated order = 3.56817
```

```
M = 1024    Error = 3.77476e-15  Approximated order = 0.0824622
M = 2048    Error = 8.88178e-16  Approximated order = 2.08746
M = 4096    Error = 1.33227e-15  Approximated order = -0.584963
Estimated rate using linear regression = 3.25256

2. ODE y' = |1.1 - y| + 1, y(0)=1

Explicit Euler


_____________________________________________________________
M = 2       Error = 0.296135
M = 4       Error = 0.176213     Approximated order = 0.748934
M = 8       Error = 0.104911     Approximated order = 0.74815
M = 16      Error = 0.0568369    Approximated order = 0.88427
M = 32      Error = 0.0303542    Approximated order = 0.904932
M = 64      Error = 0.0153476    Approximated order = 0.983887
M = 128     Error = 0.00770812   Approximated order = 0.993558
M = 256     Error = 0.00385358   Approximated order = 1.00018
M = 512     Error = 0.0019175    Approximated order = 1.00697
M = 1024    Error = 0.00094465   Approximated order = 1.02137
M = 2048    Error = 0.0004575    Approximated order = 1.04601
M = 4096    Error = 0.000213549  Approximated order = 1.09921
Estimated rate using linear regression = 0.95824

Trapezoidal rule


_____________________________________________________________
M = 2       Error = 0.0272722
M = 4       Error = 0.019933     Approximated order = 0.452272
M = 8       Error = 0.00315605   Approximated order = 2.65897
M = 16      Error = 0.00101849   Approximated order = 1.63169
M = 32      Error = 0.000278879  Approximated order = 1.86872
M = 64      Error = 4.51313e-05  Approximated order = 2.62744
M = 128     Error = 8.4172e-07   Approximated order = 5.74464
M = 256     Error = 2.80199e-06  Approximated order = -1.73504
M = 512     Error = 4.60735e-08  Approximated order = 5.92637
M = 1024    Error = 1.75031e-07  Approximated order = -1.9256
M = 2048    Error = 5.44107e-09  Approximated order = 5.00758
M = 4096    Error = 1.06517e-08  Approximated order = -0.969126
Estimated rate using linear regression = 2.19299

RK order 3


_____________________________________________________________
M = 2       Error = 0.0147288
M = 4       Error = 0.0028711    Approximated order = 2.35897
M = 8       Error = 0.000217491  Approximated order = 3.72257
M = 16      Error = 0.000773451  Approximated order = -1.83035
M = 32      Error = 3.17718e-05  Approximated order = 4.60549
```

```
M = 64      Error = 1.3791e-05   Approximated order = 1.20402
M = 128     Error = 3.99984e-06  Approximated order = 1.78571
M = 256     Error = 9.977e-07    Approximated order = 2.00327
M = 512     Error = 2.46644e-07  Approximated order = 2.01617
M = 1024    Error = 6.60285e-08  Approximated order = 1.90127
M = 2048    Error = 1.58186e-08  Approximated order = 2.06147
M = 4096    Error = 3.34817e-09  Approximated order = 2.24017
Estimated rate using linear regression = 1.94622

Classical RK order 4


-------------------------------------------------------------
M = 2       Error = 0.01995
M = 4       Error = 0.0021045    Approximated order = 3.24484
M = 8       Error = 0.000823604  Approximated order = 1.35346
M = 16      Error = 0.000651961  Approximated order = 0.337164
M = 32      Error = 3.42114e-05  Approximated order = 4.25224
M = 64      Error = 1.41017e-05  Approximated order = 1.27861
M = 128     Error = 4.03905e-06  Approximated order = 1.80378
M = 256     Error = 9.92776e-07  Approximated order = 2.02447
M = 512     Error = 2.50338e-07  Approximated order = 1.98759
M = 1024    Error = 6.55673e-08  Approximated order = 1.93283
M = 2048    Error = 1.58282e-08  Approximated order = 2.05048
M = 4096    Error = 3.34702e-09  Approximated order = 2.24154
Estimated rate using linear regression = 1.99279
```

The convergence orders of the explicit Euler and Runge–Kutta 2 methods are as expected, but we do not see any relevant improvement in the convergence orders of RK3 and RK4. This is due to the fact that the right hand side of the IVP is not continuously differentiable: the convergence theory breaks down due to *lack of smoothness* of exact solutions.

SOLUTION of (11-7.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 (11.7.2) is $\mathbb{R}^{d,d}$, a space of matrices.

SOLUTION of (11-7.b):

Writing $\mathbf{y} = [u, v]^T$, the map $\mathbf{f}$ associated to (11.7.4) 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, (11.7.4) becomes

$$\dot{\mathbf{W}} = \begin{bmatrix} 2-v(t) & -u(t) \\ v(t) & u(t)-1 \end{bmatrix} \mathbf{W}. \tag{11.7.5}$$

This is a non-autonomous ODE.

SOLUTION of (11-7.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 (11.7.4) with initial value $\mathbf{y}_0$ has period T_P, then we have $\mathbf{\Phi}(T_P, \mathbf{y}_0) = \mathbf{y}_0$, so $\mathbf{F}(\mathbf{y}_0) = 0$.

SOLUTION of (11-7.d):

By definition of $\mathbf{W}$ we immediately obtain

$$\mathrm{D}\,\mathbf{F}(\mathbf{y}) = \mathbf{W}(T_P, \mathbf{y}) - \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix.

HINT 1 for (11-7.e): Study [Lecture → § 11.1.3.3] again. Also take a look at [Lecture → Fig. 396]. ⌟

SOLUTION of (11-7.e):

If $\mathbf{y}_0 \in \mathbb{R}^2_+$ is a solution, then every state on the trajectory $\mathbf{y}_0$ will also be a solution. Only if the trajectory starting in $\mathbf{y}_0$ collapses to a point, that is, if $\mathbf{y}_0$ is a stationary point, $\mathbf{f}(\mathbf{y}_0) = 0$, and there is no other trajectory with the prescribed period, we can expect uniqueness.

HINT 1 for (11-7.f): As in (11.7.3), 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 (11-7.f):

Writing $\mathbf{w} = [u, v, W_{11}, W_{21}, W_{12}, W_{22}]^T$, b y (11.7.4) and (11.7.5) 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 11.7.6: Example code.

```
2  std::pair<Eigen::Vector2d, Eigen::Matrix2d> PhiAndW(double u0, double v0,
3                                                      double T) {
4    std::pair<Eigen::Vector2d, Eigen::Matrix2d> PaW;
5    // TODO: (11-7.f)  Calculate the time evolution of Phi(t,y0) and W(t,y0)
6    // up to the final time T, using the inital value y0=[u0,v0].
7    // Save the values of Phi and W at time T in PaW.first and
8    // PaW.second respectively.
9    // Hint: First write the ODE
10   // [ y'(t), W'(t,y0) ] = [ f(y(t)), Df(y(t))W(t,y0) ]
11   // in vectorized form, i.e. find a function g:R^6 -> R^6
12   // such that the ODE w'=g(w) is equivalent.
13   // Then use the ode45 class for the function g.
14   // START
15   auto f = [](const Eigen::VectorXd &w) {
16     Eigen::VectorXd temp(6);
17     temp(0) = (2. - w(1)) * w(0);
18     temp(1) = (w(0) - 1.) * w(1);
19     temp(2) = (2. - w(1)) * w(2) - w(0) * w(3);
20     temp(3) = w(1) * w(2) + (w(0) - 1.) * w(3);
21     temp(4) = (2. - w(1)) * w(4) - w(0) * w(5);
22     temp(5) = w(1) * w(4) + (w(0) - 1.) * w(5);
23     return temp;
24   };
25
26   Eigen::VectorXd w0(6);
27   w0 << u0, v0, 1., 0, 0, 1.;
28
29   // Construct ode solver with r.h.s
30   ode45<Eigen::VectorXd> O(f);
31   // Set options
32   O.options.rtol = 1e-14;
33   O.options.atol = 1e-12;
34   // Solve ODE
35   auto sol = O.solve(w0, T);
36   // Extract needed component
37   Eigen::VectorXd wT = sol.back().first;
38
39   PaW.first << wT(0), wT(1);
```

```
40    PaW.second << wT(2), wT(4), wT(3), wT(5);
41    // END
42    return PaW;
43  }
```

Get it on GitLab (LV.hpp).

HINT 1 for (11-7.g): The correct solutions are $u(0) \approx 3.110$ and $v(0) = 2.081$.

SOLUTION of (11-7.g):

Remember that the Newton iteration for $\mathbf{F}(\mathbf{y}) = 0$ reads:

$$\mathbf{y}^{(k+1)} = \mathbf{y}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)})\,. \tag{11.7.7}$$

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 11.7.8: Example code.

```
std::pair<double, double> findInitCond() {
  Eigen::Vector2d y;

  // TODO: (11-7.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
  constexpr double T = 5;  // Period we require

  // Compute Phi and W from initial guess
  std::pair<Eigen::Vector2d, Eigen::Matrix2d> PaW = PhiAndW(y(0), y(1), T);
  Eigen::Vector2d F = PaW.first - y;  // Value of F at initial guess
  Eigen::Matrix2d DF;                 // Declare Jacobian of F

  // Until we are happy with our approximation
  while (F.norm() > 1e-5) {  // Residual based termination
    // Calculate Jacobian
    DF = PaW.second - Eigen::MatrixXd::Identity(2, 2);
    // Use Newton iteration
    y = y - DF.lu().solve(F);
    // Test current guess
    PaW = PhiAndW(y(0), y(1), T);
    // Get value of F at the current guess
    F = PaW.first - y;
  }
  // END

  // TODO: (11-7.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 std::make_pair(y(0), y(1));
}
```

Get it on GitLab (LV.hpp).

HINT 1 for (11-8.a): Apply the chain rule, see [Lecture → § 8.5.1.15], taking into account the ODE (11.8.1) in order to substitute derivatives. ⌟

SOLUTION of (11-8.a):

For the first time derivative of $\mathbf{y}$, we just use the differential equation: $\frac{d\mathbf{y}}{dt}(t) = \mathbf{y}'(t) = \mathbf{f}(\mathbf{y}(t))$. For the second derivative, we use the previous equation, apply the chain rule and then once again insert the ODE:

$$\frac{d^2\mathbf{y}}{dt^2}(t) = \frac{d}{dt}\mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}'(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)).$$

Here $\mathbf{Df}(\mathbf{y}(t))$ is the Jacobian of $\mathbf{f}$ evaluated at $\mathbf{y}(t)$.

HINT 1 for (11-8.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 (11-8.b): This time we have to apply both the product rule [Lecture → Eq. (8.5.1.17)] and the chain rule [Lecture → Eq. (8.5.1.16)] to the expression derived in the previous sub-problem. ⌟

HINT 3 for (11-8.b): Refresh yourself on the abstract concept of a derivative discussed in [Lecture → § 8.5.1.12].

Relevant for the case $d > 1$ is the fact that the first derivative of $\mathbf{f}$ is a linear mapping $\mathbf{Df}(\mathbf{y}_0) : \mathbb{R}^n \to \mathbb{R}^n$. This linear mapping is applied by multiplying the argument with the Jacobian of $\mathbf{f}$.

Similarly, the second derivative is a *bilinear* mapping $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$. The i-th component of $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{v},\mathbf{v})$ is given by

$$\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{v},\mathbf{v})_i = \mathbf{v}^\top \mathbf{Hf}_i(\mathbf{y}_0)\mathbf{v},$$

where $\mathbf{Hf}_i(\mathbf{y}_0)$ is the Hessian of the i-th component of $\mathbf{f}$ evaluated at $\mathbf{y}_0$. ⌟

SOLUTION of (11-8.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 (11.8.3). 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 (11-8.a): $\mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t))$, with the Jacobian $\mathbf{Df}(\mathbf{y}(t))$ which contains the gradients of the components of f row-wise.

Now, we apply product rule to $y_i''(t)$ to obtain

$$y_i'''(t) = \left\{\frac{d}{dt}[\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top\right\} \cdot \mathbf{y}'(t) + [\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \cdot \mathbf{y}''(t) .$$

The second term of the sum again builds up to

$$\mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t))^2 \cdot \mathbf{f}(\mathbf{y}(t)) .$$

For the first term, we write the scalar product of the two vectors as a sum and then interchange the order of derivatives. This is possible as long as functions are sufficiently differentiable:

$$\left\{\frac{d}{dt}[\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top\right\} \cdot \mathbf{y}'(t) = \sum_{j=1}^{n} \left[\partial_{y_j} \frac{d}{dt} f_i(\mathbf{y}(t))\right] f_j(\mathbf{y}(t)) .$$

Now we apply the chain rule:

$$\begin{aligned}\sum_{j=1}^{n}\left[\partial_{y_j}\frac{d}{dt}f_i(\mathbf{y}(t))\right]f_j(\mathbf{y}(t)) &= \sum_{j=1}^{n}\left[\partial_{y_j}\sum_{l=1}^{n}\partial_{y_l}f_i(\mathbf{y}(t))y_l'(t)\right]f_j(\mathbf{y}(t)) \\ &= \sum_{j,l=1}^{n}\left[\partial_{y_j}\partial_{y_l}f_i(\mathbf{y}(t))\right]f_l(\mathbf{y}(t))f_j(\mathbf{y}(t)) \\ &= \mathbf{f}(\mathbf{y}(t))^{\top}\cdot\mathbf{H}f_i(\mathbf{y}(t))\cdot\mathbf{f}(\mathbf{y}(t))\end{aligned}$$

This is the desired result.

SOLUTION of (11-8.c):

For our particular example, recasting it into the standard form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ of an autonomous ODE, we have

$$\mathbf{y} = \begin{bmatrix} u \\ v \end{bmatrix} \quad , \quad \mathbf{f}(\mathbf{y}) = \begin{bmatrix} (\alpha_1 - \beta_1 y_2) y_1 \\ -(\alpha_2 - \beta_2 y_1) y_2 \end{bmatrix} , \tag{11.8.5}$$

and the derivatives

$$\mathbf{D}\mathbf{f}(\mathbf{y}) = \begin{bmatrix} \alpha_1 - \beta_1 y_2 & -\beta_1 y_1 \\ \beta_2 y_2 & -(\alpha_2 - \beta_2 y_1) \end{bmatrix},$$
$$\mathbf{H}f_1(\mathbf{y}) = \begin{bmatrix} 0 & -\beta_1 \\ -\beta_1 & 0 \end{bmatrix}, \qquad \mathbf{H}f_2(\mathbf{y}) = \begin{bmatrix} 0 & \beta_2 \\ \beta_2 & 0 \end{bmatrix} .$$

C++ code 11.8.6: Solution.

```
2  std::vector<Eigen::Vector2d> SolvePredPreyTaylor(double alpha1, double beta1,
3                                                   double alpha2, double beta2,
4                                                   double T,
5                                                   const Eigen::Vector2d &y0,
6                                                   unsigned int M) {
7    // Vector storing the states
8    std::vector<Eigen::Vector2d> res;
9    // TODO: (11-8.c) Solve the predator prey model using the Taylor expansion
10   // method.
11   // START
12   res.reserve(M + 1);
13
14   // Step size
15   const double h = T / M;
16
17   // Storing the initial state
18   Eigen::Vector2d y = y0;
19   res.push_back(y);
20
21   // Preparing functions for using the Taylor expansion method
22   // Lambda function for the RHS f(y)
23   auto f = [&](const Eigen::Vector2d &y) {
24     Eigen::Vector2d f;
25     f << (alpha1 - beta1 * y(1)) * y(0), (beta2 * y(0) - alpha2) * y(1);
26     return f;
27   };
28
29   // Lambda function for computing Df(y)*z
30   auto Df = [&](const Eigen::Vector2d &y, const Eigen::Vector2d &z) {
31     Eigen::Matrix2d Df;
32     Df << alpha1 - beta1 * y(1), -beta1 * y(0), beta2 * y(1),
33         -alpha2 + beta2 * y(0);
34     Eigen::Vector2d out = Df * z;
35     return out;
36   };
37
38   // Lambda function for D^2f(y) (z,z)
39   auto D2f = [&](const Eigen::Vector2d &y, const Eigen::Vector2d &z) {
40     Eigen::Matrix2d H1;
```

```
    Eigen::Matrix2d H2;
    H1 << 0, -beta1, -beta1, 0;
    H2 << 0, beta2, beta2, 0;
    Eigen::Vector2d out;
    out << z.transpose() * H1 * z, z.transpose() * H2 * z;
    return out;
  };

  for (unsigned int k = 0; k < M; ++k) {
    // evaluate terms for taylor step.
    auto fy = f(y);
    // Df(y) * f(y)
    auto dfyfy = Df(y, fy);
    // Df(y) * Df(y) * f(y)
    auto df2yfy = Df(y, dfyfy);
    // D^2f(y) (f(y),f(y))
    auto d2fyfy = D2f(y, fy);

    // evaluate taylor expansion to compute update
    y = y + h * fy + 0.5 * h * h * dfyfy +
        1.0 / 6.0 * h * h * h * (df2yfy + d2fyfy);

    // save new state:
    res.push_back(y);
  }
  // END
  return res;
}
```

Get it on GitLab (taylorintegrator.hpp).

SOLUTION of (11-8.d):

We see cubic algebraic convergence: $O(h^3)$.

C++11-code 11.8.7: Solution of (11-8.d)

```
double testCvgTaylorMethod() {
  double cvgrate = 0;
  // TODO: (11-8.d) Generate an error table and return an estimate of
     the
  // convergence rate of the Taylor method.
  // START
  // initialize parameters for the model:
  constexpr double T = 10;     // final time
  Eigen::Vector2d y0(100, 5);  // initial condition
  Eigen::Vector2d yex(0.319465882659820,
                      9.730809352326228);  // reference solution
  constexpr double alpha1 = 3.0;
  constexpr double alpha2 = 2.0;
  constexpr double beta1 = 0.1;
  constexpr double beta2 = 0.1;

  // Initialize parameters for the convergence study
  constexpr unsigned int M0 = 128;    // Minimum number of timesteps
  constexpr unsigned int numRef = 8;  // Number of refinements

  // Convergence study
  Eigen::ArrayXd error(numRef);
  Eigen::ArrayXd M(numRef);
  for (unsigned int i = 0; i < numRef; ++i) {
    M(i) = std::pow(2, i) * M0;
    auto res = SolvePredPreyTaylor(alpha1, beta1, alpha2, beta2, T, y0, M(i));
    error(i) = (res.back() - yex).norm();
  }

  PrintErrorTable(M, error);

  // calculate linear regression line:  log(error)   c0 + c1*log(M)
  Eigen::MatrixXd A(numRef, 2);
  A.col(0) = Eigen::VectorXd::Ones(numRef);
  A.col(1) = M.log();
  Eigen::VectorXd logError = error.log();
  Eigen::Vector2d coeffs = A.householderQr().solve(logError);

  // estimated convergence rate:  -c1
  cvgrate = -coeffs(1);
  // END
  return cvgrate;
}
```

Get it on GitLab (taylorintegrator.hpp).

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

SOLUTION of (11-8.e):

The error of the studied Runge-Kutta method and Taylor's method are practically identical. The obvious disadvantage of Taylor's method compared to Runge-Kutta methods is that the former involves rather complicated higher derivatives of f. If we want higher orders, those formulas get even more complicated, whereas explicit Runge-Kutta methods work with only a few evaluations of f itself, yielding results which are comparable.

Another fundamental drawback of the Taylor expansion method is that it cannot be applied to an ODE with $\mathbf{f}$ given in procedural form.

SOLUTION of (11-9.a):

By straightforward computation we find

$$\mathbf{grad}\, F(x,y) = \begin{bmatrix} 4(x^2+y^2)x - 3x^2 \\ 4(x^2+y^2)y - 3y^2 \end{bmatrix} \quad , \quad x,y \in \mathbb{R}\,. \tag{11.9.4}$$

Then we rotate the gradient counterclockwise using (11.9.2). This formula is easily explained: The linear mapping which performs this rotation is

$$\left[\boldsymbol{x} \mapsto \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \boldsymbol{x}\,, \quad \boldsymbol{x} \in \mathbb{R}^2 \right]$$

This is easily verified by looking at the unit vectors. The first unit vector pointing in x-direction $[1,0]^T$ is rotated, such that it points in y-direction, thus, it is mapped onto the second unit vector $[0,1]^T$. The second unit vector $[0,1]^T$ is further rotated in the $-x$-direction, thus, it is mapped on the opposite of the first unit vector $[-1,0]^T$.

Using (11.9.4), the differential equation (11.9.1) for F from (11.9.3) reads, with $\mathbf{y}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \frac{1}{\sqrt{(4(x^2+y^2)x - 3x^2)^2 + (4(x^2+y^2)y - 3y^2)^2}} \cdot \begin{bmatrix} -4(x^2+y^2)y + 3y^2 \\ 4(x^2+y^2)x - 3x^2 \end{bmatrix}. \tag{11.9.5}$$

The time argument of x and y is suppressed for the sake brevity.

SOLUTION of (11-9.b):

The right-hand side function to be passed to the constructor of **ode45** can conveniently be realized as a lambda function.

A conversion of the return value of `ode45::solve()` into the correct return type is needed.

C++ code 11.9.6: Sub-problem (11-9.b): The function `computeIsolinePoints()`

```
2  template <typename GradientFunctor>
3  Eigen::Matrix<double, 2, Eigen::Dynamic> computeIsolinePoints(
4      GradientFunctor &&gradF, Eigen::Vector2d y0, double T) {
5    Eigen::Matrix<double, 2, Eigen::Dynamic> States =
6        Eigen::Matrix2Xd::Zero(2, 1);
7    // TODO: (11-9.b) Solve the IVP as given in the task description.
8    // START
9    // Right-hand-side vector field of isoline ODE
10   auto rhs = [gradF](const Eigen::Vector2d &x) {
11     Eigen::Vector2d gradFx = gradF(x);
12     Eigen::Vector2d out(-gradFx(1), gradFx(0));  // rotate
13     out /= gradFx.norm();                        // scale
14     return out;
15   };
16   // Adaptive explicit Runge-Kutta method
17   ode45<Eigen::Vector2d> integrator(rhs);
18   // Set tolerances for timestep control
19   integrator.options.rtol = 0.00001;
20   integrator.options.atol = 1e-9;
21   // Perform explicit timestepping
22   std::vector<std::pair<Eigen::Vector2d, double>> sol = integrator.solve(y0, T);
23   // Convert output into requested format: points on isoline arranged into the
24   // columns of a matrix.
25   const unsigned int N = sol.size() - 1;
26   States = Eigen::MatrixXd::Zero(2, N + 1);
27   for (unsigned int i = 0; i <= N; i++) {
28     States.col(i) = sol[i].first;
29   }
30   // END
31   return States;
32 }
```

Get it on GitLab (contour.hpp).

HINT 1 for (11-9.c): It is recommended to call `computeIsolinePoints()` from Sub-problem (11-9.b

SOLUTION of (11-9.c):

The normalized rotated gradient of F from (11.9.3) can be encapsulated in a lambda function.

C++ code 11.9.7: Sub-problem (11-9.c): The function `crookedEgg()`

```
2  Eigen::Matrix<double, 2, Eigen::Dynamic> crookedEgg() {
3    // crookedEggCurve will need to be reshaped to 2*(N+1).
4    Eigen::Matrix<double, 2, Eigen::Dynamic> crookedEggCurve =
5        Eigen::Matrix2Xd::Zero(2, 1);
6    // TODO: (11-9.c) Compute the crooked egg curve using computeIsolinePoints().
7    // START
8    Eigen::Vector2d y0(1., 0.);
9    constexpr double T = 4.;
10   auto gradF = [](const Eigen::Vector2d &x) {
11     double x2 = x(0) * x(0), y2 = x(1) * x(1);
12     double x2y2 = 4. * (x2 + y2);
13     Eigen::Vector2d grad(x2y2 * x(0) - 3. * x2, x2y2 * x(1) - 3. * y2);
14     return grad;
15   };
16   crookedEggCurve = computeIsolinePoints(gradF, y0, T);
17   // END
18   return crookedEggCurve;
19 }
```

Get it on GitLab (contour.hpp).

HINT 1 for (11-9.d): Of course, you may start with copying what you implemented for `computeIsoline`

SOLUTION of (11-9.d):

In [Lecture → Ex. 1.5.4.7] we learned that the evaluation of difference quotients is affected by cancellation for very small span h. In order to curb cancellation it is recommended to use $h \approx \sqrt{\texttt{EPS}}$, where EPS is the machine precision: $\texttt{EPS} \approx 2 \cdot 10^{-16}$ for double precision floating point arithmetic, type **double**, see [Lecture → Code 1.5.3.13].

The implementation of this subproblem just boils down to augmenting the code of `computeIsolinePoi` by another lambda function supplying an approximation of $\mathbf{grad}\, F(\boldsymbol{x})$ based on two symmetric difference quotients in x_1 and x_2-directions.

C++ code 11.9.8: Sub-problem (11-9.b): The function `computeIsolinePointsDQ()`

```
template <typename FFunctor>
Eigen::Matrix<double, 2, Eigen::Dynamic> computeIsolinePointsDQ(
    FFunctor &&F, Eigen::Vector2d y0, double T) {
  // States will need to be reshaped to 2*(N+1).
  Eigen::Matrix<double, 2, Eigen::Dynamic> States =
      Eigen::Matrix2Xd::Zero(2, 1);
  // TODO: (11-9.d) Solve the IVP as given in the task description by using the
  // symmtric difference quotient for the gradient of F. You may copy code from
  // above.
  // START
  // Get gradient of F by symmetric difference quotients with a
  // span given by the root of the machine precision in order to curb
  // cancellation as much as possible.
  const double epsrt = std::sqrt(std::numeric_limits<double>::epsilon());
  auto gradF = [epsrt, F](const Eigen::Vector2d &x) {
    Eigen::Vector2d grad, dx, dy;
    dx << epsrt, 0.;
    dy << 0., epsrt;
    grad(0) = (F(x + dx) - F(x - dx)) / (2 * epsrt);
    grad(1) = (F(x + dy) - F(x - dy)) / (2 * epsrt);
    return grad;
  };
  // The remainder as in computeIsolinePoints()
  auto rhs = [gradF](const Eigen::Vector2d &x) {
    Eigen::Vector2d gradFx = gradF(x);
    Eigen::Vector2d out(-gradFx(1), gradFx(0));  // rotate
    out /= gradFx.norm();                        // scale
    return out;
  };

  ode45<Eigen::Vector2d> integrator(rhs);
  integrator.options.rtol = 0.00001;
  integrator.options.atol = 1e-9;

  std::vector<std::pair<Eigen::Vector2d, double>> sol = integrator.solve(y0, T);
  const unsigned int N = sol.size() - 1;
  States = Eigen::MatrixXd::Zero(2, N + 1);
  for (unsigned int i = 0; i <= N; i++) {
    States.col(i) = sol[i].first;
  }
  // END
  return States;
}
```

Get it on GitLab (contour.hpp).

SOLUTION of (12-1.a):

The solution is given in [Lecture → Rem. 12.3.3.6].

SOLUTION of (12-1.b):

We follow the developments of [Lecture → Rem. 12.3.3.9] and rewrite the stage equations as

$$F(\mathbf{g}) = \mathbf{0} \quad \text{with} \quad F(\mathbf{g}) := \mathbf{g} - h(\mathfrak{A} \otimes \mathbf{I}) \begin{bmatrix} \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_1) \\ \vdots \\ \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_s) \end{bmatrix} . \tag{12.1.1}$$

Then the corresponding Newton iteration reads

$$\mathbf{g}^{(k+1)} = \mathbf{g}^{(k)} - DF(\mathbf{g}^{(k)})^{-1} F(\mathbf{g}^{(k)}) . \tag{12.1.2}$$

We can write the function F in "vector-valued components":

$$F(\mathbf{g}) = F\left(\begin{bmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_s \end{bmatrix} \right) = \begin{bmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_s \end{bmatrix} - h \begin{bmatrix} \sum_{j=1}^{s} a_{1j} \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_j) \\ \vdots \\ \sum_{j=1}^{s} a_{sj} \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_j) \end{bmatrix} =: \begin{bmatrix} F_1(\mathbf{g}) \\ \vdots \\ F_s(\mathbf{g}) \end{bmatrix} .$$

Note that the components of F are functions $\mathbb{R}^{N \cdot s} \to \mathbb{R}^N$ Componentwise differentiation yields the Jacobian

$$DF(\mathbf{g}^{(k)}) = \left[\frac{\partial F_i}{\partial \mathbf{g}_k} \right]_{i,k=1}^{s} = \mathbf{I} - h \begin{bmatrix} a_{11} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_1) & \dots & a_{1s} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_s) \\ \vdots & & \vdots \\ a_{s1} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_1) & \dots & a_{ss} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_s) \end{bmatrix} . \tag{12.1.3}$$

Here $\dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{g}) \in \mathbb{R}^{N,N}$ are "partial-derivative Jacobi matrices".

SOLUTION of (12-1.c):

We implement the method discussed in Sub-problem (12-1.b). The main task is to wrap the function F from (12.1.1) and its Jacobian (12.1.3) into suitable lambda functions and pass those to `dampnewton()`.

C++11-code 12.1.4: Implementation of implicitRKIntegrator

```
2  class implicitRKIntegrator {
3   public:
4    /**
5     * \brief Constructor for the implicit RK method.
6     * Performs size checks and copies A and b into internal storage.
7     * \param A Matrix containing coefficents of Butcher tableau.
8     * \param b Vector containing coefficients of
9     * lower part of Butcher tableau.
10    */
11   implicitRKIntegrator(const Eigen::MatrixXd &A, const Eigen::VectorXd &b)
12       : A(A), b(b), s(b.size()) {
13     assert(A.cols() == A.rows() && "Matrix must be square.");
14     assert(A.cols() == b.size() && "Incompatible matrix/vector size.");
15   }
16
17   /**
18    * \brief Perform the solution of the ODE.
19    * Solve an autonomous ODE y' = f(y), y(0) = y0, using an
20    * implicit RK scheme given in the Butcher tableau provided in the
21    * constructor. Performs N equidistant steps upto time T
22    * with initial data y0.
23    * \tparam Function type for function implementing the rhs function.
24    * Must have Eigen::VectorXd operator()(Eigen::VectorXd x)
25    * \tparam Function2 type for function implementing the Jacobian of f.
26    * Must have Eigen::MatrixXd operator()(Eigen::VectorXd x)
27    * \param f function handle for rhs in y' = f(y), e.g.
28    * implemented using lambda funciton
29    * \param Jf function handle for Jf, e.g.
30    * implemented using lambda funciton
31    * \param T final time T
32    * \param y0 initial data y(0) = y0 for y' = f(y)
33    * \param N number of steps to perform.
34    * Step size is h = T / N. Steps are equidistant.
35    * \return vector containing all steps y^n (for each n)
36    * including initial and final value
37    */
38   template <class Function, class Jacobian>
39   std::vector<Eigen::VectorXd> solve(Function &&f, Jacobian &&Jf, double T,
40                                      const Eigen::VectorXd &y0,
41                                      unsigned int N) const {
42     // Init step size
43     double h = T / N;
44
45     // Will contain all steps, reserve memory for efficiency
46     std::vector<Eigen::VectorXd> res;
47     res.reserve(N + 1);
48
49     // Store initial data
50     res.push_back(y0);
```

```
51      // Initialize some memory to store temporary values
52      Eigen::VectorXd ytemp1 = y0;
53      Eigen::VectorXd ytemp2 = y0;
54      // Pointers to swap previous value
55      Eigen::VectorXd *yold = &ytemp1;
56      Eigen::VectorXd *ynew = &ytemp2;
57
58      // Loop over all fixed steps
59      for (unsigned int k = 0; k < N; ++k) {
60        // Compute, save and swap next step
61        step(f, Jf, h, *yold, *ynew);
62        res.push_back(*ynew);
63        std::swap(yold, ynew);
64      }
65      return res;
66    }
67
68  private:
69    /**
70     * \brief Perform a single step of the RK method for the
71     * solution of the autonomous ODE
72     * Compute a single explicit RK step y^{n+1} = y_n + \sum ...
73     * starting from value y0 and storing next value in y1
74     * \tparam Function type for function implementing the rhs.
75     * Must have Eigen::VectorXd operator()(Eigen::VectorXd x)
76     * \tparam Jacobian type for function implementing the Jacobian of f.
77     * Must have Eigen::MatrixXd operator()(Eigen::VectorXd x)
78     * \param f function handle for ths f, s.t. y' = f(y)
79     * \param Jf function handle for Jf, e.g. implemented using lambda
80     *function
81     * \param h step size ! \param[in] y0 initial Eigen::VectorXd
82     * \param y1 next step y^{n+1} = y^n + ...
83     */
84    template <class Function, class Jacobian>
85    void step(Function &&f, Jacobian &&Jf, double h, const Eigen::VectorXd &y0,
86              Eigen::VectorXd &y1) const {
87      // TODO: (12-1.c) Perform one step of the implicit Runge-Kutta method.
88      // START
89      const unsigned int d = y0.size();
90      Eigen::MatrixXd eye = Eigen::MatrixXd::Identity(d, d);
91
92      // Handle for the function F describing the
93      // equation satisfied by the stages g
94      auto F = [&y0, h, d, this, &f, &eye](Eigen::VectorXd gv) {
95        Eigen::VectorXd Fv = gv;
96        for (unsigned int j = 0; j < s; j++) {
97          Fv = Fv -
98               h * kroneckerProduct(A.col(j), eye) * f(y0 + gv.segment(j * d, d));
99        }
100       return Fv;
101     };
102
103     // Handle for the Jacobian of F.
104     auto JF = [&y0, h, d, &Jf, this](Eigen::VectorXd gv) {
105       Eigen::MatrixXd DF(s * d, s * d);
106       for (unsigned int j = 0; j < s; j++) {
107         DF.block(j * d, 0, d, s * d) =
```

```
108                 kroneckerProduct(A.row(j), Jf(y0 + gv.segment(j * d, d)));
109           }
110           DF = Eigen::MatrixXd::Identity(s * d, s * d) - h * DF;
111           return DF;
112         };
113
114         // Obtain stages with damped Newton method
115         Eigen::VectorXd gv = Eigen::VectorXd::Zero(s * d);
116         dampnewton(F, JF, gv);
117
118         // Calculate y1
119         Eigen::MatrixXd K(d, s);
120         for (unsigned int j = 0; j < s; j++)
121           K.col(j) = f(y0 + gv.segment(j * d, d));
122         y1 = y0 + h * K * b;
123         // END
124       }
125       //<! Matrix A in Butcher scheme
126       const Eigen::MatrixXd A;
127       //<! Vector b in Butcher scheme
128       const Eigen::VectorXd b;
129       //<! Size of Butcher matrix and vector A and b
130       unsigned int s;
131     };
```

Get it on GitLab (implicit_rkintegrator.hpp).

HINT 1 for (12-1.d): Scan the particular implicit Runge-Kutta single step methods presented in [Lecture → Section 12.3].

⌟

SOLUTION of (12-1.d):

The Butcher scheme used corresponds to the Radau RK-SSM of order 3 (see [Lecture → Ex. 12.3.4.21] for the complete scheme). Thus, the method is A-stable and L-stable.

SOLUTION of (12-1.e):

C++11-code 12.1.7: Use of implicitRKIntegrator for solving an IVP for the Lotka-Volterra ODE

```
  // Coefficients and handle for prey/predator model
  constexpr double alpha1 = 3.;
  constexpr double alpha2 = 2.;
  constexpr double beta1 = 0.1;
  constexpr double beta2 = 0.1;
  auto f = [](const Eigen::VectorXd &y) {
    auto temp = y;
    temp(0) *= alpha1 - beta1 * y(1);
    temp(1) *= -alpha2 + beta2 * y(0);
    return temp;
  };

  auto Jf = [](const Eigen::VectorXd &y) {
    Eigen::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
  constexpr unsigned int d = 2;

  // Final time for model
  constexpr 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, 32768, 65536};

  // Exact value y(10) at final time T = 10 (approximated)
  Eigen::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;
  }
```

Get it on GitLab (main.cpp).

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:

$$\mathrm{errornorm} = O(h^3) \ .$$

We expect this because the 2-stage Radau single-step method is of order 3, see [Lecture → Ex. 12.3.4.21].

HINT 1 for (12-2.a): Examine the time derivative $\frac{d}{dt}\|\mathbf{y}(t)\|_2^2$ and use the product rule. ⌟

SOLUTION of (12-2.a):

Since $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$ for all $\mathbf{x}, \mathbf{y}$, we have

$$(\mathbf{a} \times \mathbf{y} + c(\mathbf{y} \times (\mathbf{a} \times \mathbf{y}))) \cdot \mathbf{y} = 0 \quad \Rightarrow \quad D_t \|\mathbf{y}(t)\|_2^2 = 2\dot{\mathbf{y}}(t) \cdot \mathbf{y}(t) = 0 \quad \Rightarrow \quad \|\mathbf{y}(t)\|_2^2 = \text{const. } \forall t \, .$$

Thus $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}(0)\|_2 = \|\mathbf{y}_0\|_2$.

SOLUTION of (12-2.b):

The right-hand side function $\mathbf{f}$ in (12.2.1) is a quadratic polynomial in the components of $\mathbf{y}$, hence smooth and defined on all of $\mathbb{R}^3$. Based on the product rule [Lecture → Eq. (8.5.1.17)] and appealing to the bilinearity of the cross product, we find

$$\mathrm{D}\mathbf{f}(\mathbf{y})\mathbf{h} = \mathbf{a}\times\mathbf{h} + c\mathbf{h}\times(\mathbf{a}\times\mathbf{y}) + c\mathbf{y}\times(\mathbf{a}\times\mathbf{h}) \,.$$

The Jacobian $\mathrm{D}\mathbf{f}(\mathbf{y})$ is the matrix representation of this mapping $\mathbf{h}\mapsto\mathrm{D}\mathbf{f}(\mathbf{y})$. Using the definition of cross product, a simple but tedious calculation shows that

$$\mathrm{D}\mathbf{f}(\mathbf{y}) = \begin{bmatrix} -ca_2y_2 - ca_3y_3 & -a_3 + 2ca_1y_2 - ca_2y_1 & a_2 - ca_3y_1 + 2ca_1y_3 \\ a_3 - ca_1y_2 + 2ca_2y_1 & -ca_3y_3 - ca_1y_1 & -a_1 + 2ca_2y_3 - ca_3y_2 \\ -a_2 + 2ca_3y_1 - ca_1y_3 & a_1 - ca_2y_3 + 2ca_3y_2 & -ca_1y_1 - ca_2y_2 \end{bmatrix}.$$

Thus for $\mathbf{y} = \mathbf{a}$ we obtain

$$\begin{aligned} \mathrm{D}\mathbf{f}(\mathbf{a}) &= \begin{bmatrix} -c(a_2^2+a_3^2) & -a_3 + ca_1a_2 & a_2 + ca_3a_1 \\ a_3 + ca_1a_2 & -c(a_1^2+a_3^2) & -a_1 + ca_2a_3 \\ -a_2 + ca_1a_3 & a_1 + ca_2a_3 & -c(a_1^2+a_2^2) \end{bmatrix} \\ &= \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} + c\mathbf{a}\mathbf{a}^\top - c\|\mathbf{a}\|^2\mathbf{I} \,. \end{aligned}$$

The next step is an exercise in advanced linear algebra: We see that $\mathrm{D}\mathbf{f}(\mathbf{a})\mathbf{a} = \mathbf{0}$ and on the orthogonal complement $\mathbf{a}^\perp$ the matrix acts like a rotation (scaled with $\|\mathbf{a}\|$) minus $c\|\mathbf{a}\|\mathbf{I}$. This reveals that the spectrum (= set of eigenvalues) is given by

$$\sigma(\mathrm{D}\mathbf{f}(\mathbf{a})) = \{0, -c\|\mathbf{a}\|_2^2 \pm i\|\mathbf{a}\|_2\} = \{0, -c \pm i\} \,. \tag{12.2.2}$$

HINT 1 for (12-2.c): Recall from [Lecture → Section 12.2]:

An initial value problem for an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ will be stiff, if, for substantial periods of time,

$$\min\{\operatorname{Re}\lambda : \lambda \in \sigma(\mathrm{D}\,\mathbf{f}(\mathbf{y}(t)))\} \ll 0\,,$$
$$\textbf{and} \qquad \max\{\operatorname{Re}\lambda : \lambda \in \sigma(\mathrm{D}\,\mathbf{f}(\mathbf{y}(t)))\} \lesssim 0\,,$$

where $t \mapsto \mathbf{y}(t)$ is the solution trajectory and $\sigma(\mathbf{M})$ is the spectrum of the matrix $\mathbf{M}$, see [Lecture → Def. 9.1.0.1].

⌟

SOLUTION of (12-2.c):

Appealing to (12.2.2) and applying the criteria [Lecture → Eq. (12.2.0.13)] and [Lecture → Eq. (12.2.0.14)] we find that at close to the state $\mathbf{y} = \mathbf{a}$ the ODE (12.2.1) is stiff for large $c > 0$.

SOLUTION of (12-2.d):

In the plot, we see that, on the one hand for the integrator **Ode45** the number of steps rises with c. On the other hand, the A-stable method uses roughly the same amount of steps, irregardless of the value of c chosen.

As **Ode45** is an explicit solver, it suffers from stability-induced step-size constraints for large $c > 0$. The A-stable integrator need not use smaller timesteps to satisfy the tolerance for big c.

This demonstrates [Lecture $\rightarrow$ Notion 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 (12-2.b).

SOLUTION of (12-2.e):

Recall the recursion formula for the implicit mid-point method applied to an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$,

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) .$$

Its specialization for (12.2.1) reads

$$\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h} = \mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) + c\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) \times \left(\mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right)\right) .$$

SOLUTION of (12-2.f):

t	$\lvert y(t)\rvert$
0	1.73205
0.078125	1.73205
0.15625	1.73205
0.234375	1.73205
0.3125	1.73205
⋮	⋮

We observe that norm of the approximate states generated by the single step method *is constant*, $\|\mathbf{y}_k\|_2 = \|\mathbf{y}_0\|$ for all k, and equal to $1.73205 = \left\|[1,1,1]^\top\right\|_2$.

Our implementation relies on an auxiliary function that calls **implicitRKIntegrator** with parameters characteristic of the implicit midpoint method. The Butcher scheme for the implicit midpoint method can be found in [Lecture → Ex. 12.3.4.5].

The right-hand side function for (12.2.1) and its Jacobian as computed in Sub-problem (12-2.b) are realized as lambda functions.

C++-code 12.2.3: Implementation of `tab_crossprod`

```
2  void tab_crossprod() {
3    // set data
4    constexpr double T = 10.;
5    constexpr unsigned int N = 128;
6    constexpr double c = 1.;
7    Eigen::Vector3d y0;
8    y0 << 1., 1., 1.;
9    Eigen::Vector3d a;
10   a << 1., 0., 0.;
11   // define rhs
12   auto f = [&a, c](const Eigen::Vector3d &y) -> Eigen::Vector3d {
13     return a.cross(y) + c * y.cross(a.cross(y));
14   };
15   // TODO: (12-2.f) solve the cross-product ODE with the implicit RK method
16   // defined in solve_imp_mid. Tabulate the norms of the results at all steps.
17   // START
18   // define Jacobian of rhs
19   auto Jf = [&a](const Eigen::Vector3d &y) {
20     Eigen::Matrix3d temp;
21     temp << -c * (a(1) * y(1) + a(2) * y(2)),
22         c * (2 * a(0) * y(1) - a(1) * y(0)) - a(2),
23         a(1) + c * (2 * a(0) * y(2) - a(2) * y(0)),
24         a(2) - c * (a(0) * y(1) - 2 * a(1) * y(0)),
25         -c * (a(0) * y(0) + a(2) * y(2)),
26         c * (2 * a(1) * y(2) - a(2) * y(1)) - a(0),
27         -a(1) - c * (a(0) * y(2) - 2 * a(2) * y(0)),
28         a(0) - c * (a(1) * y(2) - 2 * a(2) * y(1)),
29         -c * (a(0) * y(0) + a(1) * y(1));
30     return temp;
31   };
32
33   // Solving the ODE using implicit midpoint method
34   std::vector<Eigen::VectorXd> res_imp(N + 1);
35   // Initialize implicit RK with Butcher scheme
36   constexpr unsigned int s = 1;
```

```
37    // Initialize coefficients for the implicit midpoint method
38    Eigen::MatrixXd A(s, s);
39    Eigen::VectorXd b(s);
40    A << 1. / 2.;
41    b << 1.;
42    implicitRKIntegrator RK(A, b);
43    res_imp = RK.solve(f, Jf, T, y0, N);
44
45    // Tabulating the norm of generated states
46    std::cout << "1. Implicit midpoint method" << std::endl;
47    std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
48              << std::endl;
49
50    for (unsigned int i = 0; i < N + 1; ++i) {
51      std::cout << std::setw(10) << T * i / N << std::setw(15)
52                << res_imp[i].norm() << std::endl;
53    }
54    // END
```

Get it on GitLab (cross.hpp).

HINT 1 for (12-2.g): The linearization of increment equations is also discussed in [Lecture → Exp. 12.4.0. and [Lecture → Eq. (12.4.0.5)]. ⌟

SOLUTION of (12-2.g):

The linear implicit mid-point method is obtained by expanding the right-hand side of the increment equations of the implicit mid point method in its Taylor series in h around $h = 0$:

$$\mathbf{k}_1 = \mathbf{f}\left(\mathbf{y}_k + \frac{h}{2}\mathbf{k}_1\right) = \mathbf{f}(\mathbf{y}_k) + \frac{h}{2}D\mathbf{f}(\mathbf{y}_k)\mathbf{k}_1 + O(h^2)$$

and only taking the linear terms. Since the non-linear method is given by $\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1$, the linearization reads

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1^{\text{lin.}} \quad , \qquad \mathbf{k}_1^{\text{lin.}} := \left(I - \tfrac{h}{2}D\mathbf{f}(\mathbf{y}_k)\right)^{-1}\mathbf{f}(\mathbf{y}_k) \ .$$

A step of the method just involves the solution of a single $N \times N$ linear system of equations.

SOLUTION of (12-2.h):

See file `cross.hpp`.

t	$\lvert y(t)\rvert$
0	1.73205
0.078125	1.73151
0.15625	1.731
0.234375	1.73054
0.3125	1.73015
0.390625	1.72981
0.46875	1.72954
0.546875	1.72932
0.625	1.72915
0.703125	1.72901
0.78125	1.7289
0.859375	1.72882
0.9375	1.72875
⋮	⋮

The sequence of the Euclidean norms of the states is not exactly constant: it starts at 1.73205 and converges to 1.72853. This is due to the approximation introduced with the linearization.

C++-code 12.2.4: `solve_lin_mid`

```
template <class Function, class Jacobian>
std::vector<Eigen::VectorXd> solve_lin_mid(Function &&f, Jacobian &&Jf,
                                           double T, const Eigen::VectorXd &y0,
                                           unsigned int N) {
  std::vector<Eigen::VectorXd> res;
  // TODO: (12-2.h) 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
  const double h = T / N;
  // Store initial data
  res.push_back(y0);

  // Initialize some memory to store temporary values
  Eigen::VectorXd ytemp1 = y0;
  Eigen::VectorXd ytemp2 = y0;
  // Pointers to swap previous value
  Eigen::VectorXd *yold = &ytemp1;
  Eigen::VectorXd *ynew = &ytemp2;
  Eigen::MatrixXd eye = Eigen::MatrixXd::Identity(3, 3);

  // Loop over all fixed steps
  for (unsigned int k = 0; k < 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;
```

```
33 }
```

Get it on GitLab (cross.hpp).

C++-code 12.2.5: Solve the cross-product ODE with implicit RK

```
2  // TODO: (12-2.h) solve the cross-product ODE with the implicit RK method
3  // defined in solve_lin_mid. Tabulate the norms of the results at all steps.
4  // START
5  std::vector<Eigen::VectorXd> res_lin = solve_lin_mid(f, Jf, T, y0, N);
6
7  std::cout << "\n2. Linear implicit midpoint method" << std::endl;
8  std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
9            << std::endl;
10 for (int i = 0; i < N + 1; ++i) {
11   std::cout << std::setw(10) << T * i / N << std::setw(15)
12             << res_lin[i].norm() << std::endl;
13 }
14 // END
```

Get it on GitLab (cross.hpp).

HINT 1 for (12-3.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. 12.3.3.9].

⌟

SOLUTION of (12-3.a):

For SDIRK methods the Butcher matrix is lower triangular. In this case the increment equations/stage equations can be solved separately and sequentially, computing $\mathbf{k}_1 \to \mathbf{k}_2 \to \cdots \to \mathbf{k}_s$ in that order.

So instead of solving a big $sN \times sN$ non-linear system of equations for a generic implicit Runge-Kutta single-step method, we need only solve s smaller $N \times N$ non-linear systems.

SOLUTION of (12-3.b):

The solution is essentially given in [Lecture → Def. 12.3.3.1]. The increments $\mathbf{k}_i$ are given as solutions of the two $N \times N$ non-linear systems of equations,

$$\begin{aligned} \mathbf{k}_1 &= f(t_0 + h\gamma, \boldsymbol{y}_0 + h\gamma\mathbf{k}_1) \ , \\ \mathbf{k}_2 &= f(t_0 + h(1-\gamma), \boldsymbol{y}_0 + h(1-2\gamma)\mathbf{k}_1 + h\gamma\mathbf{k}_2) \ . \end{aligned}$$

SOLUTION of (12-3.c):

The stability function $S(z)$ of a method is derived by applying the method to the scalar, linear test equation

$$\dot{y}(t) = \lambda y(t) \ .$$

The solution can be written as

$$y_{k+1} = S(z) y_k \ ,$$

where $S(z)$ is the stability function and $z := h\lambda$.

In the case of the SDIRK method, we get the increment equations

$$\begin{aligned} k_1 &= \lambda(y_k + h\gamma k_1) \ , \\ k_2 &= \lambda(y_k + h(1-2\gamma)k_1 + h\gamma k_2) \ . \end{aligned}$$

Therefore:

$$\begin{aligned} k_1 &= \frac{\lambda}{1-h\lambda\gamma} y_k , \\ k_2 &= \frac{\lambda}{1-h\lambda\gamma}(y_k + h(1-2\gamma)k_1) \ , \end{aligned}$$

which leads to the recursion

$$y_{k+1} = y_k + \frac{h}{2}(k_1 + k_2) \ .$$

With $z := h\lambda$ and by plugging in k_1 and k_2 we arrive at:

$$y_{k+1} = \underbrace{\left(1 + \frac{z}{2(1-\gamma z)}\left(2 + \frac{z(1-2\gamma)}{1-\gamma z}\right)\right)}_{=:S(z)} y_k \ .$$

Hence:

$$S(z) = \frac{2(1-\gamma z)^2 + 2z(1-\gamma z)^2 + z^2(1-2\gamma)}{2(1-\gamma z)^2}$$

and after collecting the powers of z in the numerator we get

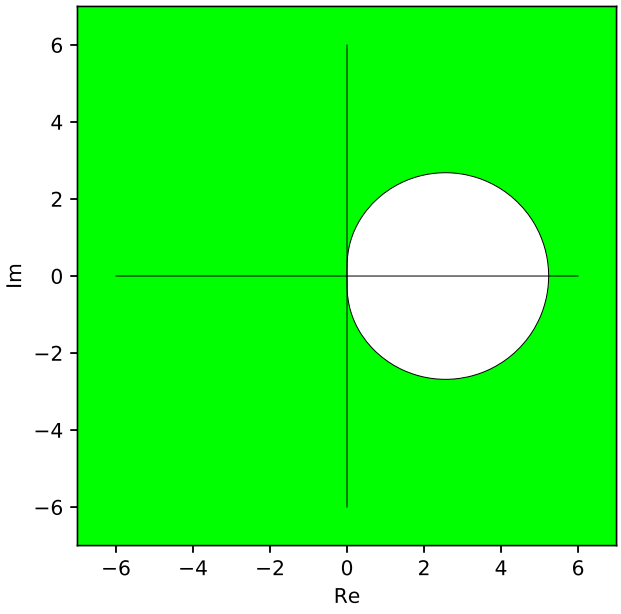
$$S(z) = \frac{1 + z(1-2\gamma) + z^2(\gamma^2 - 2\gamma + \frac{1}{2})}{(1-\gamma z)^2} \ .$$

Alternatively, you can directly appeal to the formula for the stability function of an implicit RK-SSM, in particular the determinant-based version:

Theorem [Lecture → Thm. 12.3.4.4]. Stability function of general Runge-Kutta methods

The discrete evolution Ψ_λ^h of an s-stage Runge-Kutta single step method (→ [Lecture → Def. 12.3.3.1]) with Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ *[Lecture → Eq. (12.3.3.3)] for the ODE $\dot{y} = \lambda y$ is given by a multiplication with*

$$S(z) := \underbrace{1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1}}_{\text{stability function}} = \frac{\det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T)}{\det(\mathbf{I} - z\mathfrak{A})} \ , \quad z := \lambda h \ , \quad \mathbf{1} = [1, \ldots, 1]^T \in \mathbb{R}^s \ .$$



◁ Plot of the stability domain (green) of the 2-stage SDIRK method from (12.3.3).

- The imaginary axis is tangent to the boundary of the stability domain in $z = 0$
- We see that the left half plane $\mathbb{C}^-$ is included in the stability domain.

SOLUTION of (12-3.d):

Given the previous sub-problem, for $\gamma = 1$ the stability function is:

$$S_1(z) := \frac{1 - z - \frac{z^2}{2}}{(1-z)^2} \tag{12.3.5}$$

Verification of the A-stability of (12.3.5)**:**
By definition [Lecture → § 12.3.4.7], we need to show that $|S_1(z)| \leq 1$ for all $z \in \mathbb{C}^- := \{z \in \mathbb{C} \mid \operatorname{Re} z < 0\}$. In order to do this, we consider the stability function on the imaginary axis:

$$\begin{aligned} |S_1(iy)|^2 &= \frac{|1 - iy - (iy)^2/2|^2}{|1-iy|^4} = \frac{|1 - iy + y/2|^2}{|1-iy|^4} \\ &= \frac{(1 + y^2/2)^2 + y^2}{(1+y^2)^2} = \frac{1 + 2y^2 + y^4/4}{1 + 2y^2 + y^4} \leq 1, \quad y \in \mathbb{R} \end{aligned}$$

Since the only pole ($z = 1$) of the rational function $S_1(z)$ lies in the positive half plane of $\mathbb{C}$, the function S_1 is holomorphic in the left half plane. Furthermore, S_1 is bounded by 1 on the boundary of this half plane (i.e. on the imaginary axis). So, by the maximum principle for holomorphic functions (hint), S_1 is bounded on the entire left half plane by 1. This implies in particular that S_1 is A-stable.

Verification of the L-stability of (12.3.5)**:**
S_1 is not L-stable (*cf.* definition [Lecture → § 12.3.4.14]), because:

$$\lim_{\operatorname{Re} z \to -\infty} |S_1(z)| = \lim_{\operatorname{Re} z \to -\infty} \left| \frac{1 - z - z^2/2}{1 - 2z + z^2} \right| = \frac{1}{2} \neq 0$$

SOLUTION of (12-3.e):

Define $z_1 = y$, $z_2 = \dot{y}$, then the initial value problem

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0\,, \tag{12.3.6}$$

is equivalently to the *linear* first order system of ODEs

$$\begin{aligned} \dot{z}_1 &= z_2\,, \\ \dot{z}_2 &= -z_1 - z_2 \end{aligned} \quad \Leftrightarrow \quad \frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\,, \tag{12.3.7}$$

with initial values $z_1(0) = 1$ and $z_2(0) = 0$.

SOLUTION of (12-3.f):

The first-order ODE derived in Sub-problem (12-3.e) is of the form

$$\dot{\mathbf{z}} = \mathbf{A}\mathbf{z} \quad \text{with} \quad \mathbf{A} := \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix} . \tag{12.3.8}$$

From the course we recall (note that instead of $\mathbf{z}$ this uses the notation $\mathbf{y}$):

Definition [Lecture → Def. 12.3.3.1]. General Runge-Kutta single step method

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, an s-stage Runge-Kutta single step method (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$ is defined by

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{s} a_{ij}\mathbf{k}_j) , \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i .$$

The vectors $\mathbf{k}_i \in \mathbb{R}^N$ are called **increments**.

Shorthand notation for Runge-Kutta methods

Butcher scheme ▷

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} .$$

[Lecture → Eq. (12.3.3.3)]

Note that now, in contrast to [Lecture → Eq. (11.4.0.13)], $\mathfrak{A}$ can be a general $s \times s$-matrix.

Thus we can write the increment equations for (12.3.8) as

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{A}\mathbf{z}_0 + h\gamma\mathbf{A}\mathbf{k}_1 , \\ \mathbf{k}_2 &= \mathbf{A}\mathbf{z}_0 + h(1-2\gamma)\mathbf{A}\mathbf{k}_1 + h\mathbf{A}\mathbf{k}_2 . \end{aligned}$$

This yields the explicit formulas

$$\begin{aligned} \mathbf{k}_1 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}\mathbf{A}\mathbf{z}_0 , \\ \mathbf{k}_2 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}(\mathbf{A}\mathbf{z}_0 + h(1-2\gamma)\mathbf{A}\mathbf{k}_1) . \end{aligned} \tag{12.3.9}$$

Note that a linear system of equations with the same coefficient matrix has to be solved to obtain both $\mathbf{k}_1$ and $\mathbf{k}_2$. Finally, we have the update equation

$$\mathbf{z}_1 = \mathbf{z}_0 + h(\tfrac{1}{2}\mathbf{k}_1 + \tfrac{1}{2}\mathbf{k}_2) .$$

C++11-code 12.3.10: Solution of (12-3.f)

```
2 Eigen::Vector2d sdirkStep(const Eigen::Vector2d &z0, double h, double gamma) {
3   Eigen::Vector2d res = z0;
4   // TODO: (12-3.f) compute one timestep of the ODE
5   // START
6   // Matrix A for evaluation of f
7   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
    res = z0 + h * 0.5 * (k1 + k2);
    // END
    return res;
}
```

Get it on GitLab (SDIRK.hpp).

HINT 1 for (12-3.g): The exact solution of (12.3.2) is

$$y(t) = \tfrac{1}{3}e^{-\frac{1}{2}t}\left(3\cos(\tfrac{1}{2}\sqrt{3}t) + \sqrt{3}\sin(\tfrac{1}{2}\sqrt{3}t)\right) .$$

It can be computed by finding the roots of the characteristic polynomial associated with the second-order ODE. ⌟

SOLUTION of (12-3.g):

C++11-code 12.3.11: Testing convergence of SDIRK method for an IVP for (12.3.2)

```
double cvgSDIRK() {
  double conv_rate = 0.;
  // Initial data z0 = [y(0), y'(0)]
  Eigen::Vector2d z0;
  z0 << 1, 0;
  // Final time
  constexpr double T = 10;
  // Parameter
  const double gamma = (3. + std::sqrt(3.)) / 6.;
  // Mesh sizes
  Eigen::ArrayXd err(10);
  Eigen::ArrayXd N(10);
  N << 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240;
  // TODO: (12-3.g) Study the convergence rate of the method.
  // START

  // 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) {
    unsigned int n = N(i);
    // Get solution
    std::vector<Eigen::Vector2d> sol(n + 1);
    // Equidistant step size
    const double h = T / n;
    // Push initial data
    sol.at(0) = z0;
    // Main loop
    for (unsigned int i = 1; i <= n; ++i) {
      sol.at(i) = sdirkStep(sol.at(i - 1), h, gamma);
    }
    // Compute error
    err(i) = std::abs(sol.back()(0) - yex(T));

    // Print table
    std::cout << std::setw(15) << n << std::setw(15) << err(i);
    if (i > 0) std::cout << std::setw(15) << std::log2(errold / err(i));
    std::cout << std::endl;

    // Store old error
    errold = err(i);
  }
```

```
54    Eigen::VectorXd coeffs = polyfit(N.log(), err.log(), 1);
55    conv_rate = -coeffs(0);
56    // END
57    return conv_rate;
58  }
```

Get it on GitLab (SDIRK.hpp).

N	error
20	0.000283567
40	5.05272e-05
80	7.73715e-06
160	1.07013e-06
320	1.40565e-07
640	1.80048e-08
1280	2.27798e-09
2560	2.86467e-10
5120	3.59161e-11
10240	4.49619e-12

◁ Norm of error at final time $T = 10$ on temporal mesh.

We observe an asymptotic (in the last steps) decrease of the error norm by a factor of ≈ 8 whenever the number of (equidistant) timesteps is doubled. Thus we observe *algebraic convergence* with rate 3 in the timestep size h:

$$\text{errornorm} = O(h^3) .$$

This hints that the SDIRK single-step method has order 3 .

The numerically estimated order of convergence is 3.

SOLUTION of (12-4.a):

Matching [Lecture → Eq. (12.4.0.9)] and (12.4.2) we find

- that we deal with a 2-stage method: $s = 2$,
- $d_{2,1} = a$,
- $a_{11} = a_{22} = a$, $a_{2,1} = \frac{1}{2} - a$,
- $b_1 = 0$, $b_2 = 1$.

SOLUTION of (12-4.b):

Give a formula for single step of size $h > 0$ of the RK-SSM for the scalar linear ODE $\dot{y} = \lambda y$ in terms of the method's stability function:

$$y_1 = S(h\lambda)y_0 \ .$$

SOLUTION of (12-4.c):

For a scalar ODE, the Jacobian is just the derivative w.r.t. y, whence $J = Df(y) = f'(y) = (\lambda y)' = \lambda$. The quantity $\mathbf{W}$ is, therefore a scalar quantity as well:

$$W = 1 - ahJ = 1 - ah\lambda$$

If we combine these formulas and assume h is small enough:

$$\begin{aligned}
y_1 &= y_0 + h\frac{f(y_0 + \frac{1}{2}hk_1) - ahJk_1}{W} \\
&= y_0 + h\frac{f(y_0 + \frac{1}{2}h\frac{f(y_0)}{W}) - ahJ\frac{f(y_0)}{W})}{W} \\
&= y_0 + h\frac{\lambda(y_0 + \frac{1}{2}h\frac{\lambda y_0}{1-ah\lambda}) - \frac{ah\lambda^2 y_0}{1-ah\lambda}}{1 - ah\lambda} \\
&= \frac{(1-ah\lambda)^2 + h\lambda(1-ah\lambda) + \frac{1}{2}h^2\lambda^2 - ah^2\lambda^2}{(1-ah\lambda)^2} y_0 \\
&= \frac{(1 + a^2z^2 - 2az) + z(1-az) + \frac{1}{2}z^2 - az^2}{(1-az)^2} y_0 \\
&= \frac{1 + (1-2a)z + (\frac{1}{2} - 2a + a^2)z^2}{(1-az)^2} y_0
\end{aligned}$$

Since $\frac{1}{2} - 2a + a^2 = 0$, it follows

$$S(z) = \frac{1 + (1-2a)z}{(1-az)^2} \,. \tag{12.4.3}$$

HINT 1 for (12-4.d): The idea behind this sub-problem is elucidated in [Lecture → Rem. 12.1.0.21]. Apply [Lecture → Lemma 12.1.0.23]. ⌟

SOLUTION of (12-4.d):

We compute:

$$\begin{aligned} S(0) &= 1, \\ S'(0) &= 1, \\ S''(0) &= 4a - 2a^2, \\ S'''(0) &= 18a^2 - 12a^3 \ , \end{aligned}$$

therefore

$$S(z) = 1 + z + (2a - a^2)z^2 + (3a^2 - 2a^3)z^3 + O(z^4) \quad \text{for} \quad z \to 0 \ .$$

We can also compute the expansion of $\exp$:

$$\exp(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + O(z^4) \quad \text{for} \quad z \to 0 \ .$$

Since $2a - a^2 = \frac{1}{2}$ but $3a^2 - 2a^3 = \frac{1}{2}(\sqrt{2} - 1) \neq \frac{1}{6}$, we have:

$$|S(z) - \exp(z)| = O(|z|^3) \ .$$

Note that this means that 3 is the order of the one-step error when applying the single-step method to solve initial-value problems for the ODE $\dot{y} = y$. Using [Lecture → Lemma 12.1.0.23], we deduce that the maximal order q of the scheme is $q = 2$.

SOLUTION of (12-4.e):

The implementation relies on EIGEN's direct solvers for dense linear systems.

C++11-code 12.4.4: Solution of Sub-problem (12-4.e)

```cpp
template <class Function, class Jacobian>
std::vector<Eigen::VectorXd> solveRosenbrock(Function &&f, Jacobian &&df,
                                             const Eigen::VectorXd &y0,
                                             unsigned int N, double T) {
  // Will contain all time steps
  std::vector<Eigen::VectorXd> res(N + 1);

  // TODO: (12-4.e) Implement the Rosenbrock method. Note that the function
  // f should take as argument an Eigen::VectorXd, and return one as well.
  // The Jacobian df should take as argument an Eigen::VectorXd, but return
  // a square Eigen::MatrixXd.
  // START

  res.at(0) = y0;  // Push initial data
  const double h = T / N;
  const double a = 1. / (std::sqrt(2) + 2.);

  // Some temporary variables
  Eigen::VectorXd k1, k2;
  Eigen::MatrixXd J, W;

  // Main loop: *up to N (performs N steps)*
  for (unsigned int i = 1; i <= N; ++i) {
    Eigen::VectorXd &yprev = res.at(i - 1);

    // Jacobian computation
    J = df(yprev);
    W = Eigen::MatrixXd::Identity(J.rows(), J.cols()) - 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;
  }

  // END
  return res;
}
```

Get it on GitLab (rosenbrock.hpp).

SOLUTION of (12-4.f):

The Jacobian Df is:

$$D\mathbf{f}(\mathbf{y}) = \begin{bmatrix} \lambda(1-\|\mathbf{y}\|^2) - 2\lambda y_0^2 & -1 - 2\lambda y_1 y_0 \\ 1 - 2\lambda y_1 y_0 & \lambda(1-\|\mathbf{y}\|^2) - 2\lambda y_1^2 \end{bmatrix}$$

C++11-code 12.4.6: Solution of Sub-problem (12-4.e)

```
double cvgRosenbrock() {
  double cvgRate = 0;
  // TODO: (12-4.f) Use polyfit() to estimate the rate of convergence
  // for solveRosenbrock().
  // START

  // Final time
  constexpr double T = 10;
  // Mesh sizes h=2^{-k} for k in K.
  const Eigen::ArrayXd K = Eigen::ArrayXd::LinSpaced(7, 4, 10);

  // Initial data
  Eigen::Vector2d y0;
  y0 << 1., 1.;
  // Parameter and useful matrix for f
  constexpr double lambda = 1;
  Eigen::Matrix2d R;
  R << 0., -1., 1., 0.;

  // Function and its Jacobian
  auto f = [&R](const Eigen::Vector2d &y) {
    return R * y + lambda * (1. - y.squaredNorm()) * y;
  };
  auto df = [](const Eigen::Vector2d &y) {
    double x = 1 - y.squaredNorm();
    Eigen::Matrix2d J;
    J << lambda * x - 2 * lambda * y(0) * y(0), -1 - 2 * lambda * y(1) * y(0),
        1 - 2 * lambda * y(1) * y(0), lambda * x - 2 * lambda * y(1) * y(1);
    return J;
  };

  // Reference mesh size
  const int N_ref = 10 * std::pow(2, 12);
  // Reference solution
  auto solref = solveRosenbrock(f, df, y0, N_ref, T);

  Eigen::ArrayXd Error(K.size());
  std::cout << std::setw(15) << "N" << std::setw(16) << "maxerr\n";
  // Main loop: loop over all meshes
  for (unsigned int i = 0; i < K.size(); ++i) {
    // h = 2^{-k} => N = T*h = T*2^k
    int N = T * std::pow(2, K[i]);
    // Get solution
    auto sol = solveRosenbrock(f, df, y0, N, T);
    // Compute error
    double maxerr = 0;
    for (unsigned int j = 0; j < sol.size(); ++j) {
```

```
49        maxerr =
50            std::max(maxerr, (sol.at(j) - solref.at((j * N_ref) / N)).norm());
51      }
52
53      Error[i] = maxerr;
54      std::cout << std::setw(15) << N << std::setw(16) << maxerr << std::endl;
55    }
56    // Use log(N)=log(T*2^k)=log(T)+log(2)*k to get natural logarithm of N.
57    cvgRate = -polyfit(std::log(2) * K, Error.log(), 1)(0);
58    // END
59    return cvgRate;
60  }
```

Get it on GitLab (rosenbrock.hpp).

M	maxerr
160	0.00181187
320	0.000439645
640	0.000108218
1280	2.68249e-05
2560	6.66039e-06
5120	1.64229e-06
10240	3.90643e-07

Least squares fitting of a linear function to $(\log h_i, \mathrm{err}(i))$ yields the estimated rate of algebraic convergence 2.02461 through the slope of the linear function, see [Lecture → Ex. 3.0.1.4].

Alternatively, we can read off the error table that doubling the number M of timesteps roughly makes the error decrease by a factor of four, which hints that $\texttt{maxerr} \sim M^{-2}$.

We conjecture that the Rosenbrock single-step method (12.4.2) has order 2 , an integer order as all other commonly used single-step methods.

HINT 1 for (12-4.g): To investigate A-stability, calculate the complex norm of $S(z)$ on the imaginary axis $\mathrm{Re}\, z = 0$ and apply the ⌟

SOLUTION of (12-4.g):

We start by proving that the method is A-stable [Lecture → § 12.3.4.7], meaning that $S(z) < 1, \forall z \in \mathbb{C}, \mathrm{Re}(z) < 0$. First of all, we can compute the complex norm of the stability function at $z = iy, y \in \mathbb{R}$ as:

$$|S(iy)|^2 = \frac{|1 + (1-2a)iy|^2}{|(1-aiy)^2|^2} = \frac{1 + (1-2a)^2y^2}{(1+a^2y^2)^2} = \frac{1 + (1-4a+4a^2)y^2}{(1+2a^2y^2+a^4y^4)} .$$

Notice that $1 - 4a + 4a^2 = 2a^2$, therefore:

$$|S(iy)|^2 = \frac{1 + 2a^2y^2}{1 + 2a^2y^2 + a^4y^4} < 1 .$$

The norm of the function S is bounded on the imaginary axis. Observe that $|S(z)| \to 0, |z| \to \infty$, which, follows from the fact that the degree of the denominator of S is bigger than the polynomial degree of the numerator. Notice that $1 - 2a = 1 - 2\frac{1}{\sqrt{(2)}+2} = \sqrt{2} - 2$.

The only pole of the function is at $z = 1/a$, therefore the function is holomorphic on the left complex plane.

Applying the theorem of the hint on concludes that the absolute value of the function is bounded by 1 on the left complex plane.

The L-stability follows immediately with A-stability and the fact that the absolute value of the function converges to zero as $|z| \to \infty$.

SOLUTION of (12-5.a):

We recall the notion of consistency of a single-step method:

Definition [Lecture → Def. 11.3.1.12]. Consistent single step methods

A single step method according to [Lecture → Def. 11.3.1.5] based on a discrete evolution of the form

$$\mathbf{\Psi}^h\mathbf{y} = \mathbf{y} + h\boldsymbol{\psi}(h,\mathbf{y}) \quad \text{with} \quad \begin{array}{l}\boldsymbol{\psi} : I \times D \to \mathbb{R}^N \text{ continuous,}\\ \boldsymbol{\psi}(0,\mathbf{y}) = \mathbf{f}(\mathbf{y})\ .\end{array} \quad [\text{Lecture} \to \text{Eq. (11.3.1.11)}]$$

is called **consistent** with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

In view of (12.5.1), consistency is equivalent to

$$\varphi\big(h\,\mathrm{D}\,\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0) = \mathbf{f}(\mathbf{y}_0) \quad \text{for} \quad h = 0\ .$$

By definition of the matrix exponential we have

$$\exp(hZ) = \mathsf{Id} + hZ + O(h^2) \quad \text{as} \quad h \to 0\ .$$

Therefore, for $h \to 0$,

$$\varphi(h\,\mathrm{D}\,\mathbf{f}(\mathbf{y}_0)) = (\exp(h\,\mathrm{D}\,\mathbf{f}(\mathbf{y}_0)) - \mathsf{Id})(h\,\mathrm{D}\,\mathbf{f}(\mathbf{y}_0))^{-1} = (\mathrm{D}\,\mathbf{f}(\mathbf{y}_0)) + O(h))\,\mathrm{D}\,\mathbf{f}(\mathbf{y}_0)^{-1} = \mathbf{I} + O(h)\ .$$

Whence

$$\varphi(\mathbf{O}) = \mathbf{I} \quad \Rightarrow \quad \varphi\big(h\,\mathrm{D}\,\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0) = \mathbf{f}(\mathbf{y}_0)\ ,$$

as desired.

HINT 1 for (12-5.b): Recall [Lecture → Eq. (12.1.0.34)]; the solution of the IVP is $\mathbf{y}(t) = \exp(\mathbf{A}t)\mathbf{y}_0$. To facilitate formal calculations, you may assume that $\mathbf{A}$ is regular. ⌟

SOLUTION of (12-5.b):

Given $\mathbf{y}(0)$, a step of size h of the method gives

$$\begin{aligned}\mathbf{y}_1 &= \mathbf{y}(0) + h\ \varphi\big(h\ \mathsf{D}\,\mathbf{f}(\mathbf{y}(0))\big)\ \mathbf{f}(\mathbf{y}(0)) \\ &= \mathbf{y}(0) + h\ \varphi\big(h\ \mathbf{A}\big)\ \mathbf{A}\ \mathbf{y}(0) = \mathbf{y}(0) + h\ \big(\exp(\mathbf{A}h) - \mathsf{Id}\big)\ (h\mathbf{A})^{-1}\ \mathbf{A}\ \mathbf{y}(0) \\ &= \mathbf{y}(0) + \exp(\mathbf{A}h)\ \mathbf{y}(0) - \mathbf{y}(0) = \exp(\mathbf{A}h)\ \mathbf{y}(0) = \mathbf{y}(h)\ .\end{aligned}$$

Hence the discrete evolution exactly reproduces the exact evolution for (12.5.3).

SOLUTION of (12-5.c):

The discrete evolution $\boldsymbol{\Psi}^h$ associated to (12.5.1) is given by

$$\boldsymbol{\Psi}^h(\mathbf{y}) = \mathbf{y} + h\varphi(h\,\mathrm{D}\,\mathbf{f}(\mathbf{y}))\mathbf{f}(\mathbf{y}).$$

Thus, for a scalar linear ODE of the form $\dot{y} = \lambda y$ for $\lambda \in \mathbb{C}$ we have ($f(y) = \lambda y$, $Df(y) = \lambda$)

$$\boldsymbol{\Psi}^h(y) = y + h\frac{e^{h\lambda} - 1}{h\lambda}\lambda y = e^{h\lambda}y.$$

Thus, the stability function $S\colon \mathbb{C} \to \mathbb{C}$ is given by $S(z) = e^z$. Therefore, the region of stability of this method is

$$\mathcal{S}_{\boldsymbol{\Psi}} = \{z \in \mathbb{C} : |e^z| < 1\} = \{z \in \mathbb{C} : e^{\mathrm{Re}\,z} < 1\} = \{z \in \mathbb{C} : \mathrm{Re}\,z < 0\}.$$

It is illuminating to relate this to [Lecture → Ex. 12.1.0.53].

Alternatively, one may simply appeal to Sub-problem (12-5.b) to see that the method has the "ideal" region of stability as introduced in [Lecture → § 12.3.4.10].

SOLUTION of (12-5.d):

Given the auxiliary function `phim()`, the implementation of (12.5.1) is straightforward.

C++11-code 12.5.5: Code of `exponentialEulerStep`

```
2  template <class Function, class Jacobian>
3  Eigen::VectorXd exponentialEulerStep(const Eigen::VectorXd &y0, Function &&f,
4                                       Jacobian &&df, double h) {
5    Eigen::VectorXd y1 = y0;
6    // TODO: (12-5.d) Implement one step of the exponential Euler method.
7    // START
8    y1 = y0 + h * phim(h * df(y0)) * f(y0);
9    // END
10   return y1;
11 }
```

Get it on GitLab (exponentialintegrator.hpp).

HINT 1 for (12-5.e): The exact solution is

$$y(t) = \frac{y(0)}{y(0) + \big(1 - y(0)\big)e^{-t}}.$$

SOLUTION of (12-5.e):

M	error
2	0.00168674
4	0.000452805
8	0.000117603
16	2.99917e-05
32	7.57469e-06
64	1.90346e-06
128	4.77101e-07
256	1.1943e-07
512	2.98771e-08
1024	7.4717e-09
2048	1.86823e-09
4096	4.67095e-10
8192	1.16777e-10
16384	2.91974e-11
32768	7.29886e-12

Choosing a geometrically increasing sequence of equidistant timesteps is the most suitable approach to detecting conjectured algebraic convergence $|y(T) - y_M| = O(M^{-r})$ for $M \to \infty$.
In this experiment doubling the number of timesteps leads to a decrease of the error at final time by a factor of ≈ 4, which makes us conjecture

$$|y(T) - y_M| = O(M^{-2}) \quad \text{for} \quad M \to \infty .$$

In terms of the timestep size $h = \frac{1}{M}$ this means

$$\text{error}(h) = |y(T) - y_M| = O(h^2) \quad \text{for} \quad h \to 0 .$$

We conjecture that the exponential Euler method has order 2 .

C++11-code 12.5.6: Code for `testExpEulerLogODE()`

```
2  void testExpEulerLogODE() {
3    // Final time
4    constexpr double T = 1.0;
5    // Initial value
6    Eigen::VectorXd y0(1);
7    y0 << 0.1;
8    // TODO: (12-5.e)
9    // START
10   // Function and Jacobian and exact solution
11   auto f = [](const Eigen::VectorXd &y) { return y(0) * (1.0 - y(0)); };
12   auto df = [](const Eigen::VectorXd &y) {
13     Eigen::MatrixXd dfy(1, 1);
14     dfy << 1.0 - 2.0 * y(0);
15     return dfy;
16   };
17   double exactyT = y0(0) / (y0(0) + (1.0 - y0(0)) * std::exp(-T));
18
19   // Container for errors
20   std::vector<double> error(15);
21
22   // Test many step sizes
23   for (unsigned int j = 0; j < 15; ++j) {
24     const unsigned int M = std::pow(2, j + 1);
25     Eigen::VectorXd y = y0;
26     const double h = T / M;
27     for (unsigned int k = 0; k < M; ++k) {
28       y = exponentialEulerStep(y, f, df, h);
29     }
30
31     error[j] = std::abs(y(0) - exactyT);
32     std::cout << std::left << std::setfill(' ') << std::setw(3)
33               << "M = " << std::setw(7) << M << std::setw(8)
```

```
34                << "Error = " << std::setw(13) << error[j];
35      if (j > 0) {
36        std::cout << std::left << std::setfill(' ') << std::setw(10)
37                  << "Approximated order = " << std::log2(error[j - 1] / error[j])
38                  << std::endl;
39      } else
40        std::cout << std::endl;
41    }
42    // END
43  }
```

Get it on GitLab (exponentialintegrator.hpp).

HINT 1 for (12-6.a): Recall the stage form of implicit RK-SSM from [Lecture → Rem. 12.3.3.6]. ⌟

SOLUTION of (12-6.a):

Let us define $d_{i,j} := 0$ for $j \geq i$. Replacing $\mathbf{y}_1$ in the equation for $\mathbf{g}_i$, one obtains:

$$\mathbf{g}_i := (1 - v_i)\mathbf{y}_0 + v_i \left(\mathbf{y}_0 + h \sum_{j=1}^{s} b_j \mathbf{f}(\mathbf{g}_j) \right) + h \sum_{j=1}^{i-1} d_{i,j} \mathbf{f}(\mathbf{g}_j) = \mathbf{y}_0 + h \sum_{j=1}^{s} (v_i b_j + d_{i,j}) \mathbf{f}(\mathbf{g}_j),$$

$$\mathbf{y}_1 := \mathbf{y}_0 + h \sum_{j=1}^{s} b_j \mathbf{f}(\mathbf{g}_j)$$

Set $\mathbf{k}_i := \mathbf{f}(\mathbf{g}_i)$. Then, $\mathbf{k}_i = f(\mathbf{y}_0 + h \sum_{j=1}^{s} \overbrace{(v_i b_j + d_{i,j})}^{=:a_{i,j}} \mathbf{k}_j)$ and

$$\mathbf{y}_1 := \mathbf{y}_0 + h \sum_{j=1}^{s} b_j \mathbf{k}_j \tag{12.6.2}$$

Therefore, the Butcher scheme becomes

$$\begin{array}{c|c} \mathbf{c} & \mathbf{D} + \mathbf{v}\mathbf{b}^\top \\ \hline & \mathbf{b}^\top, \end{array} \quad ,$$

where $(\mathbf{D})_{i,j} = d_{i,j}$, $\mathbf{b} = [b_1, \ldots, b_s]^\top$, $\mathbf{v} = [v_1, \ldots, v_s]^\top$, and $c_i = \sum_{j=1}^{s} a_{i,j}$.

HINT 1 for (12-6.b): The stability function can be obtained by applying the single-step method to the scalar linear ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{R}$, see [Lecture → Ex. 12.1.0.9]. ⌟

SOLUTION of (12-6.b):

We follow the policy of the classical linear model problem analysis and apply the MIRK scheme to the ODE $y' = \lambda y$. Denote by $\mathbf{g} := [g_1, g_2]^\top$ and $z := h\lambda$, then:

$$\begin{aligned} \mathbf{g} &= (\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1 + z\mathbf{D}\mathbf{g} \\ (\mathbf{I} - z\mathbf{D})\mathbf{g} &= (\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1 \,, \end{aligned}$$

where $\mathbf{1} = [1, \ldots, 1]^\top$. Consider

$$y_1 = y_0 + z\mathbf{b}^\top \mathbf{g} = y_0 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}((\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1) \,.$$

Then

$$(1 - z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}\mathbf{v})y_1 = (1 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}(\mathbf{1} - \mathbf{v}))y_0 \,,$$

whence

$$S(z) = \frac{1 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}(\mathbf{1} - \mathbf{v})}{1 - z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}\mathbf{v}} \,. \tag{12.6.3}$$

Alternatively, you may use the formula from [Lecture → Thm. 12.3.4.4] together with the Butcher scheme derived in the previous sub-problem.

SOLUTION of (12-6.c):

In the scalar case we write for the autonomous ODE $\dot{y} = f(y)$. All vectors in (12.6.1) become more real numbers, which yields

$$\mathbf{F}\left(\begin{bmatrix} g_1 \\ g_2 \\ y_1 \end{bmatrix}\right) = \begin{bmatrix} g_1 - (1-v_1)y_0 - v_1 y_1 \\ g_2 - (1-v_2)y_0 - v_2 y_1 - h d_{2,1} f(g_1) \\ y_1 - y_0 - h\sum_{j=1}^{2} b_j f(g_j) \end{bmatrix} . \tag{12.6.4}$$

SOLUTION of (12-6.d):

Well, use (12.6.4) and compute the partial derivatives for all variables according to [Lecture → Eq. (8.3.2.8)]

$$\blacktriangleright \quad \mathrm{D}\,F\left(\begin{bmatrix} g_1 \\ g_2 \\ y_1 \end{bmatrix}\right) = \mathbf{I}_3 - \begin{bmatrix} 0 & 0 & v_1 \\ hd_{2,1}f'(g_1) & 0 & v_2 \\ hb_1 f'(g_1) & hb_2 f'(g_2) & 0 \end{bmatrix}.$$

SOLUTION of (12-6.e):

We just need to recall the general formula for a Newton step for the non-linear system of equations $\mathbf{F}(\mathbf{x}) = 0$:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)}) \,.$$

Of course, the multiplication with the inverse of the Jacobi matrix indicates that a linear system has to be solved. This is done using EIGEN's built-in functions.

C++11-code 12.6.5: Solution code for `Newton2Steps()`

```
template <class Func, class Jac>
Eigen::VectorXd Newton2Steps(Func &&F, Jac &&DF, Eigen::VectorXd z) {
  // TODO: (12-6.e) Perform two steps of Newton's method.
  // START
  // First Newton step
  z = z - DF(z).lu().solve(F(z));
  // Second Newton step
  z = z - DF(z).lu().solve(F(z));
  // END
  return z;
}
```

Get it on GitLab (mirk.hpp).

SOLUTION of (12-6.f):

C++11-code 12.6.7: Solution code for `MIRKStep`

```
template <class Func, class Jac>
double MIRKStep(Func &&f, Jac &&df, double y0, double h) {
  // Coefficients of MIRK
  constexpr double v1 = 1.0;
  constexpr double v2 = 344.0 / 2025.0;
  constexpr double d21 = -164.0 / 2025.0;
  constexpr double b1 = 37.0 / 82.0;
  constexpr double b2 = 45.0 / 82.0;

  double y1 = y0;  // next state; overwrite this
  // TODO: (12-6.f) Perform one step of the MIRK scheme applied to the scalar
  // ODE given by f.
  // START

  // F derived from MIRK scheme (vector valued)
  auto F = [&f, &y0, &h](Eigen::Vector3d z) -> Eigen::Vector3d {
    Eigen::Vector3d ret;
    ret << z(0) - (1 - v1) * y0 - v1 * z(2),
        z(1) - (1 - v2) * y0 - v2 * z(2) - h * d21 * f(z(0)),
        z(2) - y0 - h * (b1 * f(z(0)) + b2 * f(z(1)));
    return ret;
  };
  // Jacobian of F (matrix-valued)
  auto DF = [&df, &h, &v1, &v2](Eigen::Vector3d z) -> Eigen::Matrix3d {
    Eigen::Matrix3d M;
    M << 0, 0, v1, h * d21 * df(z(0)), 0, v2, h * b1 * df(z(0)),
        h * b2 * df(z(1)), 0;
    return Eigen::Matrix3d::Identity() - M;
  };
  // Initial Guess for Newton steps
  Eigen::Vector3d z = {0.0, 0.0, y0};

  // Approximate z, s.t.  F(z) = 0
  z = Newton2Steps(F, DF, z);
  y1 = z(2);
  // END
  return y1;
}
```

Get it on GitLab (mirk.hpp).

SOLUTION of (12-6.g):

C++11-code 12.6.8: Solution code for `MIRKSolve`

```
template <class Func, class Jac>
double MIRKSolve(Func &&f, Jac &&df, double y0, double T, unsigned int M) {
  // Will contain next step
  double y = y0;
  // TODO: (12-6.g) Solve the ODE given by f until time T
  // START

  // Step size
  const double h = T / M;
  // Perform N quidistant steps
  for (unsigned int i = 0; i < M; ++i) {
    y = MIRKStep(f, df, y, h);
  }
  // Return final value at t = T
  // END
  return y;
}
```

Get it on GitLab (mirk.hpp).

SOLUTION of (12-6.h):

C++11-code 12.6.10: Solution code for `cvgMIRK`

```
void cvgMIRK() {
  // TODO: (12-6.h) Tabulate the errors obtained from using the MIRK scheme on
  // the given ODE.
  // START

  // r.h.s
  auto f = [](double y) -> double { return 1 + y * y; };
  // Jacobian of f
  auto df = [](double y) -> double { return 2 * y; };
  // Initial data
  constexpr double y0 = 0.;
  // Final time
  constexpr double T = 1.;
  // Exact solution at t = T = 1
  const double yex = std::tan(T);

  std::cout << "Convergence MIRK for IVP y' = 1+y^2 " << std::endl;
  // Table header
  std::cout << "M"
            << "\t"
            << "yend"
            << "\t"
            << "err" << std::endl;
  for (unsigned int M = 4; M <= 512; M *= 2) {
    // Solve up to time T = 1, using M equidistant steps
    const double yend = MIRKSolve(f, df, y0, T, M);
    // Compute error
    const double err = std::abs(yex - yend);
    // Print table
    std::cout << M << "\t" << yend << "\t" << err << std::endl;
  }
  // END
}
```

Get it on GitLab (mirk.hpp).

M	y_M	error norm
4	1.60432	0.0469136
8	1.56842	0.0110085
16	1.56009	0.00268694
32	1.55807	0.00066483
64	1.55757	0.000165414
128	1.55745	4.12584e-05
256	1.55742	1.0303e-05
512	1.55741	2.5742e-06

The error at final time is roughly reduced by a factor of four when halving the timestep size $h := \frac{1}{M}$.

The error converges like $O(h^2)$ for $h \to 0$. Thus, the experimental rate of convergence of the methods is 2 .

SOLUTION of (12-7.a):

Recall the definition:

Definition [Lecture → Def. 11.4.0.11]. Explicit Runge-Kutta single-step method

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{i-1} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, an **s-stage explicit Runge-Kutta single step method** (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$, $\mathbf{f}: \Omega \to \mathbb{R}^N$, is defined by ($\mathbf{y}_0 \in D$)

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j)\,, \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i\,.$$

The vectors $\mathbf{k}_i \in \mathbb{R}^N$, $i = 1,\ldots,s$, are called **increments**, $h > 0$ is the size of the timestep.

Also recall the shorthand notation for (explicit) Runge-Kutta methods

Butcher scheme ▷

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & 0 & \cdots & 0 \\ c_2 & a_{21} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{s,s-1} \quad 0 \\ \hline & b_1 & \cdots & b_{s-1} \quad b_s \end{array}\,.$$

(Note that the Butcher matrix $\mathfrak{A}$ is a strictly lower triangular $s \times s$-matrix)

[Lecture → Eq. (11.4.0.13)]

Given the right-hand side function $\mathbf{f}$ of the ODe $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ the implementation of an explicit Runge-Kutta single-step method is straightforward, because the increments $\mathbf{k}_i$ can be computed one after the other by plugging linear combinations of previous increments into $\mathbf{f}$.

C++-code 12.7.3: Solution of (12-7.a)

```
2  Eigen::Vector2d PredPrey(Eigen::Vector2d y0, double T, unsigned int M) {
3    const double h = T / M;
4    Eigen::Vector2d y = y0;
5    // TODO: (12-7.a) Solve the predator-prey model using the given
       RK-SSM.
6    // START
7    // Define right-hand-side function for Lotka-Volterra ODE
8    auto f = [](Eigen::Vector2d y) -> Eigen::Vector2d {
9      return {(1 - y(1)) * y(0), (y(0) - 1) * y(1)};
10   };
11   // Main timetepping loop:  uniform stepsize
12   for (unsigned int j = 0; j < M; ++j) {
13     // Compute increments and updates according to [Lecture →
         Def. 11.4.0.11] for the method
14     // described by the Butcher scheme (12.7.1)
15     Eigen::Vector2d k1 = f(y);
16     Eigen::Vector2d k2 = f(y + h * k1);
17     Eigen::Vector2d k3 = f(y + (h / 4.) * k1 + (h / 4.) * k2);
18     y = y + (h / 6.) * k1 + (h / 6.) * k2 + (2. * h / 3.) * k3;
```

```
19      }
20      // END
21      return y;
22  }
```

Get it on GitLab (stabrk3.hpp).

HINT 1 for (12-7.b): $\mathbf{y}_M$ for $M = 128$ is $[6.47563e-27, 40.7386]^\top$.

SOLUTION of (12-7.b):

M	error	rate
4	inf	
8	inf	-nan
16	inf	-nan
32	2.2565	inf
64	0.148092	3.92952
128	0.0268461	2.46371
256	0.00421868	2.66985
512	0.000596288	2.82271
1024	7.93781e-05	2.9092
2048	1.02263e-05	2.95645
4096	1.28099e-06	2.99695
8192	1.43426e-07	3.15889

The convergence rate is estimates by a two-mesh comparison

$$\texttt{rate} \approx \log_2 \frac{\text{error}(i-1)}{\text{error}(i)} .$$

We also observe that a doubling of the number of steps leads to a decrease of the error at final time by roughly a factor of 8.
Thus the scheme seems to be of order 3 .

In case of large stepsize h the scheme is *unstable*. At least 32 steps are needed for the solution to be stable.

C++-code 12.7.4: Solution of (12-7.b)

```
2  void SimulatePredPrey() {
3    // TODO: (12-7.b) Simulate the predator-prey model and perform a
         convergence
4    // study.
5    // START
6    // Parameters
7    constexpr double T = 1.0;
8    Eigen::Vector2d y0(100.0, 1.0);
9
10   // (Approximate) reference solution
11   Eigen::Vector2d y_ref = PredPrey(y0, T, std::pow(2, 14));
12
13   Eigen::ArrayXd error(12);
14   Eigen::ArrayXd M(12);
15   // Studying the error for geometrically increasing numbers of
         equidistant
16   // timesteps is the most appropriate approach to empirically exploring
17   // algebraic convergence.
18   M << 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192;
19
20   // Compute errors
21   for (unsigned int i = 0; i < M.size(); ++i) {
22     Eigen::Vector2d y = PredPrey(y0, T, M(i));
23     error(i) = (y - y_ref).norm();
24   }
25   // Print error table
26   std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::setw(15)
27             << "rate" << std::endl;
28   // Formatted output in C++
29   for (unsigned int i = 0; i < M.size(); ++i) {
30     std::cout << std::setw(15) << M(i) << std::setw(15) << error(i);
31     if (i > 0) {
32       std::cout << std::setw(15) << std::log2(error(i - 1) / error(i));
33     }
34     std::cout << std::endl;
35   }
36   // END
```

```
37 }
```

Get it on GitLab (stabrk3.hpp).

SOLUTION of (12-7.c):

We obtain the stability function $S(z)$ by applying our method to the model problem $\dot{y}(t) = \lambda y$, $y(0) = y_0$ and by writing the result in the form $y_1 = S(z)y_0$, where $z := h\lambda$. For the increment, we obtain

$$\begin{aligned} k_1 &= \lambda y_0, \\ k_2 &= \lambda(y_0 + hk_1) = \lambda y_0(1+z), \\ k_3 &= \lambda(y_0 + \tfrac{h}{4}(k_1+k_2)) = \lambda y_0(1+\tfrac{1}{2}z+\tfrac{1}{4}z^2)\,, \end{aligned}$$

and for the update

$$\begin{aligned} y_1 &= y_0 + \frac{h\lambda y_0}{6}\left(1+1+z+4\left(1+\frac{1}{2}z+\frac{1}{4}z^2\right)\right) \\ &= y_0 \underbrace{\left(1+z+\tfrac{1}{2}z^2+\tfrac{1}{6}z^3\right)}_{=:S(z)}\,. \end{aligned}$$

In agreement with [Lecture → Cor. 12.1.0.20] the stability function of the 3-stage explicit RK-SSM is a *polynomial* of degree 3.

Note that the stability function agrees with the Taylor series of $z \mapsto \exp(z)$ in the first four terms, which implies

$$|S(z) - \exp(z)| = O(z^4) \quad \text{for} \quad z \to 0\,.$$

According to [Lecture → Lemma 12.1.0.23] this is necessary for an explicit Runge-Kutta single-step method of order 3.

SOLUTION of (12-8.a):

Since $\mathbf{grad}\, V$ is continuous, $t \mapsto \mathbf{y}(t)$ is a C^1-function. Therefore, also $t \mapsto V(\mathbf{y}(t))$ is continuously differentiable. This means that $\varphi(t) := V(\mathbf{y}(t))$ is non-increasing if and only if its *derivative is* ≤ 0. By the chain rule we compute

$$\dot{\varphi}(t) = \mathbf{grad}\, V(\mathbf{y}(t))^\top \dot{\mathbf{y}}(t) \overset{(12.8.1)}{=} -\|\mathbf{grad}\, V(\mathbf{y}(t))\|^2 \leq 0\,.$$

SOLUTION of (12-8.j):

Doubling the number of timesteps roughly reduces the error by a factor of 16, which hints that the SDIRK RK-SSM is of order 4 , because the order of a Runge-Kutta single-step method predicts its algebraic rate of convergence with respect to $h \sim N^{-1} \to 0$.

SOLUTION of (12-8.b):

For $N = 1$ the gradient boils down to the standard derivative. Thus we find

$$V(y) = \tfrac{1}{2}\lambda y^2 \quad \Rightarrow \quad \frac{dV}{dy}(y) = \lambda y \; .$$

SOLUTION of (12-8.c):

Note that thanks to the lower triangular shape of the Butcher matrix $\mathfrak{A}$ the increments can be computed sequentially $\mathbf{k}_1 \rightarrow \mathbf{k}_2 \rightarrow \cdots \rightarrow \mathbf{k}_5$.

$$
\begin{aligned}
\mathbf{k}_1 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{1}{4}\mathbf{k}_1\big)\Big) &= \mathbf{0}\,, \\
\mathbf{k}_2 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{1}{2}\mathbf{k}_1 + \frac{1}{4}\mathbf{k}_2\big)\Big) &= \mathbf{0}\,, \\
\mathbf{k}_3 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{17}{50}\mathbf{k}_1 - \frac{1}{25}\mathbf{k}_2 + \frac{1}{4}\mathbf{k}_3\big)\Big) &= \mathbf{0}\,, \\
\mathbf{k}_4 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{371}{1360}\mathbf{k}_1 - \frac{137}{2720}\mathbf{k}_2 + \frac{15}{544}\mathbf{k}_3 + \frac{1}{4}\mathbf{k}_4\big)\Big) &= \mathbf{0}\,, \\
\mathbf{k}_5 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{25}{24}\mathbf{k}_1 - \frac{49}{48}\mathbf{k}_2 + \frac{125}{16}\mathbf{k}_3 - \frac{85}{12}\mathbf{k}_4 + \frac{1}{4}\mathbf{k}_5\big)\Big) &= \mathbf{0}\,.
\end{aligned}
$$

We tackle these equations from top to bottom. The highlighted increments are the unknowns in the equations, those printed in blue are known already.

Writing $\mathfrak{A} = \big[a_{i,j}\big]_{i,j=1}^{5}$ for the lower triangular Butcher matrix of the SDIRK method and using $a_{j,j} = \frac{1}{4}$, a more compact way to describe the increment equations is

$$
\mathbf{k}_j - \mathbf{f}(\mathbf{y}_0 + \tfrac{1}{4}h\mathbf{k}_j + h\sum_{\ell=1}^{j-1} a_{j,\ell}\mathbf{k}_\ell) = \mathbf{0}\,, \quad j = 1,2,3,4,5\,.
$$

SOLUTION of (12-8.d):

The stages defined in [Lecture → Eq. (12.3.3.7)] can be computed sequentially $\mathbf{g}_1 \to \mathbf{g}_2 \to \cdots \to \mathbf{g}_5$:

$$\begin{aligned}
\mathbf{g}_1 - h\Big(\frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_2 - h\Big(\frac{1}{2}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_3 - h\Big(\frac{17}{50}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) - \frac{1}{25}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_3)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_4 - h\Big(\frac{371}{1360}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) - \frac{137}{2720}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2) + \frac{15}{544}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_3) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_4)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_5 - h\Big(\frac{25}{24}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) - \frac{49}{48}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2) + \frac{125}{16}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_3) - \frac{85}{12}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_4) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_5)\Big) &= \mathbf{0}\,.
\end{aligned}$$

The generic form of a $N \times N$ non-linear system of equations to obtain another stage $\mathbf{g}$ is

$$\mathbf{g}_j - \tfrac{1}{4}h\mathbf{f}(\mathbf{y}_0+\mathbf{g}_j) = h\sum_{\ell=1}^{j-1} a_{j,\ell}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_\ell)\,,\quad j=1,2,3,4,5\,, \tag{12.8.3}$$

Here $a_{j,\ell}$ denotes entries of the Butcher matrix $\mathfrak{A} \in \mathbb{R}^{5,5}$.

HINT 1 for (12-8.e): Consult [Lecture → Section 8.5.3] to learn about ways to decide when to terminate a Newton iteration. ⌟

SOLUTION of (12-8.e):

The Newton iteration for the generic form (12.8.3) of the stage equation reads

$$\mathbf{g}^{(k+1)} = \mathbf{g}^{(k)} - \left(\mathbf{I}_N - \tfrac{1}{4}h\,\mathrm{D}\,f(\mathbf{y}_0 + \mathbf{g}^{(k)})\right)^{-1}\left(\mathbf{g}^{(k)} - \tfrac{1}{4}h\mathbf{f}(\mathbf{y}_0 + \mathbf{g}^{(k)}) - \mathbf{b}\right). \tag{12.8.4}$$

where $\mathbf{b}$ is a known vector depending on previous stages.

It is smart to write an auxiliary function

```
template <typename Functor, typename Jacobian>
Eigen::VectorXd solveGenStageEquation(
Functor &&f, Jacobian &&df,
const Eigen::VectorXd &y,
const Eigen::VectorXd &b, double h,
double rtol = 1E-6, double atol = 1E-8);
```

that solves (12.8.3) by means of the Newton iteration (12.8.4).

C++ code 12.8.5: Sub-problem (12-8.e): Auxiliary function `solveGenStageEquation()`

```
template <typename Functor, typename Jacobian>
Eigen::VectorXd solveGenStageEquation(Functor &&f, Jacobian &&J,
                                      const Eigen::VectorXd &y,
                                      const Eigen::VectorXd &b, double h,
                                      double rtol = 1E-6, double atol = 1E-8) {
  Eigen::VectorXd g = Eigen::VectorXd::Zero(
      y.size());  // initial guess g=0 for Newton iteration.

  // TODO: (12-8.e; optional) Solve one stage equation using the Newton
  // iteration.
  // START

  // Need to solve the equation lhs(g) = g - h*f(y+g)/4 - b = 0.
  auto lhs = [f, y, b, h](const Eigen::VectorXd &g) {
    // lhs(g) = g - h*f(y+g)/4 - b
    Eigen::VectorXd val = g - 0.25 * h * f(y + g) - b;
    return val;
  };
  auto Jlhs = [J, y, h](const Eigen::VectorXd &g) {
    // lhs(g) = g - h*f(y+g)/4 - b, so the Jacobian is
    // Jlhs(g) = Id - h*Jf(y+g)/4
    int dim = y.size();
    Eigen::MatrixXd Jval =
        Eigen::MatrixXd::Identity(dim, dim) - 0.25 * h * J(y + g);
    return Jval;
  };

  // Perform Newton iterations:
  Eigen::VectorXd delta =
      -Jlhs(g).lu().solve(lhs(g));  // Newton correction term.
  int iter = 0,
      maxiter = 100;  // If correction based termination does not work.
  while (delta.norm() > atol && delta.norm() > rtol * g.norm() &&
         iter < maxiter) {
    g = g + delta;
```

```
    delta = -Jlhs(g).lu().solve(lhs(g));
    iter++;
  }
  g = g + delta;  // Perform the final step
  // END

  return g;
}
```

Get it on GitLab (gradientflow.hpp).

C++ code 12.8.6: Sub-problem (12-8.e): Solution of stage equations for SDIRK method

```
template <typename Functor, typename Jacobian>
std::array<Eigen::VectorXd, 5> computeStages(Functor &&f, Jacobian &&J,
                                             const Eigen::VectorXd &y, double h,
                                             double rtol = 1E-6,
                                             double atol = 1E-8) {
  std::array<Eigen::VectorXd, 5> G;  // array of stages
  const unsigned int d = y.size();
  Eigen::MatrixXd Coeffs = ButcherMatrix();
  // TODO: (12-8.e) Solve for the stages of the SDIRK method using the
  // (undamped) Newton method.
  // Hint: You may implement solveGenStageEquation as
  // an auxiliary function.
  // START
  for (unsigned int i = 0; i < 5; i++) {
    // Calculate coefficients from previous stages:
    Eigen::VectorXd b = Eigen::VectorXd::Zero(d);
    for (unsigned int j = 0; j < i; j++) {
      b += Coeffs(i, j) * f(y + G.at(j));
    }
    b *= h;
    // Compute stage i and store in G.at(i):
    G.at(i) = solveGenStageEquation(f, J, y, b, h, rtol, atol);
  }
  // END
  return G;
}
```

Get it on GitLab (gradientflow.hpp).

SOLUTION of (12-8.f):

After computing the stages $\mathbf{g}_i$ or the increments $\mathbf{k}_i$, $i = 1, \ldots, 5$, by [Lecture → Def. 12.3.3.1]/[Lecture → Eq. (12.3.3.8)] we can evaluate

$$\Psi^h \mathbf{y}_0 = \mathbf{y}_0 + h \sum_{i=1}^{5} b_i \mathbf{k}_i \,,$$

$$\Psi^h \mathbf{y}_0 = \mathbf{y}_0 + h \sum_{i=1}^{5} b_i \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_i) \,,$$

where

$$[b_1, b_2, b_3, b_4, b_5] = \left[\frac{25}{24}, -\frac{49}{48}, \frac{125}{16}, -\frac{85}{12}, \frac{1}{4}\right] .$$

C++ code 12.8.7: Sub-problem (12-8.f): Implementation of SDIRK discrete evolution operator

```
template <typename Functor, typename Jacobian>
Eigen::VectorXd discEvolSDIRK(Functor &&f, Jacobian &&J,
                              const Eigen::VectorXd &y, double h,
                              double rtol = 1E-6, double atol = 1E-8) {
  // The b weights are in the last row of Coeffs.
  Eigen::MatrixXd Coeffs = ButcherMatrix();
  Eigen::VectorXd Psi;
  // TODO: (12-8.f) Apply the discrete evolution operator to the ODE described
  // by f.
  // START
  const unsigned int nstages = Coeffs.cols();
  Psi = y;

  // Using computeStages()
  std::array<Eigen::VectorXd, 5> G = computeStages(f, J, y, h, rtol, atol);
  for (int i = 0; i < nstages; i++)
    Psi += h * Coeffs(nstages, i) * f(y + G.at(i));
  // END
  return Psi;
}
```

Get it on GitLab (gradientflow.hpp).

SOLUTION of (12-8.g):

Using the chain rule and

$$\mathrm{D}_{\mathbf{x}}\{\mathbf{x} \mapsto \|\mathbf{x}\|^2\}(\mathbf{y}) = 2\mathbf{y} \,,$$

we find

$$-\,\mathbf{grad}\, V(\mathbf{y}) = -2\cos(\|\mathbf{y}\|^2)\mathbf{y} - \lambda 2(\mathbf{d}^\top \mathbf{y})\mathbf{d} \,,$$

which agrees with the right-hand side function of the ODE. So we arrive at

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := -2\cos(\|\mathbf{y}\|^2)\mathbf{y} - \lambda 2(\mathbf{d}^\top \mathbf{y})\mathbf{d} \,. \tag{12.8.9}$$

HINT 1 for (12-8.h): Recall the stiffness criteria [Lecture → Eq. (12.2.0.13)] and [Lecture → Eq. (12.2.0.1

SOLUTION of (12-8.h):

Taking the cue from the criteria [Lecture → Eq. (12.2.0.13)] and [Lecture → Eq. (12.2.0.14)] we have to inverstigate the Jacobian $\mathrm{D}\,\mathbf{f}(\mathbf{0})$. Differentiating $\mathbf{f}$ from (12.8.9) yields

$$\mathrm{D}\,\mathbf{f}(\mathbf{y}) = 4\sin(\|\mathbf{y}\|^2)\mathbf{y}\mathbf{y}^\top - 2\cos(\|\mathbf{y}\|^2)\mathbf{I}_N - 2\lambda\mathbf{d}\mathbf{d}^\top \in \mathbb{R}^{N,N}\,. \tag{12.8.10}$$

Here we have used the product rule for the first term and the fact that the derivative of linear expressions are the expressions themselves. In particular, we get

$$\mathrm{D}\,\mathbf{f}(\mathbf{0}) = -2\mathbf{I}_N - 2\lambda\mathbf{d}\mathbf{d}^\top\,.$$

This matrix is a rank-1 perturbation of a scaled identity matrix and has the eigenvalues (spectrum)

$$\sigma(\mathrm{D}\,\mathbf{f}(\mathbf{0})) = \{-2, -2-2\lambda\}\,.$$

Thus, if $\lambda \gg 1$ we face a stiff problem for states close to zero.

HINT 1 for (12-8.i): Create suitable lambda functions for $\mathbf{f}$ and $D\,\mathbf{f}$. ⌟

SOLUTION of (12-8.i):

C++ code 12.8.11: Sub-problem (12-8.i): SDIRK solver for gradient flow problem

```
std::vector<Eigen::VectorXd> solveGradientFlow(const Eigen::VectorXd &d,
                                               double lambda,
                                               const Eigen::VectorXd &y,
                                               double T, unsigned int M) {
  std::vector<Eigen::VectorXd> Y(M + 1);
  // TODO: (12-8.i) Solve the Gradient flow ODE with the vector potential given
  // in 12.8.8
  // START
  // Define the right hand side of the ODE y' = f(y), and the
  // Jacobian of f.
  auto f = [d, lambda](const Eigen::VectorXd &yt) {
    Eigen::VectorXd val =
        -2. * std::cos(yt.squaredNorm()) * yt - 2. * lambda * d.dot(yt) * d;
    return val;
  };
  auto J = [d, lambda](const Eigen::VectorXd &yt) {
    const unsigned int dim = yt.size();
    Eigen::MatrixXd term1 =
        4. * std::sin(yt.squaredNorm()) * yt * yt.transpose();
    Eigen::MatrixXd term2 =
        -2. * std::cos(yt.squaredNorm()) * Eigen::MatrixXd::Identity(dim, dim);
    Eigen::MatrixXd term3 = -2. * lambda * d * d.transpose();
    Eigen::MatrixXd Jval = term1 + term2 + term3;
    return Jval;
  };

  // Split the interval [0,T] into M intervals of size h.
  const double h = T / M;
  Eigen::VectorXd yt = y;
  Y.at(0) = y;
  // Evolve up to time T:
  for (unsigned int i = 1; i <= M; i++) {
    yt = discEvolSDIRK(f, J, yt, h);
    Y.at(i) = yt;
  }
  // END
  return Y;
}
```

Get it on GitLab (gradientflow.hpp).

HINT 1 for (1-1.a): First learn how EIGEN overloads the «-operator for the initialization of matrices, see [Lecture → § 1.2.1.3]. ⌟

SOLUTION of (1-1.a):

From code `ArrowMatrix.hpp` (Get it on GitLab (ArrowMatrix.hpp).) we deduce that the argument vectors are copied into the diagonal, the bottom row and rightmost column of the matrix $\mathbf{A}$, which yields

$$\mathbf{A} = \begin{bmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{bmatrix} \tag{1.1.2}$$

From this the pattern is clear.

SOLUTION of (1-1.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.

SOLUTION of (1-1.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 $\mathbf{a}_- := (a_{(1)}, \ldots, a_{(n-1)}, 0)$. We can rewrite $\mathbf{A}$ as follows:

$$\mathbf{A} = \begin{bmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \ldots & a_{n-1} & d_n \end{bmatrix} \tag{1.1.3}$$

$$= \operatorname{diag}(\mathbf{d}) + [\mathbf{0}, \ldots, \mathbf{0}, \mathbf{a}_-] + [\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{a}_-]^\top =: \mathbf{D} + \mathbf{V} + \mathbf{H}. \tag{1.1.4}$$

and $\mathbf{Ax} = \mathbf{Dx} + \mathbf{Vx} + \mathbf{Hx}$. Each of these multiplications can obviously be done in $O(n)$ operations, see Code 1.1.5.

C++11-code 1.1.5: Solution of (1-1.c)

```
2  void efficient_arrow_matrix_2_times_x(const Eigen::VectorXd &d,
3                                        const Eigen::VectorXd &a,
4                                        const Eigen::VectorXd &x,
5                                        Eigen::VectorXd &y) {
6    assert(d.size() == a.size() && a.size() == x.size() &&
7           "Vector size must be the same!");
8    const unsigned int n = d.size();
9
10   // TODO: (1-1.c) Implement an efficient version of
11   // arrow\_matrix\_2\_times\_x. Hint: Notice that performing two matrix vector
12   // multiplications is less expensive than a matrix-matrix multiplication.
13   // Therefore, as first step, we need a way to efficiently compute A*x
14
15   // START
16
17   // Notice that we can compute (A*A)*x more efficiently using
18   // A*(A*x). This is, in fact, performing two matrix vector
19   // multiplications instead of a more expensive matrix-matrix multiplication.
20   // Therefore, as a first step, we need a way to efficiently
21   // compute A*x
22
23   // This function computes A*x. You can use it by calling A\_times\_x(x).
24   // This is the syntax for lambda functions: notice the extra
25   // [variables] code. Each variable written within [] brackets
26   // will be captured (i.e. seen) inside the lambda function.
27   // Without \&, a copy of the variable will be performed.
28   // Notice that A = D + H + V, s.t. A*x = D*x + H*x + V*x
29   // D*x can be rewritten as d*x componentwise
30   // H*x is zero, except at the last component
31   // V*x is only affected by the last component of x
32   auto A_times_x = [&a, &d, n](const Eigen::VectorXd &x) {
33     // This takes care of the diagonal (D*x)
34     // Notice: we use d.array() to tell Eigen to treat
```

```
35      // a vector as an array. As a result: each operation
36      // is performed componentwise.
37      Eigen::VectorXd Ax = (d.array() * x.array()).matrix();
38
39      // H*x only affects the last component of A*x
40      // This is a dot product between a and x with the last
41      // component removed
42      Ax(n - 1) += a.head(n - 1).dot(x.head(n - 1));
43
44      // V*x is equal to the vector
45      // (a(0)*x(n-1), ..., a(n-2)*x(n-1), 0)
46      Ax.head(n - 1) += x(n - 1) * a.head(n - 1);
47
48      return Ax;
49    };
50
51    // <=> y = A*(A*x)
52    y = A_times_x(A_times_x(x));
53
54    // END
55  }
```

Get it on GitLab (ArrowMatrix.hpp).

Note the use of lambda functions [Lecture → Section 0.3.3] in order to achieve a streamlined implementation. Also note the use of the method `head` to access the top components of a vector.

SOLUTION of (1-1.d):

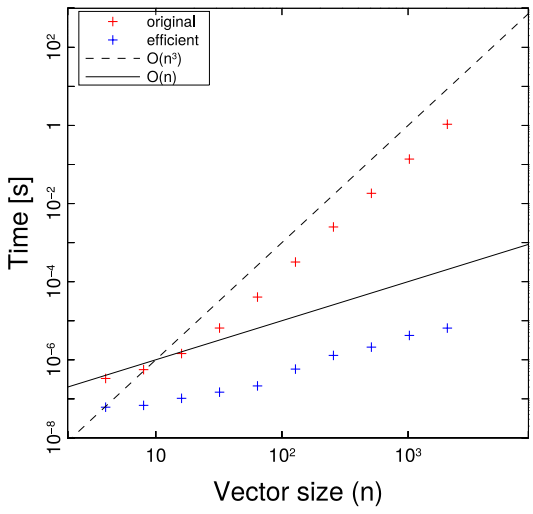
The efficient implementation only needs two vector-vector element-wise multiplications, a dot-product, and two vector-scalar multiplication. Each of them has complexity $O(n)$. Therefore the complexity is $O(n)$.

SOLUTION of (1-1.e):

The standard matrix multiplication has runtime growing with $O(n^3)$. The complexity of the more efficient implementation is $O(n)$. See Code 1.1.10 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 implementations in `ArrowMatrix.hpp` ▷

(The file `ArrowMatrix.hpp` also contains an implementation using `chrono`.)



C++11-code 1.1.10: Computation of timings of arrow matrix multiplication

```
2  void runtime_arrow_matrix() {
3    // Memory allocation for plot
4    std::vector<double> vec_size;
5    std::vector<double> elap_time, elap_time_eff;
6
7    // header for result print out
8    std::cout << std::setw(8) << "n" << std::setw(15) << "original"
9              << std::setw(15) << "efficient" << std::endl;
10
11   for (unsigned int n = 32; n <= 1024; n <<= 1) {
12     // save vector size (for plot)
13     vec_size.push_back(n);
14
15     // Number of repetitions
16     constexpr unsigned int repeats = 10;
17
18     Timer timer, timer_eff;
19     // Repeat test many times
20     for (unsigned int r = 0; r < repeats; ++r) {
21       // Create test input using random vectors
22       Eigen::VectorXd a = Eigen::VectorXd::Random(n);
23       Eigen::VectorXd d = Eigen::VectorXd::Random(n);
24       Eigen::VectorXd x = Eigen::VectorXd::Random(n);
25       Eigen::VectorXd y;
26
27       // Compute times for original implementation
28       timer.start();
29       arrow_matrix_2_times_x(d, a, x, y);
30       timer.stop();
31
32       // TODO: (1-1.e) Compute times for efficient implementation
33       // START
```

```
      timer_eff.start();
      efficient_arrow_matrix_2_times_x(d, a, x, y);
      timer_eff.stop();
      // END
    }

    // Print results (for grading): inefficient
    std::cout << std::setw(8) << n << std::scientific << std::setprecision(3)
              << std::setw(15) << timer.min() << std::setw(15)
              << timer_eff.min() << std::endl;

    // time needed
    elap_time.push_back(timer.min());
    elap_time_eff.push_back(timer_eff.min());
  }

  /* DO NOT CHANGE */
  // create plot
  plot(vec_size, elap_time, elap_time_eff, "./cx_out/comparison.eps");
}
```

Get it on GitLab (ArrowMatrix.hpp).

The file `ArrowMatrix.hpp` also contains an implementation using `chrono`.

SOLUTION of (1-10.a):

The formula use in Code 1.10.1 is vulnerable to **cancellation** for small values of the argument x, that is, for $x \approx 0$. In this case we have

$$x \approx 0 \quad \Rightarrow \quad e^x \approx 1 \quad \Rightarrow \quad e^x \approx e^{-x} \,,$$

and a subtraction of two numerical results of about the same size will occur, which leads to a huge amplification of relative errors introduced by rounding in earlier computations.

C++-code 1.10.2: Tabulating the relative errors

```
2  void sinhError() {
3    // in this solution, we also compare against an implementation based
       on the
4    // Taylor expansion of sinh in sinh_stable.hpp
5    std::cout << "--> Relative error of various implementations:" << std::endl;
6    std::cout << std::setw(10) << "k" << std::setw(15) << "own" << std::setw(15)
7              << "std" << std::setw(15) << "taylor" << std::setw(15) << "err"
8              << std::setw(15) << "err (taylor)" << std::endl;
9
10   // TODO: (1-10.a) Tabulate the relative error norms for the unstable
       sinh
11   // function.  Compare against the implementation of the standard
       library.
12   // START
13   for (int k = 1; k <= 10; ++k) {
14     // Value x
15     const double x = std::pow<double>(10., -k);
16
17     // Our own function
18     const double mySinh = sinh_unstable(x);
19     // The "standard" sinh
20     const double stdSinh = std::sinh(x);
21     // The stable sinh (look at the file "sinh_stable.hpp" if
22     // interested in advanced C++)
23     const double taylorSinh = taylor_sinh<3>(x);
24
25     // Relative error
26     const double err = std::abs(mySinh - stdSinh) / std::abs(stdSinh);
27     const double err_t = std::abs(taylorSinh - stdSinh) / std::abs(stdSinh);
28
29     // Print error
30     std::cout << std::setw(10) << k << std::setw(15) << mySinh << std::setw(15)
31               << stdSinh << std::setw(15) << taylorSinh << std::setw(15) << err
32               << std::setw(15) << err_t << std::endl;
33   }
34
35   // We compute and print the error bound computed using the
36   // remainder of the Taylor expansion
37   constexpr double x = 1e-3;
38   std::cout << "--> Relative error bound for x = " << x << ":" << std::endl;
39   std::cout << std::setw(10) << "n" << std::setw(15) << "error bound"
40             << std::endl;
41   std::cout << std::setw(10) << "1" << std::setw(15) << error_bound<1>(x)
42             << std::endl;
43   std::cout << std::setw(10) << "2" << std::setw(15) << error_bound<2>(x)
44             << std::endl;
```

```
45    std::cout << std::setw(10) << "3" << std::setw(15) << error_bound<3>(x)
46              << std::endl;
47    std::cout << std::setw(10) << "4" << std::setw(15) << error_bound<4>(x)
48              << std::endl;
49    // END
50  }
```

Get it on GitLab (sinh.hpp).

k	ϵ_{rel}
1	6.92734e-16
2	8.15306e-15
3	6.17995e-14
4	2.23075e-13
5	1.01156e-11
6	2.69221e-11
7	5.26358e-10
8	1.16286e-08
9	8.27404e-08
10	8.27404e-08

The measured error norms as given by the code.

SOLUTION of (1-10.b):

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

$$e^x = \sum_{k=0}^{m} \frac{x^k}{k!} + \frac{e^{\xi_x} x^{m+1}}{(m+1)!}. \tag{1.10.3}$$

SOLUTION of (1-10.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

$$f'(x) = e^x + e^{-x} - 2 = (e^{x/2} - e^{-x/2})^2 \geq 0 \quad \forall x \in \mathbb{R}\,.$$

SOLUTION of (1-10.d):

The idea is to use the Taylor expansion given in (1.10.3). 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 (1.10.4) and using the inequalities $e^{\xi_x} \le e^x$ and $e^{\xi_{-x}} \le e^x$, the new relative error can be bounded by

$$\epsilon'_{n,rel} := \frac{|y_n - \sinh(x)|}{\sinh(x)} \le \frac{e^x x^{2n}}{(2n+1)!}.$$

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, $y_3(x)$ already gives a relative error below 10^{-15}, as required. This is the formula we can use for the stable approximation of `sinh`.

SOLUTION of (1-11.a):

By direct computation using the binomial formula $a^2 - b^2 = (a+b)(a-b)$,

$$\begin{aligned}(x+\imath y)^2 &= x^2 - y^2 + 2\imath xy \\ &= \tfrac{1}{2}(\sqrt{u^2+v^2}+u) - \tfrac{1}{2}(\sqrt{u^2+v^2}-u) + \imath\sqrt{(\sqrt{u^2+v^2}+u)((\sqrt{u^2+v^2}-u))} \\ &= u + \imath\sqrt{(u^2+v^2)-u^2} = u+\imath v = w\,.\end{aligned}$$

SOLUTION of (1-11.b):

(I) If $|u| \gg |v|$, and $u < 0$, then $\sqrt{u^2 + v^2} \approx -u$ and cancellation will affect the term $\sqrt{u^2 + v^2} + u$ in (1.11.1).

(II) If $|u| \gg |v|$, and $u > 0$, then $\sqrt{u^2 + v^2} \approx u$ and cancellation will affect the term $\sqrt{u^2 + v^2} - u$ in (1.11.2).

Thus, if $|u| \gg |v|$, either the imaginary part or real part of the computed $\sqrt{w}$ might be affected by a huge relative error as a consequence of cancellation, which amplifies the relative errors introduced by rounding into the computation of $u^2 + v^2$.

SOLUTION of (1-11.c):

An simple calculation shows that $xy = v/2$, see Sub-problem (1-11.a).

SOLUTION of (1-11.d):

In Sub-problem (1-11.b) we observed that depending on the sign of u either (1.11.1) or (1.11.2) were affected by cancellation. The idea is to first compute either x or y, whichever is possible in a stable fashion (If $u > 0$ compute y, otherwise compute x.), and then us the formula $2xy = v$ to obtain the other variable.

C++-code 1.11.3: Stable implementation of the C++ function `myroot()`

```
2  std::complex<double> myroot(std::complex<double> w) {
3    double x, y;
4    double u = w.real();
5    double v = w.imag();
6
7    // TODO: (1-11.c) Compute the square root of w avoiding cancellation,
8    // use only real arithmetic.
9
10   // START
11   // w is actually a real number, we are done
12   if (v == 0) return std::sqrt(u);
13
14   if (u > 0) {
15     // Compute the real part, since stable
16     x = std::sqrt((std::sqrt(u * u + v * v) + u) / 2.);
17     // Use product to compute imaginary part
18     y = v / (2 * x);
19   } else {
20     // Compute the imaginary part, since stable
21     y = std::sqrt((std::sqrt(u * u + v * v) - u) / 2.);
22     // Use product to compute real part
23     x = v / (2 * y);
24   }
25   // END
26
27   return std::complex<double>(x, y);
28 }
```

Get it on GitLab (complexroot.hpp).

SOLUTION of (1-12.a):

The matrices $\mathbf{U_A} + \mathbf{D_A}$ and $\mathbf{U_A} + \mathbf{D_A}$ must be invertible, which is equivalent to $\mathbf{D_A}$ being invertible. This is equivalent to the matrix $\mathbf{A}$ having no zeros on the diagonal.

SOLUTION of (1-12.b):

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

$$\mathbf{x} = (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{b} - (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) \tag{1.12.3}$$

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

$$(\mathbf{U_A} + \mathbf{D_A})\mathbf{x} = \mathbf{b} - \mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) \tag{1.12.4}$$

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

$$(\mathbf{L_A} + \mathbf{D_A})\mathbf{y} = \mathbf{b} - \mathbf{U_A}\mathbf{x} \tag{1.12.5}$$
$$(\mathbf{U_A} + \mathbf{D_A})\mathbf{x} = \mathbf{b} - \mathbf{L_A}\mathbf{y} \tag{1.12.6}$$

Subtract the first equation to the second:

$$(\mathbf{L_A} + \mathbf{D_A})\mathbf{y} + \mathbf{U_A}\mathbf{x} - (\mathbf{U_A} + \mathbf{D_A})\mathbf{x} - \mathbf{L_A}\mathbf{y} = \mathbf{0} \tag{1.12.7}$$
$$\Leftrightarrow \quad \mathbf{D_A}(\mathbf{y} - \mathbf{x}) = \mathbf{0} \tag{1.12.8}$$
$$\Leftrightarrow \quad \mathbf{y} = \mathbf{x} \tag{1.12.9}$$

Using this last identity in (1.12.5) and get $(\mathbf{U_A} + \mathbf{D_A} + \mathbf{L_A})\mathbf{x} = \mathbf{b}$. □

SOLUTION of (1-12.c):

C++11-code 1.12.10: Code

```
2  void GSIt(const Eigen::MatrixXd& A, const Eigen::VectorXd& b,
3            Eigen::VectorXd& x, double rtol) {
4    // TODO: (1-12.c) Implement the Gauss-Seidel iteration.
5    // START
6    const auto U =
7        Eigen::TriangularView<const Eigen::MatrixXd, Eigen::StrictlyUpper>(A);
8    const auto L =
9        Eigen::TriangularView<const Eigen::MatrixXd, Eigen::StrictlyLower>(A);
10
11   const auto UpD =
12       Eigen::TriangularView<const Eigen::MatrixXd, Eigen::Upper>(A);
13   const auto LpD =
14       Eigen::TriangularView<const Eigen::MatrixXd, Eigen::Lower>(A);
15
16   // A temporary vector to store result of iteration
17   Eigen::VectorXd temp(x.size());
18
19   // We'll use pointer magic too
20   Eigen::VectorXd* xold = &x;
21   Eigen::VectorXd* xnew = &temp;
22
23   double err;
24
25   do {
26     // Compute next iteration step
27     *xnew = UpD.solve(b) - UpD.solve(L * LpD.solve(b - U * (*xold)));
28
29     // Absolute error
30     err = (*xold - *xnew).norm();
31
32     // Swap role of previous/next iteration
33     std::swap(xold, xnew);
34   } while (err > rtol * (*xnew).norm());
35
36   x = *xnew;
37   // END
38 }
```

Get it on GitLab (gsit.hpp).

SOLUTION of (1-12.d):

C++11-code 1.12.12: Code

```
double testGSIt(unsigned int n) {
  double residual_norm = 10.;
  // TODO: (1-12.d) Implement the test from the problem sheet.  Return
     the
  // residual norm.
  // START
  Eigen::MatrixXd A = Eigen::MatrixXd::Zero(n, n);
  for (unsigned int i = 0; i < n; ++i) {
    if (i > 0) A(i, i - 1) = 2;
    A(i, i) = 3;
    if (i < n - 1) A(i, i + 1) = 1;
  }
  Eigen::VectorXd b = Eigen::VectorXd::Constant(n, 1);

  Eigen::VectorXd x = b;
  GSIt(A, b, x, 1e-8);
  residual_norm = (A * x - b).norm();
  // END
  return residual_norm;
}
```

Get it on GitLab (gsit.hpp).

SOLUTION of (1-12.e):

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

HINT 1 for (1-13.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,dou`

SOLUTION of (1-13.a):

Knowing the rules of matrix×vector multiplication the loop-based implementation is straightforward.

C++11-code 1.13.3: Code

```
2  template <class Vector>
3  void xmatmult(const Vector& a, const Vector& y, Vector& x) {
4    assert(a.size() == y.size() && a.size() == x.size() &&
5           "Input vector dimensions must match");
6    const unsigned int n = a.size();
7
8    // TODO: (1-13.a) Efficiently compute the matrix-vector product for a
         matrix
9    // with nonzeros on diagonal and antidiagonal.
10   // START
11
12   // looping over half of a,
13   // if n is odd we treat the middle element differently
14   for (unsigned int i = 0; i < n / 2; ++i) {  // integer division
15     x(i) = a(i) * y(i) + a(n - i - 1) * y(n - i - 1);
16     x(n - i - 1) = x(i);
17   }
18
19   if (n % 2) {  // n is odd
20     x(n / 2) = a(n / 2) * y(n / 2);
21   }
22   // END
23 }
```

Get it on GitLab (xmatmult.hpp).

SOLUTION of (1-13.b):

The efficient solution only does a constant amount of operations per vector element, so it's asymptotic complexity is $O(n)$.

Compare this to ordinary matrix-vector multiplication which is $O(n^2)$.

HINT 1 for (1-13.c): In C++ runtimes can be measured by means of the facilities provided by the **std**::`chrono` library. ⌟

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

SOLUTION of (1-13.c):

C++11-code 1.13.4: Code

```
void compare_times() {
  std::cout << "Measuring runtimes for comparison" << std::endl;
  constexpr unsigned int repeats = 3;
  Timer t_fast, t_slow;
  Eigen::MatrixXd results(10, 3);
  // TODO: (1-13.c) Tabulate the runtimes of the efficient mvmult in comparison
  // to the normal one.
  // START
  for (unsigned int k = 14; k > 4; k--) {
    // bitshift operator '«':  1«3 == pow(2,3)
    const unsigned int n = 1 << k;
    Eigen::VectorXd a, y, x(n);
    a = y = Eigen::VectorXd::Random(n);

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

    // measure multiple times
    for (unsigned int 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();
  }
  // END

  // print results
  std::cout << std::setw(8) << "n" << std::setw(15) << "original"
            << std::setw(15) << "efficient" << std::setprecision(5)
            << std::endl;
  for (unsigned 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.hpp).

n	original	efficient
32	3.93e-07 s	7e-08 s
64	9.96e-07 s	8.9e-08 s
128	3.662e-06 s	1.31e-07 s
256	1.6476e-05 s	2.23e-07 s
512	7.0224e-05 s	4.24e-07 s
1024	0.0004936 s	8.7e-07 s
2048	0.00222 s	1.854e-06 s
4096	0.0072985 s	3.493e-06 s
8192	0.034813 s	8.058e-06 s
16384	0.11908 s	1.5899e-05 s

The runtime measurements were performed using an Intel(R) Core(TM) i7 (I7-9750H) CPU @ 2.6 GHz compiled using Apple Clang 14 and the default setup of the NumCSE build system.

HINT 1 for (1-14.a): The well-known formula for the roots of the quadratic polynomial $t \mapsto t^2 + \alpha t + \beta$ is

$$z_1 = \tfrac{1}{2}\big(-\alpha+\sqrt{D}\big) \quad , \quad z_2 = \tfrac{1}{2}\big(-\alpha-\sqrt{D}\big) \ , \tag{1.14.2}$$

valid when the discriminant $D := \alpha^2 - 4\beta$ is not negative. ⌟

HINT 2 for (1-14.a): If the polynomial $t \mapsto t^2 + \alpha t + \beta$ has two real zeros z_1 and z_2, then

$$\beta = z_1 z_2 \quad \text{and} \quad \alpha = -(z_1 + z_2) \, . \tag{1.14.3}$$

⌟

HINT 3 for (1-14.a): In EIGEN the << can be used to initialize a vector by supplying a comma-separated list of values for its components. ⌟

SOLUTION of (1-14.a):

Refer to the C++ code listed as [Lecture → Code 1.5.4.18] for a complete solution and to [Lecture → Ex. 1.5.4.17] for additional explanations.

HINT 1 for (1-14.b): Remember the "$(a-b)(a+b) = a^2 - b^2$-trick". ⌟

SOLUTION of (1-14.b):

The direct implementation

```
double arsinh(double x) { return log(std::sqrt(1 + x * x) + x); }
```

will suffer from severe amplification of roundoff errors through cancellation in the case $x < 0$ and $|x| \gg 1$. Fortunately, there is an equivalent formula

$$\begin{aligned}\log(\sqrt{x^2+1}+x) &= \log\left(\frac{(\sqrt{x^2+1}+x)(\sqrt{x^2+1}-x)}{\sqrt{x^2+1}-x}\right)\\ &= \log\left(\frac{1}{\sqrt{x^2+1}-x}\right) = -\log(\sqrt{x^2+1}-x)\,.\end{aligned}$$

In formula cancellation will occur for $x \gg 0$, but not for $x \ll 0$. Therefore, one should use it for $x < 0$, whereas (1.14.4) has not problems for $x > 0$. This suggests the following implementation:

C++ code 1.14.6: Stable implementation of $\operatorname{arsinh}()$

```
2 double arsinh(double x) {
3   if (x > 0) {
4     return std::log(std::sqrt(1 + x * x) + x);
5   } else {
6     return -std::log(std::sqrt(1 + x * x) - x);
7   }
8 }
```

SOLUTION of (1-2.a):

C++ code [Lecture → Code 0.3.5.30]: Templated function for Gram-Schmidt orthonormalization ⇝ GITLAB

```
template <typename Vec>
std::vector<Vec> gramschmidt(const std::vector<Vec> &A,double eps=1E-14) {
  const int k = A.size();   // no.  of vectors to orthogonalize
  const int n = A[0].size(); // length of vectors
  cout << "gramschmidt orthogonalization for " << k << ' ' << n << "-vectors" <<
      endl;
  std::vector<Vec> Q({A[0]/A[0].norm()}); // output vectors
  for(int j=1;(j<k) && (j<n);++j) {
    Q.push_back(A[j]);
    for(int l=0;l<j;++l) Q.back() -= (A[j]*Q[l])*Q[l];
    if (Q.back().norm() < eps*A[j].norm()) { // premature termination ?
      Q.pop_back(); break;
    }
    Q.back() /= Q.back().norm(); // normalization
  }
  return (Q); // return at end of local scope
}
```

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 $n \to \infty$.

HINT 1 for (1-2.b): Use EIGENs block operations see  EIGEN documentation and [Lecture → Code 1.2.1.6].

SOLUTION of (1-2.b):

There are many ways to code the function `gram_schmidt()`. The following code makes heavy use of EIGEN's block operations.

C++-code 1.2.1: Gram-Schmidt orthonormalization with EIGEN

```
2  Eigen::MatrixXd gram_schmidt(const Eigen::MatrixXd &A) {
3    // We create a matrix Q with the same size and data of A
4    Eigen::MatrixXd Q(A);
5
6    // The first vector just gets normalized
7    Q.col(0).normalize();
8
9    // TODO: (1-2.b) Implement the gram_schmidt procedure by iterating
       over all
10   // other columns of A.
11   // START
12   for (unsigned int j = 1; j < A.cols(); ++j) {
13     // See eigen documentation for usage of col and leftCols
14     Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() * A.col(j));
15
16     // Normalize vector, if possible
17     // (otherwise it means columns of A are
18     // almost linearly dependent)
19     constexpr double eps = std::numeric_limits<double>::denorm_min();
20     if (Q.col(j).norm() <= eps * A.col(j).norm()) {
21       std::cerr << "Gram-Schmidt failed because "
22                 << "A has (almost) linearly dependent "
23                 << "columns." << std::endl;
24       break;
25     } else {
26       Q.col(j).normalize();
27     }
28   }
29   // END
30
31   return Q;
32 }
```

Get it on GitLab (gramschmidt.hpp).

HINT 1 for (1-2.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 (1-2.c): Remember the warning given in [Lecture → Rem. 1.5.3.15] and design your test accordingly: Never compare a matrix with floating-point entries which is the result of a computation to the zero matrix, when you want to test, whether it is “equal to zero”. ⌟

HINT 3 for (1-2.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 (1-2.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. 3.4.4.17]). 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 1.2.2: Test of Gram-Schmidt orthonormalization with EIGEN

```
2  bool testGramSchmidt(unsigned int n) {
3    // Orthonormality test
4    bool orthonormal = false;
5    constexpr double eps = 1e-9;
6    Eigen::MatrixXd A, Q;
7
8    // TODO: (1-2.c) Create A, use gram_schmidt() to compute an
9    // orthonormalization of A, call it Q, and return whether it is
       orthonormal.
10   // START
11   Eigen::VectorXd linspace = Eigen::VectorXd::LinSpaced(n, 1., n);
12   A = Eigen::MatrixXd::Zero(n, n);
13   A.rowwise() += 2. * linspace.transpose();
14   A.colwise() += linspace;
15   Q = gram_schmidt(A);
16   const double err =
17       (Q.transpose() * Q - Eigen::MatrixXd::Identity(n, n)).norm();
18   orthonormal = err < eps;
19   // END
20
21   return orthonormal;
22 }
```

Get it on GitLab (gramschmidt.hpp).

SOLUTION of (1-3.a):

The matrix $\mathbf{C}$ is:

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

HINT 1 for (1-3.b): You can use Eigen “block” operations, see the  EIGEN documentation. ⌟

SOLUTION of (1-3.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 1.3.2: Kronecker product

```
2  Eigen::MatrixXd kron(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B) {
3    // Allocate enough space for the matrix
4    Eigen::MatrixXd C = Eigen::MatrixXd(A.rows() * B.rows(), A.cols() * B.cols());
5    // TODO: (1-3.b) Fill in the entries of C.
6    // Hint: Use a nested for-loop and C.block().
7    // START
8    for (unsigned int i = 0; i < A.rows(); ++i) {
9      for (unsigned int j = 0; j < A.cols(); ++j) {
10       // We use eigen block operations to set the values of
11       // each n × n block.
12       C.block(i * B.rows(), j * B.cols(), B.rows(), B.cols()) =
13           A(i, j) * B;  // ∈ R^(n×n)
14     }
15   }
16   // END
17
18   return C;
19 }
```

Get it on GitLab (kron.hpp).

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

SOLUTION of (1-3.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 $O(n^2 \cdot n^2) = O(n^4)$.

HINT 1 for (1-3.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 (1-3.d):

Define $\tilde{\mathbf{x}}_j := (x_{j\cdot n+1}, \ldots, x_{(j+1)\cdot n})^\top \in \mathbb{R}^n$ (a "segment" of $\mathbf{x}$) for $j = 0, \ldots, n-1$). Notice the following:

$$\mathbf{C}\mathbf{x} = \begin{bmatrix} a_{1,1}\mathbf{B} & \ldots & a_{1,n}\mathbf{B} \\ \vdots & & \vdots \\ a_{n,1}\mathbf{B} & \ldots & a_{n,n}\mathbf{B} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}}_0 \\ \vdots \\ \tilde{\mathbf{x}}_{n-1} \end{bmatrix} = \begin{bmatrix} a_{1,1}\mathbf{B}\tilde{\mathbf{x}}_0 + \cdots + a_{1,n}\mathbf{B}\tilde{\mathbf{x}}_{n-1} \\ \vdots \\ a_{n,1}\mathbf{B}\tilde{\mathbf{x}}_0 + \cdots + a_{n,n}\mathbf{B}\tilde{\mathbf{x}}_{n-1} \end{bmatrix}.$$

The computation of $\mathbf{z}_j := \mathbf{B}\tilde{\mathbf{x}}_j$ needs to be carried out only once and the vectors$\mathbf{z}_j$ can be reused.

C++-code 1.3.3: Kronecker product times vector

```
2  Eigen::VectorXd kron_mult(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
3                            const Eigen::VectorXd &x) {
4    assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
5           "Matrices A and B must be square matrices with same size!");
6    assert(x.size() == A.cols() * A.cols() &&
7           "Vector x must have length A.cols()^2");
8    const unsigned int n = A.rows();
9
10   // Allocate space for output
11   Eigen::VectorXd y = Eigen::VectorXd::Zero(n * n);
12
13   // TODO: (1-3.d) Fill in the entries of y.
14   // Hint: Use a nested for-loop, x.segment(), and y.segment().
15   // In the outer loop, you can perform a computation based on
16   // B and x, and save the result in a variable that is reused in
17   // each iteration of the inner loop.
18   // START
19
20   // Note: this is like doing a matrix-vector multiplication
21   // where the entries of the matrix are smaller matrices
22   // and entries of the vector are smaller vectors
23
24   // Loop over all segments of x (x̃)
25   for (unsigned int j = 0; j < n; ++j) {
26     // Reuse computation of z
27     Eigen::VectorXd z = B * x.segment(j * n, n);
28     // Loop over all segments of y
29     for (unsigned int i = 0; i < n; ++i) {
30       y.segment(i * n, n) += A(i, j) * z;
31     }
32   }
33   // END
34
35   return y;
36 }
```

Get it on GitLab (kron.hpp).

HINT 1 for (1-3.e): Study [Lecture → Rem. 1.2.3.6] about "reshaping" matrices in EIGEN. See also the  EIGEN documentation. ⌟

HINT 2 for (1-3.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 (1-3.e):

See also [Lecture → Code 1.4.3.9]. More explanations are given there.

C++-code 1.3.4: Kronecker product times vector using `Map()`

```
2  Eigen::VectorXd kron_reshape(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
3                               const Eigen::VectorXd &x) {
4    assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
5           "Matrices A and B must be square matrices with same size!");
6    const unsigned int n = A.rows();
7    Eigen::VectorXd y = Eigen::VectorXd::Zero(n * n);
8
9    // TODO: (1-3.e) Fill in the entires of y.
10   // Hint: Use Eigen::MatrixXd::Map() to reshape x into a n by n matrix.
11   // Then y is obtained by simple matrix multiplications and
12   // another reshape.
13   // START
14   Eigen::MatrixXd t = B * Eigen::MatrixXd::Map(x.data(), n, n) * A.transpose();
15   y = Eigen::MatrixXd::Map(t.data(), n * n, 1);
16   // END
17
18   return y;
19 }
```

Get it on GitLab (kron.hpp).

SOLUTION of (1-3.f):

The following code is an example of runtime computation:

C++-code 1.3.5: Runtime comparison

```
void kron_runtime() {
  Eigen::MatrixXd A, B, C;
  Eigen::VectorXd x, y;
  // We repeat each runtime measurement 10 times
  // (this is done in order to remove outliers).
  constexpr 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 = Eigen::MatrixXd::Random(M, M);
      B = Eigen::MatrixXd::Random(M, M);
      x = Eigen::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();
        C = kron(A, B);
        y = C * x;
        tm_kron.stop();
      }

      // TODO: (1-3.f) Measure the runtime of kron_mult() and kron_reshape().
      // START

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

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

```
52     }
53 }
```

Get it on GitLab (kron.hpp).

n	kron	kron_mult	kron_reshape
2	5.980e-07	5.420e-07	6.450e-07
4	9.310e-07	8.770e-07	7.410e-07
8	6.785e-06	1.780e-06	1.296e-06
16	7.987e-05	5.096e-06	2.911e-06
32	1.276e-03	2.334e-05	1.475e-05
64	nan	1.446e-04	9.057e-05
128	nan	1.195e-03	6.799e-04
256	nan	8.947e-03	4.518e-03

The runtime measurements were performed using an Intel(R) Core(TM) i7 (I7-9750H) CPU @ 2.6 GHz compiled using Apple Clang 14 and the default setup of the NumCSE build system.

On your system, the runtimes might vary but the general trend should be similar.

Observations

- From Sub-problem (1-3.c) we know that `kron()` grows as $O(n^4)$ for $n \to \infty$. Thus it is not surprising that the runtimes `kron()` grow fast with increasing problem size.
- The runtime of `kron_mult()` increases much more moderately, because we took advantage of trick presented in [Lecture → Ex. 1.4.3.8]. We observe a growth of the runtimes as $O(n^3)$.
- `kron_reshape()` uses corresponding EIGEN methods which are highly optimized, explaining the slightly lower runtimes compared to `kron_mult()`. Also note the absence of loops. It runtimes increase like $O(n^3)$ for $n \to \infty$.

It is good practice to check your results: ([Lecture → Def. 1.4.1.1]).

For `kron()` we can confirm $T(32)/T(16) = 1.276\text{e}{-03}/7.987\text{e}{-05} = 15.98 \sim 2^4$.

For `kron_mult()` we can confirm $T(256)/T(128) = 8.947\text{e}{-03}/1.195\text{e}{-03} = 7.5 < 2^3$.

For `kron_reshape()` we can confirm $T(256)/T(128) = 4.518\text{e}{-03}/6.799\text{e}{-04} = 6.6 < 2^3$

SOLUTION of (1-4.a):

The following code implements the Strassen's recursive algorithm:

C++-code 1.4.1: Strassen's algorithm with EIGEN

```
Eigen::MatrixXd strassenMatMult(const Eigen::MatrixXd &A,
                                const Eigen::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 x n/2
  // i.e.  exploits fast product of 2x2 block matrix
  if (n == 2) {  // End of recursion
    Eigen::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;
  }

  Eigen::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);

  Eigen::MatrixXd C(n, n);
  // TODO: (1-4.a) Finish Strassen's algorithm.
  // Hint:  Use strassenMatMult() to fill in the matrices Q0,..., Q6,
  // and then use Q0,..., Q6 to fill in the matrix C.
  // Note that comma-initialization works for block matrices.

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

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

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

```
52      // END
53
54      return C;
55  }
```

Get it on GitLab (strassen.hpp).

HINT 1 for (1-4.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. 3.4.4.17]. ⌟

SOLUTION of (1-4.b):

C++-code 1.4.2: Strassen's algorithm with EIGEN

```
double test_strassen() {
  // Check algorithm for correctness

  // Size of the matrix is a power of 2.
  constexpr int k = 4;
  const int n = std::pow(2, k);
  double err = 1;

  // TODO: (1-4.b) Generate two random matrices using
  // Eigen::MatrixXd::Random(n,n).  and compare the result of strassenMatMult()
  // and the built-in matrix multiplication.  Save the difference in the double
  // err.

  // START
  Eigen::MatrixXd A = Eigen::MatrixXd::Random(n, n);
  Eigen::MatrixXd B = Eigen::MatrixXd::Random(n, n);

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

  err = (AB - AxB).norm();
  // END

  return err;
}
```

Get it on GitLab (strassen.hpp).

SOLUTION of (1-4.c):

C++-code 1.4.3: Runtime of Strassen's algorithm EIGEN

```cpp
void time_strassen() {
  // Unfortunately, the time limit set by Code Expert may not be sufficiently
  // long to show the advantage of Strassen's algorithm for large matrices.

  // Minimum number of repetitions
  constexpr unsigned int repetitions = 5;

  // Display header column
  std::cout << std::setw(4) << "k" << std::setw(15) << "A*B" << std::setw(15)
            << "Strassen" << std::endl;
  for (unsigned k = 3; k <= 6; k++) {
    unsigned int n = std::pow(2, k);

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

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

    // TODO: (1-4.c) Use the .start(), and .stop() methods of timer and
    // timer_own to measure the runtimes of Eigen's built-in multiplication and
    // Strassen's algorithm.  Perform a few trials for each k (use the variable
    // repetitions).

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

      // Benchmark Eigen's matrix multiplication
      timer.start();  // start timer
      AxB = (A * B);  // do the multiplication
      timer.stop();   // stop timer

      // Benchmark Strassen's matrix multiplication
      timer_own.start();            // start timer
      AxB2 = strassenMatMult(A, B); // do the multiplication
      timer_own.stop();             // stop timer
    }
    // END

    // 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;
  }
```

```
52 }
```

Get it on GitLab (strassen.hpp).

HINT 1 for (1-5.a): The EIGEN matrix/vector methods `normalized()`, `rightCols` and `transpose` may be useful. ⌟

SOLUTION of (1-5.a):

The following code implements Algorithm 1.5.1 making use of EIGEN's matrix-block operations.

C++-code 1.5.2: C++ implementation of Code 1.5.1

```
2  void houserefl(const Eigen::VectorXd &v, Eigen::MatrixXd &Z) {
3    const unsigned int n = v.size();
4    // TODO: (1-5.a) Implement the householder refelction algorithm
       described in
5    // the pseudo code 1.5.1 to get the ONB of the space orthogonal to v.
6
7    // START
8    Eigen::VectorXd w = v.normalized();
9    Eigen::VectorXd u = w;
10   u(0) += 1;
11   Eigen::VectorXd q = u.normalized();
12   Eigen::MatrixXd X = Eigen::MatrixXd::Identity(n, n) - 2 * q * q.transpose();
13   Z = X.rightCols(n - 1);
14   // END
15 }
```

Get it on GitLab (houserefl.hpp).

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

SOLUTION of (1-5.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. □

HINT 1 for (1-5.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 (1-5.c):

Let $\mathbf{X} = [\mathbf{X}_1, \cdots, \mathbf{X}_n]\mathbb{R}^{n,n}$ be the matrix defined in Code 1.5.1. 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 $\mathbf{w} = \frac{\mathbf{v}}{|\mathbf{v}|}$. □

SOLUTION of (1-5.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 $\mathrm{Span}(\mathbf{X}_1) \overset{(1-5.c)}{=} \mathrm{Span}(\mathbf{v})$. The function `houserefl` computes an ONB of the complement of $\mathbf{v}$.

SOLUTION of (1-5.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 1.5.2.

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

SOLUTION of (1-6.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 1.6.1: Efficient matrix power

```
2  Eigen::MatrixXcd matPow(Eigen::MatrixXcd& A, unsigned int k) {
3    // Identity matrix
4    Eigen::MatrixXcd X = Eigen::MatrixXcd::Identity(A.rows(), A.cols());
5
6    // TODO: (1-6.a) Compute the k-th power of A in X.
7    // START
8
9    // p is used as binary mask to check whether,
10   // given k = sum_{i=0}^{M} b_i 2^i, b_i = 1
11   // (i.e.  if k has a 1 in the i-th binary digit).
12   // Obtaining the binary representation of p can be done in many ways,
13   // here we use ~k&p to check if b_i = 1
14   unsigned int p = 1;
15
16   // Cycle all the way up to the 1st 1 in the binary
17   // representation of k (this is called the "most significant bit")
18   for (unsigned int j = 1; j <= std::ceil(std::log2(k)); ++j) {
19     if ((~k & p) == 0) {  // if( b_i != 0 )
20       X = X * A;
21     }
22     A = A * A;
23     p = p << 1;  // p = p*2;
24   }
25   // END
26   return X;
27 }
```

Get it on GitLab (matPow.hpp).

SOLUTION of (1-6.b):

Using a "naive" implementation according to

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

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

Using the efficient implementation from Code 2.16.6, 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`):

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

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

SOLUTION of (1-6.c):

C++11-code 1.6.2: Runtime computation

```
void tabulateRuntime(unsigned int n) {
  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;
  constexpr unsigned int repeats = 10;
  // Loop from 2 to 2^31
  for (unsigned long long int K = 2; K <= (1u << 31); K = K << 1) {
    Timer tm_pow, tm_Eigen_pow;

    // Repeat the test repeat times
    for (unsigned int r = 0; r < repeats; ++r) {
      // TODO: (1-6.c) Measure runtime of your own and Eigen's
         implementation.
      // You may use construct_matrix.
      // START
      // Build Vandermonde matrix of size n
      Eigen::MatrixXcd X, A = construct_matrix(n);

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

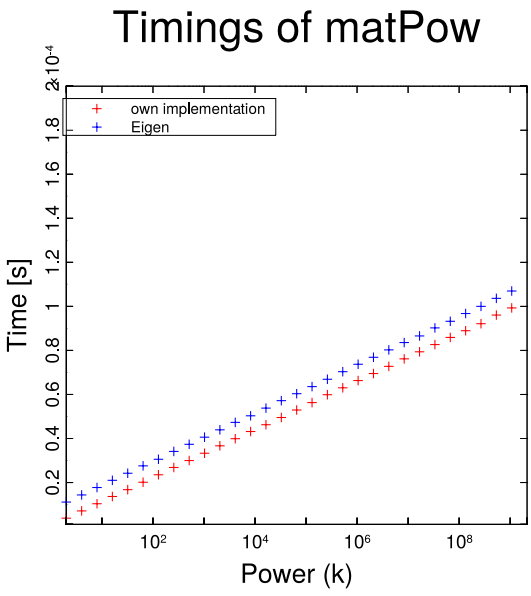
      // Compute runtime of eigen implementation
      A = construct_matrix(n);
      tm_Eigen_pow.start();
      X = A.pow(K);
      tm_Eigen_pow.stop();
      // END
    }

    // 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;
  }
}
```

Get it on GitLab (matPow.hpp).

K	Own matPow	Eigen pow
2	3.300e-07	5.260e-07
4	4.820e-07	6.800e-07
8	6.220e-07	8.170e-07
16	7.480e-07	9.670e-07
32	8.780e-07	1.107e-06
64	9.920e-07	1.243e-06
128	1.141e-06	1.380e-06
256	1.264e-06	1.521e-06
512	1.399e-06	1.662e-06
1024	1.531e-06	1.804e-06
2048	1.669e-06	1.941e-06
4096	1.809e-06	2.077e-06
8192	1.927e-06	2.228e-06
16384	2.059e-06	2.368e-06
32768	2.186e-06	2.493e-06
65536	2.317e-06	2.644e-06
131072	2.454e-06	2.776e-06
262144	2.580e-06	2.915e-06
524288	2.712e-06	3.065e-06
1048576	2.844e-06	3.202e-06
2097152	2.974e-06	3.334e-06
4194304	3.102e-06	3.480e-06
8388608	3.242e-06	3.631e-06
16777216	3.377e-06	3.755e-06
33554432	3.503e-06	3.897e-06
67108864	3.647e-06	4.046e-06
134217728	3.778e-06	4.196e-06
268435456	3.902e-06	4.317e-06
536870912	4.031e-06	4.470e-06
1073741824	4.158e-06	4.589e-06
2147483648	4.300e-06	4.757e-06

The runtime measurements were performed using an Intel(R) Core(TM) i7 (I7-9750H) CPU @ 2.6 GHz compiled using Apple Clang 14 and the default setup of the NumCSE build system.



According to the result of Sub-problem (1-6.b) we expect an asyymptotic behavior of the runtimes as $O(\log k)$ for $k \to \infty$. This should best mmanifest itself as data points lying on a straight line in *logarithmic-linear* plot

Reamrk. When reporting runtimes one always has to provide information about the machine, operating system, compilers and compiler options used. This is missing here.

Remark: All the eigenvalues of the Vandermonde matrix $\mathbf{A}$ have absolute value 1, so the powers $\mathbf{A}^k$ are "stable": the eigenvalues of $\mathbf{A}^k$ are not approaching neither 0 nor ∞ when k grows.

SOLUTION of (1-7.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 $O(n^2)$.

HINT 1 for (1-7.b): Write an expression for the components y_j of the result vector $\mathbf{y}$. Find a way to "reuse" your computations. ⌟

SOLUTION of (1-7.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

$$y_j = \sum_{k=1}^{j} k x_k + j \sum_{k=j+1}^{n} x_k = w_j + j v_{n-j}. \tag{1.7.3}$$

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 (1.7.3) 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 1.7.4: Efficiently computing $\mathbf{A}\mathbf{x}$

```
2  void multAmin(const Eigen::VectorXd &x, Eigen::VectorXd &y) {
3    const unsigned int n = x.size();
4    y = Eigen::VectorXd::Zero(n);
5
6    // TODO: (1-7.b) Fill in the entries of y.
7    // Hint: Find an expression y(j) = w(j) + j*u(j) such that
8    // the entries of w and u can be calculated recursively.
9    // START
10   Eigen::VectorXd v = Eigen::VectorXd::Zero(n);
11   Eigen::VectorXd w = Eigen::VectorXd::Zero(n);
12
13   v(0) = x(n - 1);
14   w(0) = x(0);
15
16   for (unsigned int j = 1; j < n; ++j) {
17     v(j) = v(j - 1) + x(n - j - 1);
```

```
18      w(j) = w(j - 1) + (j + 1) * x(j);
19    }
20    for (unsigned int j = 0; j < n - 1; ++j) {
21      y(j) = w(j) + v(n - j - 2) * (j + 1);
22    }
23    y(n - 1) = w(n - 1);
24    // END
25  }
```

Get it on GitLab (multAmin.hpp).

SOLUTION of (1-7.c):

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

SOLUTION of (1-7.d):

The matrix multiplication in (1.7.5) has complexity $O(n^2)$. The faster implementation has complexity $O(n)$.

C++11-code 1.7.5: Runtime test for `multAmin`

```
2   // Timing from 2^4 to 2^10 repeating "nruns" times
3   constexpr unsigned int nruns = 10;
4
5   std::cout << "--> Timings:" << std::endl;
6   // Header, see iomanip documentation
7   std::cout << std::setw(15) << "N" << std::scientific << std::setprecision(3)
8             << std::setw(15) << "multAminSlow" << std::setw(15) << "multAmin"
9             << std::endl;
10  // From 2^4 to 2^10
11  // Note: << and >> are the bitwise shift operators
12  for (unsigned int N = (1 << 4); N <= (1 << 10); N = N << 1) {
13    Timer tm_slow, tm_fast;
14    // TODO: (1-7.d) Compute runtimes of multAminSlow(x,y) and
15    // multAmin(x,y) with x = Eigen::VectorXd::Random(N). Repeat nruns times.
16    // START
17    for (unsigned int r = 0; r < nruns; ++r) {
18      Eigen::VectorXd x = Eigen::VectorXd::Random(N);
19      Eigen::VectorXd y;
20
21      // Runtime of slow method
22      tm_slow.start();
23      multAminSlow(x, y);
24      tm_slow.stop();
25
26      // Runtime of fast method
27      tm_fast.start();
28      multAmin(x, y);
29      tm_fast.stop();
30    }
31    // END
32
33    std::cout << std::setw(15) << N << std::scientific << std::setprecision(3)
34              << std::setw(15) << tm_slow.min() << std::setw(15)
35              << tm_fast.min() << std::endl;
36  }
```

Get it on GitLab (multAmin.hpp).

SOLUTION of (1-7.e):

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

$$\mathbf{B} := \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 1 \end{bmatrix} \in \mathbb{R}^{n,n} .$$

Notice the value 1 in the entry at position (n, n).

SOLUTION of (1-7.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 1.7.7: Matrix multiplication $\mathbf{AB}$

```
2  Eigen::MatrixXd multABunitv() {
3    constexpr unsigned int n = 10;
4
5    /* SAM_LISTING_BEGIN_5 */
6    Eigen::MatrixXd B = Eigen::MatrixXd::Zero(n, n);
7    for (unsigned int i = 0; i < n; ++i) {
8      B(i, i) = 2;
9      if (i < n - 1) B(i + 1, i) = -1.0;
10     if (i > 0) B(i - 1, i) = -1.0;
11   }
12   B(n - 1, n - 1) = 1.0;
13   /* SAM_LISTING_END_5 */
14
15   Eigen::MatrixXd C(n, n);
16
17   // TODO: (1-7.f) Set the columns of C to A*B*e_j for
18   // j=0,...,n-1, and print C.
19   // START
20   Eigen::VectorXd y;
21   // B*e_j is the j-th column of B*Id = B, so we need only
22   // calculate A*B.col(j) for j=0,...,n-1.
23   for (unsigned int i = 0; i < n; ++i) {
24     multAmin(B.col(i), y);
25     C.col(i) = y;
26   }
27   std::cout << "C = \n" << C << std::endl;
28   // END
29
30   return C;
31 }
```

Get it on GitLab (multAmin.hpp).

HINT 1 for (1-8.a): Use that (1.8.1) can be transformed, using the trigonometric identity

$$\sin(\varphi) - \sin(\psi) = 2\cos\left(\frac{\varphi+\psi}{2}\right)\sin\left(\frac{\varphi-\psi}{2}\right). \tag{1.8.2}$$

⌟

SOLUTION of (1-8.a):

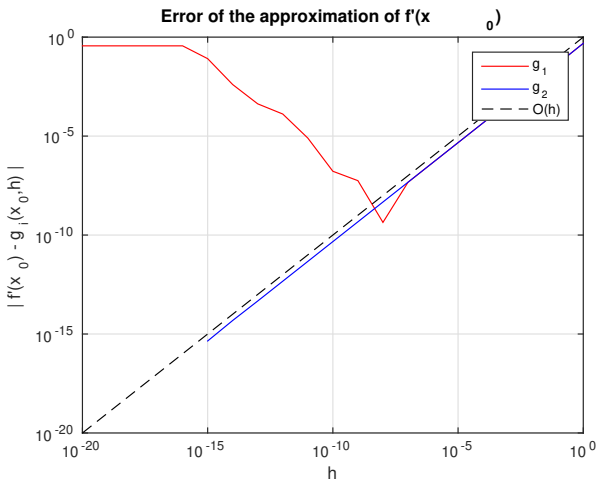
Based on (1.8.2) we get the equivalent formula

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

HINT 1 for (1-8.b): For background information refer to [Lecture → Ex. 1.5.4.7].

SOLUTION of (1-8.b):

Check the implementation in Code 1.8.4 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.



C++-code 1.8.4: C++ implementation for Problem 1-8

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

  // Derivatives
  Eigen::ArrayXd g1, g2, ex;

  // TODO: (1-8.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
    // TODO: (1-8.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.
```

```
38      // START
39      std::cout << std::setw(15) << h(i) << std::setw(15) << ex(i)
40                << std::setw(15) << g1(i) << std::setw(15)
41                << std::abs(g1(i) - ex(i)) << std::setw(15) << g2(i)
42                << std::setw(15) << std::abs(g2(i) - ex(i)) << std::endl;
43      // END
44    }
45
46    // Plots the contents of the table on a log-log scale.
47    py_plot(h, ex, g1, g2);
48  }
```

Get it on GitLab (cancellation.hpp).

HINT 1 for (1-8.c): Study the tricks applied in [Lecture → Ex. 1.5.4.21] to avoid cancellation. ⌟

SOLUTION of (1-8.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 $x = 10^8$.

SOLUTION of (1-8.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 problem we are facing with a straightforward implementation of the term $\sqrt{\frac{1}{a^2}+\frac{1}{b^2}}$, for $a, b > 0$, for instance

```
double x = std:sqrt(1./(a*a)+1./(b*b));
```

   is not that of cancellation, but of possible overflow and underflow, refer to [Lecture → Rem. 1.5.3.16]. In case

   - $|a|$ or $|b|$ are very big, their squares may exceed the largest machine number,
   - $|a|$ or $|b|$ are very small, their squares can leave the range of normalized machine numbers and "fall into the gap" between $\min|\mathbb{M}|$ and 0, which results in large roundoff errors.

   This can be remedied by the following implementation, assuming $a, b > 0$:

```
double x = [&]() -> double {
  if (b>a) {
    const double s = a/b; return (std::sqrt(1+s*s)/a);
  }
  const double s = b/a; return (std::sqrt(1+s*s)/b);
  }();
```

   Now, $|s| \leq 1$ is guaranteed and no overflow nor underflow will occur.

   Remark. The above code snippet demonstrates initialization by lambda functions, which makes it possible to use complex control structures within a local scope for the sole purpose of initializing a variable.

SOLUTION of (1-9.a):

The output is $\mathbf{y}$ s.t. $\mathbf{y} = \mathbf{A}^k\mathbf{x}$. In fact, the eigenvalue decomposition of the matrix $\mathbf{A} = \mathbf{SDW}^{-1}$ (where $\mathbf{D}$ is diagonal and $\mathbf{W}$ invertible), allows us to write:

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

$\mathbf{D}^k$ can be computed efficiently (component-wise) for diagonal matrices.

The code `getit.hpp` provides a line-by-line explanation. Get it on GitLab (getit.hpp).

SOLUTION of (1-9.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 $n \to \infty$.

SOLUTION of (2-1.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 2.1.1.

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 2.1.1: Code

```cpp
Eigen::VectorXd fruitPrice() {
  Eigen::VectorXd x = Eigen::VectorXd::Zero(6);

  // TODO: (2-1.b) Determine the fruit prices using Eigen.
  // START
  // Initialize the matrix
  Eigen::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
  Eigen::VectorXd b(6);
  b << 11.10, 17.00, 6.10, 5.25, 12.50, 7.00;

  // Initialize the solver; other decompositions also possible
  Eigen::ColPivHouseholderQR<Eigen::MatrixXd> solver;
  // Decompose A
  solver.compute(A);
  // Solve the LSE
  x = 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:
  Eigen::VectorXd x2 = A.colPivHouseholderQr().solve(b);

  // The result is the same:
  assert((x - x2).norm() < 1e-10);
  // END
  return x;
}
```

Get it on GitLab (fruits.hpp).

HINT 1 for (2-10.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 (2-10.a):

C++-code 2.10.2: Solution of (2-10.a)

```
2  double rankoneinvit(const Eigen::VectorXd& d, const double& tol) {
3    // Initialization
4    double lmin = 0, lnew = 0;
5    // TODO: (2-10.a) Port the pseudocode in algorithm (2.10.1) to C++-code.
6    // START
7    Eigen::VectorXd ev = d;
8    lnew = d.cwiseAbs().minCoeff();
9
10   while (std::abs(lnew - lmin) > tol * lmin) {
11     lmin = lnew;
12     Eigen::MatrixXd M = d.asDiagonal();
13     M += ev * ev.transpose();
14     // Instead of calculating the inverse of M directly
15     // to get ev_new = M^{-1}*ev_old,
16     // we solve the LSE M*ev_new = ev_old.
17     ev = M.lu().solve(ev);
18     // Equivalent to ev <- ev/|ev|:
19     ev.normalize();
20     lnew = ev.dot(M * ev);
21   }
22   // END
23   return lnew;
24 }
```

Get it on GitLab (rankoneinvit.hpp).

HINT 1 for (2-10.b): Find which passage mostly contributes to the overall complexity of the code. What is the corresponding asymptotic complexity? ⌟

SOLUTION of (2-10.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 $O(n^3)$.

HINT 1 for (2-10.c): Exploit the Sherman-Morrison-Woodbury formula (see [Lecture → § 2.6.0.12]). ⌟

SOLUTION of (2-10.c):

Since we want to solve $\mathbf{ev}_{new} = \mathbf{M}^{-1}\mathbf{ev}$ with $\mathbf{M} = \mathrm{diag}(\mathbf{d}) + \mathbf{ev} \cdot \mathbf{ev}^\top$, 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:

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

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

C++-code 2.10.3: Solution of (2-10.c)

```
2  double rankoneinvit_fast(const Eigen::VectorXd& d, const double& tol) {
3    // Initialization
4    double lmin = 0, lnew = 0;
5    // TODO: (2-10.c) Rewrite rankoneinvit() to have much better asymptotic
6    // complexity.
7    // START
8    Eigen::VectorXd ev = d;
9    lnew = d.cwiseAbs().minCoeff();
10
11   Eigen::VectorXd dinv = (1. / d.array()).matrix();
12   while (std::abs(lnew - lmin) > tol * lmin) {
13     lmin = lnew;
14     Eigen::VectorXd ev0 = ev;
15
16     // Here we solve the linear system
17     // with the Sherman-Morrison-Woodbury formula
18     // in the case of rank-1 perturbations.
19     // This holds from M = diag(d) + ev * ev^t
20     Eigen::VectorXd Aib = dinv.cwiseProduct(ev);
21     const double temp = ev.transpose() * Aib;
22     ev = Aib / (1 + temp);
23     ev.normalize();
24     // Better than the corresponding naive implementation.
25     // This holds from M = diag(d) + ev * ev^t, too
26     lnew = ev.transpose() * d.cwiseProduct(ev) + pow(ev.transpose() * ev0, 2);
27   }
28   // END
29   return lnew;
30 }
```

Get it on GitLab (rankoneinvit.hpp).

SOLUTION of (2-10.d):

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

SOLUTION of (2-10.e):

C++-code 2.10.4: Solution of (2-10.e)

```
2  void rankoneinvit_runtime() {
3    constexpr unsigned int repeats = 3;
4    constexpr double tol = 1e-3;
5    // TODO: (2-10.e) Tabulate the runtimes of both implementations according to
6    // the problem description. The choice of the "tol" argument affects the
7    // runtime.
8    // START
9    double lmin;
10   Eigen::VectorXd d;
11   Timer tm_slow, tm_fast;
12
13   std::cout << std::endl
14             << std::setw(15) << "n" << std::setw(15) << "Slow" << std::setw(15)
15             << "Fast" << std::endl;
16
17   for (unsigned int n = 2; n <= 256; n <<= 1) {
18     tm_slow.reset();
19     tm_fast.reset();
20     for (unsigned int r = 0; r < repeats; ++r) {
21       d = Eigen::VectorXd::LinSpaced(n, 1, 2);
22
23       tm_slow.start();
24       lmin = rankoneinvit(d, tol);
25       tm_slow.stop();
26
27       tm_fast.start();
28       lmin = rankoneinvit_fast(d, tol);
29       tm_fast.stop();
30     }
31     std::cout << std::setw(15) << n << std::scientific << std::setprecision(3)
32               << std::setw(15) << tm_slow.min() << std::setw(15)
33               << tm_fast.min() << std::endl;
34   }
35   // END
36 }
```

Get it on GitLab (rankoneinvit.hpp).

n	Slow	Fast
2	6.858e-06	1.929e-06
4	1.373e-05	3.386e-06
8	2.998e-05	4.742e-06
16	6.874e-05	6.542e-06
32	2.328e-04	9.072e-06
64	9.460e-04	9.830e-06
128	5.344e-03	1.120e-05
256	2.500e-02	1.751e-05

The runtime measurements were performed on a Intel(R) Core(TM) i7-9750H CPU @ 2.6GHz compiled using Apple Clang 14.0.0 using the Release config of CMake build system in the NumCSE repository.

On your system, the runtimes might vary but the general trend should be similar.

HINT 1 for (2-11.a): Think about the complexity of solving a linear system (Gaussian elimination, see [Lecture → § 2.3.1.5]) for n times. ⌟

SOLUTION of (2-11.a):

This function solves the linear system $\mathbf{Ax} = \mathbf{b}_i$ 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 $\mathbf{AX}_i = \mathbf{b}_i$.

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

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

HINT 1 for (2-11.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 (2-11.b):

C++11-code 2.11.2: Solution of (2-11.b)

```
2  void solvpermb(const Eigen::MatrixXd& A, Eigen::VectorXd& b,
3                 Eigen::MatrixXd& X) {
4    // Size of b, which is the size of A
5    const unsigned int n = b.size();
6    assert(n == A.rows() && n == A.cols() && "Error: size mismatch!");
7    // TODO: (2-11.b) Port the pseudocode from (3.11.1) to C++-code.
8    // START
9    X.resize(n, n);
10
11   // For each loop iteration:
12   // 1.  solve the linear system Ax = b,
13   // 2.  store the result in a column of X
14   // 3.  and shift b by one element for the next iteration.
15   for (unsigned int l = 0; l < n; ++l) {
16     X.col(l) = A.fullPivLu().solve(b);
17     shift(b);
18   }
19   // END
20 }
```

Get it on GitLab (solvepermb.hpp).

HINT 1 for (2-11.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 (2-11.c):

C++11-code 2.11.3: Solution of (2-11.c)

```
2  void solvpermb_on3(const Eigen::MatrixXd& A, Eigen::VectorXd& b,
3                     Eigen::MatrixXd& X) {
4    // Size of b, which is the size of A
5    const unsigned int n = b.size();
6    assert(n == A.cols() && n == A.rows() && "Error: size mismatch!");
7    // TODO: (2-11.c) Alter the above function such that it has asymptotic
8    // complexity O(n^3).
9    // START
10   X.resize(n, n);
11
12   // Notice that here we reuse the LU factorization
13   Eigen::FullPivLU<Eigen::MatrixXd> LU = A.fullPivLu();
14
15   for (unsigned int l = 0; l < n; ++l) {
16     X.col(l) = LU.solve(b);
17     shift(b);
18   }
19   // END
20 }
```

Get it on GitLab (solvepermb.hpp).

HINT 1 for (2-12.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 (2-12.a):

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

HINT 1 for (2-12.b): Consider a loop initialization. ⌟

SOLUTION of (2-12.b):

C++11-code 2.12.1: Solution of (2-12.b)

```
2  Eigen::MatrixXd buildA(const Eigen::VectorXd& a) {
3    // Initialization
4    const unsigned int n = a.size();
5    Eigen::MatrixXd A = Eigen::MatrixXd::Zero(n, n);
6    // TODO: (2-12.b) Build the matrix A given the vector a.
7    // START
8    for (unsigned int j = 0; j < n; ++j) {
9      for (unsigned int i = j; i < n; ++i) {
10       A(i, j) = a(j);
11     }
12   }
13   // END
14   return A;
15 }
```

Get it on GitLab (structuredLSE.hpp).

HINT 1 for (2-12.c): Remember what you did in Code 2.11.2.

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

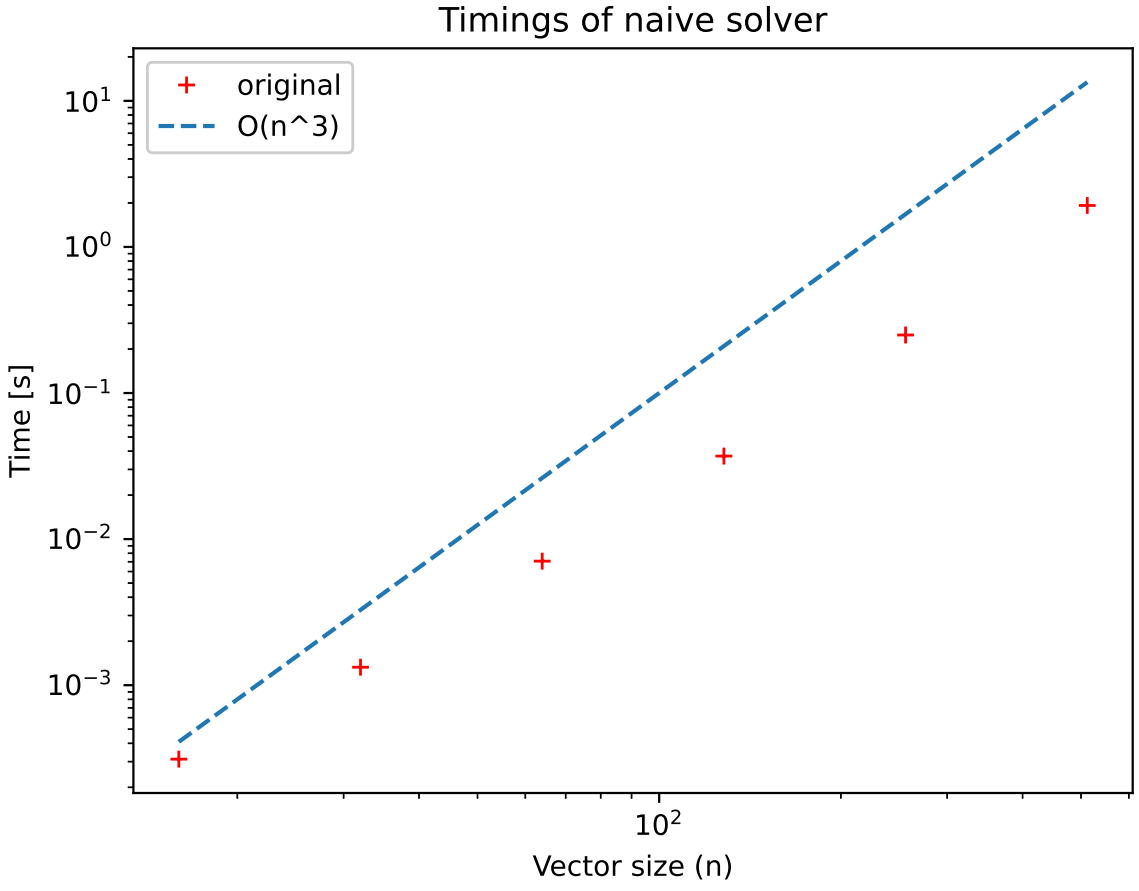
SOLUTION of (2-12.c):

C++11-code 2.12.2: Solution of (2-12.c)

```
void solveA(const Eigen::VectorXd& a, const Eigen::VectorXd& b,
            Eigen::VectorXd& x) {
  // size of b, which is the size of a
  const unsigned int n = b.size();
  assert(n == a.size() && "Error: size mismatch!");
  // TODO: (2-12.c) Solve Ax = b using "structure oblivious" Gaussian
  // elimination.
  // START
  x.resize(n);
  Eigen::MatrixXd A = buildA(a);
  x = A.fullPivLu().solve(b);
  // END
}
```

Get it on GitLab (structuredLSE.hpp).

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.



HINT 1 for (2-12.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 (2-12.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 $= 2$!

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

HINT 1 for (2-12.e): Exploit the solution of (2-12.d). ⌟

SOLUTION of (2-12.e):

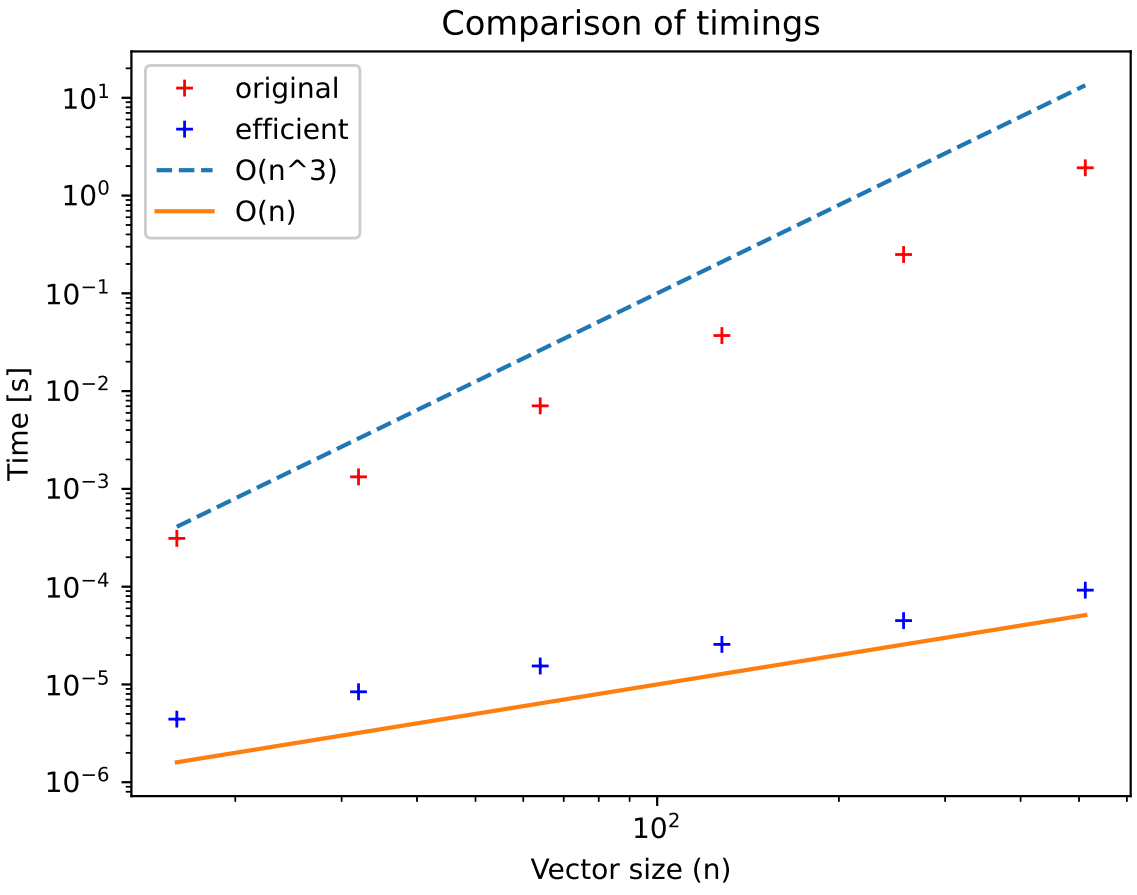
C++11-code 2.12.3: Solution of (2-12.e)

```
2  void solveA_fast(const Eigen::VectorXd& a, const Eigen::VectorXd& b,
3                   Eigen::VectorXd& x) {
4    // size of b, which is the size of a
5    const unsigned int n = b.size();
6    assert(n == a.size() && "Error: size mismatch!");
7    // TODO: (2-12.e) Efficiently (O(n)) solve Ax = b.
8    // START
9    x.resize(n);
10
11   x(0) = b(0) / a(0);
12   for (unsigned int i = 1; i < n; ++i) {
13     x(i) = (b(i) - b(i - 1)) / a(i);
14   }
15   // END
16 }
```

Get it on GitLab (structuredLSE.hpp).

HINT 1 for (2-12.f): The `main()` code to test your implementation is already provided in the template.

SOLUTION of (2-12.f):



SOLUTION of (2-13.a):

- For **TripletMatrix** the conversion is easy: Run through the vectors of triplets and add the values of a triplet to the matrix element with its indices.
- For **CRSMatrix** you fill the dense matrix row by row while iterating over the `val` and `col_ind` vectors.

C++ code 2.13.1: Implementation of `densify()` for TripletMatrix.

```
2  template <typename SCALAR>
3  Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> densify(
4      const TripletMatrix<SCALAR>& M) {
5    Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> dense =
6        Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>::Zero(M.n_rows,
7                                                                    M.n_cols);
8
9    // TODO: (2-13.a) Densify the TripletMatrix.
10   // START
11   for (auto& triplet : M.triplets) {
12     dense(std::get<0>(triplet), std::get<1>(triplet)) += std::get<2>(triplet);
13   }
14   // END
15   return dense;
16 }
```

Get it on GitLab (triplettoCRS.hpp).

C++ code 2.13.2: Implementation of `densify()` for CRSMatrix

```
2  template <typename SCALAR>
3  Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> densify(
4      const CRSMatrix<SCALAR>& M) {
5    Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> dense =
6        Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>::Zero(M.n_rows,
7                                                                    M.n_cols);
8
9    // TODO: (2-13.a) Densify the CRSMatrix.
10   // START
11   std::vector<std::size_t> row_ptr_end = M.row_ptr;
12   row_ptr_end.push_back(M.col_ind.size());
13
14   for (std::size_t i = 0; i < M.row_ptr.size(); ++i) {
15     for (std::size_t j = row_ptr_end[i]; j < row_ptr_end[i + 1]; ++j) {
16       dense(i, M.col_ind[j]) = M.val[j];
17     }
18   }
19   // END
20   return dense;
21 }
```

Get it on GitLab (triplettoCRS.hpp).

HINT 1 for (2-13.b): It might be a good idea to sort the vector of triplets beforehand in a suitable way. To that end you can invoke **std**`::sort()`.

⌟

SOLUTION of (2-13.b):

The following steps can be implemented to realize `tripletToCRS()`:

1. Copy the vector of triplets and *sort* the copy using **std::sort** with a predicate comparing two triplets according to

$$(i_1, j_1, a_1) < (i_2, j_2, a_2) \quad \Leftrightarrow \quad (i_1 < i_2) \quad \vee \quad (i_1 = i_2 \ \wedge \ j_1 < j_2) \, .$$

2. Loop over the sorted vector and count the numbers of non-zero entries per row:

$$n_i := \sharp\{j : \ (i, j, *) \in \text{triplet vector}\}; , \quad i = 1, \ldots, \texttt{n\_rows} \, .$$

   Use these values to initialize the `row_ptr` and add them to obtain the length of `col_ind` and `val`.

3. Run through the sorted triplet vector again, summing the value entries of repeated index pairs and storing the sum in the `val` vector. The corresponding column indices have to be inserted into `col_ind`.

C++ code 2.13.3: Listing of `tripletToCRS()`

```
2  template <typename SCALAR>
3  CRSMatrix<SCALAR> tripletToCRS(const TripletMatrix<SCALAR>& T) {
4    CRSMatrix<SCALAR> crs;
5    crs.n_rows = T.n_rows;
6    crs.n_cols = T.n_cols;
7
8    // TODO: (2-13.b) Convert the TripletMatrix to a CRSMatrix.  Make sure
       that you
9    // handle repeated index pairs.
10   // START
11   // sort the triplets according to their indices over a copy of the
       vector
12   auto triplets_copy = T.triplets;
13   auto sorting_predicate =
14       [](const std::tuple<std::size_t, std::size_t, SCALAR>& a,
15          const std::tuple<std::size_t, std::size_t, SCALAR>& b) {
16         return (std::get<0>(a) < std::get<0>(b)) ||
17                ((std::get<0>(a) == std::get<0>(b)) &&
18                 (std::get<1>(a) < std::get<1>(b)));
19       };
20   std::sort(triplets_copy.begin(), triplets_copy.end(), sorting_predicate);
21
22   // count the number of non-zeroes per row
23   crs.row_ptr = std::vector<std::size_t>(crs.n_rows + 1, 0);
24   std::size_t k = 0;
25   std::size_t last_j = -1;
26   std::size_t last_i = -1;
27   for (std::size_t i = 1; i <= crs.n_rows; ++i) {
28     while (k < triplets_copy.size() && std::get<0>(triplets_copy[k]) == i - 1) {
29       // do only when no duplicate index pair
30       if (std::get<1>(triplets_copy[k]) != last_j ||
31           std::get<0>(triplets_copy[k]) != last_i) {
32         last_j = std::get<1>(triplets_copy[k]);
33         last_i = std::get<0>(triplets_copy[k]);
```

```
34          ++crs.row_ptr[i];
35        }
36        ++k;
37      }
38      crs.row_ptr[i] += crs.row_ptr[i - 1];
39    }
40
41    // now, we know nnz; reserve in advance
42    crs.val.reserve(crs.row_ptr[crs.n_rows]);
43    crs.col_ind.reserve(crs.row_ptr[crs.n_rows]);
44
45    crs.val.push_back(std::get<2>(triplets_copy[0]));
46    crs.col_ind.push_back(std::get<1>(triplets_copy[0]));
47    std::size_t i = 0;
48    for (k = 1; k < triplets_copy.size(); ++k) {
49      // repeated index pairs
50      if (std::get<0>(triplets_copy[k]) == std::get<0>(triplets_copy[k - 1]) &&
51          std::get<1>(triplets_copy[k]) == std::get<1>(triplets_copy[k - 1])) {
52        crs.val[i] += std::get<2>(triplets_copy[k]);
53      } else {
54        crs.val.push_back(std::get<2>(triplets_copy[k]));
55        crs.col_ind.push_back(std::get<1>(triplets_copy[k]));
56        ++i;
57      }
58    }
59    // END
60    return crs;
61  }
```

Get it on GitLab (triplettoCRS.hpp).

HINT 1 for (2-13.c): If you use the C++ standard library functions consult the documentation to find out about their asymptotic computational effort. ⌟

SOLUTION of (2-13.c):

According to the C++ standard the algorithm underlying **std**`::sort()` has to feature an asymptotic complexity of $O(n \log n)$ for $n \to \infty$, n the number of array elements to be sorted. The sorting is the most expensive operation in Code 2.13.3. The computational cost of all other parts scales linearly in the number of triplets.

HINT 1 for (2-13.d): Note that the conversion of the matrix formats may involved adding floating point numbers, which may introduce round-off errors. Thus [Lecture → Rem. 1.5.3.15] applies. ⌟

SOLUTION of (2-13.d):

C++ code 2.13.4: Listing of `testTripletToCRS()`

```
2  bool testTripletToCRS(std::size_t n) {
3    bool areTheSame = false;
4    // TODO: (2-13.d) Test if the conversion method from above works.  To
       that end,
5    // you may use the densify() methods.
6    // START
7    TripletMatrix<double> M;
8    M.n_rows = n;
9    M.n_cols = n;
10   std::vector<std::tuple<std::size_t, std::size_t, double>> triplets(5 * n);
11   for (std::size_t i = 0; i < 5 * n; ++i) {
12     triplets[i] = {std::rand() % n, std::rand() % n, 1.};
13   }
14   M.triplets = triplets;
15   CRSMatrix<double> C = tripletToCRS(M);
16   areTheSame = (densify(C) - densify(M)).norm() < 1e-10;
17   // END
18   return areTheSame;
19 }
```

Get it on GitLab (triplettoCRS.hpp).

SOLUTION of (2-14.a):

C++11-code 2.14.1: Solution of (2-14.a)

```
2  void CCS(const Eigen::MatrixXd& A, Eigen::VectorXd& val,
3           Eigen::VectorXd& row_ind, Eigen::VectorXd& col_ptr) {
4    // number of rows and columns
5    const unsigned int m = A.rows();
6    const unsigned int n = A.cols();
7
8    // TODO: (2-14.a) Implement a function that stores the matrix A
9    // in the vectors (val,row_ind,col_ptr) in CCS format
10   // Do not use Eigen methods for Sparse matrices.
11   // START
12
13   // number of nonzeroes in the matrix
14   unsigned int nnz = 0;
15   for (unsigned int i = 0; i < m; ++i) {
16     for (unsigned int j = 0; j < n; ++j) {
17       if (A(i, j) != 0) ++nnz;
18     }
19   }
20
21   // initialization
22   val.resize(nnz);
23   row_ind.resize(nnz);
24   col_ptr.resize(n);
25
26   // store A in CCS format
27   unsigned int index = 0;
28   for (unsigned int j = 0; j < n; ++j) {  // col iterator
29     // update 'col\_ptr'
30     col_ptr(j) = index;
31
32     for (unsigned int i = 0; i < m; ++i) {  // row iterator
33       if (A(i, j) != 0) {
34         // record the value to 'val'
35         val(index) = A(i, j);
36         // record row index to 'row\_ind'
37         row_ind(index) = i;
38         ++index;
39       }
40     }
41   }
42   // END
43 }
```

Get it on GitLab (sparseCCS.hpp).

SOLUTION of (2-14.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.

HINT 1 for (2-15.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. ⌟

HINT 2 for (2-15.a): Contemplate a multi-pass implementation: The code traverses the vector of triplets several times, extracting different pieces of information.

The ultimate goal should be a code whose cost scales linearly with the number of triplets assuming that the matrix has only a small fixed number of non-zero entries per row. ⌟

HINT 3 for (2-15.a): To begin with, you will have to determine the quantity `maxcols` based on `Eigen::Triplets<double> triplets` and the number of rows m. ⌟

SOLUTION of (2-15.a):

The implementation below first scans the triplet vector in order to find out about the maximum number of non-zero entries in a row of the matrix.

C++11-code 2.15.3: Solution of Sub-problem (2-15.a): constructor of EllpackMat

```
EllpackMat::EllpackMat(const Triplets &triplets, index_t m, index_t n)
    : m(m), n(n) {
  // TODO: (2-15.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);

  // counters especially for handling repeated index
  std::vector<std::vector<unsigned int>> counters_col(m);

  // '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 &tr : triplets) {
    std::vector<unsigned int> &col_idx = counters_col[tr.row()];
    // if it's not a repeated index, then check if maxcols needs to be updated
    if (std::find(col_idx.begin(), col_idx.end(), tr.col()) == col_idx.end()) {
      // bookkeep the col of new(not repeated) triplet
      col_idx.push_back(tr.col());
      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 &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();
```

```
50          break;
51        }
52
53        // if it's a repeated index, just sum up
54        if (col[l] == tr.col()) {
55          val[l] += tr.value();
56          break;
57        }
58      }
59      assert(l < (tr.row() + 1) * maxcols &&
60             "You did not reserve enough columns!");
61    }
62    // END
63  }
```

Get it on GitLab (ellpack.hpp).

HINT 1 for (2-15.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 (2-15.b):

C++11-code 2.15.4: Solution of Sub-problem (2-15.b)

```
2  void EllpackMat::mvmult(const Vector &x, Vector &y) const {
3    assert(x.size() == n && "Incompatible vector x size!");
4    assert(y.size() == m && "Incompatible vector y size!");
5    // TODO: (2-15.b) implement the multiplication A*x using the
6    // class EllpackMat, with optimal complexity.
7
8    // START
9    for (index_t i = 0; i < m; ++i) {
10     y(i) = 0;
11     for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
12       if (col[l] == -1) break;
13       y(i) += x(col[l]) * val[l];
14     }
15   }
16   // END
17 }
```

Get it on GitLab (ellpack.hpp).

HINT 1 for (2-15.c): The solution is very similar to that of Sub-problem (2-15.b). Think about which vector of $(\mathbf{A})_{i,:}$, $\mathbf{x}$ and $\mathbf{y}$ should be reordered using **std::vector**<index_t> col to obtain the correct result. ⌟

SOLUTION of (2-15.c):

C++11-code 2.15.5: Solution of (2-15.c)

```
2  void EllpackMat::mtvmult(const Vector &x, Vector &y) const {
3    assert(x.size() == m && "Incompatible vector x size!");
4    assert(y.size() == n && "Incompatible vector y size!");
5    // TODO: (2-15.c) implement the multiplication A^T * x using the
6    // class EllpackMat, with optimal complexity.
7
8    // START
9    y = Vector::Zero(n);
10   for (index_t i = 0; i < m; ++i) {
11     for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
12       if (col[l] == -1) break;
13       y(col[l]) += x(i) * val[l];
14     }
15   }
16   // END
17 }
```

Get it on GitLab (ellpack.hpp).

SOLUTION of (2-15.d):

Due to the use of **std**`::find`, which does a sequential search, the computational effort (here measured by the numbers of comparisons needed) of the constructor of **EllpackMat** implemented in Code 2.15.3 scales quadratically in `maxcols`. However, under the *assumption* that that number is fixed and small, the overall effort it linear in the number of triplets: $O(\sharp\texttt{triplets})$.

SOLUTION of (2-16.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.

SOLUTION of (2-16.b):

The matrix $\mathbf{A}$ is given by:

$$\mathbf{A} = \begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix}$$

SOLUTION of (2-16.c):

C++11-code 2.16.1: Evaluation function

```
2  void eval(Eigen::MatrixXd& X, std::function<double(index_t, index_t)> f) {
3    // TODO: (2-16.c) fill the matrix X with values from f.
4    // Hint:  do not forget the shift of indices in C++
5    // START
6    for (index_t i = 0; i < static_cast<index_t>(X.rows()); ++i) {
7      for (index_t j = 0; j < static_cast<index_t>(X.cols()); ++j) {
8        // Notice that we shift indices due to C++ indexing.
9        X(i, j) = f(i + 1, j + 1);
10     }
11   }
12   // END
13 }
```

Get it on GitLab (gridfun.hpp).

C++11-code 2.16.2: Definition of f

```
2  std::function<double(index_t, index_t)> define_f(unsigned int n,
3                                                    unsigned int m) {
4    // TODO: (2-16.c) Define a lambda function f that takes as input n,m
5    // and outputs a double
6    // START
7    auto f = [n, m](index_t i, index_t j) -> double {
8      if (i > n / 4 && i < n * 3 / 4 && j > m / 4 && j < m * 3 / 4)
9        return 1.;
10     else
11       return 0.;
12   };
13   return f;
14   // END
15 }
```

Get it on GitLab (gridfun.hpp).

HINT 1 for (2-16.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 (2-16.d):

C++ code 2.16.3: Auxiliary function `to_vector_index()`

```cpp
inline index_t to_vector_index(index_t i, index_t j, const shape_t& size) {
  index_t index = 0;
  // TODO: (hint in (2-16.d)) for a given pair of indices (i,j)
  // return the index of vec(X) corresponding to (X)_{i,j}
  // START
  index = size[1] * i + j;
  // END
  return index;
}
```

C++11-code 2.16.4: Build matrix $\mathbf{A}$

```cpp
Eigen::SparseMatrix<double> build_matrix(const Eigen::Matrix3d& S,
                                         const shape_t& size) {
  // Will be used to store triplets
  std::vector<Eigen::Triplet<double>> triplets;
  // The matrix A will have size N x N.
  // N = n*m
  const index_t N = size[0] * size[1];

  // Now, construct A.
  Eigen::SparseMatrix<double> A(N, N);

  // TODO: (2-16.d) store the matrix in COO format in the vector "triplets"
  // then convert triplets to CRS/CCS format using the functions
  // setFromTriplets() 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) {
```

```
41          for (index_t l = 0; l < 3; ++l) {
42            if (i + k - 1 >= 0 && i + k - 1 <= size[0] - 1 && j + l - 1 >= 0 &&
43                j + l - 1 <= size[1] - 1) {
44              // The index of vec(X) w.r.t
45              // the indices (i,j) of X
46              const index_t I = to_vector_index(i, j, size);
47              // Index on vec(X) relative to index
48              // (i+k-2,j+l-2)
49              // NOTE: C++ indexing starts from 0, so we change the
50              // offset.
51              const index_t J = to_vector_index(i + k - 1, j + l - 1, size);
52              // Add triplet of corresponding stencil matrix entry
53              triplets.push_back(Eigen::Triplet<double>(J, I, S(k, l)));
54            }
55          }
56        }
57      }
58    }
59
60    // Build from triplets and compress matrix (no need to compress through
61    // makeCompressed(); already included in setFromTriplets()).
62    A.setFromTriplets(triplets.begin(), triplets.end());
63    // END
64
65    return A;
66  }
```

Get it on GitLab (gridfun.hpp).

SOLUTION of (2-16.e):

C++11-code 2.16.5: Multiplication with $\mathbf{A}$

```
2  void mult(const Eigen::SparseMatrix<double>& A, const Eigen::MatrixXd& X,
3            Eigen::MatrixXd& Y) {
4    // Size check
5    assert(A.rows() == X.rows() * X.cols() && A.cols() == X.rows() * X.cols() &&
6           "Inconsistent size of A!");
7    // Ensure correct size of Y.
8    Y.resizeLike(X);
9    // TODO: (2-16.e) compute vec(Y) = A*vec(X)
10   // Hint: Use Eigen::Map to reshape X into vec(X).
11
12   // START
13   // The second "const" here is needed to "promise" X will not be modified.
14   const Eigen::Map<const Eigen::VectorXd> x(X.data(), X.size());
15   // Notice how "Map" can also be written onto.
16   Eigen::Map<Eigen::VectorXd>(Y.data(), X.size()) = A * x;
17   // END
18 }
```

Get it on GitLab (gridfun.hpp).

SOLUTION of (2-16.f):

C++11-code 2.16.6: g
ridfun.hpp Solving the linear system

```
2  void solve(const Eigen::SparseMatrix<double>& A, const Eigen::MatrixXd& Y,
3             Eigen::MatrixXd& X) {
4    // Size check
5    assert(A.rows() == Y.rows() * Y.cols() && A.cols() == Y.rows() * Y.cols() &&
6           "Inconsistent size of A!");
7    // Ensure correct size of X.
8    X.resizeLike(Y);
9    // TODO: (2-16.f) compute X such that vec(Y) = A * vec(X) using a
10   // sparse solver in Eigen.
11   // START
12   // Sparse LU decomposition object
13   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
14
15   // Prepare LU factorization
16   solver.analyzePattern(A);
17   solver.factorize(A);
18
19   // Check if factorization failed.
20   if (solver.info() != Eigen::Success) {
21     std::cerr << "Fatal: LU decomposition of A failed!!!" << std::endl
22               << "Is the matrix invertible?" << std::endl;
23     return;
24   }
25
26   // Solve system after factorization.
27   // Notice how you can assign to a Map object.
28   Eigen::Map<Eigen::VectorXd>(X.data(), Y.size()) =
29       solver.solve(Eigen::Map<const Eigen::VectorXd>(Y.data(), Y.size()));
30   // END
31 }
```

Get it on GitLab (gridfun.hpp).

HINT 1 for (2-17.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 (2-17.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.

SOLUTION of (2-17.b):

C++11-code 2.17.1: Solution of (2-17.b)

```
2  TripVec Mat2COO(const Eigen::MatrixXd& A) {
3    TripVec triplets;
4    const unsigned int m = A.rows();
5    const unsigned int n = A.cols();
6
7    // TODO: (2-17.b) Convert A into a sparse matrix in COO format.
8    // START
9
10   // of course, if nnz(A) can be estimated a priori, space should be
       reserved in
11   // triplets
12   for (unsigned int i = 0; i < m; ++i) {
13     for (unsigned int j = 0; j < n; ++j) {
14       if (A(i, j) != 0) {
15         Trip triplet(i, j, A(i, j));
16         triplets.push_back(triplet);
17       }
18     }
19   }
20   // END
21   return triplets;
22 }
```

Get it on GitLab (matmatCOO.hpp).

HINT 1 for (2-17.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 (2-17.c):

C++11-code 2.17.2: Solution of (2-17.c)

```
TripVec COOprod_naive(const TripVec& A, const TripVec& B) {
  TripVec C;

  // TODO: (2-17.c) Compute the matrix-matrix product, naive version.
  // START
  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);
      }
    }
  }
  // END
  return C;
}
```

Get it on GitLab (matmatCOO.hpp).

HINT 1 for (2-17.d): Assume that there are no duplicates in the input triplet vectors of A and B. ⌟

SOLUTION of (2-17.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.

HINT 1 for (2-17.e): Is there a way to avoid the double loop of (2-17.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 (2-17.e):

C++11-code 2.17.3: Solution of (2-17.e)

```
TripVec COOprod_effic(TripVec& A, TripVec& B) {
  TripVec C;

  // TODO: (2-17.e) Compute the matrix-matrix product efficiently
  // START

  // sort A along columns
  std::sort(A.begin(), A.end(),
            [](const Trip& a1, const Trip& a2) { return a1.col() < a2.col(); });
  // and B along rows
  std::sort(B.begin(), B.end(),
            [](const Trip& b1, const Trip& b2) { return b1.row() < b2.row(); });

  std::size_t i_A = 0, i_B = 0;
  std::set<unsigned 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());
      ++i_A;
      ++i_B;
    }
  }

  // iteration along intersection of nonempty columns in A and rows in 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 {
            // actual multiplication
            Trip triplet(A_it->row(), B_it->col(),
                         A_it->value() * B_it->value());
            C.push_back(triplet);
          }
        }
```

```
54          }
55        }
56        A_idx = A_it;
57        B_idx = B_it;
58      }
59      // END
60      return C;
61  }
```

Get it on GitLab (matmatCOO.hpp).

HINT 1 for (2-17.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 (2-17.f):

We assume that there are no duplicates in the input triplet vectors.

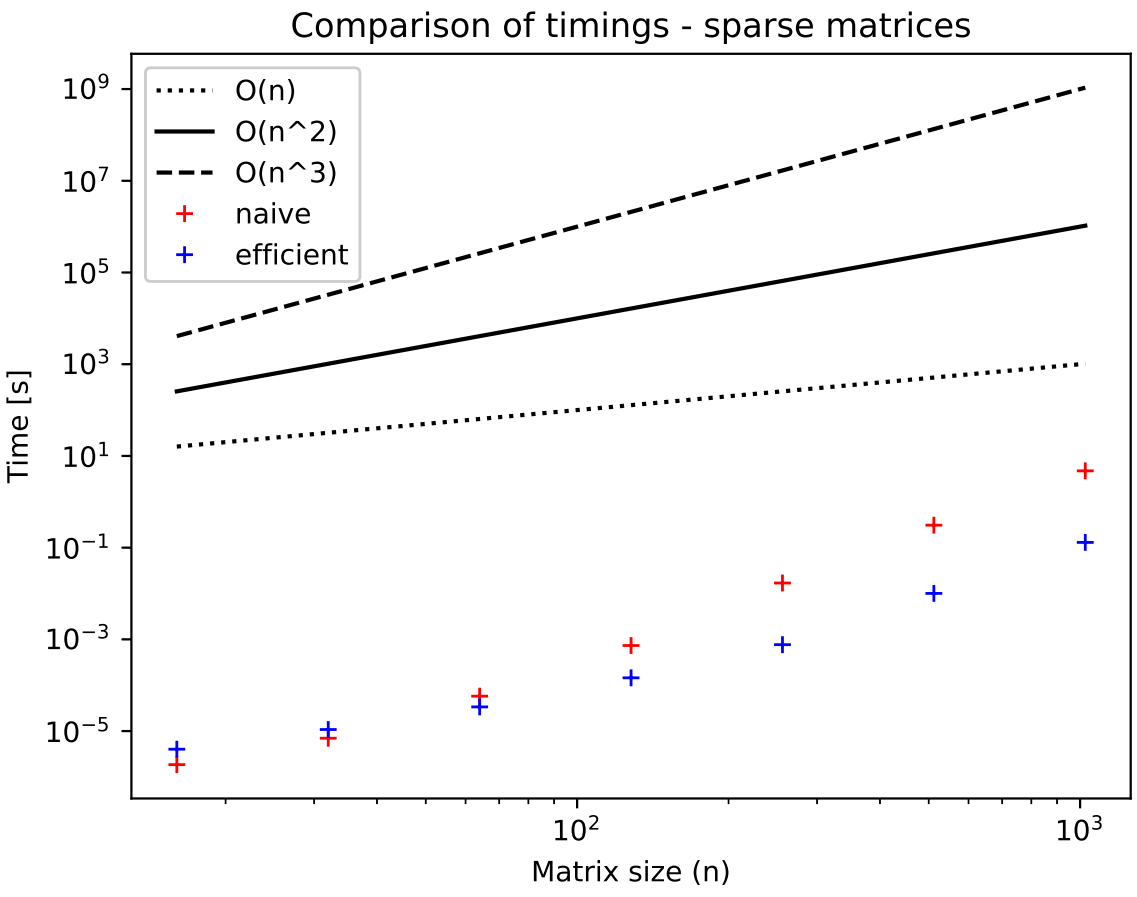
1. Sorting triplet vectors A and $B \implies O(n \cdot \log(n))$.
2. Finding the interception of common indices $k \implies O(n)$ (having already sorted both A and B).
3. Iterating along the intersection to obtain all possible triplets $a_{ik}b_{kj} \implies$ This is proportional to the product between 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.

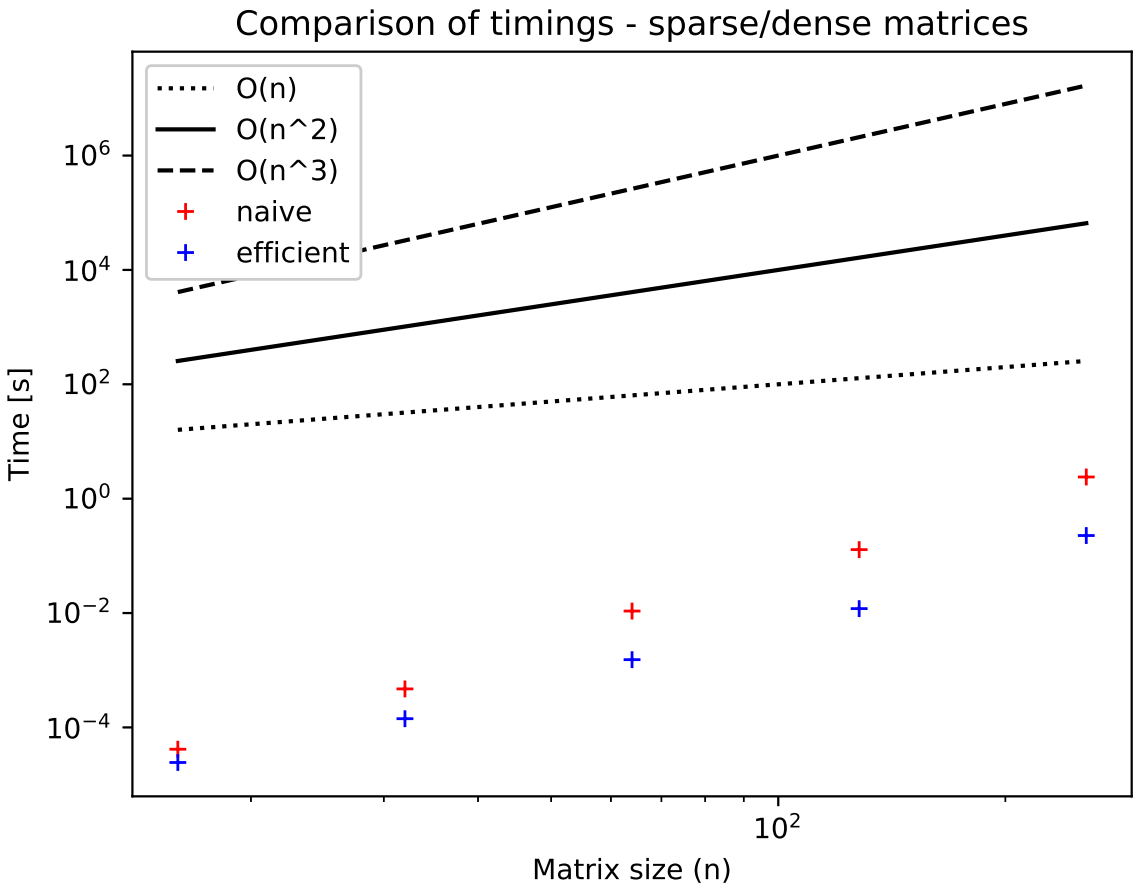
   k_{common} is certainly greater than the number of nonzero entries in matrix AB. However, $\tilde{n}$ depends on the sparsity pattern of the input matrices. In any case, as O should be an upper bound, the complexity is $O\left(k_{common}n^2\right)$.

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.

HINT 1 for (2-17.g): The `main()` code to run your simulations is already provided in the template. ⌟

SOLUTION of (2-17.g):





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 (2-17.a). However, the efficient algorithm is generally better when considering any kind of matrix.

SOLUTION of (2-18.a):

The only obstruction to (2.18.1) being well-defined is a failure of the division due to $(\mathbf{A})_{i,i} = 0$. Therefore the Gauss-Seidel iteration is well defined for all matrices, for which all diagonal entries are non-zero.

iteration (2.18.1) well-defined $\forall \mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$: $(\mathbf{A})_{i,i} \neq 0 \quad \forall i \in \{1, \ldots, n\}$.

SOLUTION of (2-18.b):

Let us first develop a code for the Gauss-Seidel iteration for a generic matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, assuming that we can access any entry:

C++ code 2.18.2: EIGEN-based implementation of `GaussSeidelstep_generic()`

```
template <typename MATRIX>
bool GaussSeidelstep_generic(const MATRIX &A, const Eigen::VectorXd &b,
                             Eigen::VectorXd &x) {
  const unsigned int n = A.cols();
  assert(n == A.rows() && "Matrix must be square");
  assert(n == b.size() && "Vector length mismatch");
  assert(n == x.size() && "Vector b length mismatch");

  for (unsigned int i = 0; i < n; ++i) {
    if (A(i, i) == 0.0) return false;
    double s = b[i];
    for (unsigned int j = 0; j < i; ++j) s -= A(i, j) * x[j];
    for (unsigned int j = i + 1; j < n; ++j) s -= A(i, j) * x[j];
    x[i] = s / A(i, i);
  }
  return true;
}
```

Get it on GitLab (auxiliary_functions.hpp).

From this code we see that the entries of $\mathbf{A}$ are accessed *row-wise sequentially*. This is exactly their arrangement in the `val` field of **CRSMatrix**, which suggests an implementation based on an outer loop running through the `val` and `col_ind` vectors.

Also remember that the CRS-format for $\mathbf{A} \in \mathbb{R}^{n,n}$ is defined by the relationship ("C++ indexing")

$$\texttt{val}[k] = (\mathbf{A})_{i,j} \quad \Leftrightarrow \quad \begin{cases} \texttt{col\_ind}[k] = j\,, \\ \texttt{row\_ptr}[i] \leq \texttt{k} < \texttt{row\_ptr}[i+1]\,, \end{cases} \qquad 0 \leq k \leq \texttt{val.size()}\,,$$

for $i, j \in \{0, \ldots, n-1\}$. Therefore the index bounds stored in `row_ptr` can be used in an inner loop.

C++ code 2.18.3: EIGEN-based implementation of `GaussSeidelstep_crs()`

```
bool GaussSeidelstep_crs(const CRSMatrix &A, const Eigen::VectorXd &b,
                         Eigen::VectorXd &x) {
  assert(A.n == A.m && "Matrix must be square");
  assert(A.n == b.size() && "Vector b length mismatch");
  assert(A.n == x.size() && "Vector x length mismatch");

  // TODO: (2-18.b) Implement a single step of the Gauss-Seidel
     iteration with
  // the system matrix in CRS format.
  // START

  // Outer loop over rows of the matrix
  for (unsigned int i = 0; i < A.n; ++i) {
    double Aii = 0.0;
    double s = b[i];
    // Inner summation loop over non-zero entries of i-th row.
    // Skip diagonal entry in the summation and store is separately.
```

```
18      for (unsigned int l = A.row_ptr[i]; l < A.row_ptr[i + 1]; ++l) {
19        const unsigned int j = A.col_ind[l];
20        if (j != i) {
21          s -= A.val[l] * x[j];
22        } else {
23          // Fetch diagonal entry of A.
24          Aii = A.val[l];
25        }
26      }
27      if (Aii != 0.0)
28        x[i] = s / Aii;
29      else
30        return false;
31    }
32    // END
33    return true;
34  }
```

Get it on GitLab (gaussseidelcrs.hpp).

SOLUTION of (2-18.c):

$\mathbf{x}^*$ is a fixed point of (2.18.1), if $\mathbf{x}^{(k)} = \mathbf{x}^*$ implies $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$, which for (2.18.1) means

$$(\mathbf{x}^*)_i = (\mathbf{A})_{i,i}^{-1}\left((\mathbf{b})_i - \sum_{j=1}^{i-1}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j - \sum_{j=i+1}^{n}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j\right), \quad i = 1,2,\ldots,n$$

$$\Leftrightarrow \quad (\mathbf{A})_{i,i}(\mathbf{x}^*)_i = (\mathbf{b})_i - \sum_{j=1}^{i-1}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j - \sum_{j=i+1}^{n}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j, \quad i = 1,2,\ldots,n$$

$$\Leftrightarrow \quad \sum_{j=1}^{n}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j = (\mathbf{b})_i, \quad i = 1,2,\ldots,n$$

$$\Leftrightarrow \quad \mathbf{A}\mathbf{x}^* = \mathbf{b} \quad \mathbf{!}.$$

Hence,

$$\{\mathbf{x}^* \in \mathbb{R}^n : \mathbf{x}^* \text{ is fixed point of (2.18.1)}\} = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} = \mathbf{b}\},$$

or, equivalently, since $\mathbf{A}$ is regular

$$\{\mathbf{x}^* \in \mathbb{R}^n : \mathbf{x}^* \text{ is fixed point of (2.18.1)}\} = \left\{\mathbf{A}^{-1}\mathbf{b}\right\}. \tag{2.18.4}$$

We conclude that there is a unique fixed point of (2.18.1), which coincides with the solution of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$.

SOLUTION of (2-18.d):

The correction-based termination criterion reads:

$$\text{STOP, when } \left\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\right\|_2 \leq \texttt{rtol}\left\|\mathbf{x}^{(k+1)}\right\|_2 \quad \text{or} \quad \left\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\right\|_2 \leq \texttt{atol}\,. \tag{2.18.5}$$

Therefore we have to store the previous iterate temporarily. This is done in the following code, which relies on `GaussSeidelstep_crs` from Sub-problem (2-18.b).

C++ code 2.18.6: EIGEN-based implementation of `GaussSeidel_iteration()`

```
2  template <typename RECORDER = std::function<void(double, double)>>
3  bool GaussSeidel_iteration_II(
4      const CRSMatrix &A, const Eigen::VectorXd &b, Eigen::VectorXd &x,
5      double atol = 1.0E-8, double rtol = 1.0E-6, unsigned int maxit = 100,
6      RECORDER &&rec = [](double, double) -> void {}) {
7    assert(A.n == A.m && "Matrix must be square");
8    assert(A.n == b.size() && "Vector b length mismatch");
9    assert(A.n == x.size() && "Vector x length mismatch");
10
11   // Main loop for Gauss-Seidel iteration
12   Eigen::VectorXd x_old{x};
13   for (unsigned int k = 0; k < maxit; ++k) {
14     // TODO: (2-18.d; optional) Second function to implement the
         Gauss-Seidel
15     // iteration, not required to implement.
16     // START
17     if (!GaussSeidelstep_crs(A, b, x)) return false;
18     const double delta_x_norm = (x - x_old).norm();
19     const double x_norm = x.norm();
20     rec(x_norm, delta_x_norm);  //
21     if ((delta_x_norm < atol) ||
22         (delta_x_norm < rtol * x_norm)) {  //
23       return true;
24     }
25     x_old = x;
26     // END
27   }
28   // No termination
29   return false;
30 }
```

Get it on GitLab (auxiliary_functions.hpp).

The stopping rule (2.18.5) is implemented in Line 22.

This code also takes a recorder object as an argument to keep track of the norms of the iterates and of the updates. This is an add-on, which was not requested in this sub-problem.

A more efficient implementation that dispenses with temporary vectors is given next.

C++ code 2.18.7: EIGEN-based implementation of `GaussSeidel_iteration()`

```
2  bool GaussSeidel_iteration(const CRSMatrix &A, const Eigen::VectorXd &b,
3                             Eigen::VectorXd &x, double atol = 1.0E-8,
4                             double rtol = 1.0E-6, unsigned int maxit = 100) {
5    assert(A.n == A.m && "Matrix must be square");
```

```
  assert(A.n == b.size() && "Vector b length mismatch");
  assert(A.n == x.size() && "Vector x length mismatch");

  // Main loop for Gauss-Seidel iteration
  for (unsigned int k = 0; k < maxit; ++k) {
    // TODO: (2-18.d) Implement the Gauss-Seidel iteration with a
    // correction-based termination criterion.
    // START
    double x_normsq = 0.0;
    double delta_x_normsq = 0.0;
    for (unsigned int i = 0; i < A.n; ++i) {
      double Aii = 0.0;
      double s = b[i];
      for (unsigned int l = A.row_ptr[i]; l < A.row_ptr[i + 1]; ++l) {
        const unsigned int j = A.col_ind[l];
        if (j != i) {
          s -= A.val[l] * x[j];
        } else {
          // Fetch diagonal entry of A
          Aii = A.val[l];
        }
      }
      if (Aii != 0.0) {
        // Update iterate and compute Eucidean norm of current iterate
        // and of correction
        const double tmp = x[i];
        x[i] = s / Aii;
        delta_x_normsq += (x[i] - tmp) * (x[i] - tmp);
        x_normsq += x[i] * x[i];
      } else
        return false;
    }
    // Correction based termination cirterion
    if ((delta_x_normsq < atol * atol) ||
        (delta_x_normsq < rtol * rtol * x_normsq)) {
      return true;
    }
    // END
  }
  // No termination
  return false;
}
```

Get it on GitLab (gaussseidelcrs.hpp).

SOLUTION of (2-18.e):

From the data we conclude that the iteration converges and that $\mathbf{x}^{(k)} \to \mathbf{0}$.

According to (2.18.4) for $\mathbf{b} = \mathbf{0}$ the only fixed point of the iteration will be $\mathbf{x}^* = \mathbf{0}$. Hence, also the iteration error $\mathbf{x}^{(k)} - \mathbf{x}^*$ is given by $\mathbf{x}^{(k)}$. From the last column of the table we see that for sufficiently large k (this is the meaning of "asymptotic" in the problem statement)

$$\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\|_2 = \left\|\mathbf{x}^{(k+1)}\right\|_2 \leq 0.18\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| = 0.18\left\|\mathbf{x}^{(k)}\right\|_2 ,$$

which means that the iteration *converges asymptotically linearly* with rate ≈ 0.18.

SOLUTION of (2-19.j):

C++ code 2.19.12: Solving Sylvester equation (2.19.4).

```
Eigen::MatrixXd solveSpecialSylvesterEq(const Eigen::SparseMatrix<double> &A) {
  const int n = A.rows();
  assert(n == A.cols() && "Matrix A must be square.");
  Eigen::MatrixXd X(n, n);

  // TODO: (2-19.j) 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.
  Eigen::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"!
  Eigen::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.
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  // Since C is s.p.d., we could also use
  // SimplicialLLT<Eigen::SparseMatrix<double>> solver;
  solver.compute(C);
  // x = Vec(X)
  Eigen::VectorXd x;
  x = solver.solve(b);
  X = Eigen::MatrixXd::Map(x.data(), n, n);
  // END

  return X;
}
```

Get it on GitLab (Sylvester.hpp).

SOLUTION of (2-19.b):

The vectorized matrix is obtained by stacking the columns of the matrix on top of each other:



$$\text{vec}(\mathbf{X}) = [x_{11}, x_{21}, x_{12}, x_{22}]^\top \in \mathbb{R}^4 \, .$$

SOLUTION of (2-19.c):

We obtain the k-th 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

$$\begin{aligned}\mathbf{C}_{:,1} &= \mathrm{vec}\left(\begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}\right) = \mathrm{vec}\left(\begin{bmatrix} s_{11} & 0 \\ s_{21} & 0 \end{bmatrix}\begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}\right) \\ &= \mathrm{vec}\left(\begin{bmatrix} s_{11}t_{11} & s_{11}t_{12} \\ s_{21}t_{11} & s_{21}t_{12} \end{bmatrix}\right) = \begin{bmatrix} s_{11}t_{11} \\ s_{21}t_{11} \\ s_{11}t_{12} \\ s_{21}t_{12} \end{bmatrix} .\end{aligned}$$

Thus we find

$$\mathbf{C} = \begin{bmatrix} s_{11}t_{11} & s_{12}t_{11} & s_{11}t_{21} & s_{12}t_{21} \\ s_{21}t_{11} & s_{22}t_{11} & s_{21}t_{21} & s_{22}t_{21} \\ s_{11}t_{12} & s_{12}t_{12} & s_{11}t_{22} & s_{12}t_{22} \\ s_{21}t_{12} & s_{22}t_{12} & s_{21}t_{22} & s_{22}t_{22} \end{bmatrix} = \mathbf{T}^\top \otimes \mathbf{S} \ .$$

HINT 1 for (2-19.d): Read .

SOLUTION of (2-19.d):

By the rules of matrix multiplication for $\mathbf{S}, \mathbf{X}, \mathbf{T} \in \mathbb{R}^{n,n}$:

$$(\mathbf{AXT})_{i,j} = \sum_{\ell=1}^{n} \sum_{\nu=1}^{n} s_{i,\ell} x_{\ell,\nu} t_{\nu,j} \,, \tag{2.19.1}$$

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 (2.19.1) we find

$$(\mathbf{S}\mathbf{E}_{k,m}\mathbf{T})_{i,j} = s_{i,k} t_{m,j} \,. \tag{2.19.2}$$

Write $P(i,j) \in \{1,\ldots,n^2\}$ for the **p**osition of the matrix entry $(\mathbf{Z})_{i,j}$ in the vector $\mathrm{vec}(Z)$, $\mathbf{Z} \in \mathbb{R}^{n,n}$, $i,j \in \{1,\ldots,n\}$. Hence, from (2.19.2) we conclude

$$(\mathbf{C})_{P(i,j),P(k,m)} = s_{i,k} t_{m,j} = \left(\mathbf{T}^\top \otimes \mathbf{S}\right)_{P(i,j),P(k,m)} \quad \Leftrightarrow \quad \boxed{\mathbf{C} = \mathbf{T}^\top \otimes \mathbf{S}} \,. \tag{2.19.3}$$

SOLUTION of (2-19.e):

According to (2.19.3) the coefficient matrices $\mathbf{C} \in \mathbb{R}^{n^2,n^2}$ are ($\mathbf{I} \in \mathbb{R}^{n,n}$ the identity matrix)

$$\text{For (2.19.4):} \quad \mathbf{C}_4 = \mathbf{A}^{-1} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{A} \ , \tag{2.19.6}$$

$$\text{For (2.19.5):} \quad \mathbf{C}_5 = \mathbf{I} \otimes \mathbf{I} + \mathbf{A} \otimes \mathbf{A} \ . \tag{2.19.7}$$

Here we have made use of the symmetry of $\mathbf{A}$.

HINT 1 for (2-19.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 (2-19.f):

By the properties of $\mathbf{A}$ we know

$$\operatorname{nnz}(\mathbf{A}) \leq 5n \ .$$

However, as explained in the hint, $\operatorname{nnz}(\mathbf{A}^{-1}) = n^2$ is possible. Moreover, for the Kronecker product we have the simple estimate

$$\operatorname{nnz}(\mathbf{S} \otimes \mathbf{T}) \leq \operatorname{nnz}(\mathbf{S}) \cdot \operatorname{nnz}(\mathbf{T}) \ . \tag{2.19.8}$$

Combining all these estimates gives us

$$\operatorname{nnz}(\mathbf{A}^{-1} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{A}) \leq n^3 + 5n^2 \ ,$$
$$\operatorname{nnz}(\mathbf{I} \otimes \mathbf{I} + \mathbf{A} \otimes \mathbf{A}) \leq 26n^2 \ .$$

Hence, (2.19.5) features a much sparser coefficient matrix than (2.19.4).

HINT 1 for (2-19.g): In linear algebra we learned that symmetric matrices can always be diagonalized.
⌟

HINT 2 for (2-19.g): Remember [Lecture → Lemma 1.1.2.7].

SOLUTION of (2-19.g):

By the very definition of the Kronecker product, both matrices $\mathbf{C}_4$ and $\mathbf{C}_5$ from (2.19.6) and (2.19.7), 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. 2.19.9 this also holds for $\mathbf{C}_4$ and $\mathbf{C}_5$. Thus, by the third criterion from [Lecture $\rightarrow$ Lemma 1.1.2.7], these matrices are s.p.d.

SOLUTION of (2-19.h):

If $\mathbf{A}$ is s.p.d. and diagonal with diagonal entries $d_\ell = (\mathbf{A})_{\ell,\ell} > 0$, then (2.19.4) becomes

$$(d_i + d_j^{-1})(\mathbf{X})_{i,j} = \delta_{i,j}\,, \quad i,j \in \{1,\ldots,n\}\,.$$

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 2.19.10: Solving a "diagonal" Sylvester equation

```
2  template <typename Vector>
3  Eigen::SparseMatrix<double> solveDiagSylvesterEq(const Vector &diagA) {
4    int n = diagA.size();
5    Eigen::SparseMatrix<double> X(n, n);
6
7    // TODO: (2-19.h) Fill in the entries of X.
8    // Don't forget to use makeCompressed().
9
10   // START
11   // As X is a diagonal matrix, we reserve space
12   // for one non-zero entry per column.
13   X.reserve(Eigen::VectorXi::Constant(n, 1));
14   for (int j = 0; j < n; j++) {
15     // Set diagonal entry X_{j,j} = d_j / (d_j^2 + 1)
16     X.insert(j, j) = diagA[j] / (diagA[j] * diagA[j] + 1.0);
17   }
18   X.makeCompressed();
19   // END
20
21   return X;
22 }
```

Get it on GitLab (Sylvester.hpp).

SOLUTION of (2-19.i):

The comments in Code 2.19.11 explain the details of the algorithm.

C++ code 2.19.11: Computing $\mathbf{M} \otimes \mathbf{M}$ for a sparse matrix $\mathbf{M}$

```
Eigen::SparseMatrix<double> sparseKron(const Eigen::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) )
  Eigen::VectorXi 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.
  Eigen::SparseMatrix<double> B(n * n, n * n);

  // TODO: (2-19.i) 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

  Eigen::VectorXi nnzColwiseB;
  // We have that nnz(B.col(j*n+l)) = nnz(M.col(j)) * nnz(M.col(l)).
  Eigen::MatrixXi tmp = nnzColwiseM * nnzColwiseM.transpose();  // O(n^2)
  nnzColwiseB = Eigen::MatrixXi::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.
```

```
52        // Loop over the column index l:
53        for (int l = 0; l < n; l++) {
54          // Loop over the non-zero entries in column l:
55          for (int t = JM[l]; t < JM[l + 1]; t++) {
56            int k = IM[t];          // Row index of non-zero entry #t.
57            double Mkl = valM[t];   // M(k,l) = Mkl
58            // Insert entry (k,l) of block (i,j).
59            B.insert(i * n + k, j * n + l) = Mij * Mkl;
60          }
61        }
62      }
63    }
64    B.makeCompressed();
65    // END
66
67    return B;
68  }
```

Get it on GitLab (Sylvester.hpp).

HINT 1 for (2-2.a): Draw the “pattern” of the matrix, that is, the positions of its non-zero entries, e.g., for $n = 3$

$$\mathbf{A} = \begin{bmatrix} * & 0 & 0 & * & 0 & 0 \\ 0 & * & 0 & 0 & * & 0 \\ 0 & 0 & * & 0 & 0 & * \\ * & 0 & 0 & * & 0 & 0 \\ 0 & * & 0 & 0 & * & 0 \\ 0 & 0 & * & 0 & 0 & * \end{bmatrix}.$$

Which rows and columns are “coupled” through non-zero entries? ⌟

SOLUTION of (2-2.a):

Let us first look at the case $n = 2$:

$$\mathbf{A} = \begin{bmatrix} d_1 & 0 & c_1 & 0 \\ 0 & d_2 & 0 & c_2 \\ c_1 & 0 & d_3 & 0 \\ 0 & c_2 & 0 & d_4 \end{bmatrix}.$$

We employ the following permutation of rows and columns: $1 \leftarrow 1, 2 \leftarrow 3, 3 \leftarrow 2, 4 \leftarrow 4$ and end up with the rearranged matrix

$$\begin{bmatrix} d_1 & 0 & c_1 & 0 \\ 0 & d_2 & 0 & c_2 \\ c_1 & 0 & d_3 & 0 \\ 0 & c_2 & 0 & d_4 \end{bmatrix} \xrightarrow{\text{row pmt.}} \begin{bmatrix} d_1 & 0 & c_1 & 0 \\ c_1 & 0 & d_3 & 0 \\ 0 & d_2 & 0 & c_2 \\ 0 & c_2 & 0 & d_4 \end{bmatrix} \xrightarrow{\text{col pmt.}} \begin{bmatrix} d_1 & c_1 & 0 & 0 \\ c_1 & d_3 & 0 & 0 \\ 0 & 0 & d_2 & c_2 \\ 0 & 0 & c_2 & d_4 \end{bmatrix}.$$

Obviously, 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.

In fact, we observe that the resulting tridiagonal matrix is rather special: It is a **block-diagonal** matrix, whose diagonal blocks are of size 2×2!

SOLUTION of (2-2.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 efficiently be implemented in C++.

C++ code 2.2.2: Implementation of `multA()`

```
2  Eigen::VectorXd multA(const Eigen::VectorXd& d1, const Eigen::VectorXd& d2,
3                        const Eigen::VectorXd& c, const Eigen::VectorXd& x) {
4    const unsigned int n = d1.size();
5    assert(n == d2.size() && n == c.size() && 2 * n == x.size() &&
6           "Size mismatch!");
7    Eigen::ArrayXd y = Eigen::VectorXd::Zero(2 * n);
8
9    // TODO: (2-2.b) Efficiently calculate A*x.  Eigen's array arithmetic
       is
10   // helpful here.
11   // START
12   auto& x1 = x.head(n).array();
13   auto& x2 = x.tail(n).array();
14
15   y << d1.array() * x1 + c.array() * x2, d2.array() * x2 + c.array() * x1;
16   // END
17   return y;
18 }
```

Get it on GitLab (blocklsepiv.hpp).

HINT 1 for (2-2.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 (2-2.c):

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

$$\mathbf{A} = \begin{bmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{C} & \mathbf{D}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{D}_1^{-1}\mathbf{C} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{0} & \mathbf{D}_2 - \mathbf{C}\mathbf{D}^{-1}\mathbf{C} \end{bmatrix} = \mathbf{L}\mathbf{U}, \tag{2.2.3}$$

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.

SOLUTION of (2-2.d):

As we see from the explicit formula (2.2.3), an LU-decomposition exists, **if and only if** $\mathbf{D}_1$ is regular, that is, if and only if the diagonal entries of $\mathbf{D}_1$ are all different from 0.

HINT 1 for (2-2.e): Examples can already be found for $n = 2$ ⌟

SOLUTION of (2-2.e):

Consider the matrices

$$\mathbf{A}_1 := \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad , \quad \mathbf{A}_2 := \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} .$$

- Matrix $\mathbf{A}_1$ is obviously singular, because its columns/rows are linearly dependent.
- Matrix $\mathbf{A}_2$ is invertible, because it has linearly independent rows.

HINT 1 for (2-2.f): Recall the observation made in Sub-problem (2-2.a), which permits you to reduce everything to the solution of n 2×2 linear systems of equations. ⌟

SOLUTION of (2-2.f):

Drawing on what we learned in Sub-problem (2-2.a), namely that only rows/column k and $n+k$ of $\mathbf{A}$, $k = 1, \ldots, n$ are coupled through non-zero entries, we see that the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ where $\mathbf{A}$ has the structure as in (2.2.1), is equivalent to

$$\begin{bmatrix} (\mathbf{d}_1)_k & (\mathbf{c})_k \\ (\mathbf{c})_k & (\mathbf{d}_2)_k \end{bmatrix} \begin{bmatrix} (\mathbf{x})_k \\ (\mathbf{x})_{n+k} \end{bmatrix} = \begin{bmatrix} (\mathbf{b})_k \\ (\mathbf{b})_{n+k} \end{bmatrix}, \quad k = 1, \ldots, n\,. \tag{2.2.4}$$

To solve the 2×2 linear systems (2.2.4) you may rely on EIGEN's direct elimination solvers as discussed in [Lecture → § 2.5.0.4].

Remark. In light of the advice given in the beginning of [Lecture → Section 2.5] it is even advisable to use library routines even for 2×2 linear systems, also for the sake of numerical stability.

To test for near singularity of a 2×2-matrix we can rely on the angle enclosed by its columns. For a $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2] \in \mathbb{R}^{2,2}$ we check

$$\left| \frac{\mathbf{x}_1^\top \mathbf{x}_2}{\|\mathbf{x}_1\|_2 \|\mathbf{x}_2\|_2} \right| \leq 1 - \mathrm{EPS}\,.$$

If that fraction attains a value ± 1, the columns $\mathbf{x}_1$, $\mathbf{x}_2$ of the matrix $\mathbf{X}$ are linearly dependent. Alternatively, you may check for a relatively small determinant

$$|\det \mathbf{X}| = |(\mathbf{X})_{11}(\mathbf{X})_{22} - (\mathbf{X})_{12}(\mathbf{X})_{21}| \leq \mathrm{EPS} \cdot \|\mathbf{x}_1\|_2 \|\mathbf{x}_2\|\,.$$

Here, inspecting the relative size of the determinant is crucial, recall [Lecture → Rem. 1.5.3.15].

C++ code 2.2.5: Implementation of `solveA()`

```
2  Eigen::VectorXd solveA(const Eigen::VectorXd& d1, const Eigen::VectorXd& d2,
3                         const Eigen::VectorXd& c, const Eigen::VectorXd& b) {
4    const unsigned int n = d1.size();
5    assert(n == d2.size() && n == c.size() && 2 * n == b.size() &&
6           "Size mismatch!");
7    Eigen::VectorXd x = Eigen::VectorXd::Zero(2 * n);
8    constexpr double eps = std::numeric_limits<double>::epsilon();
9
10   // TODO: (2-2.d) Solve Ax = b for x using Gaussian elimination without
11   // directly envoking Eigen's solvers on the full matrix
12   // START
13   Eigen::Matrix2d A;
14   Eigen::Vector2d x_, b_;
15   for (unsigned int k = 0; k < n; ++k) {
16     A << d1(k), c(k), c(k), d2(k);
17     // check for singularity
18     if (std::abs(A.col(0).dot(A.col(1)) / (A.col(0).norm() * A.col(1).norm())) >
19         1. - eps) {
20       std::cerr << "Warning: matrix nearly singular!" << std::endl;
21     }
22     b_ << b(k), b(n + k);
23     x_ = A.fullPivLu().solve(b_);
24     x(k) = x_(0);
25     x(n + k) = x_(1);
```

```
26      }
27      // END
28
29      return x;
30  }
```

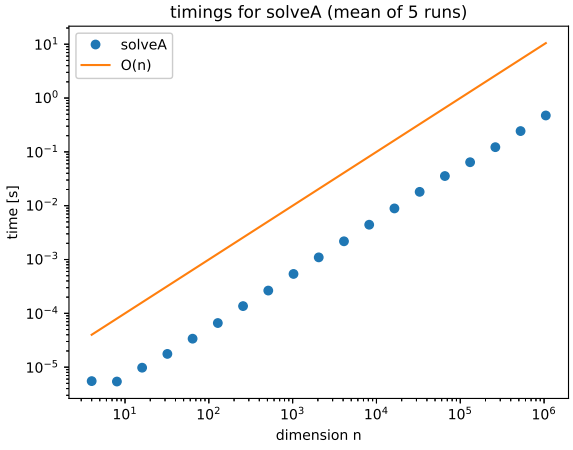
Get it on GitLab (blockIsepiv.hpp).

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

SOLUTION of (2-2.g):

In the implementation of `solveA()` we sequentially solve n small linear systems of size 2×2, which amounts to spending n times a constant effort. Thus the asymptotic complexity is $O(n)$ for $n \to \infty$.

SOLUTION of (2-2.h):



As a clear indication for linear complexity the data points should be approximately located on a straight line, as in Fig. 5.
The runtime measurements were performed using an Intel(R) Core(TM) i7 (I7-9750H) CPU @ 2.6 GHz compiled using Apple Clang 14 and the default setup of the NumCSE build system.

C++11-code 2.2.6: Solution of (2-2.h)

```
2  void numericalExperiment() {
3    constexpr unsigned int repeats = 5;
4    Eigen::VectorXd d1, d2, c, b, y;
5    plt::figure();
6
7    // TODO: (2-2.f) Tabulate runtimes of solveA and plot them using
8    // matplotlibcpp.
9    // START
10   std::vector<double> n(19), times(19), o_n(19);
11
12   // Table header
13   std::cout << std::setw(10) << "k" << std::scientific << std::setprecision(3)
14             << std::setw(15) << "runtime [s]" << std::endl;
15
16   // Loop from 4 to 2^20
17   for (unsigned int k = 4, i = 0; k <= (1 << 20); k <<= 1, ++i) {
18     d1 = Eigen::VectorXd::LinSpaced(k, 1, k);
19     d2 = -d1;
20     c = Eigen::VectorXd::Ones(k);
21     b.resize(2 * k);
22     b << d1, d1;
23
24     // Repeat measurements
25     Timer tm;
26     for (unsigned int r = 0; r < repeats; ++r) {
27       // Test run
28       tm.start();
29       y = solveA(d1, d2, c, b);
30       tm.stop();
31     }
32
33     // Keep track of data for plotting
34     n[i] = k;
35     times[i] = tm.mean();
36     o_n[i] = k * 1e-5;  // scale to beautify plot
37
```

```
38      // Print measurements
39      std::cout << std::setw(10) << k << std::scientific << std::setprecision(3)
40                << std::setw(15) << tm.duration() << std::endl;
41    }
42
43    // Plotting
44    plt::loglog(n, times, "o", {{"label", "solveA"}});
45    plt::loglog(n, o_n, {{"label", "O(n)"}});
46    plt::title("timings for solveA (mean of 5 runs)");
47    plt::xlabel("dimension n");
48    plt::ylabel("time [s]");
49    plt::legend();
50    // END
51    plt::savefig("cx_out/complexity.png");
52  }
```

Get it on GitLab (blocklsepiv.hpp).

SOLUTION of (2-20.a):

The code processes the matrix row-wise (index varaible `k`) and, thus, sequentially runs through the `val` array and the `col_ind` array (index variable `j`).

C++ code 2.20.1: Function `crsmv()`

```
2  template <typename VECTORTYPE_I, typename VECTORTYPE_II>
3  VECTORTYPE_I crsmv(const CRSMatrix &M, const VECTORTYPE_II &x) {
4    assert((x.size() == M.n) && "Size mismatch between x and M");
5    VECTORTYPE_I y(M.m);
6    for (int k = 0; k < M.m; ++k) {
7      y[k] = 0;
8      for (int j = M.row_ptr[k]; j < M.row_ptr[k + 1]; ++j) {
9        y[k] += M.val[j] * x[M.col_ind[j]];
10     }
11   }
12   return y;
13 }
```

SOLUTION of (2-21.a):

This line of code calls an EIGEN built-in function for the computation of the LU-decomposition of the dense $n \times n$-matrix $\mathbf{A}$. According to [Lecture $\rightarrow$ Thm. 2.5.0.2] this involves an asymptotic effort of $O(n^3)$ for $n \rightarrow \infty$.

SOLUTION of (2-21.b):

This line of code just copies a vector of length n, which obviously takes $O(n)$ machine operations. However, in the strict sense of

Definition [Lecture → Def. 1.4.0.1]. Computational effort

The **computational effort**/**computational cost** required/incurred by a numerical code amounts to the number of elementary operations (additions,subtractions,multiplications,divisions,square roots) executed in a run.

there is no (numerical) computational cost. Hence $O(1)$ for $n \to \infty$ would also be an acceptable answer.

SOLUTION of (2-21.c):

The variable `A_lu_dec` contains the already computed LU-factors of $\mathbf{A}$. Therefore the above line of code executes two triangulars solves (backward and forward elimination). According to [Lecture → Thm. 2.5.0.3] the cost is $O(n^2)$ for $n \to \infty$.

SOLUTION of (2-21.d):

The eigen built-in member function `norm()` for a (real) vector computes its Euclidean norm by summing the squares of its entries. Subsequently each vector component is divided by the same number. All this takes $O(n)$ elementary operations for $n \to \infty$

SOLUTION of (2-21.e):

In linear-algebra notation Line 17 of Code 2.21.1) computes

$$\lambda_{\text{new}} = \mathbf{y}_{\text{new}}^{\top} \mathbf{A} \mathbf{y}_{\text{new}} \ .$$

Obviously this involves

- a matrix-vector product of an $n \times n$-matrix with an n-vector,
- and a dot product of two n-vectors.

The matrix-vector product dominates the total cost which boils down to $O(n^2)$ for $n \to \infty$.

SOLUTION of (2-3.b):

We can simply rely on

$$\underbrace{\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{A}_{21}\mathbf{A}_{11}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ 0 & \mathbf{S} \end{bmatrix}}_{\text{block LU-factorization}} \quad , \qquad \text{with Schur complement} \quad \boxed{\mathbf{S} := \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}} \; .$$

[Lecture → Eq. (2.3.2.20)]

Here we have $\mathbf{A}_{11} = \mathbf{R}$, $\mathbf{A}_{12} = \mathbf{v}$, $\mathbf{A}_{21} = \mathbf{u}^\top$, $\mathbf{A}_{22} = \mathbf{O}$. Thus the sought LU-decomposition is

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

Here it is important that $\mathbf{R}$ was assumed to be invertible.

HINT 1 for (2-3.c): Use the LU-decomposition of $\mathbf{A}$ to compute its determinant. ⌟

SOLUTION of (2-3.c):

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

$$\det(\mathbf{A}) = \det(\mathbf{L}) \det(\mathbf{U}) = \det(\mathbf{L}) \det(\mathbf{R})(-\mathbf{u}^\top \mathbf{R}^{-1} \mathbf{v}) \tag{2.3.1}$$

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

HINT 1 for (2-3.d): You will need to solve a LSE multiple times by backwards substitution. It can be useful to implement a function `Eigen::VectorXd solve_R(const Eigen::MatrixXd& R, const Eigen::VectorXd& c);` that does that.

SOLUTION of (2-3.d):

C++ code 2.3.2: Code

```
Eigen::VectorXd solve_LSE(const Eigen::MatrixXd& R, const Eigen::VectorXd& v,
                          const Eigen::VectorXd& u, const Eigen::VectorXd& b) {
  const unsigned int 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");
  Eigen::VectorXd y(n + 1), x(n + 1);

  // TODO: (2-3.d) Solve the system Ax=b by LU-Decomposition.
  // START
  // 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
  Eigen::MatrixXd U(n + 1, n + 1);
  U << R, v, Eigen::VectorXd::Zero(n).transpose(),
      -u.transpose() * solve_R(R, v);

  // Then we solve Ux = y
  x = solve_R(U, y);
  //
  // END
  return x;
}
```

Get it on GitLab (blockLU.hpp).

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

C++ code 2.3.3: Code

```
Eigen::VectorXd solve_R(const Eigen::MatrixXd& R, const Eigen::VectorXd& c) {
  const unsigned int n = R.rows();
  assert(n == R.cols() && n == c.size() && "Input dimensions must agree");
  // initialize the return value
  Eigen::VectorXd y(n);

  // TODO: (optional) Solve Ry=c using backward substitution.  May help
  // when implementing solve_LSE.
  // START
  // 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);
    }
```

```
17        y(i) /= R(i, i);
18      }
19      // END
20      return y;
21    }
```

Get it on GitLab (blockLU.hpp).

HINT 1 for (2-4.a): For a Cholesky decomposition to be possible, show that $\mathbf{A}^\top \mathbf{A}$ is symmetric and positive definite. ⌟

SOLUTION of (2-4.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.

$\text{rank}(\mathbf{A}) = n$ 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.

HINT 1 for (2-4.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::HouseholderQ` does not make use of them. (`Eigen::ColPivHouseholderQR` and `Eigen::FullPivHousehold` 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 (2-4.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.
- $\mathbf{QR} = \left(\mathbf{R}^{-\top}\mathbf{A}^{\top}\right)^{\top}\mathbf{R} = \mathbf{A}\mathbf{R}^{-1}\mathbf{R} = \mathbf{A}$

HINT 1 for (2-4.c): Review [Lecture → Ex. 3.2.0.4]! ⌟

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

SOLUTION of (2-4.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.

SOLUTION of (2-5.b):

We write $I_{j,k}$ for the branch current flowing from node $\sharp i$ to node $\sharp k$. Moreover, denote by I_G and I_V the currents flowing through the grounding contact and the current source, respectively. According to the Kirchhoff current law (KCL) applied to every node we have

$$
\begin{aligned}
\text{Node } \sharp 1: &\quad I_{1,2} + I_{1,5} = 0 \\
\text{Node } \sharp 2: &\quad I_{2,1} + I_{2,3} + I_{2,5} + I_{2,14} = 0 \\
\text{Node } \sharp 3: &\quad I_{3,2} + I_{3,4} + I_{3,15} = 0 \\
\text{Node } \sharp 4: &\quad I_{4,3} + I_{4,6} + I_{4,15} = 0 \\
\text{Node } \sharp 5: &\quad I_{5,1} + I_{5,2} + I_{5,7} + I_{5,14} = 0 \\
\text{Node } \sharp 6: &\quad I_{6,4} + I_{6,9} + I_{6,15} + I_{6,16} = 0 \\
\text{Node } \sharp 7: &\quad I_{7,5} + I_{7,10} + I_{7,11} + I_{7,17} = 0 \\
\text{Node } \sharp 8: &\quad I_{8,9} + I_{8,12} + I_{8,13} + I_{8,15} = 0 \\
\text{Node } \sharp 9: &\quad I_{9,6} + I_{9,8} + I_{9,13} = 0 \\
\text{Node } \sharp 10: &\quad I_{10,7} + I_{10,11} = 0 \\
\text{Node } \sharp 11: &\quad I_{11,7} + I_{11,10} + I_{11,12} + I_{11,17} = 0 \\
\text{Node } \sharp 12: &\quad I_{12,8} + I_{12,11} + I_{12,13} = 0 \\
\text{Node } \sharp 13: &\quad I_{13,8} + I_{13,9} + I_{13,12} = 0 \\
\text{Node } \sharp 14: &\quad I_{14,2} + I_{14,5} + I_{14,17} + I_{14,15} = 0 \\
\text{Node } \sharp 15: &\quad I_{15,3} + I_{15,4} + I_{15,6} + I_{15,8} + I_{15,14} = 0 \\
\text{Node } \sharp 16: &\quad I_{16,6} + I_V = 0 \\
\text{Node } \sharp 17: &\quad I_{17,7} + I_{17,11} + I_{17,14} + I_G = 0 \,.
\end{aligned}
$$

Next, we plug in the circuit element relation $I_{j,k} = R_{j,k}^{-1}(U_j - U_k)$:

$$
\begin{aligned}
\text{Node } \sharp 1: &\quad \tfrac{1}{R}(U_1 - U_2) + \tfrac{1}{R}(U_1 - U_5) = 0 \\
\text{Node } \sharp 2: &\quad \tfrac{1}{R}(U_2 - U_1) + \tfrac{1}{R}(U_2 - U_3) + \tfrac{1}{R}(U_2 - U_5) + \tfrac{1}{R}(U_2 - U_{14}) = 0 \\
\text{Node } \sharp 3: &\quad \tfrac{1}{R}(U_3 - U_2) + \tfrac{1}{R}(U_3 - U_4) + \tfrac{1}{R}(U_3 - U_{15}) = 0 \\
\text{Node } \sharp 4: &\quad \tfrac{1}{R}(U_4 - U_3) + \tfrac{1}{R}(U_4 - U_6) + \tfrac{1}{R}(U_4 - U_{15}) = 0 \\
\text{Node } \sharp 5: &\quad \tfrac{1}{R}(U_5 - U_1) + \tfrac{1}{R}(U_5 - U_2) + \tfrac{1}{R}(U_5 - U_7) + \tfrac{1}{R}(U_5 - U_{14}) = 0 \\
\text{Node } \sharp 6: &\quad \tfrac{1}{R}(U_6 - U_4) + \tfrac{1}{R}(U_6 - U_9) + \tfrac{1}{R}(U_6 - U_{15}) + \tfrac{1}{R}(U_6 - U_{16}) = 0 \\
\text{Node } \sharp 7: &\quad \tfrac{1}{R}(U_7 - U_5) + \tfrac{1}{R}(U_7 - U_{10}) + \tfrac{1}{R}(U_7 - U_{11}) + \tfrac{1}{R}(U_7 - U_{17}) = 0 \\
\text{Node } \sharp 8: &\quad \tfrac{1}{R}(U_8 - U_9) + \tfrac{1}{R}(U_8 - U_{12}) + \tfrac{1}{R}(U_8 - U_{13}) + \tfrac{1}{R}(U_8 - U_{15}) = 0 \\
\text{Node } \sharp 9: &\quad \tfrac{1}{R}(U_9 - U_6) + \tfrac{1}{R}(U_9 - U_8) + \tfrac{1}{R}(U_9 - U_{13}) = 0 \\
\text{Node } \sharp 10: &\quad \tfrac{1}{R}(U_{10} - U_7) + \tfrac{1}{R}(U_{10} - U_{11}) = 0 \\
\text{Node } \sharp 11: &\quad \tfrac{1}{R}(U_{11} - U_7) + \tfrac{1}{R}(U_{11} - U_{10}) + \tfrac{1}{R}(U_{11} - U_{12}) + \tfrac{1}{R}(U_{11} - U_{17}) = 0 \\
\text{Node } \sharp 12: &\quad \tfrac{1}{R}(U_{12} - U_8) + \tfrac{1}{R}(U_{12} - U_{11}) + \tfrac{1}{R}(U_{12} - U_{13}) = 0 \\
\text{Node } \sharp 13: &\quad \tfrac{1}{R}(U_{13} - U_8) + \tfrac{1}{R}(U_{13} - U_9) + \tfrac{1}{R}(U_{13} - U_{12}) = 0 \\
\text{Node } \sharp 14: &\quad \tfrac{1}{R}(U_{14} - U_2) + \tfrac{1}{R}(U_{14} - U_5) + \tfrac{1}{R}(U_{14} - U_{17}) + \tfrac{1}{R_x}(U_{14} - U_{15}) = 0 \\
\text{Node } \sharp 15: &\quad \tfrac{1}{R}(U_{15} - U_3) + \tfrac{1}{R}(U_{15} - U_4) + \tfrac{1}{R}(U_{15} - U_6) + \tfrac{1}{R}(U_{15} - U_8) + \tfrac{1}{R_x}(U_{15} - U_{14}) = 0 \\
\text{Node } \sharp 16: &\quad \tfrac{1}{R}(U_{16} - U_6) + I_V = 0 \\
\text{Node } \sharp 17: &\quad \tfrac{1}{R}(U_{17} - U_7) + \tfrac{1}{R}(U_{17} - U_{11}) + \tfrac{1}{R}(U_{17} - U_{14}) + I_G = 0\,.
\end{aligned}
$$

This is the requested linear system of equations. The last two equations featuring unknown currents can be dropped. This yields the final linear system of equations, here given in matrix notation:

$$
\begin{bmatrix}
2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & 3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
-1 & -1 & 0 & 0 & 4 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & -1 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & 3 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3+\xi & -\xi & 0 & -1 \\
0 & 0 & -1 & -1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -\xi & 4+\xi & 0 & 0
\end{bmatrix}
\begin{bmatrix}
U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \\ U_7 \\ U_8 \\ U_9 \\ U_{10} \\ U_{11} \\ U_{12} \\ U_{13} \\ U_{14} \\ U_{15} \\ U_{16} \\ U_{17}
\end{bmatrix}
$$

where $\xi := \frac{R}{R_x}$. Note that we have multiplied ("scaled") the whole linear system with R.

SOLUTION of (2-5.c):

In the 15×17 linear system obtained in Sub-problem (2-5.b) we multiply the columns corresponding to the known nodal potentials $U_{16} = V$ and $U_{17} = 0$ with those values and move them to the right-hand. We end up with the 15×15 system matrix

$$\mathbf{A}(R_x) = \left[\begin{array}{rrrrrrrrrrrrrrr} 2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 4 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & 3 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3+\xi & -\xi \\ 0 & 0 & -1 & -1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -\xi & 4+\xi \end{array}\right]$$

with $\xi := \frac{R}{R_x}$. The right-hand side vector $\mathbf{b}$ is zero except in its sixth component:

$$(\mathbf{b})_i = \begin{cases} V & \text{for } i = 6\,, \\ 0 & \text{for } i \neq 6\,. \end{cases}$$

HINT 1 for (2-5.d): Use EIGEN's facilities for the direct solution of dense linear systems of equations [Lecture → § 2.5.0.4] to solve the LSE derived in Sub-problem (2-5.c). ⌟

HINT 2 for (2-5.d): Trick: Note that all nodal voltages depend linearly on V. Thus the nodal voltages can be precomputed in the constructor for $V = 1$ and that vector, scaled with the provided value of V, can just be returned by the evaluation operator. ⌟

SOLUTION of (2-5.d):

In the constructor we compute the solution of $\mathbf{A}(R_x)\mathbf{x} = \mathbf{b}$ for $V = 1$ and store it in data member of type **Eigen::VectorXd**. We use the **Topology** struct to build the matrix.

C++ code 2.5.1: Typedefs for Topology

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

C++ code 2.5.2: Implementation of struct Topology

```
struct Topology {
  /**
   * brief Construct a new Topology object**
      param V Source voltage V at node 16 (in Volt), ground is set to 0V
   * at node 17
   */
  Topology(double V) {
    // 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
    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));
    // We omit the resistor between node 14 and 15 for an easier matrix
       filling.

    // 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.
    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));
  }

  resistor_topology T;
  voltage_topology S;
};
```

C++ code 2.5.3: Complete definition of class NodalPotentials

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

  NodalPotentials(double R, double Rx);
  Eigen::VectorXd operator()(double V) const;

 private:
  // TODO: (2-5.d) Specify which variables should be stored.
  // START
  Eigen::VectorXd nodal_voltage_;
  double R_, Rx_, xi_;
  // END
};
```

C++ code 2.5.4: Constructor of NodalPotentials

```
NodalPotentials::NodalPotentials(double R, double Rx)
    // TODO: (2-5.d) initializer list.
    // START
    : R_(R),
      Rx_(Rx)
// END
{
  // TODO: (2-5.d) Build the matrix and the r.h.s.  vector by either
     using
  // the topology struct or by hardcoding the matrix and vector.
  // START
```

```
  assert(Rx_ != 0. && "Rx should not be equal to zero!");
  xi_ = R_ / Rx_;

  // Build the topology for a source voltage of 1.
  Topology top(1.);

  // Automatically build A_Rx filling from topology
  // A_Rx is the matrix A_R (i.e. with R_x = R)
  Eigen::MatrixXd A_Rx = Eigen::MatrixXd::Zero(15, 15);
  for (resistor& res : top.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
    A_Rx(i, i) += 1;
    A_Rx(j, j) += 1;
    // Fill the off diagonal (negative part in ΔV_{i,j},
    // the matrix is kept symmetric
    A_Rx(i, j) -= 1;
    A_Rx(j, i) -= 1;
  }

  // Fill in the rest (source and ground), i.e.
  // components with ΔV_{i,j} with j > 15.
  // Each node i connected to ground or source contributes
  // to to its own diagonal with R
  for (voltage& volt : top.S) {
    // Shift index down by 1 and get voltage in V2
    int i = std::get<0>(volt) - 1;
    // Add resistance to matrix diagonal: contribution of source
       current
    // scaled by R.
    A_Rx(i, i) += 1;
  }

  // Add resistance between node 14 and 15
  A_Rx(13, 13) += xi_;
  A_Rx(13, 14) -= xi_;
  A_Rx(14, 13) -= xi_;
  A_Rx(14, 14) += xi_;

  // Set up r.h.s.
  Eigen::VectorXd b = Eigen::VectorXd::Zero(15);
  b(5) = 1.;

  // Solve unscaled linear system and store in member.
  nodal_voltage_ = A_Rx.lu().solve(b);
  // END
}
```

C++ code 2.5.5: Evaluation operator of NodalPotentials

```
Eigen::VectorXd NodalPotentials::operator()(double V) const {
  Eigen::VectorXd return_vector = Eigen::VectorXd::Zero(15);

  // TODO: (2-5.d) Return the values of the potential at the nodes.
```

```
    // START
    return_vector = nodal_voltage_ * V;
    // END

    return return_vector;
}
```

Get it on GitLab (circuitimpedance.hpp).

HINT 1 for (2-5.e): 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 $\mathbf{v}$. ⌟

SOLUTION of (2-5.e):

Define $\xi := R/R_x$. Define the vector

$$\mathbf{u} := \sqrt{\xi} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -1 \\ 1 \end{bmatrix} \in \mathbb{R}^{15} . \tag{2.5.7}$$

Then we can write

$$\mathbf{A}(R_x) = \mathbf{A}(0) + \mathbf{u}\mathbf{u}^\top .$$

SOLUTION of (2-5.f):

Assume that we have a LU-factorization of $\mathbf{A}(0)$ at our disposal. Then we can use the **Sherman-Morrison Woodbury formula** of [Lecture → Lemma 2.6.0.21] for $k = 1$ as given in [Lecture → Eq. (2.6.0.22)] to efficiently compute $\widetilde{\mathbf{x}} := \mathbf{A}(R_x)^{-1}\mathbf{b}$, with $\mathbf{b} \in \mathbb{R}^{15}$:

$$\widetilde{\mathbf{x}} = \mathbf{A}(0)^{-1}\mathbf{b} - \frac{\mathbf{A}(0)^{-1}\mathbf{u}(\mathbf{u}^\top(\mathbf{A}(0)^{-1}\mathbf{b}))}{1+\mathbf{u}^\top(\mathbf{A}(0)^{-1}\mathbf{u})} \,. \tag{2.5.8}$$

The colors indicate the vectors that can be obtained by solving a LSE with system matrix $\mathbf{A}(0)$. These vectors can be regarded as "available".

Supplement & trick: We can simplify formula (2.5.8) even further by examining (2.5.7) and noticing that

$$\mathbf{A}(R_x) = \mathbf{A}(0) + \xi\,\mathbf{u}_0\mathbf{u}_0^\top \quad \text{with} \quad \xi = \frac{R}{R_x}\,, \quad \mathbf{u}_0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -1 \\ 1 \end{bmatrix} \in \mathbb{R}^{15} \,. \tag{2.5.9}$$

Thus (2.5.8) can be recast as

$$\widetilde{\mathbf{x}} = \mathbf{A}(0)^{-1}\mathbf{b} - \xi\frac{\mathbf{A}(0)^{-1}\mathbf{u}_0(\mathbf{u}_0^\top(\mathbf{A}(0)^{-1}\mathbf{b}))}{1+\xi\mathbf{u}_0^\top(\mathbf{A}(0)^{-1}\mathbf{u}_0)} \,.$$

Setting $\mathbf{w} := \mathbf{A}(0)^{-1}\mathbf{b}$, $\mathbf{z} := \mathbf{A}(0)^{-1}\mathbf{u}_0$, we can state this in a more lucid way:

$$\widetilde{\mathbf{x}} = \mathbf{w} - \xi\frac{\mathbf{z}(\mathbf{u}_0^\top\mathbf{w})}{1+\xi\mathbf{u}_0^\top\mathbf{z}} = \mathbf{w} - \frac{\xi\beta}{1+\xi\alpha}\,\mathbf{z}\,, \quad \beta := \mathbf{u}_0^\top\mathbf{w}\,, \quad \alpha := \mathbf{u}_0^\top\mathbf{z}\,. \tag{2.5.10}$$

Note that along with $\mathbf{u}_0$ also α and β are independent of R_x. Only ξ depends on the variable resistance.

HINT 1 for (2-5.g): Since I_V is proportional to V, we can just compute with $V = 1$. I_V itself is available as the current flowing between node ♯6 and the voltage source: According to the constitutive relation for the resistor between the two we have

$$I_V = R^{-1}(U_6 - V) \, .$$

HINT 2 for (2-5.g): An EIGEN data type that can store the LU-decomposition of a real dense martrix is `Eigen::PartialPivLU<Eigen::MatrixXd>`. A class data member of that type can be used. ⌟

SOLUTION of (2-5.g):

"Standard" approach based on (2.5.8):

1. In the constructor we set up the system matrix $\mathbf{A}(0)$ and the right-hand side vector $\mathbf{b}$ (for $V = 1$) as found in Sub-problem (2-5.c). We let EIGEN compute the LU-decomposition of $\mathbf{A}(0)$ and store it. We can also precompute $\mathbf{A}(0)^{-1}\mathbf{b}$ and keep it in another data member of type **Eigen::VectorXd**.
2. In the evaluation operator we compute $\mathbf{A}^{-1}\mathbf{u}$ and evaluate the formula (2.5.8).

"Smart" approach exploiting (2.5.10):

1. In the constructor compute $\mathbf{w} := \mathbf{A}(0)^{-1}\mathbf{b}, \mathbf{z} := \mathbf{A}(0)^{-1}\mathbf{u}_0 \in \mathbb{R}^{15}$ and $\alpha := \mathbf{u}_0^\top \mathbf{z}, \beta := \mathbf{u}_0^\top \mathbf{w} \in \mathbb{R}$, notations as in (2.5.10). Store them all in appropriate data members.
2. The evaluation operator just has to compute $\zeta := \frac{R}{R_x}$ and for a linear combination of $\mathbf{w}$ and $\mathbf{z}$ according to (2.5.10).

Of course, it is good practice to check whether denominators are "nearly vanishing", see [Lecture → Rem. 1.5.3.15].

C++ code 2.5.11: Complete definition of class ImpedanceMap

```
2  class ImpedanceMap {
3   public:
4    ImpedanceMap() = delete;
5    ImpedanceMap(const ImpedanceMap&) = default;
6    ImpedanceMap(ImpedanceMap&&) = default;
7    ImpedanceMap& operator=(const ImpedanceMap&) = default;
8    ImpedanceMap& operator=(ImpedanceMap&&) = default;
9    ~ImpedanceMap() = default;
10
11   ImpedanceMap(double R, double V);
12   double operator()(double Rx) const;
13
14  private:
15   // TODO: (2-5.g) Specify which variables should be stored.
16   // START
17   double R_, V_;  //< Resistance R and source voltage V.
18   Eigen::VectorXd w_, z_;
19   double alpha_, beta_;
20   static const std::size_t nnodes_ = 15;
21   // END
22 };
```

C++ code 2.5.12: Constructor of ImpedanceMap

```
2  ImpedanceMap::ImpedanceMap(double R, double V)
3      // TODO: (2-5.g) initializer list.
4      // START
5      : R_(R),
6        V_(V)
7  // END
8  {
9    // TODO: (2-5.g) Build the matrix and the r.h.s.  vector by either
```

```
    using
  // the topology or by hardcoding the matrix and vector.  Also, do
  // the setup phase for the solver here or find an even smarter way
  // for an efficient implementation
  // START

  // Build the topology with source voltage V_.
  Topology top(V_);

  // Automatically build A0 filling from topology
  // A0 is the matrix A_R (i.e.  with R_x = R)
  Eigen::MatrixXd A0 = Eigen::MatrixXd::Zero(nnodes_, nnodes_);
  for (resistor& res : top.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 ΔV_{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 ΔV_{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 resistance between node i and ground/source
  // node, V is voltage at sink or source)
  // and to its own diagonal with R
  Eigen::VectorXd b = Eigen::VectorXd::Zero(nnodes_);
  for (voltage& volt : top.S) {
    // Shift index down by 1 and get voltage in V2
    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;
  }

  // Following the "smart" approach for the SMW formula;
  // we don't even have to store the LU-decomposition!
  Eigen::PartialPivLU<Eigen::MatrixXd> lu(A0);
  w_ = lu.solve(b);
  Eigen::VectorXd u0 = Eigen::VectorXd::Zero(nnodes_);
  u0(13) = -1.;
  u0(14) = 1.;
  z_ = lu.solve(u0);
  alpha_ = u0.dot(z_);
  beta_ = u0.dot(w_);
  // END
}
```

C++ code 2.5.13: Evaluation operator of ImpedanceMap

```cpp
double ImpedanceMap::operator()(double Rx) const {
  double return_value = 0.;

  // TODO: (2-5.g) Return the impedance of the circuit given a specific
     Rx
  // START
  assert(Rx != 0 && "Rx should not be equal to zero!");
  // Store the scaled factor for convenience
  double xi = R_ / Rx;

  // Put the formula together, x is a column vector containing voltages
  // at each node (except 16 and 17, which are prescribed)
  Eigen::VectorXd x = w_ - xi * beta_ / (1. + xi * alpha_) * z_;

  // Compute the current $I = -\Delta V_{16,6}/R$
  // and then impedance $= V/I$.
  // Here $\Delta V_{16,6} = V_{16} - x_6 = V - x_6$.
  return_value = V_ * R_ / (x(5) - V_);
  // END

  return return_value;
}
```

Get it on GitLab (circuitimpedance.hpp).

SOLUTION of (2-6.a):

C++ code 2.6.2: Solution of (2-6.a)

```
Eigen::VectorXd multAx(Eigen::VectorXd &a, const Eigen::VectorXd &b,
                       const Eigen::VectorXd &x) {
  const unsigned int n = x.size();
  Eigen::VectorXd y = Eigen::VectorXd::Zero(n);
  if (a.size() < n - 1 || b.size() < n - 2) {
    std::cerr << "Error: size mismatch!" << std::endl;
    return y;
  }

  // TODO: (2-6.a) Multiply A*x and exploit the diagonal structure of
     the
  // matrix.
  // START
  // Handle first two rows
  if (n > 1) {
    y(0) = 2 * x(0) + a(0) * x(1);
  } else {
    y(0) = 2 * x(0);
    return y;
  }
  if (n > 2) {
    y(1) = 2 * x(1) + a(1) * x(2);
  } else {
    y(1) = 2 * x(1);
    return y;
  }

  // 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);
  // END
  return y;
}
```

Get it on GitLab (efficientbandmult.hpp).

HINT 1 for (2-6.b): Trick: Give a proof by contradiction of $\ker \mathbf{A} = \{\mathbf{0}\}$ by looking at the largest (in modulus) component of a vector $\mathbf{x} \in \ker \mathbf{A}$. ⌟

SOLUTION of (2-6.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 $i = \text{argmax}|x_i|$, with $x_i \neq 0$.

Since $\mathbf{Ax} = \{\mathbf{0}\}$, 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 $\rightarrow$ Def. 2.8.0.8]), like $\mathbf{A}$ in our example, must be multiples of $[1, 1, \ldots, 1]^\top$ (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.

HINT 1 for (2-6.c): What special linear system do we face in the case $b_i = 0$? ⌟

SOLUTION of (2-6.c):

If $b_i = 0$ we face a **upper triangular** linear system of equations, which can be solved by means of simple backward substitution.

The implementation involves only elementary operations on vector components.

C++11-code 2.6.3: Solution of (2-6.c)

```
2  Eigen::VectorXd solveIseAupper(const Eigen::VectorXd &a,
3                                 const Eigen::VectorXd &r) {
4    // Set up dimensions
5    const unsigned int n = r.size();
6    Eigen::VectorXd x = Eigen::VectorXd::Zero(n);
7    if (a.size() < n - 1) {
8      std::cerr << "Error: size mismatch!" << std::endl;
9      return x;
10   }
11
12   // TODO: (2-6.c) Solve the LSE A*x = r for x by exploiting the
         diagonal
13   // structure of A and the fact that b is zero.
14   // START
15   // Backward substitution
16   x(n - 1) = 0.5 * r(n - 1);
17   for (int j = n - 2; j >= 0; --j) {
18     x(j) = 0.5 * (r(j) - a(j) * x(j + 1));
19   }
20   // END
21   return x;
22 }
```

Get it on GitLab (efficientbandmult.hpp).

HINT 1 for (2-6.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].
- Note that Gaussian elimination fills in zero entries of $\mathbf{A}$ only along one lower off-diagonal, which can be stored as vector `Eigen::VectorXd c` (similarly to what was done in Sub-problem (2-6.a), Sub-problem (2-6.c)). If you store the diagonal (initially only made of 2s) as `Eigen::VectorXd d`, you can perform a straightforward Gaussian elimination only operating on vectors `Eigen::Vecto c`, `Eigen::VectorXd d` and `Eigen::VectorXd r`.

SOLUTION of (2-6.d):

To demonstrate the ideas, we perform Gaussian elimination without pivoting for the case $n = 5$:

$$\begin{bmatrix} d_1 & a_1 & 0 & 0 & 0 \\ 0 & d_2 & a_2 & 0 & 0 \\ 0 & c_2 & d_3 & a_3 & 0 \\ 0 & b_2 & c_3 & d_4 & a_4 \\ 0 & 0 & b_3 & c_4 & d_5 \end{bmatrix}$$

Second elimination step:
- Subtract $c_2/d_2 \cdot$ 2nd row from second row,
- and $b_2/d_2 \cdot$ 2nd row from third row,

in order to annihilate $c_2 \leftarrow 0$, $b_2 \leftarrow 0$.

$$\begin{bmatrix} d_1 & a_1 & 0 & 0 & 0 \\ 0 & d_2 & a_2 & 0 & 0 \\ 0 & 0 & d_3 & a_3 & 0 \\ 0 & 0 & c_3 & d_4 & a_4 \\ 0 & 0 & b_3 & c_4 & d_5 \end{bmatrix}$$

Third elimination step:
- Subtract $c_3/d_3 \cdot$ 3rd row from second row,
- and $b_3/d_3 \cdot$ 3rd row from third row,

in order to annihilate $c_3 \leftarrow 0$, $b_3 \leftarrow 0$.

$$\begin{bmatrix} d_1 & a_1 & 0 & 0 & 0 \\ 0 & d_2 & a_2 & 0 & 0 \\ 0 & 0 & d_3 & a_3 & 0 \\ 0 & 0 & 0 & d_4 & a_4 \\ 0 & 0 & 0 & c_4 & d_5 \end{bmatrix}$$

Third elimination step:
- Subtract $c_3/d_3 \cdot$ 3rd row from second row,
- and $b_3/d_3 \cdot$ 3rd row from third row,

in order to annihilate $c_3 \leftarrow 0$, $b_3 \leftarrow 0$.

$$\begin{bmatrix} d_1 & a_1 & 0 & 0 & 0 \\ 0 & d_2 & a_2 & 0 & 0 \\ 0 & 0 & d_3 & a_3 & 0 \\ 0 & 0 & 0 & d_4 & a_4 \\ 0 & 0 & 0 & 0 & d_5 \end{bmatrix}$$

Third elimination step:
- Subtract $c_4/d_4 \cdot$ 4th row from second row,

in order to annihilate $c_4 \leftarrow 0$.

The green matrix entries are those that have changed in the previous elimination step. In addition, the same transformations have to be applied to the right-hand-side vector $\mathbf{r}$. Subsequently, we have to do backward elimination in order to obtain the solution.

From this example we see a pattern that suggests the following general algorithm (linear-algebra indexing!).

1. Forward elimination:

```
c := 0;
for l=1 to n-2 do
  d[l+1] -= a[l]*c[l]/d[l];
  c[l+1] -= a[l]*b[l]/d[l];
  r[l+1] -= r[l]*c[l]/d[l];
  r[l+2] -= r[l]*b[l]/d[l];
end
d[n] -= a[n-1]*c[n-1]/d[n-1];
r[n] -= r[n-1]*c[n-1]/d[n-1];
```

2. Backward elimination:

```
x[n] = r[n]/d[n];
for l=n-1 to 1 step -1 do
  x[l] = (r[l] - a[l]*x[l+1])/d[l];
end
```

x will then contain the solution.

Note that the matrix entries turned set to zero are skipped in the algorithms.

C++ code 2.6.4: Solution of (2-6.d)

```
Eigen::VectorXd solveIseA(const Eigen::VectorXd &a, const Eigen::VectorXd &b,
                          const Eigen::VectorXd &r) {
  // Set up dimensions
  const unsigned int n = r.size();
  Eigen::VectorXd x = Eigen::VectorXd::Zero(n);
  if (a.size() < n - 1 || b.size() < n - 2) {
    std::cerr << "Error: size mismatch!" << std::endl;
    return x;
  }

  // TODO: (2-6.d) Compute the solution to A*x=r by Gaussian elimination and
  // without the help of Eigen's solvers.
  // START
  // Initializing vectors for the diagonal and lower off-diagonal
  Eigen::VectorXd d = Eigen::VectorXd::Constant(n, 2);
  Eigen::VectorXd c = Eigen::VectorXd::Constant(n - 1, 0);

  // A modifiable RHS vector
  Eigen::VectorXd R(r);

  // Performing Gaussian elimination
  // We only need to update the vectors c and d!
  for (unsigned i = 0; i < n - 2; ++i) {
    /*

      ... d(i)  a(i)     0       0      ...
      ... c(i)  d(i+1)   a(i+1)  0      ...
      ... b(i)  c(i+1)   d(i+2)  a(i+2) 0

                    ===>

      ... d(i)  a(i)                    0       0      ...
```

```
      ... 0     d(i+1)-c(i)/d(i)*a(i)   a(i+1)  0      ...
      ... b(i)  c(i+1)                  d(i+2)  a(i+2) 0
    */
    d(i + 1) -= c(i) / d(i) * a(i);
    // Modifying the RHS accordingly
    R(i + 1) -= c(i) / d(i) * R(i);

    /*
      ... d(i)  a(i)                    0       0      ...
      ... 0     d(i+1)-c(i)/d(i)*a(i)   a(i+1)  0      ...
      ... b(i)  c(i+1)                  d(i+2)  a(i+2) 0

                    ===>

      ... d(i)  a(i)                    0       0      ...
      ... 0     d(i+1)-c(i)/d(i)*a(i)   a(i+1)  0      ...
      ... 0     c(i+1)-b(i)/d(i)*a(i)   d(i+2)  a(i+2) 0
    */
    c(i + 1) -= b(i) / d(i) * a(i);
    // Modifying the RHS accordingly
    R(i + 2) -= b(i) / d(i) * R(i);
  }
  // Last row
  d(n - 1) -= c(n - 2) / d(n - 2) * a(n - 2);
  R(n - 1) -= c(n - 2) / d(n - 2) * R(n - 2);

  // Backward substitution
  x(n - 1) = R(n - 1) / d(n - 1);
  for (int i = n - 2; i >= 0; --i) {
    x(i) = (R(i) - a(i) * x(i + 1)) / d(i);
  }
  // END
  return x;
}
```

Get it on GitLab (efficientbandmult.hpp).

HINT 1 for (2-6.e): Figure out the computational cost of each step of the algorithm you coded and “sum them up”. ⌟

SOLUTION of (2-6.e):

In the code Code 2.6.4 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 $O(n)$ asymptotic complexity.

HINT 1 for (2-6.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 built-in function) enough space for the nonzero entries of each column (see the  EIGEN documentation). ⌟

SOLUTION of (2-6.f):

C++ code 2.6.5: Solution of (2-6.f)

```
2  Eigen::VectorXd solveIseASparse(const Eigen::VectorXd &a,
3                                  const Eigen::VectorXd &b,
4                                  const Eigen::VectorXd &r) {
5    // Set up dimensions
6    const unsigned int n = r.size();
7    Eigen::VectorXd x = Eigen::VectorXd::Zero(n);
8    if (a.size() < n - 1 || b.size() < n - 2) {
9      std::cerr << "Error: size mismatch!" << std::endl;
10     return x;
11   }
12
13   // TODO: (2-6.f) Solve A*x=r using Eigen's sparse solver.
14   // START
15   // Fill in matrix:
16   // We reserve three nonzero entries per row for Gaussian fill-in
17   Eigen::SparseMatrix<double> A(n, n);
18   A.reserve(Eigen::VectorXi::Constant(n, 3));
19   for (unsigned int i = 0; i < n; ++i) {
20     A.insert(i, i) = 2;
21     if (i < n - 1) A.insert(i, i + 1) = a(i);
22     if (i >= 2) A.insert(i, i - 2) = b(i - 2);
23   }
24   A.makeCompressed();
25
26   // Call SparseLU
27   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
28   solver.analyzePattern(A);
29   solver.compute(A);
30   x = solver.solve(r);
31   // END
32   return x;
33 }
```

Get it on GitLab (efficientbandmult.hpp).

On the use of the `reserve()` member function for the preallocation of sufficient space for a sparse matrix in CRS/CCS format see [Lecture → Code 2.7.2.1].

SOLUTION of (2-7.a):

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

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

SOLUTION of (2-7.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 $x + 3y = 0$.

SOLUTION of (2-7.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

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

SOLUTION of (2-7.d):

C++ code 2.7.2: Code

```
2 void applyHouseholder(Eigen::VectorXd& x, const Eigen::VectorXd& v) {
3   // TODO: (2-7.d) Apply the Householder transformation with optimal
        complexity,
4   // store result in x
5   // START
6   const double d = v.transpose() * x;
7   x -= 2 * v * d / v.squaredNorm();
8   // END
9 }
```

Get it on GitLab (householder.hpp).

HINT 1 for (2-7.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 (2-7.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 (2-7.e):

C++ code 2.7.3: Code

```
2  template <typename Scalar>
3  void applyHouseholder(Eigen::VectorXd& x, const Eigen::MatrixBase<Scalar>& V) {
4    const unsigned int n = V.rows();
5
6    // TODO: (2-7.e) Apply the sequence of Householder transformations to
       x
7    // START
8    for (unsigned int i = 0; i < n; ++i) {
9      Eigen::VectorXd v = V.col(i);
10     v(i) = std::sqrt(1. - V.col(i).squaredNorm());
11
12     x += 2. * v.dot(x) / (1. + 2. * v.squaredNorm()) * v;
13   }
14   // END
15 }
```

Get it on GitLab (householder.hpp).

HINT 1 for (2-8.a): Recall the definition of a linear mapping between two vector spaces from linear algebra. ⌟

SOLUTION of (2-8.a):

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

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

This confirms linearity of the mapping $\mathbf{X} \mapsto \mathbf{A}\mathbf{X} + \mathbf{X}\mathbf{A}^T$

HINT 1 for (2-8.c): Start by expressing X as $\begin{bmatrix} x_1 & x_3 \\ x_2 & x_4 \end{bmatrix}$. ⌟

SOLUTION of (2-8.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 (2.8.1) is equivalent to Eq. (2.8.2) with:

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

SOLUTION of (2-8.d):

By elementary properties of the matrix product:

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

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

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

$$\mathbf{C} = \mathbf{I} \bigotimes \mathbf{A} + \mathbf{A} \bigotimes \mathbf{I} \ . \tag{2.8.3}$$

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

HINT 1 for (2-8.e): Read [Lecture → Section 2.7.2] carefully before starting. ⌟

SOLUTION of (2-8.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 (2.8.3) to fill the triplets

C++ code 2.8.4: Solution of (2-8.e)

```
2  Eigen::SparseMatrix<double> buildC(const Eigen::MatrixXd &A) {
3    // Initialization
4    const unsigned int n = A.rows();
5    Eigen::SparseMatrix<double> C(n * n, n * n);
6    std::vector<Eigen::Triplet<double>> triplets;
7    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);
8
9    // Iterate over n*n blocks.
10   for (unsigned int i = 0; i < n; ++i) {
11     for (unsigned int j = 0; j < n; ++j) {
12       // TODO: (2-8.e) Fill in the (i,j)-th block of C.
13       // Hint: The vector triplets should contain the position
14       // and value the non-zero entries of C.
15       // Use triplets.push_back() to add a new
16       // Triplet<double> to the list.
17       // START
18       // Set diagonal
19       if (i == j) {
20         for (unsigned int k1 = 0; k1 < n; ++k1) {
21           for (unsigned int k2 = 0; k2 < n; ++k2) {
22             Eigen::Triplet<double> triplet(i * n + k1, j * n + k2,
23                                            A(i, j) * I(k1, k2) + A(k1, k2));
24             triplets.push_back(triplet);
25           }
26         }
27         // Set other cells
28       } else {
29         for (unsigned int k = 0; k < n; ++k) {
30           Eigen::Triplet<double> triplet(i * n + k, j * n + k, A(i, j));
31           triplets.push_back(triplet);
32         }
33       }
34       // END
35     }
36   }
37   C.setFromTriplets(triplets.begin(), triplets.end());
38   C.makeCompressed();
39   return C;
40 }
```

Get it on GitLab (lyapunov.hpp).

HINT 1 for (2-8.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 (2-8.f):

C++11-code 2.8.5: Solution of (2-8.f)

```
2  void solveLyapunov(const Eigen::MatrixXd &A, Eigen::MatrixXd &X) {
3    // To solve A*X + X*A^T = I, we must first write
4    // the LSE in "canonical form", C*vecX = b.
5    const unsigned int n = A.rows();
6    Eigen::SparseMatrix<double> C;
7    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);
8    Eigen::VectorXd b(n * n);
9    Eigen::VectorXd vecX(n * n);
10
11   // TODO: (2-8.f) Fill in the entries of C, b, and vecX.
12   // Hint: Use SparseLU to solve the LSE.
13   // START
14   C = buildC(A);
15   b = Eigen::Map<Eigen::MatrixXd>(I.data(), n * n, 1);
16   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
17   solver.compute(C);
18   vecX = solver.solve(b);
19   // END
20
21   X = Eigen::Map<Eigen::MatrixXd>(vecX.data(), n, n);
22 }
```

Get it on GitLab (lyapunov.hpp).

SOLUTION of (2-8.g):

C++ code 2.8.6: Solution of (2-8.g)

```cpp
bool testLyapunov() {
  // Initialization
  constexpr unsigned int n = 5;
  Eigen::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;

  constexpr double tol = 1e-8;
  bool works = false;

  // TODO: (2-8.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'
  Eigen::SparseMatrix<double> C = buildC(A);
  std::cout << "C = " << C << std::endl;

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

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

  // Verify the solution if you obtain zero
  Eigen::MatrixXd I = Eigen::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.hpp).

SOLUTION of (2-8.h):

For general matrices $\mathbf{A}$ and $\mathbf{B}$ we have that $\operatorname{nnz}(\mathbf{A}\otimes\mathbf{B}) = \operatorname{nnz}(\mathbf{A})\operatorname{nnz}(\mathbf{B})$. 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 $\operatorname{nnz}(\mathbf{C}) = \operatorname{nnz}(\mathbf{I}\otimes\mathbf{A}+\mathbf{A}\otimes\mathbf{I}) \leq \operatorname{nnz}(\mathbf{I}\otimes\mathbf{A}) + \operatorname{nnz}(\mathbf{A}\otimes\mathbf{I}) \leq 2\operatorname{nnz}(\mathbf{I})\operatorname{nnz}(\mathbf{A})$. Summarising, $\operatorname{nnz}(\mathbf{C}) < 2n\operatorname{nnz}(\mathbf{A})$. 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 $\operatorname{nnz}(\mathbf{C}) < 2n^3 \ll n^4$. 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.

HINT 1 for (2-9.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 (2-9.a):

There are many ways to come to the conclusion that all diagonal entries must be different from zero:

- The matrix $\mathbf{R}$ being upper triangular has determinant $\det(\mathbf{R}) = \prod_{i=1}^{n} (\mathbf{R})_{i,i}$, which 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.
- It is easy to verify (how?) that all columns of $\mathbf{R}$ are linearly independent, if and only if the diagonal entries do not vanish.

SOLUTION of (2-9.b):

From the first n equations of (2.9.2) we deduce

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

from which we find, using the bottom equation of (2.9.2)

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

assuming that the denominator does not vanish.

Alternatively, applying the block Gaussian 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 $s \neq 0$.

SOLUTION of (2-9.c):

Recall that the matrix $\mathbf{R}$ is known to be invertible. The square matrix $\mathbf{A}$ is regular, if the corresponding linear system has a unique solution for every right hand side vector. In the previous sub-problem, it was shown that the linear system is equivalent to the equations

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

If $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$, then ξ is uniquely determined by the former equation, and $\mathbf{z}$ by the latter. If $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} = 0$ and $\mathbf{u}^\top \mathbf{R}^{-1}\mathbf{b} = \beta$, then the equations are satisfied for any ξ (the linear system has an infinite number of solutions). If $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} = 0$ and $\mathbf{u}^\top \mathbf{R}^{-1}\mathbf{b} \neq \beta$, then the former equation is not satisfied (the linear system has no solution). This proves that $\mathbf{A}$ is regular if and only if $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$.

An alternative proof can be based on the block 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 (2-9.a).

SOLUTION of (2-9.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 2.9.3: Block Gauss based solution

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

  // TODO: (2-9.d)
  // i) Use assert() to check that R, v, u, bb all have the appropriate
     sizes.
  // ii) Use (3-9.b) to solve the LSE.
  // Hint:  Use R.triangularView<Eigen::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 Eigen::TriangularView<const Eigen::MatrixXd, Eigen::Upper>& triR =
      R.triangularView<Eigen::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
  const double sinv = -1. / u.dot(triR.solve(v));
  const double bs = (bb(n) - u.dot(triR.solve(bb.head(n))));
  const double sinvbs = sinv * bs;

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

Get it on GitLab (blockdecomp.hpp).

HINT 1 for (2-9.e): In order to obtain information about direct elimination solvers available in EIGEN consult  EIGEN documentation. ⌟

HINT 2 for (2-9.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 (2-9.e):

C++11-code 2.9.4: LU-decomposition based solution

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

  // TODO: (2-9.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 size of u, v, and size of bb is n+1
  const unsigned int n = R.rows();
  assert(n == R.cols() && n == u.size() && n == v.size() && n + 1 == b.size() &&
         "Size mismatch!");

  Eigen::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()
  Eigen::VectorXd y;
  solvelse(R, v, u, b, y);

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

  // END
  return areTheSame;
}
```

Get it on GitLab (blockdecomp.hpp).

SOLUTION of (2-9.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: $O(n^2)$.

SOLUTION of (3-1.a):

The resulting system of nonlinear equations is:

$$y_i = f(t_i) = \alpha e^{\beta t_i} \ , \quad i = 1, \ldots, n \ .$$

The non-linearity is due to the occurrence of parameter β as exponent of e.

HINT 1 for (3-1.b): Use the inverse function of the exponential. ⌟

SOLUTION of (3-1.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:

$$g(t) = \log f(t) = \log \alpha + \log \mathrm{e}^{\beta t} = a + \beta t \; .$$

Moreover, we impose $b_i := \log y_i$.

The new problem is therefore expressed by the following overdetermined linear system:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \text{with } \mathbf{A} = \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \vdots \\ 1 & t_n \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} a \\ \beta \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \log y_1 \\ \log y_2 \\ \vdots \\ \log y_n \end{bmatrix} \; .$$

HINT 1 for (3-1.c): Apply the definition of normal equations given in [Lecture → Section 3.1.2]. ⌟

SOLUTION of (3-1.c):

We determine the parameters of the normal equations $\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b}$. We obtain:

$$\mathbf{A}^\top \mathbf{A} = \begin{bmatrix} n & \sum_{i=1}^n t_i \\ \sum_{i=1}^n t_i & \sum_{i=1}^n t_i^2 \end{bmatrix}, \quad \mathbf{A}^\top \mathbf{b} = \begin{bmatrix} \sum_{i=1}^n b_i \\ \sum_{i=1}^n t_i b_i \end{bmatrix}. \tag{3.1.2}$$

Once we have found the least squares solution $\mathbf{x} = \begin{bmatrix} a \\ \beta \end{bmatrix}$, we just need to replace $\alpha = \mathrm{e}^a$ and β in function f.

SOLUTION of (3-1.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 (3.1.2).

C++ code 3.1.3: Solution of (3-1.d)

```
2  Eigen::VectorXd linReg(const Eigen::VectorXd &t, const Eigen::VectorXd &y) {
3    assert(t.size() == y.size() && "t and y must have same size");
4    Eigen::VectorXd x = Eigen::VectorXd::Zero(2);
5
6    // TODO: (3-1.d) Use the method of normal equations to solve the 1D
       linear
7    // regression problem in least-square sense.
8    // Note:  The tests anticipate the outputs in the order (alpha, beta),
       and not
9    // (beta, alpha).
10
11   // START
12
13   // Coefficient matrix of overdetermined linear system.
14   Eigen::MatrixXd A(t.size(), 2);
15   A.col(0).setOnes();
16   A.col(1) = t;
17
18   // Normal equation
19   Eigen::MatrixXd lhs = A.transpose() * A;  // Left-hand side
20   Eigen::VectorXd rhs = A.transpose() * y;  // Right-hand side
21
22   // Intercept and slope.
23   x = lhs.lu().solve(rhs);
24
25   // END
26
27   return x;
28 }
```

Get it on GitLab (adaptedlinreg.hpp).

HINT 1 for (3-1.e): Use the function `linReg` from (3-1.d). Take care to give the correct input $\mathbf{y}$ to `linReg` and to extract the right parameters α and β from its output. ⌟

SOLUTION of (3-1.e):

C++ code 3.1.4: Solution of (3-1.e)

```
2  Eigen::VectorXd expFit(const Eigen::VectorXd &t, const Eigen::VectorXd &y) {
3    assert(t.size() == y.size() && "t and y must have same size");
4    Eigen::VectorXd x = Eigen::VectorXd::Zero(2);
5
6    // TODO: (3-1.e) Implement least square estimate of alpha and beta
         using
7    // the previously implemented function linReg().
8    // Note:  You don't need to have implemented linReg() to solve this
         subproblem.
9    // The tests will compile and use the master solution for linReg().
10   // Note:  The tests anticipate the outputs in the order (alpha, beta),
         and not
11   // (beta, alpha).
12
13   // START
14
15   // Transform to a linear equation
16   // b := log(y) = x(0) + x(1)*t
17   // We use array() for coefficient-wise log() and matrix() to transform
         it
18   // back to a vector.
19   Eigen::VectorXd b = y.array().log().matrix();
20
21   // Solve and transform back to the nonlinear equation
22   // y = exp(x(0)) * exp(x(1)*t) = alpha*exp(beta*t)
23   x = linReg(t, b);
24   x(0) = exp(x(0));  // Eigen's coefficient-wise exp()
25
26   // END
27
28   return x;
29 }
```

Get it on GitLab (adaptedlinreg.hpp).

SOLUTION of (3-10.a):

Preliminary considerations, also relevant for the following sub-problems: (3.10.1) represents a linear system of $m := \frac{1}{2}n(n-1)$ equations for $n-1$ unknowns. For instance, for $n = 6$ that overdetermined linear system reads (up to permutations of the rows)

$$\begin{bmatrix} -\mathbf{1} & \mathbf{B}_5 & & & \\ & -\mathbf{1} & \mathbf{B}_4 & & \\ & & -\mathbf{1} & \mathbf{B}_3 & \\ & \mathbf{O} & & -\mathbf{1} & \mathbf{B}_2 \\ & & & & -\mathbf{1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_5 \end{bmatrix} = \begin{bmatrix} d_{12} \\ d_{13} \\ \vdots \\ \vdots \\ d_{16} \\ d_{23} \\ \vdots \\ \vdots \\ d_{5,6} \end{bmatrix}, \quad \mathbf{B}_k = \begin{bmatrix} \mathbf{I}_{k-1} \\ \mathbf{0}^\top \end{bmatrix} \in \mathbb{R}^{k,k-1}, \quad \mathbf{1} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}.$$

Again, up to permutations of the rows, its general structure is

$$\begin{bmatrix} & \mathbf{C}_{n-1} \\ \mathbf{O}_{n-2,1} & \mathbf{C}_{n-2} \\ \vdots & \vdots \\ \mathbf{O}_{k,n-1-k} & \mathbf{C}_k \\ \vdots & \vdots \\ \mathbf{O}_{2,n-3} & \mathbf{C}_2 \\ \mathbf{O}_{1,n-2} & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} d_{12} \\ d_{13} \\ \vdots \\ \vdots \\ d_{1n} \\ d_{23} \\ \vdots \\ \vdots \\ d_{n-1,n} \end{bmatrix}, \tag{3.10.2}$$

$$\mathbf{C}_k := \begin{bmatrix} -\mathbf{1} & \mathbf{B}_k \end{bmatrix} = \begin{bmatrix} -\mathbf{1} & \mathbf{I}_{k-1} \\ -1 & \mathbf{0}^\top \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 & \dots & \dots & 0 \\ \vdots & 0 & \ddots & \ddots & & \vdots \\ & \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & \ddots & \ddots & 0 \\ \vdots & \vdots & & & 0 & 1 \\ -1 & 0 & \dots & & \dots & 0 \end{bmatrix} \in \mathbb{R}^{k,k}.$$

Here $\mathbf{O}_{k,\ell}$ stands for a zero matrix of size $k \times \ell$. In $n-1$ rows there is only a single non-zero entry -1, in all other rows, there is one entry 1, another entry -1, and all remaining vanish. Hence,

$$\mathrm{nnz}(\mathbf{A}) = 2\binom{n}{2} - (n-1) = n(n-1) - (n-1) = (n-1)^2 \tag{3.10.3}$$

This implies that

- the COO format (→ [Lecture → § 2.7.1.1]) requires a minimum of $2(n-1)^2$ integers and $(n-1)^2$ real numbers (two integers and one real number for every non-zero entry),

- the CRS format needs $(n-1)^2 + \frac{1}{2}n(n-1) + 1 = (\frac{3}{2}n - 1)(n-1)$ integers (one for every non-zero entry, array `col_idx` and another one for every row plus one sentinel, array `row_ptr`) and $\mathrm{nnz}(A) = (n-1)^2$ real numbers, array `val`, see [Lecture → § 2.7.1.4].

One may add two integers for storing the matrix dimensions.

Remark. There is no other way to define the matrix $\mathbf{A}$ other than that given in (3.10.2), because the linear system of equations is completely defined by (3.10.1).

SOLUTION of (3-10.b):

There are two options:

1. Since we know that $\mathbf{A}$ has at most two non-zero entries per row, we can give this information through the `reserve()` method of **Eigen::SparseMatrix**. Then, the `insert()` method can be used to set the entries without the need to move data, see [Lecture → Code 2.7.2.1].
2. We can first build the matrix in COO format by creating a vector of suitable triplets and then call the `setFromTriplets()` method, see [Lecture → Section 2.7.2].

C++ code 3.10.4: Sub-problem (3-10.b): The function `initA()`

```
2  Eigen::SparseMatrix<double> initA(unsigned int n) {
3    const unsigned int rows = n * (n - 1) / 2;
4    const unsigned int cols = n - 1;
5    Eigen::SparseMatrix<double> A(rows, cols);
6    // TODO: (3-10.b) Initialize the sparse coefficient matrix for
7    // the distance fitting problem
8    // START
9    // Tell Eigen the number of non-zeros per column
10   A.reserve(Eigen::VectorXi::Constant(n - 1, n - 1));
11
12   // Loop through the n-1 columns of the matrix
13   for (unsigned int i = 0; i < cols; ++i) {
14     const unsigned int offset = cols * i - i * (i - 1) / 2;
15     for (unsigned int k = 0; k < cols - i; ++k) {
16       // Set the -1's in C_k from (3.10.2)
17       A.insert(offset + k, i) = -1;
18       // Set the identity blocks of C_k
19       if (k < cols - i - 1) {
20         A.insert(offset + k, i + k + 1) = 1;
21       }
22     }
23   }
24   // END
25   A.makeCompressed();
26   return A;
27 }
```

Get it on GitLab (distfitting.hpp).

HINT 1 for (3-10.c): You may use `initA()` from Sub-problem (3-10.b) and the helper function

```
std::vector<Eigen::Triplet<double>>
convertToTriplets(Eigen::SparseMatrix<double> &A);
```

which is provided in the file `totriplets.hpp`. With an effort proportional to $\mathrm{nnz}(\mathbf{A})$ it extracts the COO format of a sparse matrix and returns a corresponding vector of triplets. ⌟

SOLUTION of (3-10.c):

The main task is to build the coefficient matrix

$$\mathbf{A}_{\text{ext}} := \begin{bmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^\top & \mathbf{O} \end{bmatrix} \in \mathbb{R}^{n-1+m,n-1+m} \tag{3.10.6}$$

as a sparse matrix in CRS format, where $\mathbf{A} \in \mathbb{R}^{m,n-1}$ is generated by `initA()` from Sub-problem (3-10.b)

Remark. Unfortunately EIGEN's << operation for dense matrices, see [Lecture → § 1.2.1.3], is not available for sparse matrices.

The implementation can rely on `initA()` from Sub-problem (3-10.b), then use `convertToTriplets(` duplicate the triplets, adjust their indices, and, finally, build another sparse matrix. More precisely, we separately deal with the four different blocks of $\mathbf{A}_{\text{ext}}$:

1. $-\mathbf{I}$: triplets $(i,i,-1.0)$ for $i \in \{1,\ldots,m\}$,
2. $\mathbf{A}$: triplets of $\mathbf{A}$ with m added to column index,
3. $\mathbf{A}^\top$: triplets of $\mathbf{A}$ with swapped row/column indices and, subsequently, m added to row index,
4. $\mathbf{O}$: nothing to be done.

The following implementation uses the ordering of the equations is given in (3.10.2).

C++ code 3.10.7: Sub-problem (3-10.c): The function `solveExtendedNormalEquations()`

```
2  Eigen::VectorXd solveExtendedNormalEquations(const Eigen::MatrixXd& D) {
3    Eigen::VectorXd x;
4    const unsigned int n = D.cols();
5    const unsigned int m = n * (n - 1) / 2;
6    // TODO: (3-10.c) Solve the extended normal equations with a sparse
       solver
7    // that Eigen provides
8    // START
9    // initialize right hand side vector b as in (3.10.2)
10   Eigen::VectorXd b_ext = Eigen::VectorXd::Zero(m + n - 1);
11   unsigned int curr = 0;
12   for (unsigned int i = 0; i < n - 1; ++i) {
13     for (unsigned int k = i + 1; k < n; ++k) {
14       b_ext(curr) = D(i, k);
15       ++curr;
16     }
17   }
18   // initialize matrix A and convert to triplet format
19   Eigen::SparseMatrix<double> A = initA(n);
20   std::vector<Eigen::Triplet<double>> tripletsA = convertToTriplets(A);
21   // define triplets for extended normal equation
22   std::vector<Eigen::Triplet<double>> tripletsAext;
23   for (unsigned int k = 0; k < m; ++k) {
24     // - I block
25     tripletsAext.push_back(Eigen::Triplet<double>(k, k, -1.0));
26   }
27   for (auto t : tripletsA) {
28     // A block
29     tripletsAext.push_back(
```

```
30            Eigen::Triplet<double>(t.row(), m + t.col(), t.value()));
31        // A.transpose() block
32        tripletsAext.push_back(
33            Eigen::Triplet<double>(m + t.col(), t.row(), t.value()));
34    }
35
36    Eigen::SparseMatrix<double> A_ext(n - 1 + m, n - 1 + m);
37    A_ext.setFromTriplets(tripletsAext.begin(), tripletsAext.end());
38    // solve extended system and extract the last n-1 entries
39    // of the solution vector
40    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver(A_ext);
41    x = solver.solve(b_ext).tail(n - 1);
42    // END
43    return x;
44 }
```

Get it on GitLab (distfitting.hpp).

Alternatively, the matrix of the extended normal equations can be built from scratch based on an intermediate COO format. This time, direct initialization of a sparse matrix and the use of `reserve()` is complicated, because of different numbers of non-zeros entries in different row/columns of the system matrix of the extended normal equations.

HINT 1 for (3-10.d): Exploit the Sherman-Morrison-Woodbury formula from [Lecture → Lemma 2.6.0.21]

SOLUTION of (3-10.d):

By the formula $\mathbf{M} = \mathbf{A}^\top \mathbf{A}$ we see that the entries of $\mathbf{M}$ are the inner products of the columns of $\mathbf{A}$. From the structure of $\mathbf{A}$ as presented in (3.10.2) we see that different columns of $\mathbf{A}$ have both non-vanishing entries ± 1 (with different signs) in exactly a single row. Thus, their inner product will always be -1. The inner product of every column with itself is $= n-1$ for all columns. This means

$$[\mathbf{M}]_{i,j} = \begin{cases} -1 & \text{, if } i \neq j \text{ ,} \\ n-1 & \text{, if } i = j \end{cases} \quad \blacktriangleright \quad \mathbf{M} = n\mathbf{I}_{n-1} - \mathbf{1}\mathbf{1}^\top \text{ .} \tag{3.10.9}$$

We conclude that

$$\mathbf{D} = n\mathbf{I}_{n-1} \quad , \quad \mathbf{z} = [1, \ldots, 1]^\top \in \mathbb{R}^{n-1} \text{ .}$$

Note that the ordering of rows of $\mathbf{A}$ does not matter for $\mathbf{M}$.

Recall the formula for the inverse of a matrix under low-rank modification:

Lemma [Lecture → Lemma 2.6.0.21]. Sherman-Morrison-Woodbury formula

For regular $\mathbf{A} \in \mathbb{K}^{n,n}$, and $\mathbf{U}, \mathbf{V} \in \mathbb{K}^{n,k}$, $n, k \in \mathbb{N}$, $k \leq n$, holds

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I}_k + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^H\mathbf{A}^{-1} \text{ ,}$$

if $\mathbf{I}_k + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U}$ *is regular.*

We apply this for $k = 1$ with

- $\mathbf{A} \leftarrow n\mathbf{I}_{n-1}$, which means $\mathbf{A}^{-1} = \frac{1}{n}\mathbf{I}_{n-1}$,
- $\mathbf{U} \leftarrow -\mathbf{1}$ and $\mathbf{V} \leftarrow \mathbf{1}$.

$$\blacktriangleright \quad \mathbf{M}^{-1} = \tfrac{1}{n}I_{n-1} + \tfrac{1}{n}\mathbf{1}\frac{1}{1 - \frac{1}{n}\mathbf{1}^\top\mathbf{1}}\tfrac{1}{n}\mathbf{1}^\top = \tfrac{1}{n}\left(\mathbf{I}_{n-1} + \mathbf{1}\mathbf{1}^\top\right) \text{ ,} \tag{3.10.10}$$

since $\mathbf{1}^\top\mathbf{1} = n - 1$. This formula can be verified directly

$$(n\mathbf{I}_{n-1} - \mathbf{1}\mathbf{1}^\top)\frac{1}{n}\left(\mathbf{I}_{n-1} + \mathbf{1}\mathbf{1}^\top\right) = \mathbf{I}_{n-1} + (-\frac{1}{n} + 1 - \frac{n-1}{n})\mathbf{1}\mathbf{1}^\top = \mathbf{I}_{n-1} \text{ .}$$

HINT 1 for (3-10.e): Your routine may call the function `initA()` from Sub-problem (3-10.b). ⌟

SOLUTION of (3-10.e):

The normal equations are $\mathbf{M}\mathbf{x} = \mathbf{A}^\top\mathbf{b}$, with $\mathbf{M}^{-1}$ given in (3.10.10), hence

$$\mathbf{x} = \frac{1}{n}\left(\mathbf{I}_{n-1} + \mathbf{1}\mathbf{1}^\top\right)\mathbf{A}^\top\mathbf{b}\,.$$

This means that the main effort has to go into computing $\mathbf{A}^\top\mathbf{b}$. You may simply use `initA()` from Sub-problem (3-10.b) to that end.

C++ code 3.10.11: Sub-problem (3-10.e): The function `solveNormalEquations()`

```
2  Eigen::VectorXd solveNormalEquations(const Eigen::MatrixXd& D) {
3    Eigen::VectorXd x;
4    const unsigned int n = D.cols();
5    const unsigned int m = n * (n - 1) / 2;
6    // TODO: (3-10.e) Solve the normal equations by exploiting the
7    // Sherman-Morrison-Woodbury formula, i.e. by using your results from the
8    // previous subproblem
9    // START
10   // initialize right hand side
11   Eigen::VectorXd b = Eigen::VectorXd::Zero(m);
12   unsigned int curr = 0;
13   for (unsigned int i = 0; i < n - 1; ++i) {
14     for (unsigned int k = i + 1; k < n; ++k) {
15       b(curr) = D(i, k);
16       ++curr;
17     }
18   }
19   // Direct computation of A^T*b
20   Eigen::SparseMatrix<double> AT = initA(n).transpose();
21   Eigen::VectorXd ATb = AT * b;
22
23   // Alternative version
24
25   // Initialize the extended D
26   Eigen::MatrixXd D_full = Eigen::MatrixXd::Zero(n, n);
27   D_full.triangularView<Eigen::Upper>() =
28       D.triangularView<Eigen::Upper>();  // in case D is not just a upper
29                                          // triangular matrix
30   Eigen::MatrixXd D_fullT = D_full.transpose();
31   D_full = D_full - D_fullT;
32   // Calculate ATb in a more efficient way
33   Eigen::VectorXd ATb_alt =
34       D_full.transpose().block(0, 0, n - 1, n) * Eigen::VectorXd::Ones(n);
35
36   // We use ATb on forward but ATb == ATb_alt
37
38   // Use explicit expression of the inverse of A^T*A
39   // to compute the solution
40   x = (Eigen::MatrixXd::Identity(n - 1, n - 1) +
41        Eigen::MatrixXd::Ones(n - 1, n - 1)) *
42       ATb / n;
43   // END
44   return x;
45 }
```

Get it on GitLab (distfitting.hpp).

Of course, the structure of $\mathbf{A}$ could be exploited. First, one completely fills the matrix $\mathbf{D}$:

$$[\mathbf{D}]_{i,j} = \begin{cases} d_{i,j} & \text{, if } i < j \text{ ,} \\ 0 & \text{, if } i = j \text{ ,} \\ -d_{i,j} & \text{, if } i > j \text{ .} \end{cases}$$

Then we have

$$\left[\mathbf{A}^\top \mathbf{b}\right]_\ell = \sum_{k=1}^{n} [\mathbf{D}]_{k,\ell} \text{ , } \quad \ell = 1, \ldots, n-1 \quad \Rightarrow \quad \mathbf{A}^\top \mathbf{b} = \left[\mathbf{D}^\top\right]_{1:n-1,:} \mathbf{1} \text{ .}$$

HINT 1 for (3-10.f): Heed the advice given in [Lecture → Rem. 1.5.3.15].

SOLUTION of (3-10.f):

You must not test for equality of results, but you have to check whether they are relatively close.

C++ code 3.10.12: Sub-problem (3-10.f): The function `testNormalEquations()`

```
2  bool testNormalEquations(const Eigen::MatrixXd& D) {
3    constexpr double tol = 1e-9;  // tolerance to check for
4    // TODO: (3-10.f) Call your implementations of solveExtendedNormalEquations()
5    // and solveNormalEquations() and return true, if their results agree.
6    // START
7    Eigen::VectorXd x_ext = solveExtendedNormalEquations(D);
8    Eigen::VectorXd x_fast = solveNormalEquations(D);
9    if ((x_ext - x_fast).norm() < x_ext.norm() * tol) return true;
10   // END
11   return false;
12 }
```

Get it on GitLab (distfitting.hpp).

SOLUTION of (3-10.g):

The straightforward matrix$\times$vector multiplication $\mathbf{A}^\top\mathbf{b}$ incurs an asymptotic effort of $O(n^2)$ for $n \to \infty$, because, thanks to the sparse matrix format, the zero entries of $\mathbf{A}$ are skipped when doing the matrix$\times$vector product.

The more sophisticated implementation making use of the structure of $\mathbf{A}$ also costs $O(n^2)$ for $n \to \infty$. This is the optimal complexity, because it agrees with the amount of data.

HINT 1 for (3-11.a): You may appeal to the singular-value decomposition of $\mathbf{M}$, in particular [Lecture → Thm. 3.4.1.1].

Theorem [Lecture → Thm. 3.4.1.1]. Singular value decomposition

For any $\mathbf{A} \in \mathbb{R}^{m,n}$ *there are* unitary *matrices* $\mathbf{U} \in \mathbb{R}^{m,m}$, $\mathbf{V} \in \mathbb{R}^{n,n}$ *and a (generalized) diagonal matrix* $\mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m,n}$, $p := \min\{m,n\}$, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$ *such that*

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top .$$

SOLUTION of (3-11.a):

Let $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, be the singular-value decomposition of $\mathbf{M}$ according to [Lecture → Thm. 3.4.1.1] with unitary matrices $\mathbf{U} \in \mathbb{R}^{m,m}$, $\mathbf{V} \in \mathbb{R}^{n,n}$ and a generalized diagonal matrix $\mathbf{\Sigma}$ with diagonal entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min\{m,n\}}$. Thanks to [Lecture → Lemma 3.4.1.13], if $\operatorname{rank}(\mathbf{M}) = r \leq \min\{m,n\}$, then $\sigma_\ell = 0$ for all $\ell > r$. Hence,

$$\mathbf{M} = \mathbf{U}[\mathbf{\Sigma}]_{:,1:r}\left([\mathbf{V}]_{:,1:r}\right)^\top \quad \blacktriangleright \quad \begin{aligned} &\mathbf{A} := \mathbf{U}[\mathbf{\Sigma}]_{:,1:r} \in \mathbb{R}^{m,r} \,, \\ &\mathbf{B} := [\mathbf{V}]_{:,1:r} \in \mathbb{R}^{n,r} \,. \end{aligned}$$

Note that $\mathbf{A}$ and $\mathbf{B}$ are not unique and other combinations are possible as well.

Alternative proof starts from a basis $\{\mathbf{a}_1, \ldots, \mathbf{a}_r\} \subset \mathbb{R}^m$ of $\mathcal{R}\mathbf{M}$:

$$\mathbf{M}\mathbf{x} = \sum_{j=1}^{r} \eta_j(\mathbf{x})\, \mathbf{a}_j \,, \quad \mathbf{x} \in \mathbb{R}^n \,.$$

The mapping $\mathbf{x} \mapsto \eta_j(\mathbf{x})$ is linear, hence, a linear form on $\mathbb{R}^n$. Every such linear form can be represented by a vector:

$$\exists \mathbf{b}_j \in \mathbb{R}^n\!: \quad \eta_j(\mathbf{x}) = \mathbf{b}_j^\top \mathbf{x} \,, \quad j = 1, \ldots, r \,,$$

$$\blacktriangleright \quad \mathbf{M}\mathbf{x} = \sum_{j=1}^{r} \mathbf{a}_j \mathbf{b}_j^\top \mathbf{x} = \mathbf{A}\mathbf{B}^\top \mathbf{x} \,, \quad \begin{aligned} &\mathbf{A} := [\mathbf{a}_1, \ldots, \mathbf{a}_r] \,, \\ &\mathbf{B} := [\mathbf{b}_1, \ldots, \mathbf{b}_r] \,. \end{aligned}$$

SOLUTION of (3-11.b):

Of course, for the sake of efficient implementation, compute the matrix products in the right order, recall [Lecture → Ex. 1.4.3.1]:

$$\mathbf{MX} = \mathbf{A}(\mathbf{B}^\top \mathbf{X}) , \quad \mathbf{X} \in \mathbb{R}^{n,k} .$$

C++ code 3.11.2: Sub-problem (3-11.b): Multiplication of a low-rank matrix with a generic matrix

```
2  Eigen::MatrixXd MatrixLowRank::operator*(const Eigen::MatrixXd &X) const {
3    Eigen::MatrixXd MX =
4        Eigen::MatrixXd::Zero(_m, _n);  // will contain the result.
5    // TODO: (3-11.b) Implement the multiplication of the m × n-matrix
6    // M = AB^T stored in this class with the given eigen matrix X.
7    // Note: The implementation should be efficient.
8
9    // START
10   // Efficient associative matrix multiplication
11   MX = _A * (_B.transpose() * X);
12   // END
13
14   return MX;
15 }
```

Get it on GitLab (matrixlowrank.hpp).

SOLUTION of (3-11.c):

Let $\mathbf{X} \in \mathbb{R}^{n,k}$ denote the argument matrix. The matrix product $\mathbf{B}^\top \mathbf{X}$ requires rnk operations. The second matrix product can be done with rmk operations. Hence the total operation count is $r(m+n)k$, with an asymptotic behavior of $O(k(m+n))$, if r is kept fixed.

SOLUTION of (3-11.d):

From linear algebra we know that

$$\operatorname{rank}(\mathbf{XY}) \leq \min\{\operatorname{rank}(\mathbf{X}), \operatorname{rank}(\mathbf{Y})\} \quad \forall \mathbf{X} \in \mathbb{R}^{m,k},\ \mathbf{Y} \in \mathbb{R}^{k,n} .$$

Hence, if the argument matrix $\mathbf{X}$ of **operator** $\star=$ is an $n \times k$-matrix, we know that

$$\operatorname{rank}(\mathbf{M} \cdot \mathbf{X}) \leq \min\{r, k\} , \quad r := \operatorname{rank}(\mathbf{M}) ,$$

where $\mathbf{M} \in \mathbb{R}^{m,n}$ is the matrix stored in the **MatrixLowRank** object. Thus, the *new rank* of that **MatrixLowRank** object will be at most $r' := \min\{r, k\} \leq r$.

However, since the problem statement asked for preserving the internal rank, the new factors simply are

$$\mathbf{A} := \mathbf{A} \quad , \quad \mathbf{B}' := \mathbf{X}^\top \mathbf{B} \in \mathbb{R}^{k,r} .$$

C++ code 3.11.3: Sub-problem (3-11.d): In-situ multiplication of a low-rank matrix with a generic matrix

```
2  MatrixLowRank &MatrixLowRank::operator*=(const Eigen::MatrixXd &X) {
3    // TODO: (3-11.d) Implement the *= operator for the in-situ multiplication of
4    // the m x n-matrix M = AB^T stored in this class with the given
5    // eigen matrix X.
6
7    // START
8    _B = X.transpose() * _B;
9    _n = X.cols();
10   // END
11
12   return *this;
13 }
```

Get it on GitLab (matrixlowrank.hpp).

You must not forget to set `_n`!

HINT 1 for (3-11.e): The sum of two low-rank matrices given in factorized representation is another low-rank matrix, whose factors are readily available. ⌟

HINT 2 for (3-11.e): The algorithm is based on

- the SVD [Lecture → Thm. 3.4.4.19],
- and the economical QR-decompositions (→ [Lecture → Thm. 3.3.3.4]) of the factors representing a low-rank matrix.

⌟

SOLUTION of (3-11.e):

For additional explanations you may also refer to Problem 3-8.

Using

$$\begin{aligned}\mathbf{M} &= \mathbf{A}\mathbf{B}^\top\,, \qquad \mathbf{A}\in\mathbb{R}^{m,r}\,,\quad \mathbf{B}\in\mathbb{R}^{n,r}\,,\\ \mathbf{X} &= \mathbf{A}_X\mathbf{B}_X^\top\,, \qquad \mathbf{A}_X\in\mathbb{R}^{m,s}\,,\quad \mathbf{B}_X\in\mathbb{R}^{n,s}\,,\end{aligned}$$

with ranks $r,s\le\min\{m,n\}$, $r+s\le\min\{m,n\}$, we find

$$\mathbf{M}+\mathbf{X} = \mathbf{A}\mathbf{B}^\top+\mathbf{A}_X\mathbf{B}_X^\top = \big[\mathbf{A},\mathbf{A}_X\big]\big[\mathbf{B},\mathbf{B}_X\big]^\top =: \widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top\,.$$

with $\widetilde{\mathbf{A}}\in\mathbb{R}^{m,r+s}$, $\widetilde{\mathbf{B}}\in\mathbb{R}^{n,r+s}$. Use the economical QR-decompositions

$$\begin{aligned}\widetilde{\mathbf{A}} &= \mathbf{Q}_A\mathbf{R}_A \quad \mathbf{Q}_A\in\mathbb{R}^{m,r+s},\ \mathbf{Q}_A^\top\mathbf{Q}_A=\mathbf{I}_{r+s}\,,\quad \mathbf{R}_A\in\mathbb{R}^{r+s,r+s}\text{ upper triangular}\,,\\ \widetilde{\mathbf{B}} &= \mathbf{Q}_B\mathbf{R}_B \quad \mathbf{Q}_B\in\mathbb{R}^{n,r+s},\ \mathbf{Q}_B^\top\mathbf{Q}_B=\mathbf{I}_{r+s}\,,\quad \mathbf{R}_B\in\mathbb{R}^{r+s,r+s}\text{ upper triangular}\,.\end{aligned}$$

Writing

$$\mathbf{R}_A\mathbf{R}_B^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top\,,\quad \mathbf{U},\mathbf{V}\in\mathbb{R}^{r+s,r+s}\text{ orthogonal},\ \mathbf{\Sigma}\in\mathbb{R}^{r+s,r+s}\text{ diagonal}\,,$$

for the singular-value decomposition of $\mathbf{R}_A\mathbf{R}_B^\top$, we obtain an economical singular value decomposition of $\widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top$:

$$\widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top = \widetilde{\mathbf{U}}\widetilde{\mathbf{\Sigma}}\widetilde{\mathbf{V}}^\top\quad,\quad \begin{aligned}\widetilde{\mathbf{U}} &:= \mathbf{Q}_A\mathbf{U}\in\mathbb{R}^{m,r+s}\,,\\ \widetilde{\mathbf{\Sigma}} &:= \mathbf{\Sigma}\in\mathbb{R}^{r+s,r+s}\\ \widetilde{\mathbf{V}} &:= \mathbf{Q}_B\mathbf{V}\in\mathbb{R}^{n,r+s}\,.\end{aligned}$$

Denote by $\sigma_1\ge\sigma_2\ge\cdots\ge\sigma_{r+s}\ge 0$ the sorted diagonal entries of $\widetilde{\mathbf{\Sigma}}$, that is, the singular values of $\widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top$. Define (for $\texttt{tol}<1$)

$$\widetilde{r} := \max\left\{r'\in\{1,\ldots,r+s\}:\ \frac{\sigma_{r'}}{\sigma_1}>\texttt{rtol},\ \sigma_{r'}>\texttt{atol}\right\}\,. \tag{3.11.4}$$

By [Lecture → Thm. 3.4.4.19] $\widetilde{r}$ is the smallest rank of a matrix that approximates $\widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top$ with a relative error $\le\texttt{rtol}$ and an absolute error $\le\texttt{atol}$ in the Euclidean matrix norm. That matrix, realizing the best approximation is

$$\texttt{trunc}_{\texttt{tol}} = \underbrace{\Big(\mathbf{Q}_A[\mathbf{U}]_{:,1:\widetilde{r}}[\mathbf{\Sigma}]_{1:\widetilde{r},1:\widetilde{r}}\Big)}_{=:\mathbf{A}_{\text{new}}}\underbrace{\Big(\mathbf{Q}_B[\mathbf{V}]_{:,1:\widetilde{r}}\Big)}_{=:\mathbf{B}_{\text{new}}}^\top\,,$$

which already gives us the new low-rank factors $\mathbf{A}_{\text{new}}$ and $\mathbf{B}_{\text{new}}$ together with the new rank $\widetilde{r}$, which has to be stored in `_r`.

In case $\sigma_1\le\texttt{atol}$, we conclude that $\mathbf{X}+\mathbf{M}=\mathbf{O}$. In this case, following the convention, we set `_r = 1` and the low-rank factors to zero, to indicate that the matrix stored in the **MatrixLowRank** object vanishes.

C++ code 3.11.5: Sub-problem (3-11.e): Addition and recompression of two low-rank matrices

```
MatrixLowRank &MatrixLowRank::addTo(const MatrixLowRank &X, double rtol,
                                    double atol) {
  assert(_m == X._m && _n == X._n && "Matrices must have the same shape!");
  assert((rank() + X.rank() <= std::min(_n, _m)) && "Sum of ranks too large!");
  assert(rtol > 0 && atol > 0 && "Tolerances must be positive!");
  // TODO: (3-11.e) Implement the function s.t. the m×n-matrix stored
  // in this class is replaced by trunc_tol(M + X).
  // Note: X is not an eigen matrix but a matrix of type MatrixLowRank.
  // Note: We have this = _A*_B.transpose()
  // Note: We have X = X._A*X._B.transpose()
  // Note: This function should be efficient.

  // START
  const unsigned int r = _r, s = X._r;
  // We will have X+this = Atilde*Btilde.transpose()
  Eigen::MatrixXd Atilde(_m, r + s), Btilde(_n, r + s);
  Atilde << _A, X._A;
  Btilde << _B, X._B;

  // Calculate the QR decomposition of Atilde and Btilde.
  Eigen::HouseholderQR<Eigen::MatrixXd> qrA(Atilde);
  Eigen::HouseholderQR<Eigen::MatrixXd> qrB(Btilde);
  Eigen::MatrixXd RA = qrA.matrixQR().triangularView<Eigen::Upper>();
  Eigen::MatrixXd RB = qrB.matrixQR().triangularView<Eigen::Upper>();

  // We need the full SVD of RR = RA * RB.transpose()
  Eigen::JacobiSVD<Eigen::MatrixXd> svdRR(
      RA * RB.transpose(), Eigen::ComputeFullU | Eigen::ComputeFullV);

  // Rank of the sum X+this is the (numerical) rank of RA*RB.transpose().
  Eigen::VectorXd Sigma = svdRR.singularValues();
  const double sigma1 = Sigma[0];
  // Take care of the case sigma1=0, i.e. X+this = 0.
  // Follow the convention for storing the zero matrix
  if (sigma1 < atol) {
    _r = 1;
    _A = Eigen::MatrixXd::Zero(_m, 1);
    _B = Eigen::MatrixXd::Zero(_n, 1);

    return *this;
  }

  unsigned int rtilde = 1;
  // Numerical rank is at least 1.
  while (rtilde < Sigma.size() && Sigma[rtilde] > atol &&
         Sigma[rtilde] > rtol * sigma1) {
    // Numerical rank is at least rtilde+1.
    rtilde++;
  }

  _r = rtilde;
  _A = qrA.householderQ() * svdRR.matrixU().leftCols(rtilde) *
       Sigma.head(rtilde).asDiagonal();
  _B = qrB.householderQ() * svdRR.matrixV().leftCols(rtilde);
  // END
```

```
57
58     return *this;
59 }
```

Get it on GitLab (matrixlowrank.hpp).

HINT 1 for (3-12.a): Thm. 3.12.1 can be shown based on the singular value decomposition (SVD) of $\mathbf{X}$ [Lecture $\rightarrow$ Thm. 3.4.1.1]. ⌟

SOLUTION of (3-12.a):

Recall the economical SVD of $\mathbf{X} \in \mathbb{R}^{m,n}$, $m \geq n$:

$$\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top\,, \quad \mathbf{U} \in \mathbb{R}^{m,n},\ \mathbf{U}^\top\mathbf{U} = \mathbf{I}_n\,, \quad \mathbf{V} \in \mathbb{R}^{n,n} \text{ orthogonal}\,,$$

where $\boldsymbol{\Sigma} \in \mathbb{R}^{n,n}$ is a diagonal matrix with non-negative diagonal entries. Setting

$$\mathbf{Q} := \mathbf{U}\mathbf{V}^\top \quad \text{and} \quad \mathbf{M} := \mathbf{V}\boldsymbol{\Sigma}\mathbf{V}^\top \tag{3.12.2}$$

we have already found the polar decomposition of $\mathbf{X}$, because

$$\begin{aligned}
&\mathbf{M}^\top = \left(\mathbf{V}\boldsymbol{\Sigma}\mathbf{V}^\top\right)^\top = \left(\mathbf{V}^\top\right)^\top\boldsymbol{\Sigma}^\top\mathbf{V}^\top = \mathbf{V}\boldsymbol{\Sigma}\mathbf{V}^\top = \mathbf{M}\,,\\
&\mathbf{Q}\mathbf{M} = \mathbf{U}\mathbf{V}^\top\mathbf{V}\boldsymbol{\Sigma}\mathbf{V}^\top = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top = \mathbf{X} \quad \mathbf{V} \text{ orthogonal!}\,,\\
&\mathbf{Q}^\top\mathbf{Q} = \mathbf{V}\mathbf{U}^\top\mathbf{U}\mathbf{V}^\top = \mathbf{V}\mathbf{V}^\top = \mathbf{I}_n\,,\\
&\mathbf{z}^\top\mathbf{M}\mathbf{z} = (\mathbf{V}\mathbf{z})^\top\boldsymbol{\Sigma}(\mathbf{V}\mathbf{z}) = \sum_{j=1}^{n} \underbrace{(\boldsymbol{\Sigma})_{j,j}}_{\geq 0}(\mathbf{V}\mathbf{z})_j^2 \geq 0 \quad \forall \mathbf{z} \in \mathbb{R}^n\,.
\end{aligned}$$

SOLUTION of (3-12.b):

The minimum cost is $O(mn + n^2)$ for $m, n \to \infty$, because the dense-matrix data members `_Q` ($\leftrightarrow \mathbf{Q} \in \mathbb{R}^{m,n}$) and `_M` ($\leftrightarrow \mathbf{M} \in \mathbb{R}^{n,n}$) have to be initialized, which involves accessing/computing each of their entries at least once.

HINT 1 for (3-12.c): You can rely on EIGEN's built-in facilities for computing the SVD of dense matrices, see [Lecture → Code 3.4.2.1]. ⌟

SOLUTION of (3-12.c):

We follow the SVD-based proof requested in Sub-problem (3-12.a), starting with the economical SVD $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ of $\mathbf{X}$ and then setting $\mathbf{Q} = \mathbf{U}\mathbf{V}^\top$, $\mathbf{M} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^\top$. Concerning the use of EIGEN's built-in SVD refer to [Lecture → Code 3.4.2.1].

C++ code 3.12.3: Constructing a polar decomposition.

```
2  void PolarDecomposition::initialize(const Eigen::MatrixXd &X) {
3    assert(X.rows() >= X.cols());
4    // TODO: (3-12.c) Implement a method to initialize the data members Q_ and M_
5    // corresponding to Q and M in Theorem 0.3.1, where X = QM
6    // START
7    // Compute SVD
8    Eigen::JacobiSVD<Eigen::MatrixXd> svd(
9        X, Eigen::ComputeThinU | Eigen::ComputeFullV);
10   // Extract factor matrices
11   const Eigen::MatrixXd &V{svd.matrixV()};
12   const Eigen::MatrixXd &U{svd.matrixU()};
13   const auto Sigma{svd.singularValues().asDiagonal()};
14   // Form factors of polar decomposition
15   Q_ = U * V.transpose();
16   M_ = V * Sigma * V.transpose();
17   // END
18 }
```

Get it on GitLab (polardecomposition.hpp).

HINT 1 for (3-12.d): One idea is to harness economical/thin QR-decompositions. ⌟

SOLUTION of (3-12.d):

You cannot simply call `initialize(A*B.transpose())`, which would incur an asymptotic computational effort of $O(mn^2)$ for $m,n \to \infty$.

In Sub-problem (3-12.b) we have found that the best possible asymptotic complexity is $O(mn+n^2)$, which is proportional to the number of matrix entries to be set for the data members `Q_` and `M_` of **PolarDecomposition**. This what we are aiming for.

A flawed approach. We start with the economical QR-decompositions of both $\mathbf{A}$ and $\mathbf{B}$

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

(computational cost = $O(k^2m)$ and $O(k^2n)$, respectively, see [Lecture → § 3.3.3.24], [Lecture → § 3.3.3.37 [Lecture → Eq. (3.3.3.25)]) and then use the SVD of the square matrix

$$\mathbf{R}_A\mathbf{R}_B^\top \in \mathbb{R}^{k,k}\colon \quad \mathbf{R}_A\mathbf{R}_B^\top = \widehat{\mathbf{U}}\widehat{\mathbf{\Sigma}}\widehat{\mathbf{V}}^\top \tag{3.12.4}$$

which comes at a fixed "$O(1)$" computational cost, because the effort depends on k alone. All its factors are $\in \mathbb{R}^{k,k}$ and both $\widehat{\mathbf{U}}$ and $\widehat{\mathbf{V}}$ are orthogonal. The SVD from (3.12.4) can be used to obtain an economical SVD of $\mathbf{X}$:

$$\mathbf{X} = \mathbf{A}\mathbf{B}^\top = \mathbf{Q}_A\mathbf{R}_A\mathbf{R}_B^\top\mathbf{Q}_B^\top = \underbrace{\mathbf{Q}_A\widehat{\mathbf{U}}}_{=:\mathbf{U}}\underbrace{\widehat{\mathbf{\Sigma}}}_{=:\mathbf{\Sigma}}\underbrace{\widehat{\mathbf{V}}^\top\mathbf{Q}_B^\top}_{=:\mathbf{V}^\top}\,, \tag{3.12.5}$$

$$\text{with} \quad \mathbf{U} \in \mathbb{R}^{m,k}\,, \quad \mathbf{V} \in \mathbb{R}^{n,k}\,, \quad \mathbf{\Sigma} \in \mathbb{R}^{k,k}\,,$$

and

$$\begin{aligned}\mathbf{U}^\top\mathbf{U} &= \widehat{\mathbf{U}}^\top\mathbf{Q}_A^\top\mathbf{Q}_A\widehat{\mathbf{U}} = \widehat{\mathbf{U}}^\top\widehat{\mathbf{U}} = \mathbf{I}_k\,,\\ \mathbf{V}^\top\mathbf{V} &= \widehat{\mathbf{V}}^\top\mathbf{Q}_B^\top\mathbf{Q}_B\widehat{\mathbf{V}} = \widehat{\mathbf{U}}^\top\widehat{\mathbf{U}} = \mathbf{I}_k\,.\end{aligned}$$

These matrix factors can be computed with a cost of $O(m+n)$ for $m,n \to \infty$, k small and fixed.

Now it is tempting to form the factors of the polar decomposition as in (3.12.2),

$$\begin{aligned}\mathbf{Q} &:= \mathbf{U}\mathbf{V}^\top \quad \text{cost} = O(mn)\,,\\ \mathbf{M} &:= \mathbf{V}\mathbf{\Sigma}\mathbf{V}^\top \quad \text{cost} = O(n^2)\,.\end{aligned}$$

However, the matrix $\mathbf{Q}$ fails to meet the requirements laid out in Thm. 3.12.1, because

$$\mathbf{Q}^\top\mathbf{Q} = \mathbf{V}\mathbf{U}^\top\mathbf{U}\mathbf{V}^\top = \mathbf{V}\mathbf{V}^\top \neq \mathbf{I}_n\,, \quad \text{if} \quad k < n\,.$$

The correct approach. We start with the full QR-decompositions of both $\mathbf{A}$ and $\mathbf{B}$:

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

From [Lecture → § 3.3.3.37], [Lecture → § 3.3.3.36] we remember two key facts:

- The orthogonal matrices $\mathbf{Q}_A \in \mathbb{R}^{m,m}$ and $\mathbf{Q}_B \in \mathbb{R}^{n,n}$ are stored in factorized form as products of k Householder reflections as explained in [Lecture → Rem. 3.3.3.21],
- and therefore the cost of computing the QR-factorizations of $\mathbf{A}$ and $\mathbf{B}$ merely is $O(mk^2)$ and $O(nk^2)$, respectively, for $m, n \to \infty$.

Also be aware that

cost($\mathbf{Q}_A \times$vector) = $O(mk)$ for $m, k \to \infty$,
cost($\mathbf{Q}_B \times$vector) = $O(nk)$ for $n, k \to \infty$.

Then we continue with the SVD of $\mathbf{R}_A \mathbf{R}_B^\top$

$$\mathbf{R}_A \mathbf{R}_B^\top \in \mathbb{R}^{k,k}: \quad \mathbf{R}_A \mathbf{R}_B^\top = \widehat{\mathbf{U}} \widehat{\mathbf{\Sigma}} \widehat{\mathbf{V}}^\top \,. \tag{3.12.4}$$

Next we introduce

$$\mathbf{U} = \mathbf{Q}_A \begin{bmatrix} \widehat{\mathbf{U}} & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{I}_{n-k} \\ \mathbf{O}_{m-n,k} & \mathbf{O}_{m-n,n-k} \end{bmatrix} \in \mathbb{R}^{m,n} \quad , \quad \mathbf{V} = \mathbf{Q}_B \begin{bmatrix} \widehat{\mathbf{V}} & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{I}_{n-k} \end{bmatrix} \in \mathbb{R}^{n,n} \,.$$

Observe that both block matrices in these expressions have orthonormal columns. Since $\mathbf{Q}_A$ and $\mathbf{Q}_B$ are orthogonal, this property carries over to $\mathbf{U}$ and $\mathbf{V}$:

$$\mathbf{U}^\top \mathbf{U} = \mathbf{I}_n \quad \text{and} \quad \mathbf{V}^\top \mathbf{V} = \mathbf{I}_n \,. \tag{3.12.6}$$

Moreover, $\mathbf{V}$ is an *orthogonal* matrix. Therefore we can set

$$\mathbf{Q} := \mathbf{U}\mathbf{V}^\top \in \mathbb{R}^{m,n} \quad , \quad \mathbf{M} := \mathbf{V} \begin{bmatrix} \widehat{\mathbf{\Sigma}} & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{O}_{n-k,n-k} \end{bmatrix} \mathbf{V}^\top \in \mathbb{R}^{n,n} \,,$$

and easily verify

(i) $\mathbf{Q}^\top \mathbf{Q} = \mathbf{V}\mathbf{U}^\top \mathbf{U}\mathbf{V}^\top = \mathbf{V}\mathbf{V}^\top = \mathbf{I}_n$,

(ii) $\mathbf{M} = \mathbf{M}^\top$ positive semi-definite ,

(iii)
$$\begin{aligned} \mathbf{Q}\mathbf{M} &= \mathbf{U}\mathbf{V}^\top \mathbf{V} \begin{bmatrix} \widehat{\mathbf{\Sigma}} & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{O}_{n-k,n-k} \end{bmatrix} \mathbf{V}^\top \\ &= (\mathbf{U})_{:,1:k} \widehat{\mathbf{\Sigma}} (\mathbf{V})_{:,1:k}^\top \\ &= (\mathbf{Q}_A)_{:,1:k} \widehat{\mathbf{U}} \widehat{\mathbf{\Sigma}} \widehat{\mathbf{V}} (\mathbf{Q}_B)_{:,1:k}^\top \\ &= (\mathbf{Q}_A)_{:,1:k} \mathbf{R}_A \mathbf{R}_B^\top (\mathbf{Q}_B)_{:,1:k}^\top \\ &= \mathbf{Q}_A \begin{bmatrix} \mathbf{R}_A \\ \mathbf{O}_{m-k,k} \end{bmatrix} \left(\mathbf{Q}_B \begin{bmatrix} \mathbf{R}_B \\ \mathbf{O}_{n-k,k} \end{bmatrix} \right)^\top = \mathbf{A}\mathbf{B}^\top = \mathbf{X} \,. \end{aligned}$$

Algorithmically, we compute the Q-factor of the polar decomposition as

$$\begin{aligned} \mathbf{Q} &= \mathbf{Q}_A \begin{bmatrix} \widehat{\mathbf{U}} & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{I}_{n-k} \\ \mathbf{O}_{m-n,k} & \mathbf{O}_{m-n,n-k} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{V}}^\top & \mathbf{O}_{n-k,k} \\ \mathbf{O}_{k,n-k} & \mathbf{I}_{n-k} \end{bmatrix} \mathbf{Q}_B^\top \\ &= \left(\mathbf{Q}_B \left(\mathbf{Q}_A \begin{bmatrix} \widehat{\mathbf{U}}\widehat{\mathbf{V}}^\top & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{I}_{n-k} \\ \mathbf{O}_{m-n,k} & \mathbf{O}_{m-n,n-k} \end{bmatrix} \right)^\top \right)^\top \,. \end{aligned}$$

The evaluation follows the order indicated by the brackets. In light of the cost of left-multiplying a vector with $\mathbf{Q}_*$, $* = A, B$, this incurs an asymptotic effort of $O(mn + n^2)$.

The M-factor can be calculated according to

$$\begin{aligned}\mathbf{M} &= \mathbf{Q}_B \begin{bmatrix} \widehat{\mathbf{V}} & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{I}_{n-k} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{\Sigma}} & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{O}_{n-k,n-k} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{V}}^\top & \mathbf{O}_{n-k,k} \\ \mathbf{O}_{k,n-k} & \mathbf{I}_{n-k} \end{bmatrix} \mathbf{Q}_B^\top \\ &= \left(\mathbf{Q}_B \left(\mathbf{Q}_B \begin{bmatrix} \widehat{\mathbf{V}}\widehat{\mathbf{\Sigma}}\widehat{\mathbf{V}}^\top & \mathbf{O}_{k,n-k} \\ \mathbf{O}_{n-k,k} & \mathbf{O}_{n-k,n-k} \end{bmatrix} \right)^\top \right)^\top .\end{aligned}$$

This can be done with an asymptotic computational effort of $O(kn^2)$ since every multiplication with $\mathbf{Q}_B$ amounts to multiplication with k reflection matrices.

Summing up the costs of every step we see that the total asymptotic complexity of the algorithm attains the lower bound $O(mn + n^2)$.

C++ code 3.12.7: Constructor of PolarDecomposition: Polar decomposition of a low-rank matrix.

```
PolarDecomposition::PolarDecomposition(const Eigen::MatrixXd &A,
                                       const Eigen::MatrixXd &B) {
  const long m = A.rows();  // No. of rows of X
  const long n = B.rows();  // No. of columns of X
  const long k = A.cols();  // Maximal rank of X
  // We assume k ≤ n ≤ m
  assert(m >= n);
  assert(k < n);
  assert(B.cols() == k);
  // TODO: (3-12.d) Implement a method to initialize the data members Q_ and M_
  // for X := AB^T = QM, with optimal complexity
  // START
  // Compute QR-decompositions in an encoded way, see \lref{par:ecovsfull}
  Eigen::HouseholderQR<Eigen::MatrixXd> QRA(A);  // cost = O(mk^2)
  Eigen::HouseholderQR<Eigen::MatrixXd> QRB(B);  // cost = O(nk^2)
  const Eigen::MatrixXd RA{
      QRA.matrixQR().block(0, 0, k, k).template triangularView<Eigen::Upper>()};
  const Eigen::MatrixXd RB{
      QRB.matrixQR().block(0, 0, k, k).template triangularView<Eigen::Upper>()};
  // SVD of small k × k-matrix R_A R_B^T
  // cost = O(k^3)
  Eigen::JacobiSVD<Eigen::MatrixXd> svdh(
      RA * RB.transpose(), Eigen::ComputeFullU | Eigen::ComputeFullV);
  // Extract kxk orthogonal factor matrices
  const Eigen::MatrixXd &Vh{svdh.matrixV()};
  const Eigen::MatrixXd &Uh{svdh.matrixU()};
  // Build auxiliary mxn block matrix
  Eigen::MatrixXd W{Eigen::MatrixXd::Zero(m, n)};  // cost = O(mn)
  W.block(0, 0, k, k) = Uh * Vh.transpose();       // cost = O(k^3)
  W.block(k, k, n - k, n - k) = Eigen::MatrixXd::Identity(n - k, n - k);
  // Compute the Q-factor, cost = O(kmn + kn^2)
  Q_ =
      (QRB.householderQ() * ((QRA.householderQ() * W).transpose())).transpose();
  // Small kxk matrix containing singular values on diagonal
  const auto Sigma{svdh.singularValues().asDiagonal()};
  // Form M-factor of polar decomposition
  // Auxiliary matrices
```

```
39      Eigen::MatrixXd S{Eigen::MatrixXd::Zero(n, k)};
40      S.block(0, 0, k, k) = Vh * Sigma * Vh.transpose();   // cost = O(k^3)
41      Eigen::MatrixXd T{Eigen::MatrixXd::Zero(n, n)};      // cost = O(n^2)
42      T.block(0, 0, k, n) =
43          (QRB.householderQ() * S).transpose();    // cost = O(kn^2)
44      M_ = (QRB.householderQ() * T).transpose();   // cost = O(kn^2)
45      // END
46  }
```

Get it on GitLab (polardecomposition.hpp).

HINT 1 for (3-13.a): Remember the connection of QR-decomposition and Gram-Schmidt orthogonalization and [Lecture → Thm. 3.3.3.2]. ⌟

SOLUTION of (3-13.a):

Two possible arguments:

❶ The leftmost m columns of the Q-factor of a QR-decomposition of a regular matrix span the same space as the corresponding columns of that matrix. One can see this from the connection of Gram-Schmidt orthonormalization and the QR-decomposition expressed in [Lecture → Thm. 3.3.3.2]. You can also see it directly from the identity (LA indexing)

$$(\mathbf{T})_{:,1:k} = (\mathbf{Q})_{:,1:k}(\mathbf{R})_{1:k,1:k} \quad \Leftrightarrow \quad (\mathbf{Q})_{:,1:k} = (\mathbf{T})_{:,1:k}(\mathbf{R})^{-1}_{1:k,1:k}\,,$$

which we owe to the invertibility of every upper left $k \times k$ block of the regular upper diagonal matrix $\mathbf{R}$.

Since $\mathbf{T} \in \mathbb{R}^{n,n}$ is tridiagonal, we have

$$\mathrm{Span}\{(\mathbf{Q})_{:,1},\ldots,(\mathbf{Q})_{:,m}\} = \mathrm{Span}\{(\mathbf{T})_{:,1},\ldots,(\mathbf{T})_{:,m}\} \subset \mathrm{Span}\{\mathbf{e}_1,\ldots,\mathbf{e}_{m+1}\}\,,$$

$m \in \{1,\ldots,n\}$, where $\mathbf{e}_\ell \in \mathbb{R}^n$ stands for the ℓ-th coordinate vector. Hence,

$$\begin{array}{c} (\mathbf{Q})_{:,m} \in \mathrm{Span}\{\mathbf{e}_1,\ldots,\mathbf{e}_{m+1}\} \\ \Updownarrow \\ (\mathbf{Q})_{:,m} = \xi_1\mathbf{e}_1 + \xi_2\mathbf{e}_2 + \cdots + \xi_m\mathbf{e}_m + \xi_{m+1}\mathbf{e}_{m+1}\,, \quad \xi_1,\ldots,\xi_{m+1} \in \mathbb{R}\,, \end{array}$$

which immediately implies the assertion of the lemma.

❷ Since $\mathbf{T} \in \mathbb{R}^{n,n}$ is supposed to be regular, so will be the upper triangular matrix $\mathbf{R}$. Thus we can write $\mathbf{Q} = \mathbf{T}\mathbf{R}^{-1}$. Then appeal to [Lecture → Lemma 1.3.1.9].

Lemma [Lecture → Lemma 1.3.1.9]. Group of regular diagonal/triangular matrices

$$\mathbf{A},\mathbf{B} \left\{ \begin{array}{c} \textit{diagonal} \\ \textit{upper triangular} \\ \textit{lower triangular} \end{array} \right. \Rightarrow \mathbf{AB} \text{ and } \mathbf{A}^{-1} \text{ are } \left\{ \begin{array}{c} \textit{diagonal} \\ \textit{upper triangular} \\ \textit{lower triangular} \end{array} \right. .$$

(assumes that $\mathbf{A}$ is regular)

This ensures that $\mathbf{R}^{-1}$ is upper triangular, too:

$$i > j \quad \Rightarrow \quad \left(\mathbf{R}^{-1}\right)_{i,j} = 0\,. \tag{3.13.2}$$

Thus, if $i > j+1$ we get

$$(\mathbf{Q})_{i,j} = \left(\mathbf{T}\mathbf{R}^{-1}\right)_{i,j} = \sum_{\ell=1}^{n} (\mathbf{T})_{i,\ell}\left(\mathbf{R}^{-1}\right)_{\ell,j} \overset{(3.13.2)}{=} \sum_{\ell=1}^{j} \underbrace{(\mathbf{T})_{i,\ell}}_{=0}\left(\mathbf{R}^{-1}\right)_{\ell,j}\,,$$

because $(\mathbf{T})_{i,\ell} = 0$ for $\ell \in \{1,\ldots,j\}$, since $\mathbf{T}$ is tridiagonal.

The fact that $\mathbf{T}$ is regular is crucial. To see this, just consider the case $\mathbf{T} = \mathbf{O}$ and $\mathbf{R} = \mathbf{O}$, where $\mathbf{Q}$ can be any orthogonal matrix.

HINT 1 for (3-13.b): Lemma 3.13.1 may be useful for the proof and can be used even if you have not succeeded in proving it. ⌟

SOLUTION of (3-13.b):

Since $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ has lower bandwidth ≤ 1 by Lemma 3.13.1,

$$\begin{aligned} i,j \in \{1,\ldots,n\} \quad \text{and} \quad i > j \quad &\Rightarrow \quad (\mathbf{R})_{i,j} = 0\,, \\ i,j \in \{1,\ldots,n\} \quad \text{and} \quad i > j+1 \quad &\Rightarrow \quad (\mathbf{Q})_{i,j} = 0\,, \end{aligned}$$

we find that for $i,j \in \{1,\ldots,n\}$, $i > j+1$,

$$(\mathbf{T}')_{i,j} = \sum_{\ell=1}^{n} (\mathbf{R})_{i,\ell}(\mathbf{Q})_{\ell,j} = \sum_{\ell=1}^{i-1} \underbrace{(\mathbf{R})_{i,\ell}}_{=0}(\mathbf{Q})_{\ell,j} + \sum_{\ell=i}^{n} (\mathbf{R})_{i,\ell} \underbrace{(\mathbf{Q})_{\ell,j}}_{=0} = 0\,.$$

In words, $\mathbf{T}'$ has lower bandwidth ≤ 1.

Since $\mathbf{Q}$ is *orthogonal*, $\mathbf{Q}^{-1} = \mathbf{Q}^{\top}$, we deduce the identity

$$\mathbf{T}' = \mathbf{R}\mathbf{Q} = \mathbf{Q}^{\top}\mathbf{Q}\mathbf{R}\mathbf{Q} = \mathbf{Q}^{\top}\mathbf{T}\mathbf{Q}\,,$$

which implies that $\mathbf{T}'$ is symmetric:

$$(\mathbf{T}')^{\top} = \left(\mathbf{Q}^{\top}\mathbf{T}\mathbf{Q}\right)^{\top} = \mathbf{Q}^{\top}\mathbf{T}\mathbf{Q} = \mathbf{T}'\,.$$

Finally, a symmetric matrix with lower bandwidth ≤ 1 must be tridiagonal

$$i,j \in \{1,\ldots,n\}: \begin{array}{c} i > j+1 \quad \Rightarrow \quad (\mathbf{M})_{i,j} = 0\,, \\ (\mathbf{M})_{i,j} = (\mathbf{M})_{j,i} \end{array} \quad \blacktriangleright \quad (\mathbf{M})_{i,j} = 0 \quad \text{for} \quad |i-j| > 1\,.$$

HINT 1 for (3-13.c): In the file `qriteration.hpp` you have at your disposal a function

```
Eigen::Vector2d givens(Eigen::Vector2d a);
```

that, when supplied with any vector $\mathbf{a} \in \mathbb{R}^2$ returns a vector $\begin{bmatrix} c \\ s \end{bmatrix} \in \mathbb{R}^2$ such that

$$s^2 + c^2 = 1 \quad \text{and} \quad \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \mathbf{a} = \begin{bmatrix} \pm \|\mathbf{a}\| \\ 0 \end{bmatrix} .$$

HINT 2 for (3-13.c): Lemma 3.13.3 is crucial for the design of the algorithm! ⌟

SOLUTION of (3-13.c):

We write $\mathbf{G}_{i,j}^{\ell}(\mathbf{M}) \in \mathbb{R}^{n,n}$ for the orthogonal matrix describing a **Givens rotation** [Lecture → § 3.3.3.15] that through left-multiplication acts on rows i and j of $\mathbf{M} \in \mathbb{R}^{n,n}$, $i,j \in \{1,\ldots,n\}$, $i \neq j$, such that a particular entry of $\mathbf{M}$ is annihilated:

$$\left(\mathbf{G}_{i,j}^{\ell}(\mathbf{M})\mathbf{M}\right)_{j,\ell} = 0\,, \quad \ell = 1,\ldots,n\,.$$

From [Lecture → Rem. 3.3.3.26] we know that a tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ can be transformed into an upper triangular matrix $\mathbf{R}$ by means of $n-1$ successive Givens rotations:

$$\mathbf{T}^{(1)} := \mathbf{T}\,, \quad \mathbf{T}^{(k)} := \mathbf{G}_{k-1,k}^{k-1}(\mathbf{T}^{(k-1)})\mathbf{T}^{(k-1)}\,, \quad k = 2,\ldots,n\,, \quad \mathbf{R} = \mathbf{T}^{(n)}\,: \tag{3.13.4}$$

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{1,2}^{1}} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{2,3}^{2}\cdots\mathbf{G}_{n-1,n}^{n-1}} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

Abbreviating the Givens rotations, we have

$$\mathbf{G}_{n-1,n}^{n-1} \cdot \cdots \cdot \mathbf{G}_{2,3}^{2}\mathbf{G}_{1,2}^{1}\mathbf{T} = \mathbf{R}\,,$$

which supplies a representation of the Q-factor $\mathbf{Q}$ of the QR-decomposition of $\mathbf{T}$ as a product of Givens rotations:

$$\mathbf{Q} = \left(\mathbf{G}_{1,2}^{1}\right)^{\top}\left(\mathbf{G}_{2,3}^{2}\right)^{\top} \cdots \cdot \left(\mathbf{G}_{n-1,1}^{n-1}\right)^{\top}\,.$$

We conclude

$$\mathbf{T}' = \mathbf{G}_{n-1,1}^{n-1} \cdot \cdots \cdot \mathbf{G}_{2,3}^{2}\mathbf{G}_{1,2}^{1}\,\mathbf{T}\left(\mathbf{G}_{1,2}^{1}\right)^{\top}\left(\mathbf{G}_{2,3}^{2}\right)^{\top} \cdots \cdot \left(\mathbf{G}_{n-1,1}^{n-1}\right)^{\top}\,.$$

This suggests a two-stage algorithm for computing $\mathbf{T}'$:

(I) Compute $\mathbf{R}$ according to (3.13.4) by applying $n-1$ Givens rotations to intermediate matrices.

(II) Successively right-multiply $\mathbf{R}$ with the transposed Givens rotations from step (I), in the same order.

In order to achieve an optimal computational complexity $O(n)$ for $n \to \infty$ we must not operate on densely populated square matrices, but we have to store non-zero entries only, guided by the observation that all intermediate matrices in (3.13.4) possess at most four bands with non-zero entries, see also [Lecture → Thm. 3.3.3.27]:

$$\left(\mathbf{T}^{(k)}\right)_{i,j} = 0 \quad \text{, if} \quad i-j > 1 \quad \text{or} \quad j-i > 2\,. \tag{3.13.5}$$

Thus we can store the bands of $\mathbf{T}^{(k)}$ in the rows of a $4 \times n$ auxiliary matrix $\mathbf{U}$:

$$(\mathbf{U})_{i-j+3,j} := \left(\mathbf{T}^{(k)}\right)_{i,j} \quad \text{for} \quad i-j \in \{-2,-1,0,1\}\,, \quad j = 1,\ldots,n\,:$$

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \dots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \dots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \dots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \dots & t_{n,n-1} & 0 \end{bmatrix}, \quad t_{i,j} := \left(\mathbf{T}^{(k)}\right)_{i,j}.$$

The third row contains the diagonal of the matrix, the bottom row the first sub-diagonal. In the first step of (3.13.4) we apply $\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)})$ to $\mathbf{T}^{(1)}$. This affects six entries of the matrix

$$\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)}) \cdot \begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix},$$

which correspond to the following entries of $\mathbf{U}$, highlighted with color:

$$\begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \dots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \dots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \dots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \dots & t_{n,n-1} & 0 \end{bmatrix}.$$

We write `givens` for a function computing the relevant parameters $c := \cos\varphi$ and $s := \sin\varphi$ of the Givens rotation. Then

$$\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)}) = \begin{bmatrix} c & -s & 0 & \dots & 0 \\ s & c & & & \vdots \\ 0 & & 1 & & \\ \vdots & & & \ddots & \vdots \\ 0 & \dots & & \dots & 1 \end{bmatrix}, \quad \begin{bmatrix} c \\ s \end{bmatrix} = \texttt{givens}\left(\begin{bmatrix} t_{11} \\ t_{21} \end{bmatrix}\right),$$

is the matrix effecting the Givens transformation in the first step, which amounts to the following changes in $\mathbf{U}$:

$$\begin{bmatrix} (\mathbf{U})_{3,1} \\ (\mathbf{U})_{4,1} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{3,1} \\ (\mathbf{U})_{4,1} \end{bmatrix}, \quad \begin{bmatrix} (\mathbf{U})_{2,2} \\ (\mathbf{U})_{3,2} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{2,2} \\ (\mathbf{U})_{3,2} \end{bmatrix}, \quad \begin{bmatrix} (\mathbf{U})_{1,3} \\ (\mathbf{U})_{2,3} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{1,3} \\ (\mathbf{U})_{2,3} \end{bmatrix}.$$

In the second step of (3.13.4) we multiply $\mathbf{T}^{(2)}$ from the left with $\mathbf{G}^2_{2,3}(\mathbf{T}^{(2)})$.

$$\mathbf{G}^1_{2,3}(\mathbf{T}^{(2)}) \cdot \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix}.$$

Again, we have to operate on six entries, and for $\mathbf{U}$ the transformation affects the following entries

$$\begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \dots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \dots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \dots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \dots & t_{n,n-1} & 0 \end{bmatrix},$$

and reads

$$\begin{bmatrix}(\mathbf{U})_{3,2}\\(\mathbf{U})_{4,2}\end{bmatrix} \leftarrow \begin{bmatrix}c & -s\\ s & c\end{bmatrix}\begin{bmatrix}(\mathbf{U})_{3,2}\\(\mathbf{U})_{4,2}\end{bmatrix}, \quad \begin{bmatrix}(\mathbf{U})_{2,3}\\(\mathbf{U})_{3,3}\end{bmatrix} \leftarrow \begin{bmatrix}c & -s\\ s & c\end{bmatrix}\begin{bmatrix}(\mathbf{U})_{2,3}\\(\mathbf{U})_{3,3}\end{bmatrix}, \quad \begin{bmatrix}(\mathbf{U})_{1,4}\\(\mathbf{U})_{2,4}\end{bmatrix} \leftarrow \begin{bmatrix}c & -s\\ s & c\end{bmatrix}\begin{bmatrix}(\mathbf{U})_{1,4}\\(\mathbf{U})_{2,4}\end{bmatrix}.$$

Now a general pattern should have emerged: In the k-th step, $\mathbf{T}^{(k)} \to \mathbf{T}^{(k+1)}$, $k = 1, \ldots, n-1$, we have to perform the following transformation of $\mathbf{U}$: For $k = 1, 2, \ldots, n-1$

$$\begin{bmatrix}c\\s\end{bmatrix} := \texttt{givens}\left(\begin{bmatrix}(\mathbf{U})_{3,k}\\(\mathbf{U})_{4,k}\end{bmatrix}\right),$$
$$\begin{bmatrix}(\mathbf{U})_{3,k}\\(\mathbf{U})_{4,k}\end{bmatrix} \leftarrow \begin{bmatrix}c & -s\\ s & c\end{bmatrix}\begin{bmatrix}(\mathbf{U})_{3,k}\\(\mathbf{U})_{4,k}\end{bmatrix}, \quad \begin{bmatrix}(\mathbf{U})_{2,k+1}\\(\mathbf{U})_{3,k+1}\end{bmatrix} \leftarrow \begin{bmatrix}c & -s\\ s & c\end{bmatrix}\begin{bmatrix}(\mathbf{U})_{2,k+1}\\(\mathbf{U})_{3,k+1}\end{bmatrix},$$
$$\begin{bmatrix}(\mathbf{U})_{1,k+2}\\(\mathbf{U})_{2,k+2}\end{bmatrix} \leftarrow \begin{bmatrix}c & -s\\ s & c\end{bmatrix}\begin{bmatrix}(\mathbf{U})_{1,k+2}\\(\mathbf{U})_{2,k+2}\end{bmatrix} \quad \text{, if} \quad k < n-1\,.$$

Eventually, $\mathbf{U}$ will encode the $\mathbf{R}$-factor of the QR-decomposition of $\mathbf{T}$:

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & r_{1,3} & r_{2,4} & r_{3,5} & \ldots & r_{n-3,n-1} & r_{n-2,n}\\ 0 & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,5} & \ldots & r_{n-2,n-2} & r_{n-1,n}\\ r_{1,1} & r_{2,2} & r_{3,3} & r_{4,4} & r_{5,5} & \ldots & r_{n-1,n-1} & r_{n,n}\\ 0 & 0 & 0 & 0 & 0 & \ldots & 0 & 0\end{bmatrix}, \quad r_{i,j} := (\mathbf{R})_{i,j}\,.$$

In stage (II) of the algorithm we have to compute

$$\mathbf{T}' := \mathbf{R}\left(\mathbf{G}_{1,2}^{1}\right)^{\top}\left(\mathbf{G}_{2,3}^{2}\right)^{\top}\cdot\cdots\cdot\left(\mathbf{G}_{n-1,1}^{n-1}\right)^{\top}.$$

We find that right-multiplication of a matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ with

$$\left(\mathbf{G}_{i,j}^{\ell}\right)^{\top} = \begin{bmatrix}
1 & 0 & \ldots & & & & & & \ldots & 0\\
0 & \ddots & \ddots & & & & & & & \vdots\\
\vdots & \ddots & 1 & 0 & & & & & & \\
 & & 0 & c & 0 & \ldots & 0 & s & & \\
 & & & 0 & 1 & & & 0 & & \\
 & & & \vdots & & \ddots & & \vdots & & \\
 & & & 0 & & & 1 & 0 & & \\
 & & & -s & 0 & \ldots & 0 & c & 0 & \\
 & & & & & & & 0 & 1 & \ddots & \vdots\\
\vdots & & & & & & & & \ddots & \ddots & 0\\
0 & \ldots & & & & & & & \ldots & 0 & 1
\end{bmatrix}, \quad i,j \in \{1,\ldots,n\},\ i \neq j\,,$$

boils down to combining the i-th and j-th column of $\mathbf{M}$:

$$\begin{bmatrix}(\mathbf{M})_{:,i} & (\mathbf{M})_{:,j}\end{bmatrix} \leftarrow \begin{bmatrix}c(\mathbf{M})_{:,i} - s(\mathbf{M})_{:,j} & s(\mathbf{M})_{:,i} + c(\mathbf{M})_{:,j}\end{bmatrix}.$$

So the first step of stage (II), right multiplication with the transpose of $\mathbf{G}_{1,2}^{1}(\mathbf{T}^{(1)})$, involves the two leftmost

columns of $\mathbf{R}$:

$$\begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}_{1,2}^{1}\right)^{\top} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

The following entries of the matrix $\mathbf{U}$ are affected:

$$\begin{bmatrix} 0 & 0 & r_{1,3} & r_{2,4} & r_{3,5} & \dots & r_{n-3,n-1} & r_{n-2,n} \\ 0 & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,5} & \dots & r_{n-2,n-2} & r_{n-1,n} \\ r_{1,1} & r_{2,2} & r_{3,3} & r_{4,4} & r_{5,5} & \dots & r_{n-1,n-1} & r_{n,n} \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

The next step combines the second and third column of the matrix:

$$\begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}_{2,3}^{2}\right)^{\top} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & * & * & * & * & 0 \\ 0 & 0 & 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix},$$

which impacts the following entries of $\mathbf{U}$:

$$\begin{bmatrix} 0 & 0 & r_{1,3} & r_{2,4} & r_{3,5} & \dots & r_{n-3,n-1} & r_{n-2,n} \\ 0 & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,5} & \dots & r_{n-2,n-2} & r_{n-1,n} \\ r_{1,1} & r_{2,2} & r_{3,3} & r_{4,4} & r_{5,5} & \dots & r_{n-1,n-1} & r_{n,n} \\ r_{1,2} & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}.$$

The following step targets the third and fourth column:

$$\begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}_{2,3}^{2}\right)^{\top} = \begin{bmatrix} * & * & * & \star & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

We make that bewildering observation that the transformation introduces an entry outside the four bands stored in $\mathbf{U}$! This is where Lemma 3.13.3 comes to the rescue. It tells us that the entry $\star$ must be zero (in exact arithmetic) and, thus, safely can be ignored. So in the next step, it is again only six entries that

we have to deal with:

$$\begin{bmatrix} * & * & * & \mathbf{0} & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}_{2,3}^2\right)^\top = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & \star & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & * & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

A pattern has emerged and suggests the following operations on $\mathbf{U}$: For $k = 1, 2, \ldots, n-1$

$$\begin{aligned} \begin{bmatrix} (\mathbf{U})_{4,k-1} \\ (\mathbf{U})_{3,k} \end{bmatrix} &\leftarrow \begin{bmatrix} c^{(k)} & -s^{(k)} \\ s^{(k)} & c^{(k)} \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{4,k-1} \\ (\mathbf{U})_{3,k} \end{bmatrix}, \\ \begin{bmatrix} (\mathbf{U})_{3,k-1} \\ (\mathbf{U})_{2,k} \end{bmatrix} &\leftarrow \begin{bmatrix} c^{(k)} & -s^{(k)} \\ s^{(k)} & c^{(k)} \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{3,k-1} \\ (\mathbf{U})_{2,k} \end{bmatrix}, \\ \begin{bmatrix} (\mathbf{U})_{2,k-1} \\ (\mathbf{U})_{1,k} \end{bmatrix} &\leftarrow \begin{bmatrix} c^{(k)} & -s^{(k)} \\ s^{(k)} & c^{(k)} \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{2,k-1} \\ (\mathbf{U})_{1,k} \end{bmatrix}, \end{aligned} \tag{3.13.6}$$

where $c^{(k)}$ and $s^{(k)}$ are the parameters of the Givens rotation used in the k-th step of (3.13.4). In the end, $(\mathbf{U})_{3,:}$ contains the diagonal entries of $\mathbf{T}'$ and $(\mathbf{U})_{4,1:n-1}$ provides the super- and sub-diagonal.

C++ code 3.13.7: Implementation of `qrStep()`

```
2  std::pair<Eigen::VectorXd, Eigen::VectorXd> qrStep(const Eigen::VectorXd &dT,
3                                                     const Eigen::VectorXd &uT) {
4    const long n = dT.size();
5    assert(uT.size() == n - 1);
6    // Defining vectors for T' (Written as T_p)
7    Eigen::VectorXd dT_p = Eigen::VectorXd::Zero(n);
8    Eigen::VectorXd uT_p = Eigen::VectorXd::Zero(n - 1);
9
10   // TODO: (3-13.c) Compute the defining vectors d(T') and u(T') of T':=
         RQ with
11   // O(n) asymptotic complexity, given the defining vectors d(T) and
         U(T) of T =
12   // QR
13   // START
14   // Auxiliary 4xn matrix U ∈ R^{4,n}
15   Eigen::MatrixXd U{Eigen::MatrixXd::Zero(4, n)};
16   U.block(1, 1, 1, n - 1) = uT.transpose();
17   U.block(2, 0, 1, n) = dT.transpose();
18   U.block(3, 0, 1, n - 1) = uT.transpose();
19   // Stage I: QR decomposition by n-1 Givens rotations
20   std::vector<Eigen::Matrix2d> givensrots;  // Store rotations!
21   for (long i = 0; i < n - 1; ++i) {
22     const Eigen::Vector2d g = givens(Eigen::Vector2d(U(2, i), U(3, i)));
23     const Eigen::Matrix2d G{
24         (Eigen::Matrix2d(2, 2) << g[0], -g[1], g[1], g[0]).finished()};
25     U.block(2, i, 2, 1).applyOnTheLeft(G);
26     U.block(1, i + 1, 2, 1).applyOnTheLeft(G);
27     if (i < n - 2) {
28       U.block(0, i + 2, 2, 1).applyOnTheLeft(G);
29     }
```

```
30      givensrots.push_back(G);
31    }
32    // At this point U encodes the R-factor of the QR-decomposition of
33    // T.  In particular, the bottom row of U should contain
34    // only entries that are "numerically zero".
35
36    // Stage II: Apply transposed rotations from right to R according to
37    // formulas (3.13.6)
38    for (long i = 1; i < n; ++i) {
39      const Eigen::Matrix2d &G{givensrots[i - 1]};  // An alias
40      Eigen::Vector2d tmp;
41      tmp = G * Eigen::Vector2d(U(3, i - 1), U(2, i));
42      U(3, i - 1) = tmp[0];
43      U(2, i) = tmp[1];
44      tmp = G * Eigen::Vector2d(U(2, i - 1), U(1, i));
45      U(2, i - 1) = tmp[0];
46      U(1, i) = tmp[1];
47      tmp = G * Eigen::Vector2d(U(1, i - 1), U(0, i));
48      U(1, i - 1) = tmp[0];
49      U(0, i) = tmp[1];
50    }
51    // Extract diagonal and sub-/super-diagonal of T'.'
52    dT_p = U.block(2, 0, 1, n).transpose();
53    uT_p = U.block(1, 1, 1, n - 1).transpose();
54    // END
55
56    return {dT_p, uT_p};
57  }
```

Get it on GitLab (qriteration.hpp).

The EIGEN method `applyOnTheLeft()` performs in-situ left-multiplication with a matrix. Of course, the code uses C++ indexing!

Obviously, for the implementation of Code 3.13.7 a call to `qrStep()` for n-vectors can be processed with $O(n)$ elementary operations for $n \to \infty$.

HINT 1 for (3-14.a): Both cases $m \geq n$ and $m \leq n$ must be taken into account. ⌟

SOLUTION of (3-14.a):

$$\mathbf{U} \in \mathbb{R}^{m,\min\{m,n\}} \quad , \quad \mathbf{\Sigma} \in \mathbb{R}^{\min\{m,n\},\min\{m,n\}} \quad , \quad \mathbf{V} \in \mathbb{R}^{n,\min\{m,n\}} \, .$$

SOLUTION of (3-14.b):

1. FALSE, because $\mathbf{U}$ need not be a square matrix. Only for the full SVD the U-factor would always be orthogonal.
2. FALSE, because if $m \geq n$, $\mathrm{rank}(\mathbf{A}) < n$, then the columns of $\mathbf{U}$ will span a space of dimension n, while $\mathcal{R}(\mathbf{A})$ has a smaller dimension.
3. FALSE, because only the columns of $\mathbf{V}$ are orthogonal, if $m < n$, $\mathbf{V}\mathbf{V}^{\top} \neq \mathbf{I}$.
4. TRUE, because $\mathbf{V}$ will always have orthonormal columns.
5. TRUE, because $\mathbf{\Sigma}$ is a *diagonal matrix*.
6. FALSE, because this matrix product may no even be defined. Even if it is, the rows of $\mathbf{U}$ and $\mathbf{V}$ are not related in any respect.

Also refer to [Lecture → § 3.4.1.4].

SOLUTION of (3-14.c):

According to [Lecture → § 3.4.2.2] we have

$$\text{cost}(\text{economical SVD of } \mathbf{A} \in \mathbb{R}^{m,n}) = O(\min\{m,n\}^2 \max\{m,n\}) \quad \text{for} \quad m,n \to \infty \,.$$

SOLUTION of (3-15.a):

For the range space of the matrix $\mathbf{X}$ we find

$$\mathcal{R}(\mathbf{X}) = \mathcal{R}(\mathbf{X}_1) + \mathcal{R}(\mathbf{X}_2) \; .$$

$$\Rightarrow \quad \dim \mathcal{R}(\mathbf{X}) \leq \dim \mathcal{R}(\mathbf{X}_1) + \dim \mathcal{R}(\mathbf{X}_2) \; ,$$

$$\Rightarrow \quad \operatorname{rank}(\mathbf{X}) \leq 2k \; .$$

In addition, obviously $\dim \mathcal{R}(\mathbf{X}) \leq m$, which means

$$\operatorname{rank}(\mathbf{X}) \leq \min\{2k, m\} \; .$$

Simple examples show that this bound is sharp: In the case $2k \leq m$ choose $\mathbf{A}_1$ and $\mathbf{A}_2$ such that their combined columns yield a linearly independent set of vectors $\in \mathbb{R}^m$.

HINT 1 for (3-15.b): Employ the economical QR-decomposition of the low-rank factors $\mathbf{B}_1$, $\mathbf{B}_2$. ⌟

SOLUTION of (3-15.b):

Start with the **economical QR decompositions** [Lecture → Thm. 3.3.3.4] of the matrices $\mathbf{B}_1$ and $\mathbf{B}_2$:

$$\mathbf{B}_1 = \mathbf{Q}_1\mathbf{R}_1\,,\quad \mathbf{Q}_1 \in \mathbb{R}^{n,k}\,,\quad \mathbf{R}_1 \in \mathbb{R}^{k,k}\,,\quad \mathbf{Q}_1^\top\mathbf{Q}_1 = \mathbf{I}_k\,,$$
$$\mathbf{B}_2 = \mathbf{Q}_2\mathbf{R}_2\,,\quad \mathbf{Q}_2 \in \mathbb{R}^{n,k}\,,\quad \mathbf{R}_2 \in \mathbb{R}^{k,k}\,,\quad \mathbf{Q}_2^\top\mathbf{Q}_2 = \mathbf{I}_k\,.$$

They permit us to obtain a factorization of $\mathbf{X} \in \mathbb{R}^{m,2n}$:

$$\mathbf{X} = [\mathbf{X}_1\,\mathbf{X}_2] = \left[\mathbf{A}_1\mathbf{R}_1^\top\mathbf{Q}_1^\top\ \mathbf{A}_2\mathbf{R}_2^\top\mathbf{Q}_2^\top\right] = [\underbrace{\mathbf{A}_1\mathbf{R}_1^\top}_{\in\mathbb{R}^{m,k}}\ \underbrace{\mathbf{A}_2\mathbf{R}_2^\top}_{\in\mathbb{R}^{m,k}}]\begin{bmatrix}\mathbf{Q}_1^\top & \mathbf{O}\\ \mathbf{O} & \mathbf{Q}_2^\top\end{bmatrix} \tag{3.15.4}$$

Note that the matrix

$$\mathbf{Q} := \begin{bmatrix}\mathbf{Q}_1 & \mathbf{O}\\ \mathbf{O} & \mathbf{Q}_2\end{bmatrix} \in \mathbb{R}^{n,2k}$$

has orthonormal columns: $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}_{2k}$, which is clear from the properties of $\mathbf{Q}_1, \mathbf{Q}_2$. Define the matrix

$$\mathbf{Z} := \left[\mathbf{A}_1\mathbf{R}_1^\top\ \mathbf{A}_2\mathbf{R}_2^\top\right] \in \mathbb{R}^{m,2k} \tag{3.15.5}$$

and consider its **economical SVD** [Lecture → § 3.4.1.4]

$$\mathbf{Z} := \mathbf{U}_Z\mathbf{\Sigma}_Z\mathbf{V}_Z^\top\,,\quad \mathbf{U}_Z \in \mathbb{R}^{m,2k},\ \mathbf{V}_Z \in \mathbb{R}^{2k,2k},\ \mathbf{\Sigma}_Z \in \mathbb{R}^{2k,2k} \text{ diagonal.}$$

From (3.15.4) we have

$$\mathbf{X} = \mathbf{Z}\mathbf{Q}^\top = \mathbf{U}_Z\mathbf{\Sigma}_Z\mathbf{V}_Z^\top\mathbf{Q}^\top\,,$$

from which we read off the factors of the economical SVD of $\mathbf{X}$, $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$:

$$\mathbf{U} = \mathbf{U}_Z\quad,\quad \mathbf{\Sigma} = \mathbf{\Sigma}_Z\quad,\quad \mathbf{V} = \mathbf{Q}\mathbf{V}_Z\,,\quad \mathbf{V} = \begin{bmatrix}\mathbf{Q}_1(\mathbf{V}_Z)_{1:k,:}\\ \mathbf{Q}_2(\mathbf{V}_Z)_{k+1:2k,:}\end{bmatrix}\,.$$

We also verify

$$\mathbf{V}^\top\mathbf{V} = \mathbf{V}_Z^\top\mathbf{Q}^\top\mathbf{Q}\mathbf{V}_Z = \mathbf{V}_Z^\top\mathbf{V}_Z = \mathbf{I}_{2k}\,.$$

C++ code 3.15.6: EIGEN-based implementation of `eco_svd_concat()`

```
2  std::tuple<Eigen::MatrixXd, Eigen::VectorXd, Eigen::MatrixXd> eco_svd_concat(
3      const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
4      const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2) {
5    // Matrix dimensions and consistency checks
6    const unsigned int m = A1.rows();
7    assert(A2.rows() == m);
8    const unsigned int n = B1.rows();
9    assert(B2.rows() == n);
10   const unsigned int k = A1.cols();
11   assert((k == A2.cols()) && (k == B1.cols()) && (k == B2.cols()));
12   // Matrices/vector to be returned by the function
```

```
13    Eigen::MatrixXd U = Eigen::MatrixXd::Zero(m, 2 * k);
14    Eigen::VectorXd s = Eigen::VectorXd::Zero(2 * k);
15    Eigen::MatrixXd V = Eigen::MatrixXd::Zero(2 * n, 2 * k);
16
17    // TODO: (3-15.b) Compute the economical SVD with optimal complexity
          and store
18    // the results in U, s and V.
19    // START
20
21    // Computation of economical QR-decompositions of B-factors
22    // incurs a computational effort of
23    // O(nk^2)
24    Eigen::HouseholderQR<Eigen::MatrixXd> qrB1(B1);  //
25    const Eigen::MatrixXd Q1{qrB1.householderQ() *
26                             Eigen::MatrixXd::Identity(n, k)};  //
27    Eigen::HouseholderQR<Eigen::MatrixXd> qrB2(B2);             //
28    const Eigen::MatrixXd Q2{qrB2.householderQ() *
29                             Eigen::MatrixXd::Identity(n, k)};  //
30    // Computation of the matrix Z as defined in (3.15.5)
31    // Uses EIGEN's matrix concatenation operator
32
33    Eigen::MatrixXd Z(m, 2 * k);
34    Z << (A1 * (qrB1.matrixQR()
35                   .block(0, 0, k, k)
36                   .template triangularView<Eigen::Upper>())
37                  .transpose()),
38         (A2 * (qrB2.matrixQR()
39                   .block(0, 0, k, k)
40                   .template triangularView<Eigen::Upper>())
41                  .transpose());
42
43    // Economical SVD of Z
44    Eigen::JacobiSVD<Eigen::MatrixXd> svdZ(
45        Z, Eigen::ComputeThinU | Eigen::ComputeThinV);  //
46    // Extract desired factors of final economical SVD of X
47    U = svdZ.matrixU();         // left factor of eco-SVD of X
48    s = svdZ.singularValues();  // singular values in a vector
49    // Build right factor of final economical SVD
50    Eigen::MatrixXd VZ{svdZ.matrixV()};
51    V.block(0, 0, n, 2 * k) = Q1 * VZ.block(0, 0, k, 2 * k);  //
52    V.block(n, 0, n, 2 * k) = Q2 * VZ.block(k, 0, k, 2 * k);  //
53    // END
54
55    return {U, s, V};
56  }
```

Get it on GitLab (concatmatlrc.hpp).

The construction of the $n \times k$-matrices `Q1` and `Q2` in Line 26 and Line 29 is wasteful, but inevitable due to a lack of access methods to the Q-factor of a QR-decomposition in EIGEN.

This is the asymptotic computational cost of the individual steps for $k, m, n \to \infty$, $2k < \min\{m, n\}$:

- Economical QR-decompositions of $\mathbf{B}_1$ and $\mathbf{B}_2$ in Line 24 and Line 27: $O(k^2 n)$
- Construction of the $n \times k$-matrices `Q1` and `Q2` in Line 26 and Line 29: $O(k^2 n)$, because the Q-factors of the QR-decompositions are internally stored in the form of k Householder transformations [Lecture → Rem. 3.3.3.21].

- Assembly of matrix $\mathbf{Z} \in \mathbb{R}^{n,2k}$ through matrix products: $O(mk^2)$
- Economical SVD of $\mathbf{Z}$ in Line 45: $O(nk^2)$, see [Lecture → § 3.4.2.2]
- Matrix products in Line 51 and Line 52: $O(nk^2)$

Hence the total computational effort scales like $O((n+m)k^2)$ as $m,n,k \to \infty$. This is optimal in terms of m,n, because this amounts to the number of floating point data in both the input and output.

Remark. We do not gain any benefit from a QR-decomposition of the low-rank factors $\mathbf{A}_1$, $\mathbf{A}_2$. Starting with

$$\begin{aligned} \mathbf{A}_1 &= \mathbf{Q}_1\mathbf{R}_1 \quad \text{with} \quad \mathbf{Q}_1 \in \mathbb{R}^{m,k},\ \mathbf{R}_1 \in \mathbb{R}^{k,k}\ , \\ \mathbf{A}_2 &= \mathbf{Q}_2\mathbf{R}_2 \quad \text{with} \quad \mathbf{Q}_2 \in \mathbb{R}^{m,k},\ \mathbf{R}_2 \in \mathbb{R}^{k,k}\ . \end{aligned}$$

we can factor $\mathbf{X}$ according to

$$\mathbf{X} = \left[\mathbf{Q}_1\mathbf{R}_1\mathbf{B}_1^\top\ \mathbf{Q}_2\mathbf{R}_2\mathbf{B}_2^\top\right] = [\mathbf{Q}_1\ \mathbf{Q}_2]\begin{bmatrix}\mathbf{R}_1\mathbf{B}_1^\top & \mathbf{O} \\ \mathbf{O} & \mathbf{R}_2\mathbf{B}_2^\top\end{bmatrix}.$$

Unfortunately, the matrix $[\mathbf{Q}_1\ \mathbf{Q}_2]$ does not have orthonormal columns and, therefore, we cannot immediately build the first factor of an SVD from it.

HINT 1 for (3-15.c): The method `norm()` for EIGEN's matrix types computes the Frobenius norm of a matrix. It can be used to check whether two matrices are close. ⌟

SOLUTION of (3-15.c):

Let us write $\mathbf{U}$, $\mathbf{s}$, and $\mathbf{V}$ for the matrices and the vector returned by `eco_svd_concat()`. The test has to involve the following verifications:

- Make sure that the sizes of $\mathbf{U}$, $\mathbf{s}$, and $\mathbf{V}$ match those of $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{B}_1$, $\mathbf{B}_2$.
- Check, whether the computed singular values are non-negative and ordered.
- Ensure that $\mathbf{U}$ has orthogonal columns by testing whether

$$\left\| \mathbf{U}^\top \mathbf{U} - \mathbf{I}_{2k} \right\|_F \leq \tau \,. \tag{3.15.7}$$

- Check the orthonormality of the columns of $\mathbf{V}$ by the test

$$\left\| \mathbf{V}^\top \mathbf{V} - \mathbf{I}_{2k} \right\|_F \leq \tau \,. \tag{3.15.8}$$

- Examine, if the matrix product yields $\mathbf{X}$:

$$\left\| \mathbf{X} - \mathbf{U}\,\mathrm{diag}(\mathbf{s})\mathbf{V}^\top \right\|_F \leq \tau \|\mathbf{s}\|_2 \quad ? \tag{3.15.9}$$

In (3.15.7) and (3.15.7) need not use a data-dependent bound, because the “exact value” is the identity matrix, which has a norm depending only on k.

Note that, in (3.15.9) we *have to test for relative smallness*, because we want to check equality to zero for quantities that are the results of numerical computations. See [Lecture → Rem. 1.5.3.15] for a discussion. The tolerance τ should be a positive number of the same order of magnitude as the machine precision $\mathrm{EPS} \approx 10^{-16}$. A sophisticated solution may choose $\tau = 100nm\mathrm{EPS}$ in order to compensate the larger accumulation of round-off errors in the case of larger matrix sizes, but simply setting $\tau := 10^{-d}$, $d \in \{6, \ldots, 12\}$ is also acceptable. Setting $\tau := \mathrm{EPS}$ is too tight a requirement, because it will make the test fail even in case of a mild accumulation of round-off errors.

Above, $\|\cdot\|_F$ stands for the Frobenius norm of a matrix, see [Lecture → Def. 3.4.4.17]. Yet, it may be replaced with any other matrix norm presented in [Lecture → Rem. 1.5.5.5].

We point out that $\|\mathbf{X}\|_F = \|\mathbf{s}\|_2$ according to [Lecture → Eq. (3.4.4.18)], which explains the right-hand side in (3.15.9). Replacing $\|\mathbf{s}\|_2$ with the largest singular value $(\mathbf{s})_1 = \|\mathbf{X}\|_2$ is an equivalent option.

C++ code 3.15.10: EIGEN-based implementation of `test_eco_svd_concat()`

```
2  bool test_eco_svd_concat(const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
3                           const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2) {
4    auto [U, s, V] = eco_svd_concat(A1, B1, A2, B2);
5    // Obtain matrix dimensions and perform consistency checks
6    const unsigned int m = A1.rows();
7    const unsigned int n = B1.rows();
8    const unsigned int k = A1.cols();
9    bool ok = ((U.rows() == m) && (U.cols() == 2 * k) && (V.rows() == 2 * n) &&
10             (V.cols() == 2 * k) && (s.size() == 2 * k));
11   // Define a suitable tolerance
12   const double tau = 100 * m * n * std::numeric_limits<double>::epsilon();
13   // Ensure that the singular values are non-negative and ordered
```

```
14    ok = ok && (s[0] >= 0.0);
15    for (unsigned int i = 1; i < 2 * k; ++i) {
16      ok = ok && (s[i - 1] >= s[i]) && (s[i] >= 0.0);
17    }
18
19    // TODO: (3-15.c) Check orthonormality and correctness.
20    // START
21    // Check whether the columns of U are orthonormal
22    ok = ok &&
23         ((U.transpose() * U - Eigen::MatrixXd::Identity(2 * k, 2 * k)).norm() <=
24          tau);
25    // Check whether the columns of V are orthonormal
26    ok = ok &&
27         ((V.transpose() * V - Eigen::MatrixXd::Identity(2 * k, 2 * k)).norm() <=
28          tau);
29    // Check correctness of matrix factorization
30    const Eigen::MatrixXd P{U * s.asDiagonal() * V.transpose()};
31    const double ref = tau * s.norm();
32    ok = ok && ((A1 * B1.transpose() - P.block(0, 0, m, n)).norm() <= ref);
33    ok = ok && ((A2 * B2.transpose() - P.block(0, n, m, n)).norm() <= ref);
34    // END
35
36    return ok;
37  }
```

Get it on GitLab (concatmatlrc.hpp).

SOLUTION of (3-15.d):

The function `eco_svd_concat()` supplies an **economical SVD** of $\mathbf{X}$:

$$\mathbf{X} = \mathbf{U}\,\mathrm{diag}(\sigma_1,\sigma_2,\ldots,\sigma_{2k})\mathbf{V}^\top\,,\quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{2k} \geq 0\,. \tag{3.15.13}$$

Appealing to [Lecture → Thm. 3.4.4.19] we find

$$r = \min\{\ell \in \{1,\ldots,2k\}:\ \sigma_\ell : \sigma_1 \leq \tau\}\,, \tag{3.15.14}$$

because $\sigma_1 = \|\mathbf{X}\|_2$. Directly using the formula given in [Lecture → Thm. 3.4.4.19] we have

$$\mathbf{M} = \underbrace{(\mathbf{U})_{:,1:r}\,\mathrm{diag}(\sigma_1,\ldots,\sigma_r)}_{=:\mathbf{A}_M}\underbrace{(\mathbf{V})_{:,1:r}^\top}_{=:\mathbf{B}_M^\top}\,. \tag{3.15.15}$$

Thus, $\mathbf{A}_M$ and $\mathbf{B}_M$ can simply be extracted from the SVD-factors by block access operations.

C++ code 3.15.16: EIGEN-based implementation of `concat_low_rank_best()`

```
2  std::pair<Eigen::MatrixXd, Eigen::MatrixXd> concat_low_rank_best(
3      const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
4      const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2, double tol) {
5    assert((tol > 0) && (tol < 1.0));
6    // Obtain economical SVD of X, Sub-problem (3-15.b)
7    auto [U, s, V] = eco_svd_concat(A1, B1, A2, B2);
8    unsigned int L = s.size();
9    Eigen::MatrixXd A_M = Eigen::MatrixXd::Zero(L, L);  // return dummy
10   Eigen::MatrixXd B_M = Eigen::MatrixXd::Zero(L, L);  // return dummy
11
12   // TODO: (3-15.d) Compute best low-rank approximation and return
13   // factorization.
14   // START
15
16   // Find minimal rank r, see
17   // (3.15.14)
18   unsigned int r;
19   for (r = 1; (r < L) && (s[r] / s[0] > tol); ++r)
20     ;
21   // Rescale columns of U
22   for (unsigned int j = 0; j < r; ++j) U.col(j) *= s[j];
23   // Extract sub-matrices
24   A_M = U.leftCols(r);
25   B_M = V.leftCols(r);
26   // END
27   return {A_M, B_M};
28 }
```

Get it on GitLab (concatmatlrc.hpp).

SOLUTION of (3-16.a):

By the associativity of matrix multiplication

$$\mathbf{AX} = \mathbf{Q}(\mathbf{RX}) = \mathbf{Q}_1\big(\mathbf{Q}_2\big(\dots\mathbf{Q}_{n-1}(\mathbf{RX})\dots\big)\big)\,, \quad \mathbf{Q}_j := \mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top\,.$$

As discussed in [Lecture → Ex. 1.4.3.1], for the sake of efficiency the multiplication of a matrix $\mathbf{Y} \in \mathbb{R}^{n,k}$ with $\mathbf{Q}_j$ has to be implemented as follows

$$\mathbf{Q}_j\mathbf{Y} = \mathbf{Y} - 2\mathbf{u}_j\Big(\mathbf{u}_j^\top\mathbf{Y}\Big)\,,$$

which requires an asymptotic computational effort of $O(nk)$ for $n,k \to \infty$. These matrix multiplications have to be embedded into an outer loop over $j = n-1, n-2, \dots, 1$.

C++ code 3.16.5: Method `matmult()` of CompactStorageQR

```
2  Eigen::MatrixXd CompactStorageQR::matmult(const Eigen::MatrixXd &X) const {
3    assert((X.rows() == n_) && "Wrong size of matrix factor");
4    Eigen::MatrixXd Y = Eigen::MatrixXd::Zero(n_, X.rows());
5
6    // TODO: (3-16.a) Implement matrix multiplication with computational
         cost at
7    // most O(n^2k)
8    // START
9    // Multiplication with R-factor
10   Y = M_.template triangularView<Eigen::Upper>() * X;
11   // Multiplication with Q-factors
12   Eigen::VectorXd u{Eigen::VectorXd::Zero(n_)};
13   for (int k = n_ - 2; k >= 0; --k) {
14     u.tail(n_ - k - 1) = M_.block(k + 1, k, n_ - k - 1, 1);
15     u[k] = std::sqrt(1.0 - u.squaredNorm());
16     Y -= 2 * u * (u.transpose() * Y);
17   }
18   // END
19
20   return Y;
21 }
```

Get it on GitLab (compactstorageqr.hpp).

An even more efficient implementation is possible by taking into account that the first $j-1$ entries of the vector $\mathbf{u}_j$ are zero.

HINT 1 for (3-16.b): You must exploit that the factor matrices in (3.16.2) are Householder matrices [Lecture → § 3.3.3.11], in particular, *orthogonal*. ⌟

SOLUTION of (3-16.b):

Based on the factorization of $\mathbf{A}$,

$$\mathbf{A} = \mathbf{QR} = \mathbf{Q}_1\mathbf{Q}_2 \cdot \cdots \cdot \mathbf{Q}_{n-1}\mathbf{R}\,, \quad \mathbf{Q}_j := \mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top\,,$$

we have

$$\mathbf{A}^{-1}\mathbf{B} = \mathbf{R}^{-1}\mathbf{Q}_{n-1}^{-1}\mathbf{Q}_{n-2}^{-1} \cdot \cdots \cdot \mathbf{Q}_1^{-1}\mathbf{B}\,.$$

Note that, since the vectors $\mathbf{u}_j$ from (3.16.1) are normalized, $\left\|\mathbf{u}_j\right\|_2 = 1$, the matrices $\mathbf{Q}_j$ are *orthogonal* and symmetric **Householder matrices** [Lecture → § 3.3.3.11] satisfying

$$\mathbf{Q}_j^{-1} = \mathbf{Q}_j^\top = \mathbf{Q}_j\,, \quad j = 1,\ldots,n-1\,. \tag{3.16.6}$$

We infer

$$\mathbf{A}^{-1}\mathbf{B} = \mathbf{R}^{-1}\mathbf{Q}_{n-1}\mathbf{Q}_{n-2} \cdot \cdots \cdot \mathbf{Q}_1\mathbf{B}\,,$$

which suggests an implementation very similar to `CompactStorageQR::matmult()` in Code 3.16.5. Of course $\mathbf{R}^{-1}\mathbf{Y}$ for $\mathbf{Y} \in \mathbb{R}^{n,k}$ is obtained by solving the k linear systems of equations $\mathbf{Rx} = (\mathbf{Y})_\ell$, $\ell = 1,\ldots,k$, which can be done with an asymptotic computational effort of $O(n^2)$ for $n \to \infty$, because $\mathbf{R}$ is *upper triangular*.

C++ code 3.16.7: Method `solve()` of class CompactStorageQR

```
2  Eigen::MatrixXd CompactStorageQR::solve(const Eigen::MatrixXd &B) const {
3    assert((B.rows() == n_) && "Wrong size of right-hand side");
4    Eigen::MatrixXd X{B};
5
6    // TODO: (3-16.b) Solve the linear system of equations given by AX = B
       with
7    // complexity O(n^2k).
8    // START
9    // Check invertibility by inspecting diagonal
10   // elements of R-factor
11   constexpr double atol = 1.0E-16;
12   constexpr double rtol = 1.0E-8;
13   double maxrii = std::abs(M_(0, 0));
14   for (unsigned int k = 1; k < n_; ++k) {
15     const double tmp = std::abs(M_(k, k));
16     if (maxrii < tmp) {
17       maxrii = tmp;
18     }
19   }
20   for (unsigned int k = 1; k < n_; ++k) {
21     const double rii = std::abs(M_(k, k));
22     if ((rii < rtol * maxrii) || (rii < atol)) {
23       throw std::runtime_error("CompactStorageQR::solve: matrix not regular");
24     }
25   }
26   // Multiplication with transposed Q-factors
27   Eigen::VectorXd u(n_);
28   for (unsigned int k = 0; k < n_ - 1; ++k) {
29     double sn{0};
```

```
30      for (unsigned int j = k + 1; j < n_; ++j) {
31        u[j] = M_(j, k);
32        sn += u[j] * u[j];
33      }
34      u[k] = std::sqrt(1.0 - sn);
35      // Multiplication with current Q-factor
36      X -= 2 * u * (u.transpose() * X);
37      u[k] = 0.0;
38    }
39    // Solve upper triangular linear system RX = Q^T B
40    X = M_.template triangularView<Eigen::Upper>().solve(X);
41    // END
42
43    return X;
44  }
```

Get it on GitLab (compactstorageqr.hpp).

This code also checks for the invertibility of the matrix $\mathbf{A}$, which was not requested in this sub-problem.

Inspecting the loop we see that the total asymptotic computational cost of `solve()` for $\mathbf{B} \in \mathbb{R}^{n,k}$ is $O(n^2k)$, which is optimal.

HINT 1 for (3-16.c): From linear algebra remember that $\det(\mathbf{AX}) = \det \mathbf{A} \cdot \det \mathbf{X}$ for all $\mathbf{AX} \in \mathbb{R}^{n,n}$. ⌟

SOLUTION of (3-16.c):

Thanks for multiplication theorem for determinants the factorization

$$\mathbf{A} = \mathbf{QR} = \mathbf{Q}_1\mathbf{Q}_2 \cdot \cdots \cdot \mathbf{Q}_{n-1}\mathbf{R}\,, \quad \mathbf{Q}_j := \mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top\,,$$

immediately implies

$$\det \mathbf{A} = \prod_{j=1}^{n-1} \det \mathbf{Q}_j \cdot \det \mathbf{R}\,.$$

Note that

- the matrices $\mathbf{Q}_j$ are *orthogonal* and symmetric **Householder matrices** [Lecture → § 3.3.3.11] satisfying $\det \mathbf{Q}_j = -1$,
- the determinant of the *upper triangular* matrix $\mathbf{R}$ is equal to the product of its diagonal elements.

Remark. To see that $\det \mathbf{Q}_j = -1$, let $\mathbf{U}_j \in \mathbb{R}^{n,n}$ an *orthogonal* matrix with first column $\mathbf{u}_j$:

$$\mathbf{U}_j^\top \mathbf{Q}_j \mathbf{U}_j = \mathbf{U}_j^\top (\mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top)\mathbf{U}_j = \operatorname{diag}(-1, 1, \ldots, 1)\,.$$

This is an immediate consequence of the orthonormality of the columns of $\mathbf{U}_j$. Thus $\mathbf{U}_j$ effects a similarity transform of $\mathbf{Q}_j$ to $\operatorname{diag}(-1, 1, \ldots, 1)$, which means that the determinants of the two matrices have the same value.

C++ code 3.16.8: Method `det` of class CompactStorageQR

```
2  double CompactStorageQR::det() const {
3    double d = 0.;
4
5    // TODO: (3-16.c) Compute the determinant of A.
6    // START
7    // The determinant is just the product of the diagonal entries of the
8    // R-factor, because the determinant of the Q-factor is 1.
9    d = 1.0;
10   for (unsigned int j = 0; j < n_; ++j) {
11     d *= M_(j, j);
12   }
13   // Sign correction, because Householder transformations
14   // have determinant -1 and n-1 of them are involved.
15   d *= ((n_ % 2 == 0) ? -1.0 : 1.0);
16   // END
17
18   return d;
19 }
```

Get it on GitLab (compactstorageqr.hpp).

SOLUTION of (3-17.a):

As in [Lecture → § 3.2.0.7] we rewrite the normal equations for $\mathbf{Ax} = \mathbf{b}$

$$\mathbf{A}^\top \mathbf{AX} = \mathbf{A}^\top \mathbf{b} \quad \Leftrightarrow \quad \mathbf{A}^\top (\mathbf{Ax} - \mathbf{b}) = \mathbf{0} \quad \Leftrightarrow \quad \mathbf{A}^\top \mathbf{r} = \mathbf{0}\,.$$

This equation is combined with the definition of the residual

$$\begin{aligned} -\mathbf{r} + \mathbf{Ax} &= \mathbf{b} \\ \mathbf{A}^\top \mathbf{r} &= \mathbf{0} \end{aligned} \quad \Leftrightarrow \quad \begin{bmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^\top & \mathbf{O} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}\,. \tag{3.17.2}$$

Here $\mathbf{I} = \mathbf{I}_m$ is the $m \times m$ identity matrix, and $\mathbf{O} = \mathbf{O}_{n,n}$ is the $n \times n$ zero matrix.

SOLUTION of (3-17.b):

From the formula for $\mathbf{A}_e$ from (3.17.2) we immediately see

$$\mathrm{nnz}(\mathbf{A}_e) = 2\,\mathrm{nnz}(\mathbf{A}) + m \quad .$$

HINT 1 for (3-17.c): This is the definition of EIGEN's triplet class:

```
template<typename Scalar, typename StorageIndex=typename
  SparseMatrix<Scalar>::StorageIndex >
 class Triplet
 {
 public:
   Triplet() : m_row(0), m_col(0), m_value(0) {}
   Triplet(const StorageIndex& i, const StorageIndex& j, const Scalar&
      v = Scalar(0))
     : m_row(i), m_col(j), m_value(v) {}
   const StorageIndex& row() const { return m_row; }
   const StorageIndex& col() const { return m_col; }
   const Scalar& value() const { return m_value; }
 protected:
   StorageIndex m_row, m_col;
   Scalar m_value;
 };
```

The type **StorageIndex** can be read as **unsigned int**.

SOLUTION of (3-17.c):

C++ code 3.17.4: Initialization of system matrix of extended normal equations in triplet format.

```
using Triplet = Eigen::Triplet<double>;
using TripletMatrix = std::vector<Triplet>;
TripletMatrix extNeqSysMatCOO(unsigned int m, unsigned int /*n*/,
                              const TripletMatrix &A_coo) {
  TripletMatrix A_ext_coo{};
  // Set up the m × m identity matrix block
  for (int j = 0; j < (int)m; ++j) {
    A_ext_coo.push_back(Triplet{j, j, -1.0});
  }
  for (const auto &t : A_coo) {
    // Right upper block A
    A_ext_coo.push_back(Triplet{t.row(), t.col() + (int)m, t.value()});
    // Left lower block A^T
    A_ext_coo.push_back(Triplet{t.col() + (int)m, t.row(), t.value()});
  }
  return A_ext_coo;
}
```

Note that the typecast (**int**) is included in order to suppress annoying compiler warnings about a “narrowing conversion”.

HINT 1 for (3-2.a): Compute the matrix$\times$vector product $\mathbf{T}_{\alpha,\beta}\mathbf{z}$ for using $\mathbf{z} = [z_1, z_2, \ldots, z_n]^\top$. Express the resulting vector as the product of a matrix and the vector $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$. ⌟

SOLUTION of (3-2.a):

Clearly we set $\mathbf{x} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$ ($k = 2$) and $\mathbf{b} := \mathbf{c}$ ($m = n$). The vector $\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c}$ whose norm has to be minimized in (3.2.1) can be written as:

$$\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} = \begin{bmatrix} \alpha z_1 + \beta z_2 \\ \alpha z_2 + \beta(z_1 + z_3) \\ \dots \\ \alpha z_{n-1} + \beta(z_{n-2} + z_n) \\ \alpha z_n + \beta z_{n-1} \end{bmatrix} - \mathbf{c} = \begin{bmatrix} z_1 & z_2 \\ z_2 & z_1 + z_3 \\ \vdots & \vdots \\ z_{n-1} & z_{n-2} + z_n \\ z_n & z_{n-1} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} - \mathbf{c} =: \mathbf{A}\mathbf{x} - \mathbf{b} \qquad (3.2.4)$$

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 k of unknowns is equal to 2 and the number m of equations is equal to n.

SOLUTION of (3-2.b):

The normal equations are

$$\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{c}\,, \qquad \text{for} \quad \mathbf{A} := \begin{bmatrix} z_1 & z_2 \\ z_2 & z_1 + z_3 \\ \vdots & \vdots \\ z_{n-1} & z_{n-2} + z_n \\ z_n & z_{n-1} \end{bmatrix} \in \mathbb{R}^{n,2}\,.$$

Computing the matrix product, we find

$$\mathbf{A}^\top \mathbf{A} = \begin{bmatrix} \sum_{\ell=1}^{n} z_\ell^2 & 2\sum_{\ell=1}^{n-1} z_\ell z_{\ell+1} \\ 2\sum_{\ell=1}^{n-1} z_\ell z_{\ell+1} & z_1^2 + 2\sum_{\ell=2}^{n-1}\left(z_\ell^2 + z_{\ell-1}z_{\ell+1}\right) + z_n^2 \end{bmatrix} \in \mathbb{R}^{2,2}\,.$$

HINT 1 for (3-2.c): Similarly to Problem 3-1, $\mathbf{Ax} = \mathbf{b}$ 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 (3-2.c):

C++ code 3.2.5: Solution of (3-2.c)

```
2  Eigen::VectorXd lsqEst(const Eigen::VectorXd& z, const Eigen::VectorXd& c) {
3    assert(z.size() == c.size() && "z and c must have same size");
4    Eigen::VectorXd x = Eigen::VectorXd::Zero(2);  // (α,β).
5
6    // TODO: (3-2.c) Solve eq.  (3.2.1) using the method of normal
         equations.
7    // START
8
9    // Reformulating T_{α,β}z = c as Ax = b with x = (α,β)^T,
10   // we have b = c, and A defined as follows.
11   const unsigned int n = z.size();
12   Eigen::MatrixXd A(n, 2);
13   // Initialize all entries of A:
14   A.col(0) = z;
15   A(0, 1) = 0;
16   A.col(1).tail(n - 1) = z.head(n - 1);
17   // Add remaining terms:
18   A.col(1).head(n - 1) += z.tail(n - 1);
19   // Normal equation linear system
20   Eigen::MatrixXd lhs = A.transpose() * A;  // Left-hand side
21   Eigen::VectorXd rhs = A.transpose() * c;  // Right-hand side
22   // Least squares estimate of (α,β) as solution
23   // of normal equations.
24   x = lhs.lu().solve(rhs);
25   // END
26   return x;
27 }
```

Get it on GitLab (tridiagleastsquares.hpp).

SOLUTION of (3-3.a):

We have to recast (3.3.3) as a least squares problem arising from an overdetermined linear system of equations $\mathbf{Ax} = \mathbf{b}$. 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. The argument vector `t` passes $[t_i]_{i=1}^{10}$.

C++-code 3.3.4: Creating the matrix A

```
2  Eigen::MatrixXd make_A(const Eigen::VectorXd &t) {
3    std::size_t n = t.size();
4    Eigen::MatrixXd A = Eigen::MatrixXd::Zero(n, 4);
5    // TODO: (3-3.a) Build the matrix A
6    // Hint: Evaluate the functions \phi_j at the time points defined in b
7
8    // START
9    for (std::size_t i = 0; i < n; i++) {
10     const double t_i = t[i];
11     // Initialize row i of the matrix
12     A(i, 0) = 1.0 / t_i;
13     A(i, 1) = 1.0 / (t_i * t_i);
14     A(i, 2) = std::exp(-(t_i - 1.0));
15     A(i, 3) = std::exp(-2.0 * (t_i - 1.0));
16   }
17   // END
18
19   return A;
20 }
```

Get it on GitLab (linfit.hpp).

In `data_fit_normal`, we use `make_A` to create the matrix. For the **normal equations**

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b} ,$$

we calculate $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{b}$ and then solve the system using EIGEN's built-in dense linear system solver, see [Lecture → § 2.5.0.8].

C++-code 3.3.5: Solving the Normal Equations

```
2  Eigen::VectorXd data_fit_normal(const Eigen::VectorXd &t,
3                                  const Eigen::VectorXd &f) {
4    Eigen::VectorXd gamma = Eigen::VectorXd::Zero(4);
5
```

```
    // TODO: (3-3.a) Solve normal equations to find the coefficients of the
    // linear fitting
    // START
    // Create matrix A
    Eigen::MatrixXd A = make_A(t);

    // Transpose matrix A
    Eigen::MatrixXd At = A.transpose();

    // Compute LHS (A^T*A) of the normal equation
    Eigen::MatrixXd AtA = At * A;

    // Compute RHS (A^T*b) of the normal equation
    Eigen::VectorXd Atf = At * f;

    // Solve normal equation to get coefficients
    // we use the Cholesky decomposition as the matrix AtA is positive
    // semidefinite
    gamma = AtA.ldlt().solve(Atf);
    // END

    return gamma;
}
```

Get it on GitLab (linfit.hpp).

SOLUTION of (3-3.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 3.3.6: Solving with QR-Decomposition

```
2  Eigen::VectorXd data_fit_qr(const Eigen::VectorXd &t,
3                              const Eigen::VectorXd &f) {
4    Eigen::VectorXd gamma = Eigen::VectorXd::Zero(4);
5
6    // TODO: (3-3.b) Find the coefficients for the linear
7    // fitting by means of the QR decomposition of A
8    // START
9    // Make matrix A
10   Eigen::MatrixXd A = make_A(t);
11
12   // Compute QR decomposition
13   gamma = A.colPivHouseholderQr().solve(f);
14   // END
15
16   return gamma;
17 }
```

Get it on GitLab (linfit.hpp).

SOLUTION of (3-3.c):

We use the following code for this and the next part:

C++-code 3.3.7: `fitted_function()`: computing values of $f(t)$ using the estimated parameters

```
Eigen::VectorXd fitted_function(const Eigen::VectorXd &gamma,
                                const Eigen::VectorXd &t_vec) {
  const std::size_t n = t_vec.size();
  Eigen::VectorXd y = Eigen::VectorXd::Zero(n);

  // TODO: (3-3.c) Compute the values of f(t) using the estimated parameters.
  // START
  Eigen::MatrixXd A = make_A(t_vec);
  y = A * gamma;
  // END

  return y;
}
```

Get it on GitLab (linfit.hpp).

SOLUTION of (3-3.d):

C++-code 3.3.8: `fitting_error()` : computing the deviation from measured data

```
Eigen::VectorXd fitting_error(const Eigen::VectorXd &gamma) {
  Eigen::VectorXd err = 1000. * Eigen::VectorXd::Ones(10);
  Eigen::VectorXd t_vec = Eigen::VectorXd::LinSpaced(10, 0.1, 1.0);
  Eigen::VectorXd f(10);
  f << 100., 34., 17., 12., 9., 6., 5., 4., 4., 2.;

  // TODO: (3-3.d) Compute the vector of squared errors of your
  // fit at the provided sample points
  // START
  err = fitted_function(gamma, t_vec) - f;
  err = err.cwiseProduct(err);
  // END

  return err;
}
```

Get it on GitLab (linfit.hpp).

HINT 1 for (3-4.a): There is a direct proof just employing the rules for multiplying matrices. First establish the property of $\mathbf{Q} = \mathbf{A}\mathbf{R}^{-1}$ and then that of $\mathbf{R}$. The following result is useful:

Lemma [Lecture → Lemma 1.3.1.9]. Group of regular diagonal/triangular matrices

$$\mathbf{A}, \mathbf{B} \left\{ \begin{array}{c} \textit{diagonal} \\ \textit{upper triangular} \\ \textit{lower triangular} \end{array} \right. \Rightarrow \quad \mathbf{A}\mathbf{B} \quad \textit{and} \quad \mathbf{A}^{-1} \left\{ \begin{array}{c} \textit{diagonal} \\ \textit{upper triangular} \\ \textit{lower triangular} \end{array} \right. .$$

(assumes that $\mathbf{A}$ is regular)

SOLUTION of (3-4.a):

The assertion (3.4.1) can be proved in different ways:

I. Based on the hint 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

$$\begin{aligned}
(\mathbf{Q})_{j,k} &= \sum_{0\le l\le n} a_{j,l}s_{l,k} = \sum_{l\in\{j-1,j,j+1\}} a_{j,l}s_{l,k} \quad \text{, because } \mathbf{A} \text{ is tridiagonal,} \\
&= \sum_{l\in\{j-1,j,j+1\},\, l\le k} a_{j,l}s_{l,k} \quad \text{, because } \mathbf{R}^{-1} \text{ is upper triangular,} \\
&= 0 \quad \text{, if } j > k+1\,, \quad j,k \in \{1,\ldots,n\}\,.
\end{aligned} \tag{3.4.2}$$

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 $i > j+1$)

$$\begin{aligned}
(\mathbf{R})_{j,k} &= \sum_{0\le l\le n} \left(\mathbf{Q}^\top\right)_{j,l} a_{l,k} = \sum_{0\le l\le n} q_{l,j}a_{l,k} \\
&= \sum_{l\in\{k-1,k,k+1\}} q_{l,j}a_{l,k} \quad \text{, because } \mathbf{A} \text{ is tridiagonal,} \\
&= \sum_{l\in\{k-1,k,k+1\},\, l\le j+1} q_{l,j}a_{l,k} \quad \text{, because of the pattern of } \mathbf{Q}, \\
&= 0 \quad \text{, if } j+1 < k-1 \quad \text{, i.e., if } k > j+2\,, \quad j,k \in \{1,\ldots,n\}\,.
\end{aligned} \tag{3.4.3}$$

In words, if $k > j+2$ the non-zero entries in the k-th column of $\mathbf{A}$ meet only the trailing zeros in the j-th row of $\mathbf{Q}^\top$: $\mathbf{Q}^\top\mathbf{A} =$

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ * & * & * & * & * & 0 & 0 & 0 \\ * & * & * & * & * & * & 0 & 0 \\ * & * & * & * & * & * & * & 0 \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \end{bmatrix} \begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ \star & * & * & * & 0 & 0 & 0 & 0 \\ \star & \star & * & * & * & 0 & 0 & 0 \\ \star & \star & \star & * & * & * & 0 & 0 \\ \star & \star & \star & \star & * & * & * & 0 \\ \star & \star & \star & \star & \star & * & * & * \\ \star & \star & \star & \star & \star & \star & * & * \\ \star & \star & \star & \star & \star & \star & \star & * \end{bmatrix}.$$

Here the entries marked with $\star$ are zero by the very definition of the QR-decomposition, no by the rules of matrix multiplication.

II. We can exploit the algebraic equivalence of QR-decomposition and Gram-Schmidt orthonormalization explained in [Lecture → Section 3.3.3.1]. [Lecture → Thm. 3.3.3.2] tells us that the spans of the k leftmost columns of $\mathbf{Q}$ and $\mathbf{A}$ are the same,

$$\mathrm{Span}\{(\mathbf{Q})_{:,1},\ldots,(\mathbf{Q})_{:,k}\} = \mathrm{Span}\{(\mathbf{A})_{:,1},\ldots,(\mathbf{A})_{:,k}\}\,, \tag{3.4.4}$$

because $\mathbf{A}$ has full rank n, since it is regular. Obviously, for a tridiagonal matrix $\mathbf{A}$ we have

$$\operatorname{Span}\{(\mathbf{A})_{:,1},\ldots,(\mathbf{A})_{:,k}\} \subset \operatorname{Span}\{\mathbf{e}_1,\ldots,\mathbf{e}_k\} = \left\{ \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ \xi_j \in \mathbb{R} \right\}. \tag{3.4.5}$$

Combining (3.4.4) and (3.4.5) gives the assertion on the pattern of $\mathbf{Q}$. Then proceed as in (3.4.3).

III. 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. 3.3.3.26]. Applying these Givens rotations to an identity matrix reveals the structure of $\mathbf{Q}^\top$.

SOLUTION of (3-4.b):

The first part of the implementation amounts to copying the code of [Lecture → Code 3.3.3.17]. Then use the formulas from [Lecture → Rem. 3.3.3.21] to define the code ρ depending on the entries of the rotation matrix, see [Lecture → Eq. (3.3.3.22)].

C++ code 3.4.7: Computing end enconding a Givens rotation

```
2  std::tuple<double, double, double> compGivensRotation(Eigen::Vector2d a) {
3    double rho, gamma, sigma;
4    // TODO (3-4.b):  Compute the parameters of a rotation that when
5    // applied to a results in the vector (‖a‖,0)^T or (−‖a‖,0)^T.
6    // Hint:  You can use the function sign() above when determining rho
7    // from gamma and sigma.
8    // START
9    if (a(1) != 0.0) {
10     double t;
11     // Separate cases to avoid cancellation/overflow
12     if (std::abs(a(1)) > std::abs(a(0))) {
13       t = -a(0) / a(1);
14       sigma = 1.0 / std::sqrt(1.0 + t * t);
15       gamma = sigma * t;
16     } else {             // Then abs(a(0)) >= abs(a(1)) > 0
17       t = -a(1) / a(0);  // a(0) != 0
18       gamma = 1.0 / std::sqrt(1.0 + t * t);
19       sigma = gamma * t;
20     }
21   } else {
22     // No rotation.
23     gamma = 1.0;
24     sigma = 0.0;
25   }
26   // Determine rho
27   if (gamma == 0.0)
28     rho = sign(sigma);
29   else if (std::abs(gamma) > std::abs(sigma))
30     rho = std::sqrt(2.0) * sign(gamma) * sigma;
31   else
32     rho = 0.5 * std::sqrt(2.0) * sign(sigma) / gamma;
33   // END
34   return std::make_tuple(rho, gamma, sigma);
35 }
```

Get it on GitLab (tridiagqr.hpp).

The function `sign()` just returns the sign of a number:

```
template <typename T>
int sign(T val) {
  return (T(0) < val) - (val < T(0));
}
```

HINT 1 for (3-4.c): First copy the data from `A` to `B` and then perform in-situ (in-place) transformations on `B`. ⌟

HINT 2 for (3-4.c): An implementation, which uses a different data structure, however, can be found in the first part of [Lecture → Code 3.3.4.5]. ⌟

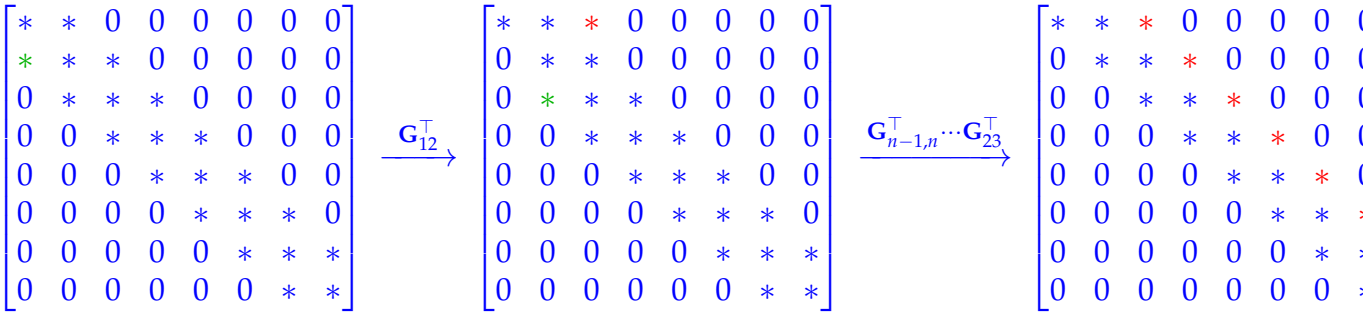
SOLUTION of (3-4.c):

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:



$*$ ≙ entry set to zero by Givens rotation, $*$ ≙ new non-zero entry

C++ code 3.4.8: 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 (unsigned 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);
    // TODO (3-4.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
  }
```

```
34 }
```

Get it on GitLab (tridiagqr.hpp).

Note that a more compact way to handle multiple return values is presented in [Lecture → Code 0.3.4.3].

HINT 1 for (3-4.d): Look up [Lecture → Eq. (3.3.3.23)] to understand how to recover the rotation from the code ρ. ⌟

SOLUTION of (3-4.d):

The following code reconstructs the rotation matrix from ρ.

C++ code 3.4.9: Decoding compact storage of Givens matrices

```
inline Eigen::Matrix2d Givens(double rho) {
  double gamma, sigma;
  if (std::abs(rho) < 1.0) {
    sigma = 0.5 * std::sqrt(2.0) * rho;
    gamma = std::sqrt(1.0 - sigma * sigma);
  } else if (std::abs(rho) > 1.0) {
    gamma = 0.5 * std::sqrt(2.0) / rho;
    sigma = std::sqrt(1 - gamma * gamma);
  } else {
    gamma = 0.0;
    sigma = sign(rho);
  }
  Eigen::Matrix2d G;
  G << gamma, sigma, -sigma, gamma;
  return G;
}
```

Get it on GitLab (tridiagqr.hpp).

This code applies the $n-1$ rotations encoded in `rho` to consecutive entries of the vector $\mathbf{x}$.

C++ code 3.4.10: 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);
  // TODO (3-4.d):  Fill in the entries of y.
  // START
  for (unsigned int k = 0; k < n; ++k) {
    y[k] = x[k];
  }
  Eigen::Matrix2d G;
  for (unsigned int k = 0; k < n - 1; ++k) {
    G = Givens(rho[k]).transpose();
    y.segment(k, 2) = G * y.segment(k, 2);
  }
  // END
  return y;
}
```

Get it on GitLab (tridiagqr.hpp).

HINT 1 for (3-4.e): You may get some inspiration from the second part of [Lecture → Code 3.3.4.5]. ⌟

SOLUTION of (3-4.e):

As explained in [Lecture → Rem. 3.3.4.3], solving the linear system $\mathbf{Ax} = \mathbf{b}$ with a QR-decomposition of $\mathbf{A}$ already available involves the following two steps:

1. First step: $\mathbf{c} = \mathbf{Q}^\top \mathbf{b}$ by `applyQT()`
2. Second step: back-substitution for a tridiagonal matrix

The challenge in Step 2 (back-substitution) is to access the right entries of the array $\mathbf{B}$. It is advisable to elaborate the algorithm for the matrix $\mathbf{R}$ first and then translate to the "$\mathbf{B}$ storage scheme".

C++ code 3.4.11: Method `solve` of TriDiagonalQR

```
2  template <typename VecType>
3  Eigen::VectorXd TriDiagonalQR::solve(const VecType &b) const {
4    assert(n == b.size() &&
5           "Input must have size equal to the number of rows in Q.");
6    Eigen::VectorXd x(n);
7    // TODO (3-4.e): Solve the linear system A*x = b if A is invertible,
8    // otherwise throw an std::runtime_error.
9    // START
10   if ((B.col(0).array().abs() < 1E-10).any()) {
11     throw std::runtime_error("A is not invertible!");
12   }
13   // Multiply by Q^T on both sides of Q*R*x=b:
14   // R*x = Q^T*b = c
15   Eigen::VectorXd c = applyQT(b);
16
17   // Back-substitution
18   int k = n - 1;
19   x(k) = c(k) / B(k, 0);
20   k--;
21   x(k) = (c(k) - B(k, 1) * x(k + 1)) / B(k, 0);
22   k--;
23   while (k >= 0) {
24     x(k) = (c(k) - B(k, 1) * x(k + 1) - B(k, 2) * x(k + 2)) / B(k, 0);
25     k--;
26   }
27   // END
28   return x;
29 }
```

Get it on GitLab (tridiagqr.hpp).

SOLUTION of (3-4.f):

All functions/methods enjoy the asymptotic complexity $O(n)$ for $n \to \infty$ as they all involve a single loop.

HINT 1 for (3-4.g): Take into account [Lecture → Rem. 2.5.0.10] for the sake of efficient implementation. ⌟

SOLUTION of (3-4.g):

For the sake of efficiency it is important to compute the QR-decomposition outside the loop, recall [Lecture → Rem. 2.5.0.10]. This is done in the constructor of **TriDiagonalQR**.

C++ code 3.4.12: Implementation og `invit()` for TriDiagonalMartrix

```
2  template <>
3  unsigned int invit<TriDiagonalMatrix>(const TriDiagonalMatrix &A,
4                                        Eigen::VectorXd &x, double TOL,
5                                        unsigned int maxit) {
6    unsigned int it = 0;
7    // TODO (3-4.g): Specialize (efficiently) the template function invit()
8    // for the TriDiagonalMatrix class.
9    // START
10   x.normalize();
11   Eigen::VectorXd x_old(x.size());
12   TriDiagonalQR Aqr(A);
13   do {
14     x_old = x;
15     x = Aqr.solve(x_old);
16     x.normalize();
17   } while (((x - x_old).norm() > TOL) && (it++ < maxit));
18   // END
19   return it;
20 }
```

Get it on GitLab (tridiagqr.hpp).

Remark. Computing the QR-decomposition outside the loop does not make a difference in terms of asymptotic complexity, because it merely requires $O(n)$ operations.

HINT 1 for (3-5.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 $\mathbf{\Sigma}$ of a singular value decomposition presents the diagonal entries sorted in decreasing order: $(\mathbf{\Sigma})_{1,1} \geq (\mathbf{\Sigma})_{2,2} \cdots \geq (\mathbf{\Sigma})_{n,n}$. ⌟

SOLUTION of (3-5.a):

See also [Lecture → Section 3.4.4.1]. For any orthogonal matrix $\mathbf{V}$ and any vector $\mathbf{x}$, the following holds:

$$\|\mathbf{V}\mathbf{x}\|^2 = (\mathbf{V}\mathbf{x})^\top \mathbf{V}\mathbf{x} = \mathbf{x}^\top \mathbf{V}^\top \mathbf{V}\mathbf{x} = \mathbf{x}^\top \mathbf{x} = \|\mathbf{x}\|^2. \tag{3.5.4}$$

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:

$$\|\mathbf{A}\mathbf{x}\| = \|\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \mathbf{x}\| = \|\mathbf{U}\mathbf{\Sigma}\mathbf{y}\| = \|\mathbf{\Sigma}\mathbf{y}\| \tag{3.5.5}$$

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:

$$\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{v}_n\|_2 = \sigma_n. \tag{3.5.6}$$

Firstly, we prove "$\geq$". We have the following:

$$\begin{aligned}
\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| &= \min_{\|\mathbf{y}\|=1} \|\mathbf{\Sigma}\mathbf{y}\| && (3.5.7)\\
&= \min_{\|\mathbf{y}\|=1} \sqrt{\sigma_1^2 y_1^2 + \ldots + \sigma_n^2 y_n^2} && (3.5.8)\\
&\geq \min_{\|\mathbf{y}\|=1} \sqrt{\sigma_n^2 y_1^2 + \ldots + \sigma_n^2 y_n^2} && (3.5.9)\\
&= \sigma_n \min_{\|\mathbf{y}\|=1} \sqrt{y_1^2 + \ldots + y_n^2} && (3.5.10)\\
&= \sigma_n \min_{\|\mathbf{y}\|=1} \|\mathbf{y}\| = \sigma_n. && (3.5.11)
\end{aligned}$$

Here we first use the previously obtained equation and then compute $\mathbf{\Sigma}\mathbf{y}$ using the fact that $\mathbf{\Sigma}$ is a diagonal matrix (when computing the norm).

Secondly, we prove "$\leq$". We can bound the minimum by:

$$\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \min_{\|\mathbf{y}\|=1} \|\mathbf{\Sigma}\mathbf{y}\| \leq \|\mathbf{\Sigma}\mathbf{e}_n\| = \sigma_n \ .$$

We have therefore proven the second equality (3.5.6), i.e. that $\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \sigma_n$, and $\mathbf{e}_n = \operatorname{argmin}_{\|\mathbf{y}\|}$

Let us reconstruct $\mathbf{x}$ from $\mathbf{y} = \mathbf{V}^\top \mathbf{x}$. Since $\mathbf{V}$ is orthogonal, we have that:

$$\mathbf{e}_n = \mathbf{y} = \mathbf{V}^\top \mathbf{x} \quad \Longrightarrow \quad \mathbf{x} = \mathbf{V}\mathbf{y} = \mathbf{V}\mathbf{e}_n = \mathbf{V}_{:,n} =: \mathbf{v}_n. \tag{3.5.12}$$

HINT 1 for (3-6.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 (3-6.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 (3.6.2), respectively. Notice, that the values in the i-th column of $\mathbf{I} - \mathbf{AX}$ depend only on the values in the vector $\mathbf{x}_i$. Hence, in order to minimize the Frobenius norm $\|\mathbf{I} - \mathbf{AX}\|_F^2$ one needs to (independently) minimize each of the terms $\|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2$:

$$\|\mathbf{I} - \mathbf{AX}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{I}\mathbf{e}_i - \mathbf{AX}\mathbf{e}_i\|_2^2 = \sum_{i=1}^{n} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2.$$

The corresponding minimization problems read

$$\mathbf{b}_i := \operatorname*{argmin}_{\mathbf{x}_i \in \mathcal{P}\left((\mathbf{A})_{:,i}\right)} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \ldots, N, \tag{3.6.3}$$

where for a vector $\mathbf{a} \in \mathbb{R}^N$ we define $\mathcal{P}(\mathbf{a}) := \{\mathbf{x} \in \mathbb{R}^N : (\mathbf{a})_i = 0 \;\Rightarrow\; (\mathbf{x})_i = 0\}$, which is the set of vectors of the same length, with potentially non-zero components at the same positions as $\mathbf{a}$

Note that (3.6.3) is not yet a least-squares problem in standard form. To convert it into such, we have to introduce

$$\mathcal{J} := \{j : \; (\mathbf{A})_{j,i} \neq 0\} \quad , \quad m := \sharp\mathcal{J} \; .$$

and define

$$\mathbf{A}_i := (\mathbf{A})_{:,\mathcal{J}} \in \mathbb{R}^{N,m} \; .$$

Then (3.6.3) can be stated as

$$\mathbf{y} \in \mathbb{R}^m\text{:} \quad \|\mathbf{e}_i - \mathbf{A}_i\mathbf{y}\|_2 \to \min \; .$$

HINT 1 for (3-6.b): For an efficient implementation, you can exploit the internal CRS data structure underlying `Eigen::SparseMatrix<double>`. Use

- `Eigen::SparseMatrix<double>::valuePtr()`,
- `Eigen::SparseMatrix<double>::innerIndexPtr()` and
- `Eigen::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 **B** in EIGEN using the intermediate triplet format (`std::vector<Triplet`
see [Lecture → Section 2.7.2], before converting it to `Eigen::SparseMatrix<double>`. Use the method `setFromTriplets()`. ⌟

SOLUTION of (3-6.b):

The formula (3.6.3) 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 $\mathbf{x} \mapsto [x_{j_1}, \ldots, x_{j_{m_i}}] =: \tilde{\mathbf{x}}$, where j_k, $k = 1, \ldots, m_i$ 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. $a_i = 0$:

$$\mathbf{C} := [(\mathbf{A})_{:,j_1}, \ldots, (\mathbf{A})_{:,j_{m_i}}] .$$

We obtain, *cf.* the solution of Sub-problem (3-6.a),

$$\mathbf{b}_i := \underset{\mathbf{x}_i \in \mathbb{R}^{m_i}}{\operatorname{argmin}} \|\mathbf{e}_i - \mathbf{C}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \ldots, N.$$

This canonical form can be solved with the standard normal formula.

See Equation 3.6.6 for the EIGEN-based implementation in C++.

C++11-code 3.6.4: Computation of $\mathbf{B}$.

```
2  Eigen::SparseMatrix<double> spai(Eigen::SparseMatrix<double>& A) {
3    // Size check
4    assert(A.rows() == A.cols() && "Matrix must be square!");
5    unsigned int N = A.rows();
6
7    Eigen::SparseMatrix<double> B(N, N);
8
9    // TODO: (3-6.b) Compute the sparse approximate inverse of A by
       exploiting
10   // Eigen's internal CCS data structure
11   // START
12
13   // Needed to make sure ...Ptr functions return
14   // arrays specified in CCS format
15   A.makeCompressed();
16
17   // Obtain pointers to data of A
18   double* valPtr = A.valuePtr();
19   index_t* innPtr = A.innerIndexPtr();
20   index_t* outPtr = A.outerIndexPtr();
21
22   // Create vector for triplets of B and reserve enough space
23   std::vector<Eigen::Triplet<double>> triplets;
24   triplets.reserve(A.nonZeros());
25
26   // Loop over each column of A
27   for (unsigned int i = 0; i < N; ++i) {
28     // Number of non zeros in current column of A
29     index_t nnz_i = outPtr[i + 1] - outPtr[i];
30     // If column is empty:  skip column (matrix is not invertible)
31     if (nnz_i == 0) continue;
32
33     // Temporarily create a (small, dense) matrix on which normal
         formula
34     // will be applied.  We project the space P(A)
35     // onto P(a_i)
```

```
36      Eigen::SparseMatrix<double> C(N, nnz_i);
37      std::vector<Eigen::Triplet<double>> C_triplets;
38      C_triplets.reserve(nnz_i * nnz_i);
39
40      // Need to build matrix C.  To this end we remove all columns
41      // from A for which a_i = 0.
42      // To this end:  loop over all non-zero entries of the i-th column
43      // This loop has length n
44      for (unsigned int k = outPtr[i]; k < outPtr[i + 1]; ++k) {
45        // Row of this non-zero entry
46        index_t row_k = innPtr[k];
47        // Number of non-zero entries for row_k-th column
48        index_t nnz_k = outPtr[row_k + 1] - outPtr[row_k];
49        // Loop over all non-zeros of row_k-th-column
50        // This loop has length complexity n
51        for (unsigned int l = 0; l < nnz_k; ++l) {
52          unsigned int innIdx = outPtr[row_k] + l;
53          C_triplets.emplace_back(Eigen::Triplet<double>(
54              innPtr[innIdx], k - outPtr[i], valPtr[innIdx]));
55        }
56      }
57      C.setFromTriplets(C_triplets.begin(), C_triplets.end());
58      C.makeCompressed();
59
60      // Compute C^T C and solve normal equation.
61      // Complexity of product:  O(n^3)
62      // Complexity of solve:  O(n^3)
63      // Size of b is at most n.
64      Eigen::SparseMatrix<double> S = C.transpose() * C;
65      Eigen::MatrixXd M = Eigen::MatrixXd(S);
66      Eigen::VectorXd xt = C.row(i).transpose();
67      Eigen::VectorXd b = M.partialPivLu().solve(xt);
68
69      // Loop of length at most n.
70      for (unsigned int k = 0; k < b.size(); ++k) {
71        triplets.emplace_back(
72            Eigen::Triplet<double>(innPtr[outPtr[i] + k], i, b(k)));
73      }
74    }
75
76    // Build and return SPAI preconditioner
77    B.setFromTriplets(triplets.begin(), triplets.end());
78    // END
79
80    B.makeCompressed();
81    return B;
82  }
```

Get it on GitLab (spai.hpp).

SOLUTION of (3-6.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 $O(Nn^3)$.

HINT 1 for (3-6.d): To use EIGEN's **ConjugateGradient** class without a preconditioner you can use EIGEN's **IdentityPreconditioner** class.
Be aware that this is not the default preconditioner of **ConjugateGradient**! ⌟

HINT 2 for (3-6.d): To use EIGEN's **ConjugateGradient** class with $\frac{1}{2}(\mathbf{B}+\mathbf{B}^T)$ as a preconditiioner, use the method `precoditioner()` and the provided class **SymmetrizedSPAIPreconditioner**. ⌟

SOLUTION of (3-6.d):

C++11-code 3.6.6: Testing the SPAI preconditioner.

```
std::vector<std::pair<unsigned int, unsigned int>> testSPAIPrecCG(
    unsigned int L) {
  std::vector<std::pair<unsigned int, unsigned int>> cg_iterations(L);

  // TODO: (3-6.d) Compute the iterations needed by the
     ConjugateGradient solver
  // without a preconditioner as well as with one

  // START

  // initialize r.h.s.  with
  // its maximal size we will use head() to get smaller versions of it
  Eigen::VectorXd b = Eigen::VectorXd::Ones(1 << (2 * L));

  // loop over n = 2^{1,...,L}, keep two running variables
  for (unsigned int n = 2, i = 0; n <= (1 << L); n <<= 1, ++i) {
    unsigned int N = n * n;
    Eigen::SparseMatrix<double> A = init_A(n);
    Eigen::SparseMatrix<double> B = spai(A);
    // compute Ax = b without preconditioner
    Eigen::ConjugateGradient<Eigen::SparseMatrix<double>,
                             Eigen::Lower | Eigen::Upper,
                             Eigen::IdentityPreconditioner>
        cg(A);
    Eigen::VectorXd sol = cg.solve(b.head(N));

    // compute Ax = b with preconditioner
    Eigen::ConjugateGradient<
        Eigen::SparseMatrix<double>, Eigen::Lower | Eigen::Upper,
        SymmetrizedSPAIPreconditioner<Eigen::SparseMatrix<double>>>
        cg_preconditioner;
    // note the wrapper around B.transpose():
    // this changes the storage order which has to match when executing
    // operator+
    SymmetrizedSPAIPreconditioner<Eigen::SparseMatrix<double>> preconditioner(
        0.5 * (B + Eigen::SparseMatrix<double>(B.transpose())));
    cg_preconditioner.preconditioner() = preconditioner;
    cg_preconditioner.compute(A);
    Eigen::VectorXd sol_prec = cg_preconditioner.solve(b.head(N));

    // check if both solutions coincide
    assert((sol - sol_prec).norm() <= 1e-9);

    cg_iterations[i] = {cg.iterations(), cg_preconditioner.iterations()};
  }
  // END

  return cg_iterations;
}
```

Get it on GitLab (spai.hpp).

SOLUTION of (3-7.a):

Let $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$. Then, for every $i = 1, \ldots, n$:

$$\begin{bmatrix} p_1^i \\ p_2^i \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1^i \\ x_2^i \end{bmatrix} = \begin{bmatrix} a_{11}x_1^i + a_{12}x_2^i \\ a_{21}x_1^i + a_{22}x_2^i \end{bmatrix} \tag{3.7.2}$$

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:

$$\mathbf{v} = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{bmatrix} \tag{3.7.3}$$

Rewriting (3.7.2):

$$\begin{bmatrix} p_1^i \\ p_2^i \end{bmatrix} = \begin{bmatrix} x_1^i & x_2^i & 0 & 0 \\ 0 & 0 & x_1^i & x_2^i \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{bmatrix} \tag{3.7.4}$$

Vertically stacking all the equations, for $i = 1, \ldots, n$:

$$\underbrace{\begin{bmatrix} p_1^1 \\ p_2^1 \\ p_1^2 \\ p_2^2 \\ \vdots \\ p_1^n \\ p_2^n \end{bmatrix}}_{\mathbf{w}} = \underbrace{\begin{bmatrix} x_1^1 & x_2^1 & 0 & 0 \\ 0 & 0 & x_1^1 & x_2^1 \\ x_1^2 & x_2^2 & 0 & 0 \\ 0 & 0 & x_1^2 & x_2^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_1^n & x_2^n & 0 & 0 \\ 0 & 0 & x_1^n & x_2^n \end{bmatrix}}_{\mathbf{B}} \underbrace{\begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{bmatrix}}_{\mathbf{v}}.$$

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.

Note that the order of the matrix entries in the vector $\mathbf{v}$ can be different, of course, leading to a matrix $\mathbf{B}$ with permuted columns.

SOLUTION of (3-7.b):

The matrix has $\mathrm{rank}(\mathbf{B}) \leq 4$. 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.

SOLUTION of (3-7.c):

C++-code 3.7.5: Building the system matrix $\mathbf{B}$.

```
2  Eigen::MatrixXd shape_ident_matrix(const Eigen::MatrixXd &X) {
3    assert(X.rows() == 2 && "X must have 2 rows!");
4
5    unsigned int n = X.cols();
6    Eigen::MatrixXd B = Eigen::MatrixXd::Zero(2 * n, 4);
7
8    // TODO: (3-7.c) Build system matrix B.
9
10   // START
11   for (unsigned int row = 0; row < n; ++row) {
12     // Odd row, first two columns
13     B(2 * row, 0) = X(0, row);
14     B(2 * row, 1) = X(1, row);
15     // Even row, last two columns
16     B(2 * row + 1, 2) = X(0, row);
17     B(2 * row + 1, 3) = X(1, row);
18   }
19   // END
20
21   return B;
22 }
```

Get it on GitLab (shape_ident.hpp).

SOLUTION of (3-7.d):

The following code demonstrates advanced use of EIGEN's facilities. Of course, everything can be done "in a pedestrian way". Nevertheless it is worth while trying to understand that very compact computation of the matrix $\mathbf{A}$.

C++-cde 3.7.6: Solving the least square problem.

```
double solve_lsq(const Eigen::MatrixXd &X, const Eigen::MatrixXd &P,
                 Eigen::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();
  double norm_of_residual = 0.;

  // TODO: (3-7.d) solve LSQ problem, return best linear approximation
      and
  // residual

  // START
  // Build system matrix
  Eigen::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 = Eigen::Map<Eigen::MatrixXd>(
          Eigen::MatrixXd(
              // Solve LSQ problem using normal equation
              (B.transpose() * B)
                  .ldlt()
                  .solve(B.transpose() *
                         // Need to vectorize matrix
                         Eigen::Map<const Eigen::MatrixXd>(P.data(), 2 * n, 1))
              // Pass an array to "EIgen::Map"
              )
              .data(),
          // Need to reshape vector to 2x2 matrix
          2, 2)
          .transpose();

  // Residual
  norm_of_residual =
      (P - A * X).norm();
  // END

  return norm_of_residual;
}
```

Get it on GitLab (shape_ident.hpp).

SOLUTION of (3-7.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++-code 3.7.7: Code for shape identification.

```
2  Shape identify(const Eigen::MatrixXd Xstop, const Eigen::MatrixXd Xpriority,
3                 const Eigen::MatrixXd &P, Eigen::MatrixXd &A) {
4    // TODO: (3-7.e) Use residual do decide wether shape defines stop or
       priority
5    // road sign.
6
7    // START
8    // Best linear transformation for both "model" shape
9    Eigen::MatrixXd Astop, Apriority;
10
11   // Residuals and linear transform for both "models"
12   double res_stop = solve_lsq(Xstop, P, Astop);
13   double res_priority = solve_lsq(Xpriority, P, Apriority);
14
15   // If residual with stop Model is bigger than the one with priority
       sign,
16   // probably it it is priority sign, otherwise is a stop sign.
17   if (res_stop <= res_priority) {
18     A = Astop;
19     return Stop;
20   } else {
21     A = Apriority;
22     return Priority;
23   }
24   // END
25 }
```

Get it on GitLab (shape_ident.hpp).

HINT 1 for (3-8.a): Recall [Lecture → Thm. 3.4.1.1] and [Lecture → Lemma 3.4.1.13]. ⌟

SOLUTION of (3-8.a):

See also [Lecture → Lemma 3.4.4.15].

❶ Denote by σ_i the singular values of the matrix $\mathbf{X}$ [Lecture → Def. 3.4.1.3]. According to [Lecture → Lemma 3.4.1.13] 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 $\boldsymbol{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_k, 0, \ldots, 0)$, see [Lecture → Eq. (3.4.1.

Thus, we can omit the last rows/columns of the matrices in the decomposition: $\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top = \mathbf{U}_{:,1:k}\boldsymbol{\Sigma}_{1:k,1:k}$

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.

❷ An alternate argument for (i)⇒(ii): As $\operatorname{rank}(\mathbf{X}) = \dim \mathcal{R}(\mathbf{X}) = k$, we can find a basis of $\mathcal{R}(\mathbf{X}) \subset \mathbb{R}^m$ composed of k linearly independent vectors and collect them into the columns of a matrix $\mathbf{A} \in \mathbb{R}^{m,k}$. Thus we can write

$$\mathbf{X}\mathbf{z} = \sum_{\ell=1}^{k} f_\ell(\mathbf{z})(\mathbf{A})_{:,\ell} \quad \forall \mathbf{z} \in \mathbb{R}^n \,,$$

with *linear functionals* $f_\ell : \mathbb{R}^n \to \mathbb{R}$. Every f_ℓ has a matrix representation through a row vector:

$$\exists \mathbf{b}_\ell \in \mathbb{R}^n\colon \quad f_\ell(\mathbf{z}) = \mathbf{b}_\ell^\top \mathbf{z} \quad \forall \mathbf{z} \in \mathbb{R}^n \,.$$

$$\blacktriangleright \qquad \mathbf{X}\mathbf{z} = \sum_{\ell=1}^{k} (\mathbf{A})_{:,\ell} \boldsymbol{v}_\ell^\top \mathbf{z} = \mathbf{A}\mathbf{B}^\top \mathbf{z} \quad \forall \mathbf{z} \in \mathbb{R}^n \quad \text{with} \quad \mathbf{B} := \left[\mathbf{b}_1, \cdots, \mathbf{b}_k\right] \in \mathbb{R}^{n,k} \,.$$

Of course, there are many more ways to prove the asserted equivalence.

HINT 1 for (3-8.b): Study [Lecture → Code 3.4.2.1], use EIGEN's facilities for *economical SVD* [Lecture → Section 3.4.2] and pay attention on how to deal with the SVD-factors $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$ properly. ⌟

SOLUTION of (3-8.b):

The following code uses the economical SVD of $\mathbf{X} \in \mathbb{R}^{m,n}$ [Lecture → § 3.4.1.4]:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top , \quad \mathbf{U} \in \mathbb{R}^{m,\min\{m,n\}} , \quad \mathbf{V} \in \mathbb{R}^{n,\min\{m,n\}} , \quad \mathbf{\Sigma} \in \mathbb{R}^{\min\{m,n\},\min\{m,n\}} .$$

If $\operatorname{rank}(X) = k \leq \min\{m,n\}$, then only the upper left $k \times k$-block of $\mathbf{\Sigma}$ is different from zero. Therefore, in this case

$$\mathbf{X} = \mathbf{A}\mathbf{B}^\top \quad \text{with} \quad \mathbf{A} := (\mathbf{U})_{:,1:k}(\mathbf{\Sigma})_{1:k,1:k} \in \mathbb{R}^{m,k} , \quad \mathbf{B} := (\mathbf{V})_{:,1:k} \in \mathbb{R}^{n,k} . \tag{3.8.1}$$

Note that testing from $\operatorname{rank}(\mathbf{X}) = k$ has to rely on the relative size of its singular values, compare [Lecture → Code 3.4.2.5].

The following code implements (3.8.1). Also consult [Lecture → Code 3.4.2.1] as regards the use of EIGEN's built-in functions.

C++11-code 3.8.2: Solution of (3-8.b)

```
2  std::pair<Eigen::MatrixXd, Eigen::MatrixXd> factorize_X_AB(
3      const Eigen::MatrixXd& X, const unsigned int k) {
4    std::size_t m = X.rows(), n = X.cols();
5    Eigen::MatrixXd A, B;
6    constexpr double tol =
7        1e-6;  // Tolerance for numerical rank: \lref{ex:svdrank}
8    // TODO: (3-8.b) Define A,B with k columns such that X = A*B^T
9    // Hint: warnings can be displayed with std::cerr, error messages with assert
10   // START
11   assert(k <= std::min(m, n) && "Rank k cannot be larger than dimensions of X");
12
13   Eigen::JacobiSVD<Eigen::MatrixXd> svd(
14       X, Eigen::ComputeThinU | Eigen::ComputeThinV);
15   // With Eigen::svd you can ask for thin U or V to be computed.
16   // In case of a rectangular m x n matrix,
17   // with j the smaller value among m and n,
18   // there can only be at most j singular values.
19   // The remaining columns of U and V do not correspond
20   // to actual singular vectors and are not computed in thin format.
21
22   Eigen::VectorXd s = svd.singularValues();
23
24   if (k + 1 <= std::min(m, n) && s(k) > tol * s(0)) {
25     // 1.\ condition checks if there is a singular value after k.
26     // (S cannot be larger than the dimensions of X.)
27     // 2.\ condition checks if such singular value
28     // (by definition non-negative) is "numerically" different from 0.
29     std::cerr << "Rank of matrix X is greater than required rank k"
30               << std::endl;
31   }
32   // We know that k <= std::min(m,n) by the assert
33   A = svd.matrixU().leftCols(k) * s.asDiagonal();
34   B = svd.matrixV().leftCols(k);
35   // END
36   return std::make_pair(A, B);
37 }
```

Get it on GitLab (lowrankrep.hpp).

To understand the test for $\mathrm{rank}(\mathbf{X}) > k$ please revisit [Lecture $\rightarrow$ Ex. 3.4.2.3], which explains how to check, if a singular value is “relatively close to zero”.

SOLUTION of (3-8.c):

We insert both QR-factorizations:

$$\mathbf{A}\mathbf{B}^\top = \mathbf{Q}_A\mathbf{R}_A\mathbf{R}_B^\top\mathbf{Q}_B^\top = \mathbf{Q}_A\mathbf{S}\mathbf{Q}_B \quad \text{with} \quad \mathbf{S} := \mathbf{R}_A\mathbf{R}_B^\top \in \mathbb{R}^{k,k} . \tag{3.8.3}$$

Next, we rely on the singular value decomposition of the (small) square matrix $\mathbf{S}$:

$$\mathbf{S} = \tilde{\mathbf{U}}\mathbf{\Sigma}\tilde{\mathbf{V}}^\top \quad , \quad \tilde{\mathbf{U}}, \mathbf{\Sigma}, \tilde{\mathbf{V}} \in \mathbb{R}^{k,k} , \quad \tilde{\mathbf{U}}, \tilde{\mathbf{V}} \text{ orthogonal} , \quad \mathbf{\Sigma} \text{ diagonal} . \tag{3.8.4}$$

Then, insert (3.8.4) into (3.8.3):

$$\mathbf{A}\mathbf{B}^\top = \mathbf{Q}_A \underbrace{\mathbf{R}_A\mathbf{R}_B^\top}_{=\tilde{\mathbf{U}}\mathbf{\Sigma}\tilde{\mathbf{V}}^\top} \mathbf{Q}_B^\top = (\underbrace{\mathbf{Q}_A\tilde{\mathbf{U}}}_{=:\mathbf{U}})\mathbf{\Sigma}(\underbrace{\mathbf{Q}_B\tilde{\mathbf{V}}}_{=:\mathbf{V}})^\top . \tag{3.8.5}$$

Note that $\mathbf{U} \in \mathbb{R}^{m,k}$, $\mathbf{V} \in \mathbb{R}^{m,k}$, and

$$\mathbf{U}^\top\mathbf{U} = \left(\mathbf{Q}_A\tilde{\mathbf{U}}\right)^\top\mathbf{Q}_A\tilde{\mathbf{U}} = \tilde{\mathbf{U}}^\top \underbrace{\mathbf{Q}_A^\top\mathbf{Q}_A}_{=\mathbf{I}} \tilde{\mathbf{U}} = \tilde{\mathbf{U}}^\top\tilde{\mathbf{U}} = \mathbf{I} ,$$

$$\mathbf{V}^\top\mathbf{V} = \left(\mathbf{Q}_B\tilde{\mathbf{V}}\right)^\top\mathbf{Q}_B\tilde{\mathbf{V}} = \tilde{\mathbf{V}}^\top \underbrace{\mathbf{Q}_B^\top\mathbf{Q}_B}_{=\mathbf{I}} \tilde{\mathbf{V}} = \tilde{\mathbf{V}}^\top\tilde{\mathbf{V}} = \mathbf{I} .$$

This shows that (3.8.5) really provides an economical SVD of $\mathbf{A}\mathbf{B}^\top$.

HINT 1 for (3-8.d): If you directly compute an SVD of $\mathbf{A}\mathbf{B}^\top$, you will always have to deal with dimensions m and n, regardless of $k \ll m, n$. As elaborated in Sub-problem (3-8.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 (3-8.d):

Of course, (3.8.5) is cheaper than the direct SVD of the $m \times n$ matrix $\mathbf{A}\mathbf{B}^\top$, 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 3.8.6: Solution of (3-8.d)

```
std::tuple<Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd> svd_AB(
    const Eigen::MatrixXd& A, const Eigen::MatrixXd& B) {
  assert(A.cols() == B.cols() &&
         "Matrices A and B should have the same column number");
  assert(A.cols() < A.rows() && A.cols() < B.rows() &&
         "Matrices A and B should have less columns than rows");
  std::size_t m = A.rows();
  std::size_t n = B.rows();
  std::size_t k = A.cols();
  Eigen::MatrixXd U, S, V;
  // TODO: (3-8.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
  Eigen::HouseholderQR<Eigen::MatrixXd> QRA = A.householderQr();
  Eigen::MatrixXd QA =
      QRA.householderQ() * Eigen::MatrixXd::Identity(m, std::min(m, k));
  Eigen::MatrixXd RA = Eigen::MatrixXd::Identity(std::min(m, k), m) *
                       QRA.matrixQR().triangularView<Eigen::Upper>();
  Eigen::HouseholderQR<Eigen::MatrixXd> QRB = B.householderQr();
  Eigen::MatrixXd QB =
      QRB.householderQ() * Eigen::MatrixXd::Identity(n, std::min(n, k));
  Eigen::MatrixXd RB = Eigen::MatrixXd::Identity(std::min(n, k), n) *
                       QRB.matrixQR().triangularView<Eigen::Upper>();

  // U,V: k x k
  Eigen::JacobiSVD<Eigen::MatrixXd> svd(
      RA * RB.transpose(), Eigen::ComputeFullU | Eigen::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!

  S = svd.singularValues().asDiagonal();
  U = svd.matrixU();
  V = svd.matrixV();
  // U: m x k; V: n x k
  U = QA * U;
  V = QB * V;
  // END
  return std::make_tuple(U, S, V);
}
```

Get it on GitLab (lowrankrep.hpp).

Remark. If the matrices $\mathbf{A}$ and $\mathbf{B}$ came from the decomposition of a matrix $\mathbf{X}$ as in the previous sub-

problems, 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 3.8.6. However, our code works for every choice of matrices $\mathbf{A}, \mathbf{B}$ with the same number of columns k.

SOLUTION of (3-8.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 $n-1, n-2, \ldots, n-k+1$): each is worth $O(nk^2)$ operations (see [Lecture → Exp. 3.3.3.39]).
2. $(\mathbf{R}_A, \mathbf{R}_B) \mapsto \mathbf{R}_A \mathbf{R}_B^\top$, i.e. a matrix multiplication $O(k^3)$, 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 $O(nk^2)$.

The total computational effort is linear in n.

SOLUTION of (3-8.f):

From $\mathrm{rank}(\mathbf{X}) = k$ 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, $\mathrm{rank}(\mathbf{X} + \mathbf{Y}) \leq 2k$.

SOLUTION of (3-8.g):

Solving this problem requires deeper understanding of matrix multiplication.

We can define $\mathbf{A} := [\mathbf{A}_X \ \mathbf{A}_Y] \in \mathbb{R}^{m,2k}$ and $\mathbf{B} := [\mathbf{B}_X \ \mathbf{B}_Y] \in \mathbb{R}^{n,2k}$ by column-wise concatenation of matrices.

Hence, we have that

$$\mathbf{A}\mathbf{B}^\top = [\mathbf{A}_X \ \mathbf{A}_Y] \begin{bmatrix} \mathbf{B}_X^\top \\ \mathbf{B}_Y^\top \end{bmatrix} = \mathbf{A}_X\mathbf{B}_X^\top + \mathbf{A}_Y\mathbf{B}_Y^\top = \mathbf{X} + \mathbf{Y} \, .$$

HINT 1 for (3-8.h): $\|\cdot\|_F$ stands for the Frobenius norm of a matrix (see [Lecture → Def. 3.4.4.17]). Recall [Lecture → Thm. 3.4.4.19] and apply the function implemented in Sub-problem (3-8.d) to the product representation of $\mathbf{X} + \mathbf{Y}$ derived in Sub-problem (3-8.g). ⌟

SOLUTION of (3-8.h):

First we appeal to the result of Sub-problem (3-8.g) to write $\mathbf{X}+\mathbf{Y}$ as a product $\mathbf{A}\mathbf{B}^\top$ of "tall" matrices. To this product we apply `svd_AB()` from Sub-problem (3-8.d) to compute its economical SVD.

According to [Lecture → Thm. 3.4.4.19], 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 3.8.7: Solution of (3-8.h)

```
2  std::pair<Eigen::MatrixXd, Eigen::MatrixXd> rank_k_approx(
3      const Eigen::MatrixXd& Ax, const Eigen::MatrixXd& Ay,
4      const Eigen::MatrixXd& Bx, const Eigen::MatrixXd& By) {
5    assert(Ax.rows() == Ay.rows() && Ax.cols() == Ay.cols() &&
6           "Matrices Ax and Ay should have the same dimensions");
7    assert(Bx.rows() == By.rows() && Bx.cols() == By.cols() &&
8           "Matrices Bx and By should have the same dimensions");
9    assert(Ax.cols() == Bx.cols() &&
10          "Matrices Ax, Ay, Bx and By should have the same amount of columns");
11   assert(Ax.cols() * 2 < Ax.rows() && Bx.cols() * 2 < Bx.rows() &&
12          "2*k should not be bigger equal min(m, n)");
13
14   Eigen::MatrixXd Az, Bz;
15   // TODO: (3-8.h) Use the function svd_AB with the appropriate A,B and
16   // return the pair Az, Bz of matrices of rank k with Z = Az*Bz
17   // Hint: you can define pairs of objects using std::make_pair
18   // Hint: use std::get<i>(my_tuple) to extract the i-th element of a tuple
19   // START
20   Eigen::MatrixXd A(Ax.rows(), Ax.cols() + Ay.cols());
21   A << Ax, Ay;
22   Eigen::MatrixXd B(Bx.rows(), Bx.cols() + By.cols());
23   B << Bx, By;
24   // U: m x 2k; S: 2k x 2k; V: n x 2k
25   auto [U, S, V] = svd_AB(A, B);
26   std::size_t k = Ax.cols();
27   Az = U.leftCols(k) * S.topLeftCorner(k, k);
28   Bz = V.leftCols(k);
29   // END
30   return std::make_pair(Az, Bz);
31 }
```

Get it on GitLab (lowrankrep.hpp).

SOLUTION of (3-8.i):

`svd_AB()` has linear complexity in n and m, as derived in (3-8.e). The multiplication of the truncated matrices $\mathbf{U}$ and $\mathbf{S}$ requires $O(mk)$ operations (since $\mathbf{S}$ is diagonal). Hence, the total computational effort is $O(n+m)$ for $m, n \to \infty$.

SOLUTION of (3-9.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,

$$\mathbf{C} = \begin{bmatrix} \mathbf{z}^\top & \mathbf{0}^\top & \cdots & \mathbf{0}^\top \\ \mathbf{0}^\top & \mathbf{z}^\top & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{0} \\ \mathbf{0}^\top & \cdots & \mathbf{0}^\top & \mathbf{z}^\top \end{bmatrix} \in \mathbb{R}^{n,n^2} . \tag{3.9.3}$$

HINT 1 for (3-9.j): What special structure for $\mathbf{M}^*$, solution of (3.9.10), is implied by one of the saddle point equations? ⌟

SOLUTION of (3-9.j):

Plugging (3.9.14) into (3.9.15) we obtain:

$$-\frac{1}{2}\mathbf{m}\|\mathbf{z}\|_2^2 - \mathbf{g} = 0 \quad \Longrightarrow \quad \mathbf{m} = -2\frac{\mathbf{g}}{\|\mathbf{z}\|_2^2} \quad \overset{((3.9.14))}{\Longrightarrow} \quad \mathbf{M}^* = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2}\,. \tag{3.9.16}$$

The matrix $\mathbf{M}^*$ is the tensor product of two vectors, see [Lecture → Section 1.3.1].

SOLUTION of (3-9.b):

$\mathbf{C}$ has full rank, if $\mathbf{z}$ has at least one non-zero component ζ, because in this case we can certainly find n linearly independent columns, because $\mathbf{C}$ contains $\zeta\mathbf{I} \in \mathbb{R}^{n,n}$ as a sub-matrix.

Equivalently, an argument can be made based on the n rows:

$$x_1 \begin{bmatrix} \mathbf{z} \\ \mathbf{0} \\ \vdots \\ \vdots \\ \mathbf{0} \end{bmatrix} + x_2 \begin{bmatrix} \mathbf{0} \\ \mathbf{z} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} + \cdots + x_n \begin{bmatrix} \mathbf{0} \\ \vdots \\ \vdots \\ \mathbf{0} \\ \mathbf{z} \end{bmatrix} = \mathbf{0}$$

has a unique solution $x_1 = x_2 = \ldots = x_n = 0$, if $\mathbf{z} \neq \mathbf{0}$.

HINT 1 for (3-9.c): Generic **augmented normal equations** associated with (3.9.2) read

$$\mathbf{S}\begin{bmatrix}\mathbf{m}^*\\ \mathbf{p}\end{bmatrix} := \begin{bmatrix}\mathsf{Id}_{n^2} & \mathbf{C}^\top\\ \mathbf{C} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{m}^*\\ \mathbf{p}\end{bmatrix} = \begin{bmatrix}\mathbf{0}\\ \mathbf{g}\end{bmatrix} =: \mathbf{h}. \tag{3.9.4}$$

Then carry out a block Gaussian elimination [Lecture → Rem. 2.3.1.14] of the unknown $\mathbf{m}^*$. Which linear system for $\mathbf{p}$ emerges? ⌟

SOLUTION of (3-9.c):

By block Gaussian elimination of the unknown $\mathbf{m}^*$ from the augmented normal equations (3.9.4) we obtain

$$-\mathbf{C}\mathbf{C}^\top \mathbf{p} = \mathbf{g}\,. \tag{3.9.5}$$

By [Lecture $\rightarrow$ 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 (3-9.b).

SOLUTION of (3-9.d):

See code in Equation 3.9.6. As pointed out in the solution of Sub-problem (3-9.a) the matrix $\mathbf{C}$ is a Kronecker product.

C++ code 3.9.6: Solution of minimization problem

```
Eigen::MatrixXd min_frob(const Eigen::VectorXd &z, const Eigen::VectorXd &g) {
  assert(z.size() == g.size() && "Size mismatch!");
  const unsigned int n = g.size();
  Eigen::MatrixXd M_mat = Eigen::MatrixXd::Zero(n, n);
  // TODO: (3-9.d) solve the augmented normal equations
  // Hint:  Use Map to reshape a vector into a matrix
  // START
  // Build temporary matrix C.
  Eigen::MatrixXd C =
      kroneckerProduct(Eigen::MatrixXd::Identity(n, n), z.transpose());
  Eigen::VectorXd minus_p = (C * C.transpose()).ldlt().solve(g);
  Eigen::VectorXd M = C.transpose() * minus_p;
  M_mat = Eigen::Map<Eigen::MatrixXd>(M.data(), n, n).transpose();
  // END
  return M_mat;
}
```

Get it on GitLab (matrixfit.hpp).

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 more efficiently.

HINT 1 for (3-9.e): Heed the advice given in [Lecture → Rem. 1.5.3.15].

SOLUTION of (3-9.e):

See code in Code 3.9.8. Note that we test for the relative deviation of the two computed matrices with respect to the Frobeniusnorm, which is evaluated by EIGEN's `norm()` member function. If this test is passed the norm of the difference between $\mathbf{M}$ and $\mathbf{M}^*$ is zero almost up to precision and we conclude that $\mathbf{M} = \mathbf{M}^*$.

C++ code 3.9.8: Testing code

```
2  bool testMformula(unsigned int n) {
3    constexpr double tol = 1e-10;
4    // TODO: (3-9.e) compare the formula:  return true if the relative
5    // error in Frobenius norm is less than tol.
6    // Hint:  Define z using LinSpaced and g using replicate()
7    // START
8    Eigen::VectorXd z = Eigen::VectorXd::LinSpaced(n, 1, n);
9    Eigen::Vector2d v = {1, -1};
10   Eigen::VectorXd g = v.replicate(static_cast<int>(n) / 2, 1);
11
12   Eigen::MatrixXd Mstar = min_frob(z, g);
13
14   Eigen::MatrixXd M = g * z.transpose() / z.squaredNorm();  // M
15   // Safe test for realtive smallness of error
16   return ((Mstar - M).norm() < tol * M.norm());
17   // END
18   return false;
19 }
```

Get it on GitLab (matrixfit.hpp).

SOLUTION of (3-9.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 (3.9.4) is

$$\mathbf{p} = -\|\mathbf{z}\|_2^{-2}\mathbf{g} \quad , \quad \mathbf{m}^* = \mathbf{C}^\top \|\mathbf{z}\|_2^{-2}\mathbf{g} = \|\mathbf{z}\|_2^{-2}\mathbf{g} \otimes \mathbf{z} \in \mathbb{R}^{n^2} \ . \tag{3.9.9}$$

Then "un-vectorize" $\mathbf{m}^*$!

HINT 1 for (3-9.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 (3-9.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 $\mathbf{Mz} = \mathbf{g}$) 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 $\mathbf{Mz} = \mathbf{g}$, i.e. $\mathbf{Mz} - \mathbf{g}$. The functional L is therefore:

$$L(\mathbf{M}, \mathbf{m}) = \|\mathbf{M}\|_F^2 + \mathbf{m}^\top(\mathbf{Mz} - \mathbf{g}) \quad . \tag{3.9.11}$$

HINT 1 for (3-9.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. stacking 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 (3-9.h):

The simplest approach is by component-wise partial derivatives:

$$\frac{\partial}{\partial x_{ij}}\Phi(\mathbf{X}) = 2(\mathbf{X})_{ij} \quad \Rightarrow \quad \mathbf{grad}\,\Phi(\mathbf{X}) = 2\mathbf{X}\,. \tag{3.9.13}$$

HINT 1 for (3-9.i): What are the necessary and sufficient condition for minimizing (3.9.10)? Remember [Lecture → Eq. (3.6.1.5)] and set

$$\frac{\partial L}{\partial \mathbf{M}} = 0 \quad \text{and} \quad \frac{\partial L}{\partial \mathbf{m}} = 0 \, .$$

These are the saddle-point equations. ⌟

SOLUTION of (3-9.i):

Recall the formula for L from (3.9.11), Sub-problem (3-9.g):

$$L(\mathbf{M},\mathbf{m}) = \|\mathbf{M}\|_F^2 + \mathbf{m}^\top(\mathbf{M}\mathbf{z}-\mathbf{g}) = \Phi(\mathbf{M}) + \mathbf{m}^\top\mathbf{M}\mathbf{z} - \mathbf{g}^\top\mathbf{m}\ .$$

We can easily compute the partial derivatives of

$$\mathbf{M} \mapsto \varphi(\mathbf{M}) := \mathbf{m}^\top\mathbf{M}\mathbf{z} = \sum_{j,k=1}^{n} (\mathbf{m})_j m_{j,k} (\mathbf{z})_k\ , \quad \mathbf{M} = \left[m_{j,k}\right]_{j,k=1}^{n}\ ,$$

with respect to the matrix entries $m_{j,k}$ of $\mathbf{M}$:

$$\frac{\partial\varphi}{\partial m_{j,k}}(\mathbf{M}) = (\mathbf{m})_j (\mathbf{z})_k \quad \Rightarrow \quad D\varphi(\mathbf{M}) = \mathbf{m}\mathbf{z}^\top \in \mathbb{R}^{n,n}\ .$$

That derivative is the constant rank-1 matrix $\mathbf{m}\mathbf{z}^\top$ (tensor product of vectors).

Thus, using $\mathbf{grad}\,\Phi(\mathbf{X}) = 2\mathbf{X}$, the derivative of L w.r.t. $\mathbf{M}$ is:

$$\frac{\partial L}{\partial \mathbf{M}}(\mathbf{M},\mathbf{m}) = 2\mathbf{M} + \mathbf{m}\mathbf{z}^\top = 0\ . \tag{3.9.14}$$

Obviously, L is linear in $\mathbf{m}$. Thus, the derivative of L w.r.t. $\mathbf{m}$ is:

$$\frac{\partial L}{\partial \mathbf{m}}(\mathbf{M},\mathbf{m}) = \mathbf{M}\mathbf{z} - \mathbf{g} = 0\ . \tag{3.9.15}$$

SOLUTION of (4-1.a):

C++11-code 4.1.2: Rendering of images with different focus settings.

```
2  void save_image(double focus) {
3    // Create empty object
4    PGMObject q;
5
6    // TODO: (4-1.a) Read matrix of image generated
7    // by "set_focus" and save as an image in format ".pgm".
8    // START
9    // Set data using function "set_data"
10   // Data obtained from "set_focus"
11   q.set_data(set_focus(focus));
12
13   // Create and save file
14   std::stringstream ss;
15   ss << "./cx_out/image_focus=" << (int)focus << ".pgm";
16   std::ofstream file(ss.str());
17   file << q;
18   // END
19 }
```

Get it on GitLab (autofocus.hpp).

HINT 1 for (4-1.b): MATPLOTLIBCPP provides many facilities for rendering 2D data, for instance `matplotlibcpp::imshow()`, see this example code. You can select the underlying color map through the `cmap` argument or just use the default. ⌟

SOLUTION of (4-1.b):

C++ code 4.1.4: Plotting of spectrum of the images.

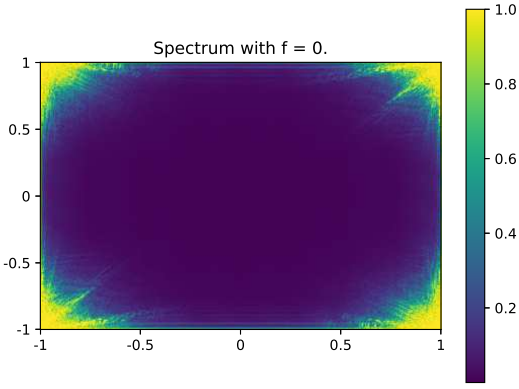
```cpp
void plot_freq(double focus) {
  constexpr unsigned int a = 0;
  constexpr unsigned int b = 8000;
  auto clamp = [](double x) { return x < a ? a : x > b ? b : x; };

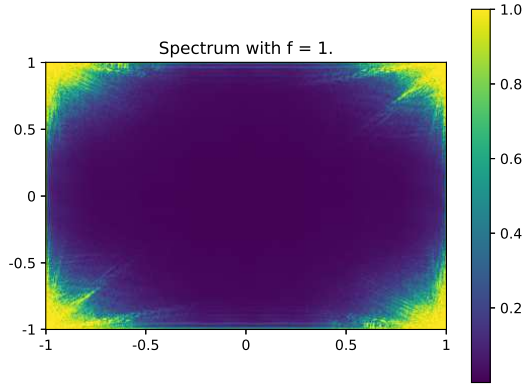
  // TODO: (4-1.b) compute D containing the spectrum of
     set_focus(focus).
  // "clamp" the data between 0 and 8000.
  // START
  Eigen::MatrixXd D = fft2r(set_focus(focus)).cwiseAbs().unaryExpr(clamp) / b;
  // END

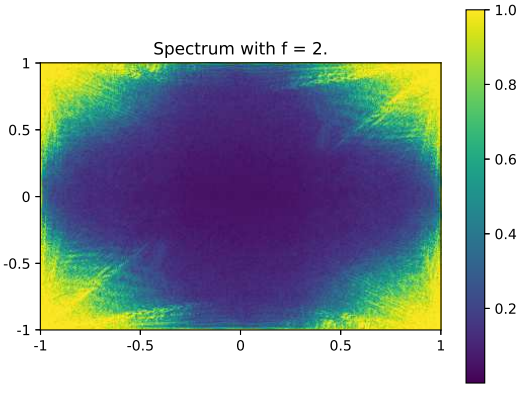
  // Plot values of X.
  // Axis labels
  std::vector<double> xticks(5);
  for (int i = 0; i < 5; ++i) {
    xticks[i] = i * (D.cols() - 1) / 4;
  }
  std::vector<double> yticks(5);
  for (int i = 0; i < 5; ++i) {
    yticks[i] = i * (D.rows() - 1) / 4;
  }
  std::vector<std::string> labels{"-1", "-0.5", "0", "0.5", "1"};

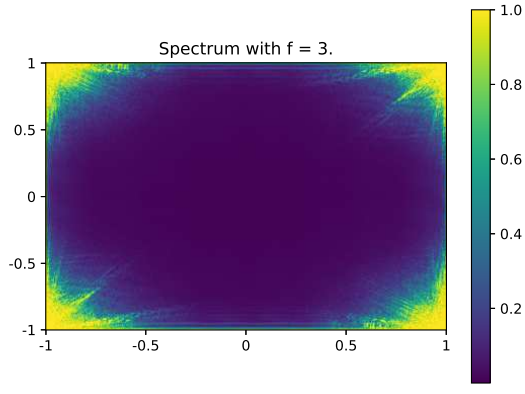
  plt::figure();
  plt::imshow(D, {{"cmap", "viridis"}, {"origin", "lower"}});
  plt::colorbar();
  plt::xticks(xticks, labels);
  plt::yticks(yticks, labels);
  std::stringstream ss;
  ss << "Spectrum with f = " << focus << ".";
  plt::title(ss.str().c_str());
  std::stringstream ss2;
  ss2 << "./cx_out/spectrum_focus=" << focus << ".png";
  plt::savefig(ss2.str().c_str());
}
```

Get it on GitLab (autofocus.hpp).


Spectrum with f = 0.
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1.0
0.8
0.6
0.4
0.2


Spectrum with f = 1.
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1.0
0.8
0.6
0.4
0.2


Spectrum with f = 2.
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1.0
0.8
0.6
0.4
0.2


Spectrum with f = 3.
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1.0
0.8
0.6
0.4
0.2

HINT 1 for (4-1.c): Look at [Lecture → § 4.2.3.6] and [Lecture → Fig. 127] to understand the relationship of Fourier coefficients and low and high frequency content of a signal. ⌟

SOLUTION of (4-1.c):

Centers of the plots correspond to the high frequencies, whereas corners correspond to the low frequencies.

Remark. From the plots we see that all the images are rich in “low-frequency content”, because the dominant 2D frequencies are concentrated in the corners of the matrix arising from 2D DFT. However, one can also observe that for some values of the autofocus parameter f also “high-frequency modes” in the center of the matrix become more important.

SOLUTION of (4-1.d):

The function `frequency_content()` evaluates $V(\mathbf{M})$ as defined in (4.1.1) for a given (real) matrix $\mathbf{M} \in \mathbb{R}^{n,m}$.

C++ code 4.1.5: Computation of high-frequency content.

```
double high_frequency_content(const Eigen::MatrixXd& M) {
  const unsigned int n = M.rows(), m = M.cols();
  double V = 0;
  // TODO: (4-1.d) compute V(M).
  // START
  for (unsigned int i = 0; i < M.rows(); ++i) {
    for (unsigned int j = 0; j < M.cols(); ++j) {
      double a = n / 2. - std::abs(i - n / 2.);
      double b = m / 2. - std::abs(j - m / 2.);
      V += (a * a + b * b) * M(i, j) * M(i, j);
    }
  }
  // END
  return V;
}
```

Get it on GitLab (autofocus.hpp).

C++ code 4.1.6: Plot creation for $V(\mathbf{B}(f))$.

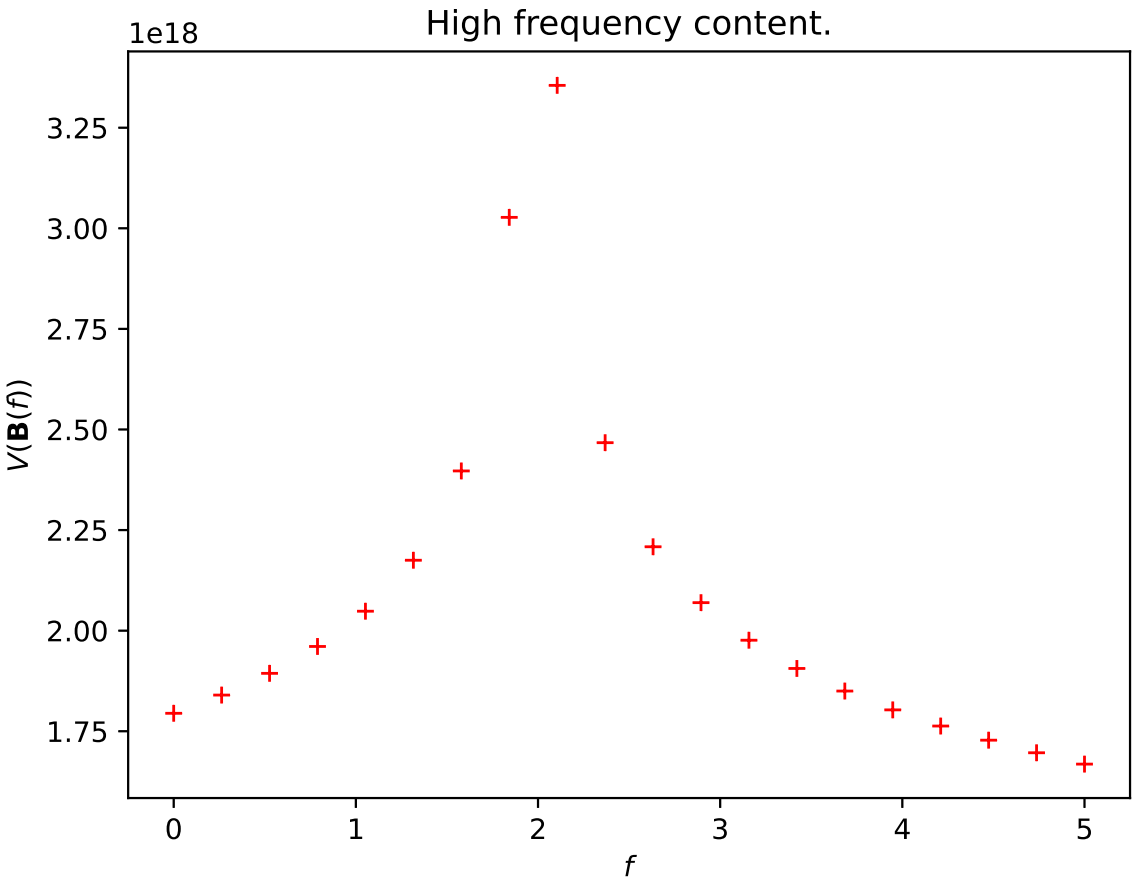
```
void plotV() {
  constexpr unsigned int N = 20;

  Eigen::VectorXd x(N), y(N);

  // TODO: (4-1.d) Plot V(B(f)).
  // START
  for (unsigned int i = 0; i < N; ++i) {
    double V = high_frequency_content(
        // Find 2D spectrum of matrix B(t)
        fft2r(
            // Evaluate set_focus at equidistant points
            set_focus(5. / (N - 1) * i))
            .cwiseAbs());
    x(i) = 5. / (N - 1) * i;
    y(i) = V;
  }

  // END
  plt::figure();
  plt::title("High frequency content.");
  plt::plot(x, y, "r+", {{"label", "V(mathbfB(f))"}});
  plt::xlabel("f");
  plt::ylabel("V(mathbfB(f))");
  plt::savefig("./cx_out/focus_plot.png");
}
```

Get it on GitLab (autofocus.hpp).



There is a pronounced peak of $f \to V(\mathbf{C}(f))$ for a particular value of f! For that value the blurred picture has maximal high-frequency content.

SOLUTION of (4-1.e):

The auto-focus determines that focus parameter should be set to $f \approx 2.0$.

C++ code 4.1.7: Finding best value of focus parameter.

```cpp
double autofocus() {
  // Minimum focus
  constexpr double min_focus = 0;
  // Maximum focus
  constexpr double max_focus = 5;
  // Min step
  constexpr double min_step = 0.05;
  // Starting guess
  double f0 = (max_focus - min_focus) / 2.;
  // Finite differences increment
  double df = min_step;
  // Starting step
  double step = max_focus / 2.;
  // Max number of iteration
  const unsigned int Niter =
      std::ceil(std::log2((max_focus - min_focus) / min_step));
  // TODO: (4-1.e) Use bisection method to find best focus.
  // START
  // Returns V(B(f))
  auto computeV = [](double focus) {
    return high_frequency_content(fft2r(set_focus(focus)).cwiseAbs());
  };

  // Bisection method
  for (unsigned int i = 0; i < Niter; ++i) {
    double dV = computeV(f0 + df) - computeV(f0 - df);

    step = step / 2.;
    f0 = f0 + (dV > 0 ? 1 : -1) * step;
  }
  // END
  return f0;
}
```

Get it on GitLab (autofocus.hpp).

SOLUTION of (4-2.a):

Of course, the coefficients of the trigonometric polynomial p are the unknown parameters to be estimates. Those will be collected in the vector $\mathbf{x}$:

$$\mathbf{x} := [c_0, \ldots, c_m] \in \mathbb{R}^{m+1} .$$

We have

$$p(\varphi_j) = p\left(2\pi\frac{j}{n}\right) = \sum_{k=0}^{m} c_k \cos\left(2\pi k\frac{j}{n}\right) = \sum_{k=0}^{m} c_k \cos\left(\frac{2\pi}{n}kj\right) , \quad j = 0, \ldots, n-1 .$$

Hence,

$$\sum_{j=0}^{n-1} \left|p(\varphi_j) - d_j\right|^2 = \left\| \begin{bmatrix} \sum_{k=0}^{m} c_k \cos\left(\frac{2\pi}{n}k \cdot 0\right) - d_0 \\ \sum_{k=0}^{m} c_k \cos\left(\frac{2\pi}{n}k \cdot 1\right) - d_1 \\ \vdots \\ \sum_{k=0}^{m} c_k \cos\left(\frac{2\pi}{n}k(n-1)\right) - d_{n-1} \end{bmatrix} \right\|_2^2 \stackrel{!}{=} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 .$$

Recalling the definition of the Euclidean norm and the rules of matrix×vector multiplication, we find

$$\mathbf{A} = \begin{bmatrix} \cos(0) & \cos(0) & \cos(0) & \ldots & \cos(0) \\ \cos(0) & \cos(\frac{2\pi}{n}) & \cos(\frac{4\pi}{n}) & \ldots & \cos(\frac{2\pi m}{n}) \\ \cos(0) & \cos(\frac{4\pi}{n}) & \cos(\frac{8\pi}{n}) & \ldots & \cos(\frac{4\pi m}{n}) \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \cos(0) & \cos(\frac{2\pi(n-1)}{n}) & \cos(\frac{4\pi(n-1)}{n}) & \ldots & \cos(\frac{2\pi(n-1)m}{n}) \end{bmatrix} \in \mathbb{R}^{n,m+1}, \ \mathbf{b} = \begin{bmatrix} d_0 \\ d_1 \\ \ldots \\ d_{n-1} \end{bmatrix} \in \mathbb{R}^n .$$

More compactly, letting cos act on a matrix componentwise, we can write

$$\mathbf{A} = \left[\cos\left(\frac{2\pi}{n}, \ell k\right)\right]_{\substack{\ell=0,\ldots,n-1 \\ k=0,\ldots,m}} = \cos\left(\frac{2\pi}{n}\begin{bmatrix} 0 \\ 1 \\ \vdots \\ n-1 \end{bmatrix}\begin{bmatrix} 0 & 1 & \ldots & m \end{bmatrix}\right) \in \mathbb{R}^{n,m+1} . \tag{4.2.4}$$

HINT 1 for (4-2.b): Use Euler's formula

$$\cos(2\pi x) = \tfrac{1}{2}(e^{2\pi i x} + e^{-2\pi i x}) \,,$$

together with the geometric sum formula [Lecture → Eq. (4.2.1.9)]

$$\sum_{\ell=0}^{n-1} q^{\ell} = \frac{1-q^{n}}{1-q} \,, \tag{4.2.5}$$

applied with $q := \exp(2\pi\imath\frac{\nu}{n})$, $n \in \mathbb{Z}$, which yields, *cf.* [Lecture → § 4.2.1.6],

$$\sum_{\ell=0}^{n-1} \exp(\tfrac{2\pi\imath}{n}\ell\nu) = \begin{cases} n & \text{, if } \nu \in n\mathbb{Z} \,, \\ 0 & \text{else.} \end{cases}$$

⌟

SOLUTION of (4-2.b):

As we have learned in [Lecture → Section 3.1.2], [Lecture → Thm. 3.1.2.1], the normal equations for a linear least-squares problem $\|\mathbf{A}\mathbf{x}-\mathbf{b}\| \to \min$ are

$$\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b}\,. \tag{4.2.6}$$

The entries of $\mathbf{A}^\top\mathbf{A} \in \mathbb{R}^{m+1,m+1}$ are inner products of columns of $\mathbf{A}$ as given in (4.2.4): for $j,k \in \{0,\ldots,m$

$$\begin{aligned}
(\mathbf{A}^\top\mathbf{A})_{j,k} &= \sum_{\ell=0}^{n-1} \cos\left(\frac{2\pi j\ell}{n}\right)\cos\left(\frac{2\pi k\ell}{n}\right)\\
&= \frac{1}{4}\sum_{\ell=0}^{n-1}\left(e^{\frac{2\pi j\ell i}{n}}+e^{-\frac{2\pi j\ell i}{n}}\right)\left(e^{\frac{2\pi k\ell i}{n}}+e^{-\frac{2\pi k\ell i}{n}}\right)\\
&= \frac{1}{4}\sum_{\ell=0}^{n-1}\left(e^{\frac{2\pi (j+k)\ell i}{n}}+e^{\frac{2\pi (j-k)\ell i}{n}}+e^{\frac{2\pi (k-j)\ell i}{n}}+e^{-\frac{2\pi (j+k)\ell i}{n}}\right) = \begin{cases} n, & j=k=0,\\ n/2, & j=k\neq 0,\\ 0, & j\neq k\,. \end{cases}
\end{aligned}$$

Note that the assumption $2m < n$ rules out $j+k=n$. As a consequence we find that $\mathbf{A}^\top\mathbf{A}$ is *diagonal*!

$$\mathbf{A}^\top\mathbf{A} = \begin{bmatrix} n & & & \\ & n/2 & & \\ & & \ddots & \\ & & & n/2 \end{bmatrix} \in \mathbb{R}^{m+1,m+1}\,. \tag{4.2.7}$$

SOLUTION of (4-2.c):

The formula (4.2.4) can be implemented directly in EIGEN, which supplies componentwise application of cos to the entries of a matrix EIGEN documentation. The test of relative closeness can rely on the Frobeniusnorm provided by EIGEN's `norm()` method, but any other matrix norm can be used as well. As tolerance we may set $mn \cdot \texttt{TOL}$ with $\texttt{TOL} \ll 1$, for instance $\texttt{TOL} \approx \texttt{EPS}$ or slightly larger.

C++11-code 4.2.8: Implementation of `testNormEqMatrix()`.

```
2  bool testNormEqMatrix(unsigned int n, unsigned int m) {
3    // TODO: (4-2.c) Test if the two definitions, given in (4.2.4) and
4    // (4.2.7), of A^T*A are equal.
5
6    // START
7    Eigen::MatrixXd A(n, m + 1);
8
9    // Initializing the vectors described in (4.2.4)
10   Eigen::VectorXd n_linear = Eigen::VectorXd::LinSpaced(n, 0, n - 1);
11   Eigen::VectorXd m_linear = Eigen::VectorXd::LinSpaced(m + 1, 0, m);
12
13   // Building A as described in (4.2.4)
14   A = 2.0 * M_PI / n * n_linear * m_linear.transpose();
15   A = A.array().cos().matrix();
16
17   // Using the knowledge gained in (4-2.b) to initialize A^T*A directly
       as a
18   // diagonal matrix.
19   Eigen::MatrixXd ATA = Eigen::MatrixXd::Zero(m + 1, m + 1);
20   ATA.diagonal() = Eigen::VectorXd::Constant(m + 1, n / 2.0);
21   ATA(0, 0) = n;
22
23   // Comparing the two approaches to calculate A^T*A.
24   // Test for relative smallness of the Frobenius norm of the difference
       of the
25   // matrices
26   if ((ATA - A.transpose() * A).norm() < 1e-10 * m * n) {
27     return true;
28   }
29   // END
30
31   return false;
32 }
```

Get it on GitLab (fftlsq.hpp).

HINT 1 for (4-2.d): Note the identity

$$\sum_{k=0}^{n-1} e^{\frac{2\pi i}{n} jk} d_k = \overline{\sum_{k=0}^{n-1} e^{-\frac{2\pi i}{n} jk} d_k}, \quad j = 0, \ldots, n-1 \,, \tag{4.2.9}$$

where the over-line denotes the complex conjugation. ⌟

SOLUTION of (4-2.d):

Of course, we solve the normal equations $\mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{a}^\top\mathbf{b}$ to obtain the coefficients of p^*. From (4.2.7) we already know that $\mathbf{A}^\top\mathbf{A}$ is diagonal. So there is no linear system to be solved, just a scaling of the right-hand side vector to perform. Yet, we have to compute that right-hand side vector $\mathbf{A}^\top\mathbf{b}$. If we did it using straightforward matrix×vector multiplication, we would incur a computational cost of $O(mn)$ for $m, n \to \infty$.

To understand how $\mathbf{A}^\top\mathbf{b}$ can be formed more efficiently, we compute the j-th entry of the right hand-side vector of the normal equations: $\mathbf{A}^\top\mathbf{b}$ is:

$$\begin{aligned}\left(\mathbf{A}^\top\mathbf{b}\right)_j &= \sum_{k=0}^{n-1} \cos\left(\frac{2\pi jk}{n}\right) d_k = \frac{1}{2}\sum_{k=0}^{n-1}\left(e^{-\frac{2\pi i}{n}jk} + e^{\frac{2\pi i}{n}jk}\right) d_k = \frac{1}{2}\sum_{k=0}^{n-1} e^{-\frac{2\pi i}{n}jk} d_k + \frac{1}{2}\sum_{k=0}^{n-1} e^{\frac{2\pi i}{n}jk} d_k \\ &= \frac{1}{2}\left((\mathbf{F}_n\mathbf{b})_j + \overline{(\mathbf{F}_n\mathbf{b})_j}\right) = \left(\mathrm{Re}(\mathbf{F}_n\mathbf{b})\right)_j ,\end{aligned}$$

where $\mathbf{F}_n$ stands for the Fourier matrix, see [Lecture → Eq. (4.2.1.13)]. This tells us that $\mathbf{A}^\top\mathbf{b} \in \mathbb{R}^n$ is the real parts of a section of the discrete Fourier transform [Lecture → Section 4.2] of the data vector $\mathbf{b} = [d_0, \ldots, d_{n-1}]^\top$. As we have learned in [Lecture → Section 4.3] that DFT is available with an effort of $O(n \log n)$ for $n \to \infty$.

Hence, if $m > O(\log n)$, this is a more efficient way to compute $\mathbf{A}^\top\mathbf{b}$ than direct multiplication.

C++11-code 4.2.10: Implementation of `find_c`.

```
2  Eigen::VectorXd find_c(const Eigen::VectorXd &d, unsigned int m) {
3    unsigned int n = d.size();
4
5    // We will use a real to complex, discrete Fourier transform.
6    Eigen::FFT<double> fft;
7    Eigen::VectorXd rhs = Eigen::VectorXd::Zero(m + 1);
8
9    // TODO: (4-2.d) find the coefficients
10   // START
11
12   // Computing DFT of d
13   Eigen::VectorXcd fourier = fft.fwd(d);
14
15   // Gathering what is important for the rhs.
16   rhs = fourier.real().head(m + 1);
17
18   // Solving normal equation by inverting diagonal matrix.
19   rhs /= n;
20   rhs.tail(m) *= 2.0;
21   // END
22
23   return rhs;
24 }
```

Get it on GitLab (fftlsq.hpp).

HINT 1 for (4-2.e): Actually MATPLOTLIBCPP's `plot()` function can merely draw polygons. Therefore you have to give it many points on the ellipse and the approximating curves: use, for instance,

$$\begin{bmatrix} d(\frac{2\pi}{N}j)\cos(\frac{2\pi}{N}j) \\ d(\frac{2\pi}{N}j)\sin(\frac{2\pi}{N}j) \end{bmatrix}, \quad j = 0, \ldots, N-1, \quad N = 10^4 .$$

SOLUTION of (4-2.e):

C++11-code 4.2.12: Implementation of `fitEllipse()`.

```
void fitEllipse() {
  unsigned int n = 10;
  unsigned int m = 3;

  // Test points
  constexpr unsigned int npoints = 10;
  Eigen::VectorXd d(npoints);
  d << 0.987214, 1.03579, 0.997689, 0.917471, 1.00474, 0.92209, 1.03517,
      1.08863, 0.904992, 0.956089;

  plt::figure();

  // TODO: (4-2.e) Tabulate the coefficients of find_c for all m = 1,2,3
  // plot the ellipse and also the curves of the trigonimetric
     polynomials
  // START
  constexpr double c = 0.8;
  auto d_func =
      [](const double phi) {  // no need to capture c because it is
         constexpr
        assert(0. <= phi && phi <= 2. * M_PI);
        return 1. / std::sqrt(1. - std::pow(c * std::cos(phi), 2));
      };

  // Sample distance data, we reuse d from above
  n = 100;
  d.resize(n);
  for (std::size_t i = 0; i < n; ++i) {
    d(i) = d_func((2. * M_PI * i) / n);
  }

  plt::title("Orbit of planet");
  plt::xlim(-2, 2);
  plt::ylim(-2, 2);

  // Plot the ellipse given by (5.2.10) using 10000 points
  constexpr unsigned int N = 10000;
  Eigen::VectorXd x_ellipse(N), y_ellipse(N);
  double temp_d;
  for (std::size_t i = 0; i < N; ++i) {
    temp_d = d_func((2.0 * M_PI * i) / N);
    x_ellipse(i) = temp_d * std::cos((2.0 * M_PI * i) / N);
    y_ellipse(i) = temp_d * std::sin((2.0 * M_PI * i) / N);
  }
  plt::plot(x_ellipse, y_ellipse, "k", {{"label", "exact orbit"}});

  // Calculate, tabulate and plot the coefficients in one go
  x_ellipse.resize(n);
  y_ellipse.resize(n);

  // Create vector of colors for plots
  std::vector<std::string> colors = {"m", "y--", "r"};
  std::cout << "degree of polynomial | coefficients" << std::endl;
```

```
  for (m = 1; m <= 3; ++m) {
    Eigen::VectorXd coefficients = find_c(d, m);

    std::cout << std::setw(20) << m << " | " << std::setw(5)
              << coefficients.transpose() << std::endl;

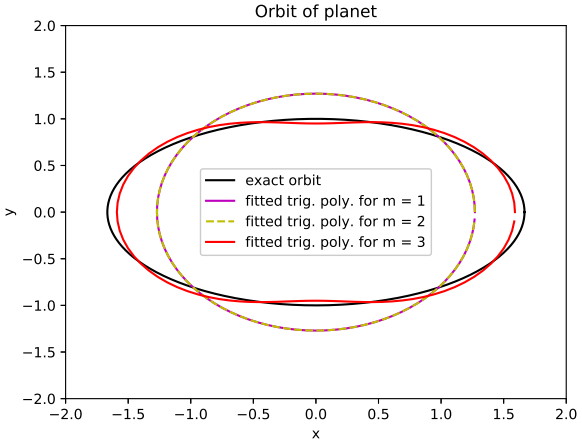
    Eigen::VectorXd e = eval_p(coefficients, n);
    for (std::size_t i = 0; i < n; ++i) {
      x_ellipse(i) = e(i) * std::cos((2 * M_PI * i) / n);
      y_ellipse(i) = e(i) * std::sin((2 * M_PI * i) / n);
    }
    const std::string label = "fitted trig. poly. for m = " + std::to_string(m);
    plt::plot(x_ellipse, y_ellipse, colors[m - 1], {{"label", label}});
  }

  plt::xlabel("x");
  plt::ylabel("y");
  plt::legend();

  // END
  plt::savefig("cx_out/orbit.png");
}
```

Get it on GitLab (fftlsq.hpp).

Picture produced by the function ▷



These are the coefficients of the fitted trigonometric polynomials:

m	$[c_0, \ldots, c_m]^\top$
0	$[1.27025]^\top$
1	$[1.27025\ 3.60268 \cdot 10^{-17}]^\top$
2	$[1.27025\ 3.60268 \cdot 10^{-17}\ 0.320124]^\top$

We see, as expected, a better fit for largers degree m. The fact $c_1 = 0$ is due to the symmetry of the ellipse.

HINT 1 for (4-3.a): Consult [Lecture → § 4.1.3.6].

SOLUTION of (4-3.a):

The following code provides a straightforward double-loop implementation of the convolution of two vectors.

C++ code 4.3.3: Solution of (4-3.a)

```cpp
Eigen::VectorXd polyMult_naive(const Eigen::VectorXd &u,
                               const Eigen::VectorXd &v) {
  // Fetch degrees of input polynomials
  const int degu = u.size() - 1;
  const int degv = v.size() - 1;

  // Object for product polynomial p = uv
  const int degp = degu + degv;

  Eigen::VectorXd uv(degp + 1);

  // TODO: (4-3.a) Multiply polynomials u and v naively.
  // START
  for (int i = 0; i <= degp; ++i) {
    const int fst = std::max(0, i - degv);
    const int lst = std::min(degu, i);
    for (int j = fst; j <= lst; ++j) {
      uv(i) += u(j) * v(i - j);
    }
  }
  // END

  return uv;
}
```

Get it on GitLab (polydiv.hpp).

Since a double loop is used, the complexity is $\mathcal{O}(nm)$.

HINT 1 for (4-3.b): As elaborated in [Lecture → § 4.1.3.6], the coefficient vector of $p = uv$ is the discrete convolution [Lecture → Def. 4.1.3.3] of the coefficient vectors of u and v. [Lecture → Code 4.2.2.5] demonstrates the efficient FTT-based discrete convolution of two vectors of the *same size*.

⌟

SOLUTION of (4-3.b):

C++ code 4.3.4: Solution of (4-3.b)

```
Eigen::VectorXd polyMult_fast(const Eigen::VectorXd &u,
                              const Eigen::VectorXd &v) {
  // Initialization
  const unsigned int m = u.size();
  const unsigned int n = v.size();

  Eigen::VectorXd u_tmp(m + n - 1);
  u_tmp.head(m) = u;
  u_tmp.tail(n - 1).setZero();

  Eigen::VectorXd v_tmp(m + n - 1);
  v_tmp.head(n) = v;
  v_tmp.tail(m - 1).setZero();

  Eigen::VectorXd uv;

  // TODO: (4-3.b) Multiply polynomials u and v efficiently.
  // START
  Eigen::FFT<double> fft;
  Eigen::VectorXcd u_tmp_ = fft.fwd(u_tmp);
  Eigen::VectorXcd v_tmp_ = fft.fwd(v_tmp);
  Eigen::VectorXcd tmp = u_tmp_.cwiseProduct(v_tmp_);
  uv = fft.inv(tmp).real();
  // END

  return uv;
}
```

Get it on GitLab (polydiv.hpp).

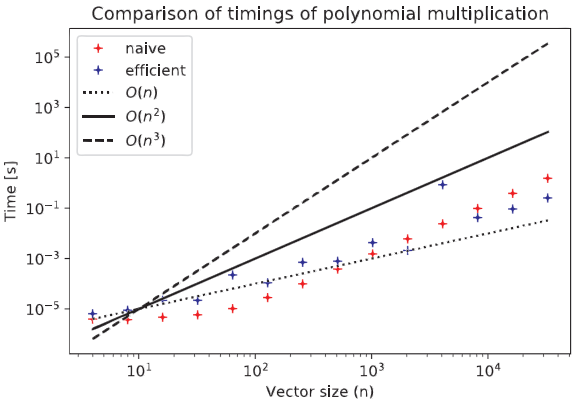
Compare this code with [Lecture → Code 4.2.2.5], which implements the discrete convolution of two vectors. The EIGEN method `conservativeResizeLike()` is used for zero padding here.

HINT 1 for (4-3.c): The asymptotic complexity of `polyMult_fast` is clearly inherited by that of a DFT. ⌟

SOLUTION of (4-3.c):

Assuming `Eigen::VectorXd u` and `Eigen::VectorXd v` represent polynomials of the degrees m and n, respectively, the complexity of the algorithm is $\mathcal{O}((n+m)\log(n+m))$.

The greater efficiency of the FTT-based implementation is noticeable only $m, n \geq 10^4$. The runtimes reported in the plot beside were measured for $m = n$ and using EIGEN's built-in FTT implementation.



HINT 1 for (4-3.d): When computing a DFT of $\mathbf{u} \in \mathbb{C}^n$, $\mathbf{c} := \mathrm{DFT}_n \mathbf{u} = \mathbf{F}_n \mathbf{u}$, we evaluate the following expressions, (see [Lecture → Def. 4.2.1.18]:

$$c_k = \sum_{j=0}^{n-1} y_j \, \omega_n^{kj} \;, \quad k = 0, \ldots, n-1 \;, \tag{4.3.5}$$

where ω_n is the n-th root of unity $\exp(-\frac{2\pi i}{n})$. For which arguments z do you have to evaluate the polynomial

$$p(z) := \sum_{j=0}^{n-1} (\mathbf{u})_j z^j \;, \quad z \in \mathbb{C} \;,$$

to obtain the complex numbers c_j? ⌟

SOLUTION of (4-3.d):

(4.3.5) implies that each entry of the DFT vector $\mathbf{c} = \mathbf{F}_n\mathbf{u}$ is the evaluation of the polynomial with coefficients $(\mathbf{u})_j$ in the roots of unity $\omega_n^k = \exp(-\frac{2\pi ki}{n})$. Thus the DFT provides an alternative "root-of-unity representation" of a polynomial through "equidistant samples on the unit circle". Notice that no information is lost when switching between these representation, as a polynomial with degree $n-1$ is uniquely determined by its values in n points.

Remark. Hence, the polynomial multiplication carried out via DFT in (4-3.b) means that you first transform the polynomials u and v from the "coefficient" representation to a "unit root evaluation" representation. In this representation, each entry k of n-dimensional vectors $\mathcal{F}(\mathbf{u})$ and $\mathcal{F}(\mathbf{v})$ is the evaluation of polynomials u and v in one root of unity ω_n^k. In the unit root representation, multiplication of polynomials is straightforward. Indeed, the component-wise product of the value vectors is the root-of-unity representation of the product polynomial uv. The inverse Fourier transform in the convolution theorem used in (4-3.b) finally transforms back the product polynomial in the coefficient representation, thus solving the polynomial multiplication problem.

HINT 1 for (4-3.e): If $p \in \mathcal{P}_N$, $u \in \mathcal{P}_m$, $p = uv$ for some $v \in \mathcal{P}_n$, $n := N - m$, then, according to [Lecture → § 4.1.3.6] and the considerations in [Lecture → Section 4.2.2],

$$\mathbf{p} = \widehat{\mathbf{u}} *_{N+1} \widehat{\mathbf{v}} \,, \tag{4.3.7}$$

where $\mathbf{p} \in \mathbb{C}^{N+1}$ is the coefficient vector for p, and $\widehat{\mathbf{v}}, \widehat{\mathbf{u}} \in \mathbb{C}^{N+1}$ are the coefficient vectors of u and v, extended to length $N+1$ by zero padding. As in [Lecture → Section 4.1.4], $*_{N+1}$ denotes the discrete periodic convolution of two vectors of length $N+1$. ⌟

HINT 2 for (4-3.e): Tests whether results of floating-point computations vanish have to be implemented in a numerically safe way according to [Lecture → Rem. 1.5.3.15] ⌟

SOLUTION of (4-3.e):

By the convolution theorem [Lecture → Thm. 4.2.2.2], (4.3.7) implies

$$\mathrm{DFT}_{N+1}\mathbf{p} = \mathrm{DFT}_{N+1}(\widehat{\mathbf{u}}) \odot \mathrm{DFT}_{N+1}(\widehat{\mathbf{v}}) \,,$$

where $\odot$ stands for componenwise multiplication of vectors. Hence, formally we get

$$\widehat{\mathbf{v}} = \mathrm{DFT}_{N+1}^{-1}(\mathrm{DFT}_{N+1}(\mathbf{p}) \oslash \mathrm{DFT}_{N+1}(\widehat{\mathbf{u}})) \,, \tag{4.3.8}$$

with componentwise division written as $\oslash$.

However, two complications arise and those have to be taken care of in a code.

(i) If $(\mathrm{DFT}_{N+1}\widehat{\mathbf{u}})_j = 0$ for some $j = 0, \ldots, N$ (C++ indexing), the componentwise division will not be possible.

(ii) The result vector $\widehat{\mathbf{v}}$ may not be a valid coefficient vector for a polynomial $v \in \mathcal{P}_n$ (notations as in the first hint), because $(\widehat{\mathbf{v}})_j = 0$ is not guaranteed for $j > n$.

To deal with (i), we have to check, whether $(\mathrm{DFT}_{N+1}\widehat{\mathbf{u}})_j = 0$ for some $j = 0, \ldots, N$. If also $(\mathrm{DFT}_{N+1}\widehat{\mathbf{p}})_j =$ then set the result of the division to zero, otherwise u does not divide p. Complication (ii) can be detected by a simple test: If any $(\widehat{\mathbf{v}})_j$, $j > n$, does not vanish we conclude that u does not divide p.

C++ code 4.3.9: Solution of (4-3.e)

```
2  Eigen::VectorXd polyDiv(const Eigen::VectorXd &uv, const Eigen::VectorXd &u) {
3    // Initialization
4    const unsigned int mn = uv.size();
5    const unsigned int m = u.size();
6    // need well behaved input
7    if (mn < m) {
8      std::cerr << "uv can't be divided by u\n";
9      return Eigen::VectorXd(0);
10   }
11   const unsigned int dim = mn;
12   const unsigned int n = mn - m + 1;
13
14   Eigen::VectorXd v;
15
16   // TODO: (4-3.e) Divide polynomials uv and u efficiently.
17   // START
18
19   // zero padding
20   Eigen::VectorXd uv_tmp = uv;
21   uv_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));
22   Eigen::VectorXd u_tmp = u;
23   u_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));
24
25   Eigen::FFT<double> fft;
26   Eigen::VectorXcd uv_tmp_ = fft.fwd(uv_tmp);
27   Eigen::VectorXcd u_tmp_ = fft.fwd(u_tmp);
28
29   // check divisibility:  case(i)
30   for (unsigned int i = 0; i < dim; ++i) {
31     if (abs(uv_tmp_(i)) < 1e-13) {
```

```
32        if (abs(u_tmp_(i)) < 1e-13) {
33          // make cwiseQuotient at i-th position equal 0
34          uv_tmp_(i) = 0.0;  // complex assignment (0., 0.)
35          u_tmp_(i) = 1.0;   // complex assignment (1., 0.)
36        } else {
37          std::cerr << "uv can't be divided by u\n";
38          return Eigen::VectorXd(0);
39        }
40      }
41    }
42
43    Eigen::VectorXcd tmp = uv_tmp_.cwiseQuotient(u_tmp_);
44
45    v = fft.inv(tmp).real();
46    // check divisibility:  case(ii)
47    for (unsigned int i = n; i < dim; ++i) {
48      if (abs(v(i)) > 1e-13) {
49        std::cerr << "uv can't be divided by u\n";
50        return Eigen::VectorXd(0);
51      }
52    }
53
54    // reshape v to a suitable size
55    v.conservativeResizeLike(Eigen::VectorXd::Zero(n));
56    // (mn-1) - (m-1) + 1
57    // END
58
59    return v;
60  }
```

Get it on GitLab (polydiv.hpp).

HINT 1 for (4-4.a): Refer to [Lecture → Section 4.5.2].

SOLUTION of (4-4.a):

We start from the Toeplitz matrix

$$\mathbf{T} = \begin{bmatrix} t_0 & t_1 & \cdots & t_{n-1} \\ t_{-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_1 \\ t_{-n+1} & \cdots & t_{-1} & t_0 \end{bmatrix}.$$

If we define

$$\mathbf{S} = \begin{bmatrix} 0 & t_{-n+1} & \cdots & t_{-1} \\ t_{n-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_{-n+1} \\ t_1 & \cdots & t_{n-1} & 0 \end{bmatrix}$$

then, the $2n \times 2n$-matrix

$$\mathbf{C} := \begin{bmatrix} \mathbf{T} & \mathbf{S} \\ \mathbf{S} & \mathbf{T} \end{bmatrix} = \begin{bmatrix} t_0 & t_1 & \cdots & t_{n-1} & 0 & t_{-n+1} & \cdots & t_{-1} \\ t_{-1} & \ddots & \ddots & \vdots & t_{n-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_1 & \vdots & \ddots & \ddots & t_{-n+1} \\ t_{-n+1} & \cdots & t_{-1} & t_0 & t_1 & \cdots & t_{n-1} & 0 \\ 0 & t_{-n+1} & \cdots & t_{-1} & t_0 & t_1 & \cdots & t_{n-1} \\ t_{n-1} & \ddots & \ddots & \vdots & t_{-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_{-n+1} & \vdots & \ddots & \ddots & t_1 \\ t_1 & \cdots & t_{n-1} & 0 & t_{-n+1} & \cdots & t_{-1} & t_0 \end{bmatrix}$$

will be circulant, because its columns are generated by repeated cyclic permutations.

SOLUTION of (4-4.b):

C++11-code 4.4.2: Solution of (4-4.b)

```
Eigen::MatrixXd toeplitz(const Eigen::VectorXd &c, const Eigen::VectorXd &r) {
  // Find size of Toeplitz matrix
  const unsigned int m = c.size();
  const unsigned int n = r.size();
  Eigen::MatrixXd T = Eigen::MatrixXd::Zero(m, n);

  // TODO: (4-4.b) Build Toeplitz matrix T
  // START
  for (unsigned int i = 0; i < n; ++i) {
    T.col(i).tail(m - i) = c.head(m - i);
  }
  for (unsigned int i = 0; i < m; ++i) {
    T.row(i).tail(n - i - 1) = r.segment(1, n - i - 1);
  }  // Do not reassign the diagonal!
  // END

  return T;
}
```

Get it on GitLab (toeplitz.hpp).

HINT 1 for (4-4.c): When analyzing function `toepmult()`, recall (4-4.a) and the properties of the product between a circulant matrix and a vector [Lecture → Rem. 4.1.4.15]. ⌟

SOLUTION of (4-4.c):

First, we consider function `toepmatmult()`. It takes two vectors `c` and `r` of the same size n as arguments. These vectors determine the first column and the first row of a Toeplitz matrix according to

$$T = \begin{bmatrix} (\mathbf{c})_1 & (\mathbf{r})_2 & \cdots & (\mathbf{r})_n \\ (\mathbf{c})_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & (\mathbf{r})_2 \\ (\mathbf{c})_n & \cdots & (\mathbf{c})_2 & (\mathbf{c})_1 \end{bmatrix}.$$

Thus, the first C++ code `toepmatmult()` multiplies this matrix with the vector x using the standard matrix-vector product.

Now we look `toepmult`. This function uses the function `pconvfft` given in [Lecture → Code 4.2.2.4], which implements the periodic convolution of two vectors of the same size, which is equivalent to multiplying one vector with the circulant matrix defined by the other, see [Lecture → Section 4.1.4].

C++11-code 4.4.5: Function `pconvfft()` for the periodic convolution of two vectors

```
2 Eigen::VectorXcd pconvfft(const Eigen::VectorXcd &u,
3                           const Eigen::VectorXcd &x) {
4   Eigen::FFT<double> fft;
5   return fft.inv((fft.fwd(u)).cwiseProduct(fft.fwd(x))).eval());
6 }
```

By using the result from Sub-problem (4-4.a), we extend the Toeplitz matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ to a circulant matrix $\mathbf{C} \in \mathbb{R}^{2n,2n}$. For circulant matrices the product with a vector is a periodic convolution, which can be computed via the discrete Fourier transform.

Since the matrix is now of a bigger size, we have to extend the vector $\mathbf{x}$. This is done by zero padding.

$$\mathbf{C}\begin{bmatrix}\mathbf{x}\\ \mathbf{0}\end{bmatrix} = \begin{bmatrix}\mathbf{T} & \mathbf{S}\\ \mathbf{S} & \mathbf{T}\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \mathbf{0}\end{bmatrix} = \begin{bmatrix}\mathbf{Tx}+\mathbf{S0}\\ \mathbf{Sx}+\mathbf{T0}\end{bmatrix} = \begin{bmatrix}\mathbf{Tx}\\ \mathbf{Sx}\end{bmatrix}$$

Using this result, we find the result of $\mathbf{Tx}$ in the first half of the vector that we get from periodic convolution.

SOLUTION of (4-4.d):

- `toepmatmult()` uses standard matrix-vector multiplication. Thus, the complexity is $\mathcal{O}(n^2)$ for $n \to \infty$.
- The complexity of `toepmult` is mainly determined by that of the discrete Fourier transforms. At the same time, since we extended the matrix and the vector, we have to compute three Fourier transforms of length $2n$. Thus, the asymptotic complexity is $\mathcal{O}(n\log(n))$ for $n \to \infty$.

HINT 1 for (4-4.e): Consider the following 2×2 block partitioning of the linear system of equations in the case of even system size n:

$$\mathbf{H}_n \mathbf{x} = \mathbf{y} \quad \Leftrightarrow \quad \begin{bmatrix} \mathbf{H}_m & 0 \\ \mathbf{T}_m & \mathbf{H}_m \end{bmatrix} \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \end{bmatrix} = \begin{bmatrix} \mathbf{y}^1 \\ \mathbf{y}^2 \end{bmatrix}, \quad \mathbf{T}_m \in \mathbb{R}^{m,m}, \quad m := n/2$$

SOLUTION of (4-4.e):

In `ttmatsolve()` we first build the following triangular Toeplitz matrix:

$$\mathbf{H}_n = \begin{bmatrix} h_1 & 0 & \cdots & 0 \\ h_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ h_n & \cdots & h_2 & h_1 \end{bmatrix}.$$

Then, we solve the triangular system. Thus, $\mathbf{x} = \mathbf{H}_n^{-1}\mathbf{y}$.

To understand the purpose of `ttrecsolve()`, we first take into account the hint. For $n = 2m$, we state the linear system of equations as a 2×2 block system:

$$\mathbf{H}_n\mathbf{x} = \mathbf{y} \quad \Longleftrightarrow \quad \begin{bmatrix} \mathbf{H}_m & \mathbf{0} \\ \mathbf{T}_m & \mathbf{H}_m \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix}.$$

The block matrices are defined as follows:

$$\mathbf{H}_m = \begin{bmatrix} h_1 & 0 & \cdots & 0 \\ h_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ h_{n/2} & \cdots & h_2 & h_1 \end{bmatrix}, \quad \mathbf{T}_m = \begin{bmatrix} h_{n/2+1} & h_{n/2} & \cdots & h_2 \\ h_{n/2+2} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & h_{n/2} \\ h_n & \cdots & h_{n/2+2} & h_{n/2+1} \end{bmatrix}.$$

The solution of the block-triangular system is given by

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{H}_m^{-1}\mathbf{y}_1 \\ \mathbf{x}_2 &= \mathbf{H}_m^{-1}(\mathbf{y}_2 - \mathbf{T}_m\mathbf{x}_1) \end{aligned}$$

Note that we only have to invert $\mathbf{H}_m$, which is also a triangular Toeplitz matrix. This suggests a *recursive approach*.

The function `ttrecsolve()` first finds out whether the vectors are of length 1 or not. If they are of length 1, the solution is trivial and is calculated by just dividing y by h. If the length is larger than 1, the system is divided in the two subsystems as shown above. The first subsystem is solved by applying the `ttrecsolve()` again. Then, the second subsystem is set up by calculating the right hand side of the second system. Note that $\mathbf{T}_m$ is a Toeplitz matrix. Thus, the multiplication is performed by using the `toepmult()` function from Sub-problem (4-4.c). Then, the second system is again solved by applying `ttrecsolve()`.

Since the system is always divided in half, this algorithm will not work if the size of the original system is not a power of 2.

SOLUTION of (4-4.f):

The algorithm is called “divide & conquer” because we divide the problem into two problems of the same kind, both of only half the original size, until we end up with a trivial one-dimensional problem. The solutions of the two sub-problems are then assembled into the solution of the full-size problem.

HINT 1 for (4-4.g): Refer to § 1.1.6 for detailed instructions on how to measure runtimes in a C++ code using **std**`::chrono`. Alternatively, you may use the tools provided in `timer.h`. ⌟

SOLUTION of (4-4.g):

C++11-code 4.4.8: Solution of (4-4.g)

```
void runtime_toeplitz() {
  // memory allocation for plot
  std::vector<double> vec_size;
  std::vector<double> elap_time_matmult, elap_time_mult, elap_time_ttmat,
      elap_time_ttrec;

  // header for the results to print out
  std::cout << std::setw(8) << "n" << std::setw(15) << "toepmatmult"
            << std::setw(15) << "toepmult" << std::setw(20) << "ttmatsolve"
            << std::setw(15) << "ttrecsolve" << std::endl;

  for (unsigned int l = 3; l <= 9; l += 1) {
    // vector size
    unsigned int n = std::pow(2, l);
    // save vector size n
    vec_size.push_back(n);

    // number of repetitions
    constexpr unsigned int repeats = 3;
    Timer tm_matmult, tm_mult, tm_ttmat, tm_ttrec;
    // TODO: (4-4.g) Perform a runtime comparison by repeating time
       computation
    // 'repeats' times
    // START
    // repeat test 'repeats' times
    for (unsigned int rr = 0; rr < repeats; ++rr) {
      // create runtime test using random vectors and given vector h
      Eigen::VectorXd h = Eigen::VectorXd::LinSpaced(n, 1, n).cwiseInverse();
      Eigen::VectorXd c = Eigen::VectorXd::Random(n);
      Eigen::VectorXd r = Eigen::VectorXd::Random(n);
      Eigen::VectorXd x = Eigen::VectorXd::Random(n);
      Eigen::VectorXd y = Eigen::VectorXd::Random(n);
      r(0) = c(0);

      // compute times for toepmatmult implementation
      tm_matmult.start();
      toepmatmult(c, r, x);
      tm_matmult.stop();

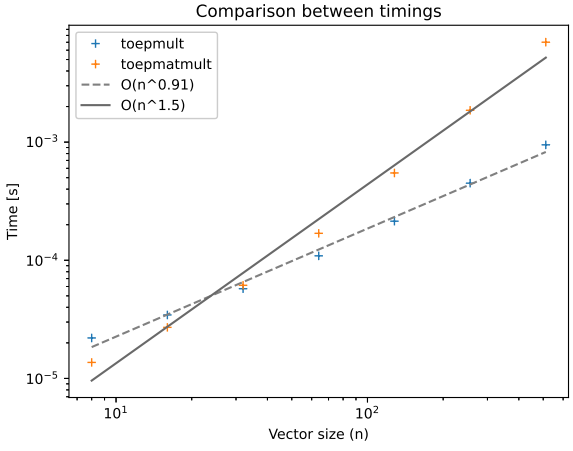
      // compute times for toepmult implementation
      tm_mult.start();
      toepmult(c, r, x);
      tm_mult.stop();

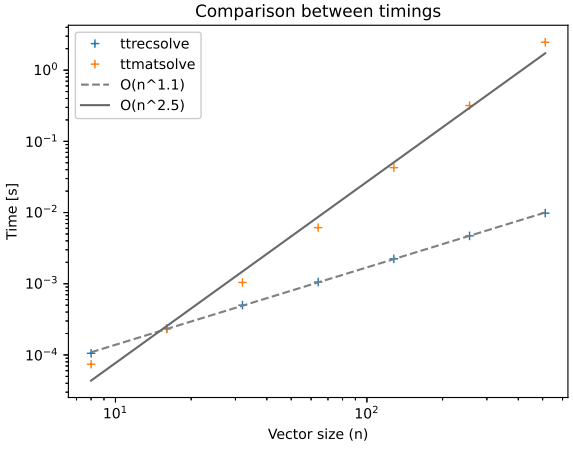
      // compute times for ttmatsolve implementation
      tm_ttmat.start();
      ttmatsolve(h, y);
      tm_ttmat.stop();
      // compute times
      tm_ttrec.start();
      ttrecsolve(h, y, l);
      tm_ttrec.stop();
```

```
53      }
54      // END
55
56      // print the results:  toepmult vs toepmatmult
57      std::cout << std::setw(8) << n << std::scientific << std::setprecision(3)
58                << std::setw(15) << tm_matmult.min() << std::setw(15)
59                << tm_mult.min() << std::setw(20) << tm_ttmat.min()
60                << std::setw(15) << tm_ttrec.min() << std::endl;
61
62      // save elapsed time for plot:  toepmatmult vs toepmult
63      elap_time_matmult.push_back(tm_matmult.min());
64      elap_time_mult.push_back(tm_mult.min());
65      // save elapsed time for plot:  ttmatsove vs ttrecsolve
66      elap_time_ttmat.push_back(tm_ttmat.min());
67      elap_time_ttrec.push_back(tm_ttrec.min());
68    }
69
70    // create plot
71    plot(vec_size, elap_time_mult, elap_time_matmult, "./cx_out/fig1.png",
72         "toepmult", "toepmatmult");
73    plot(vec_size, elap_time_ttrec, elap_time_ttmat, "./cx_out/fig2.png",
74         "ttrecsolve", "ttmatsolve");
75  }
```

Get it on GitLab (toeplitz.hpp).

This function produces the following pictures.





For large problem sizes n, `ttrecsolve()` is much faster than `ttmatsolve()`. The runtime of the latter seems to grow like $O(n^3)$, whereas the former seems to scale (almost) linearly in n.

SOLUTION of (4-4.h):

For `ttmatsolve()`, the complexity is determined by the complexity of the solver employed, which is generally $\mathcal{O}(n^3)$. Note that the implementation of Code 4.4.3 fails to communicate to EIGEN that the matrix is actually triangular. Thus EIGEN will blindly carry out full Gaussian elimination, *cf.* [Lecture → Rem. 2.5.0.5].

For `ttrecsolve()`, the number of operations in each step is given by the multiplication with the Toeplitz matrix and the vector subtraction in $\mathbf{x}_2 = \mathbf{H}_m^{-1}(\mathbf{y}_2 - \mathbf{T}_m\mathbf{x}_1)$. Thus, if we have a problem of size $\bar{n} = 2^{\bar{l}}$, we have

$$\approx \bar{n}\log(\bar{n}) + \bar{n} \approx 2^{\bar{l}}\bar{l} + 2^{\bar{l}} = 2^{\bar{l}}(\bar{l}+1)$$

operations in this step. The first term comes from the complexity found in (4-4.d).

Now, we have to add these terms for all the steps. If the original problem is of size $n = 2^l$, recursion enters into l levels, and on each level $\tilde{l} = 1,\ldots,l$ we have $2^{\tilde{l}}$ subproblems, each of size $2^{l-\tilde{l}}$.

Thus, the total number of operations is:

$$\begin{aligned}\sum_{\tilde{l}=1}^{l} 2^{l-\tilde{l}}(l-\tilde{l}+1)2^{\tilde{l}} &= 2^l \sum_{\tilde{l}=1}^{l}(l-\tilde{l}+1) \\ &= n\left(l^2 - \frac{l(l+1)}{2} + l\right) = n\frac{l(l+1)}{2} = O\left(n\log^2(n)\right) \quad \text{for} \quad n \to \infty\,.\end{aligned}$$

In the last step we exploited the equality $l = \log_2(n) = \frac{\log(n)}{\log(2)}$.

HINT 1 for (4-4.i): Just note that a lower triangular linear system of equations can be extended to a larger lower triangular LSE by adding additional rows/columns at the bottom/right of the coefficient matrix without affecting the first components of the solution vector. Thus, a triangular Toeplitz LSE can always be enlarged to a triangular Toeplitz LSE of a size that is a power of 2. ⌟

SOLUTION of (4-4.i):

The wrapper function has to provide a solution $\mathbf{H}_n\mathbf{x} = \mathbf{y}$ for the problem sizes n which are not powers of 2. If we have a problem of size m and $n = 2^l$ is the next larger power of 2, we can choose the following extension:

$$\mathbf{H}\mathbf{x}_{\text{long}} = \begin{bmatrix} \mathbf{H}_m & \mathbf{0} \\ \mathbf{X}_{n-m} & \mathbf{H}_{n-m} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \widetilde{\mathbf{x}} \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \widetilde{\mathbf{y}} \end{bmatrix} = \mathbf{y} \,.$$

Here the matrix $\mathbf{X} \in \mathbb{R}^{n-m,m}$ has to be chosen such that $\mathbf{H} \in \mathbb{R}^{n,n}$ becomes a lower triangular Toeplitz matrix. The solution of this linear system, which can efficiently be computed with `ttrecsolve()` is:

$$\begin{aligned} \mathbf{x} &= \mathbf{H}_m^{-1}\mathbf{y} \,, \\ \widetilde{\mathbf{x}} &= \mathbf{H}_{n-m}^{-1}(\widetilde{\mathbf{y}} - \mathbf{T}_{n-m}\mathbf{y}) \,, \end{aligned}$$

and $\mathbf{x}$ is what we are looking for.

Thus, we must find a suitable extension $\mathbf{H}_{n-m}$ and $\mathbf{X}$ such that $\mathbf{H}$ is a lower triangular Toeplitz matrix of the right size n. The solution of the original system of equations would then be available as the first m entries of $\mathbf{x}_{\text{long}}$.

$\mathbf{H}$ is therefore the $n \times n$ triangular Toeplitz matrix whose vector $\mathbf{h}$ of elements along the diagonal is determined by $\mathbf{h}_m$ extended with $n - m$ zeros.

For example, in the case $m = 3$ and given

$$\mathbf{H}_m = \begin{bmatrix} h_1 & 0 & 0 \\ h_2 & h_1 & 0 \\ h_3 & h_2 & h_1 \end{bmatrix} \,,$$

we have that $n = 4$ and

$$\mathbf{H} = \begin{bmatrix} h_1 & 0 & 0 & 0 \\ h_2 & h_1 & 0 & 0 \\ h_3 & h_2 & h_1 & 0 \\ 0 & h_3 & h_2 & h_1 \end{bmatrix} \,.$$

C++ code 4.4.9: Solution of (4-4.i)

```
2  Eigen::VectorXd ttsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y) {
3    assert(h.size() == y.size() && "h and y have different lengths!");
4    const unsigned int n = h.size();
5
6    Eigen::VectorXd x;
7
8    // TODO: (4-4.i) Wrap 'ttrecsolve' for any size n
9    // START
10   const unsigned int l = std::ceil(std::log2(n));
11   const unsigned int m = std::pow(2, l);
12
13   Eigen::VectorXd h_tmp = h;
14   h_tmp.conservativeResize(m);
15   h_tmp.bottomRows(m - n).setZero();
```

```
16
17      Eigen::VectorXd y_tmp = y;
18      y_tmp.conservativeResize(m);
19      y_tmp.bottomRows(m - n).setZero();
20
21      x = ttrecsolve(h_tmp, y_tmp, l);
22      x.conservativeResize(n);
23      // END
24
25      return x;
26  }
```

Get it on GitLab (toeplitz.hpp).

SOLUTION of (4-5.a):

We can rewrite the discrete convolution of two vectors [Lecture → Eq. (4.1.3.4)] as (C++ indexing used)

$$(\mathbf{h} * \mathbf{x})_k = \sum_{j=\max\{0,k-n+1\}}^{\min\{m-1,k\}} (\mathbf{x})_j (\mathbf{h})_{k-j} \,, \quad k = 0, \ldots, m+n-2 \,. \tag{4.5.2}$$

We take the cue from [Lecture → Rem. 4.1.4.15] and borrow the crucial trick of zero padding. For $N \geq \max\{m,n\}$ we introduce the zero-padded vectors

$$\widetilde{\mathbf{x}} := \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N \quad , \quad \widetilde{\mathbf{h}} := \begin{bmatrix} \mathbf{h} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N \,.$$

Next we closely examine the periodic convolution of these two vectors, exploiting our knowledge about zero components of $\widetilde{\mathbf{x}}$ and $\widetilde{\mathbf{h}}$:

$$\begin{aligned} \left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_k &= \sum_{j=0}^{N-1} (\widetilde{\mathbf{x}})_j \left(\widetilde{\mathbf{h}}\right)_{(k-j) \bmod N} = \sum_{j=0}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{(k-j) \bmod N} \\ &= \sum_{j=0}^{\min\{m-1,k\}} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k-j} + \sum_{j=\min\{m,k+1\}}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k+N-j} \,. \end{aligned}$$

❶ Case $k \in \{m-1, \ldots, N-1\}$: In this case the second sum becomes void and we find

$$\left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_k = \sum_{j=0}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k-j} = \sum_{j=\max\{0,k-n+1\}}^{\min\{m-1,k\}} (\mathbf{x})_j (\mathbf{h})_{k-j} = (\mathbf{h} * \mathbf{x})_k \,,$$

that is, for these vector components both discrete convolutions produce the same result for any $N \geq \max\{n$

❷ Case $k \in \{0, \ldots, m-2\}$: The corresponding components of the discrete periodic convolution are

$$\begin{aligned} \left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_k &= \sum_{j=0}^{k} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k-j} + \sum_{j=k+1}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k+N-j} \\ &= \sum_{j=\max\{0,k-n+1\}}^{k} (\mathbf{x})_j (\mathbf{h})_{k-j} + \sum_{j=k+1}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k+N-j} \,. \end{aligned}$$

Clearly, the second sum should evaluate to zero if both discrete convolutions are to yield the same vector components. Thus, we have to demand

$$\begin{aligned} & \left(\widetilde{\mathbf{h}}\right)_{k+N-j} = 0 \qquad \forall j \in \{k+1, \ldots, m-1\} \,, \quad \forall k \in \{0, \ldots, m-2\} \\ \Leftrightarrow \quad & k+N-j \geq n \qquad \forall j \in \{k+1, \ldots, m-1\} \,, \quad \forall k \in \{0, \ldots, m-2\} \\ \Leftrightarrow \quad & N-(m-1) \geq n \\ \Leftrightarrow \quad & N \geq n+m-1 \,. \end{aligned}$$

Hence, $N \geq n+m-1$ is a necessary and sufficient condition for the agreement of the vector components with indices $k \in \{0, \ldots, m-2\}$.

Summing up we have found $N = n + m - 1$ and

$$\mathbf{h} * \mathbf{x} = \left(\widetilde{\mathbf{h}} *_{m+n-1} \widetilde{\mathbf{x}}\right)_{0:n+m-2} .$$

In fact, the range specification for the vector components could be dropped, because for $N = n + m - 1$ all components of the vector produced by the discrete periodic convolution $*_{m+n-1}$ are relevant.

The following code demonstrates the implementation of this idea:

C++ code 4.5.3: Discrete convolution by periodic convolution

```
2  Eigen::VectorXd dconv_p(const Eigen::VectorXd &h, const Eigen::VectorXd &x) {
3    const int n = h.size();   // length of vector h
4    const int m = x.size();   // length of vector x
5    const int N = m + n - 1;  // Minimal length of periodic convolution
6    // Zero-padded vectors of length m+n-1
7    Eigen::VectorXd ht = Eigen::VectorXd::Zero(N);
8    Eigen::VectorXd xt = Eigen::VectorXd::Zero(N);
9    ht.head(n) = h;
10   xt.head(m) = x;
11   // Discrete periodic convolution, see [Lecture → Code 4.2.2.1]
12   return pconv(ht, xt);
13 }
```

Get it on GitLab (discreteconvolution.hpp).

A basic loop-based implementation of `pconv()` is shown in [Lecture → Code 4.2.2.1].

HINT 1 for (4-5.b): Please note that EIGEN's FFT implementation can only deal with vectors with complex-valued entries, that is, the type **Eigen::VectorXcd**. In EIGEN you have to use the method `cast<std::complex<double>>()` when using both real-valued and complex-valued vectors in an expression. ⌟

SOLUTION of (4-5.b):

We elaborate how to realize a general periodic discrete convolution $\mathbf{p} *_n \mathbf{x}$, $\mathbf{p}, \mathbf{x} \in \mathbb{K}^n$, $n \in \mathbb{N}$, by means of a discrete periodic convolution of vectors of length $N > n$. We assume $N \geq 2n$!

We first rewrite the formula for the discrete periodic convolution of $\mathbf{p} = [p_0, \ldots, p_{n-1}]^\top \in \mathbb{K}^n$, $\mathbf{x} = [x_0, \ldots,$ from [Lecture → Def. 4.1.4.7], see also [Lecture → Code 4.2.2.1]:

$$(\mathbf{p} *_n \mathbf{x})_k = \sum_{j=0}^{n-1} x_j p_{(k-j) \bmod n} = \sum_{j=0}^{k} x_j p_{k-j} + \sum_{j=k+1}^{n-1} x_j p_{n+k-j} \quad , \quad k = 0, \ldots, n-1 \, . \tag{4.5.4}$$

Next, we look at a discrete periodic convolution of length $N > n$ of two vectors $\widetilde{\mathbf{p}} = [\widetilde{p}_0, \ldots, \widetilde{p}_{N-1}]^\top \in \mathbb{K}^N$ $\widetilde{\mathbf{x}} = [\widetilde{x}_0, \ldots, \widetilde{x}_{N-1}]^\top \in \mathbb{K}^N$,

$$(\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} \widetilde{x}_j \widetilde{p}_{k-j} + \sum_{j=k+1}^{N-1} \widetilde{x}_j \widetilde{p}_{N+k-j} \quad , \quad k = 0, \ldots, N-1 \, . \tag{4.5.5}$$

and wonder, how the vectors $\widetilde{\mathbf{p}}/\widetilde{\mathbf{x}}$ and $\mathbf{p}/\mathbf{x}$ have to be related so that the two periodic convolutions produce the same values for $k = 0, \ldots, n-1$.



Find $\widetilde{\mathbf{p}} \in \mathbb{K}^N$ and $\widetilde{\mathbf{x}} \in \mathbb{K}^N$ such that both the -term and the -term in (4.5.4) and (4.5.5) yield the same values for all $k \in \{0, \ldots, n-1\}$.

(I) Term : the sums agree, provided that

$$\widetilde{x}_j = x_j \quad , \quad \widetilde{p}_j = p_j \quad \text{for all} \quad j = 0, \ldots, n-1 \, . \tag{4.5.6}$$

(II) Term : In order to end up with the same summation bounds we set

$$\widetilde{x}_j = 0 \quad \text{for} \quad j = n, \ldots, N-1 \, . \tag{4.5.7}$$

(III) Term : in light of (4.5.7) we now need

$$\widetilde{p}_{N-\ell} = p_{n-\ell} \quad \text{for} \quad \ell = 1, \ldots, n-1 \tag{4.5.8}$$

to make the second sums agree. However, this is only compatible with (4.5.6), if

$$n - 1 < N - (n-1) \quad \Leftrightarrow \quad N > 2n - 2 \, ,$$

because otherwise the requirements (4.5.6) and (4.5.8) will clash.

Based on these considerations we introduce auxiliary vectors by zero padding and periodic padding, respectively:

$$\widetilde{\mathbf{x}} = \begin{bmatrix} \widetilde{x}_0 \\ \vdots \\ \widetilde{x}_{N-1} \end{bmatrix} := \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N \quad , \quad \widetilde{\mathbf{p}} = \begin{bmatrix} \widetilde{p}_0 \\ \vdots \\ \widetilde{p}_{N-1} \end{bmatrix} := \begin{bmatrix} \mathbf{p} \\ \mathbf{0} \\ \mathbf{p} \end{bmatrix} \in \mathbb{K}^N \, .$$

Written in components this amounts to

$$(\widetilde{\mathbf{x}})_j := \begin{cases} x_j & \text{for} \quad j = 0,\ldots,n-1\,, \\ 0 & \text{for} \quad j = 1n,\ldots,N-1\,, \end{cases}$$

$$(\widetilde{\mathbf{p}})_j := \begin{cases} p_j & \text{for} \quad j = 0,\ldots,n-1\,, \\ 0 & \text{for} \quad j = n+1,\ldots,N-n-1\,, \\ p_{j+(n-N)} & \text{for} \quad j = n-N,\ldots,N-1\,. \end{cases}$$

In particular, this means

$$\widetilde{p}_j = p_j \quad \text{for} \quad j = 0,\ldots,n-1 \quad , \quad \widetilde{p}_{N-\ell} = p_{n-\ell} \quad \text{for} \quad \ell = 1,\ldots,n\,. \tag{4.5.9}$$

Based on (4.5.4), we examine the size-N discrete periodic convolution of $\widetilde{\mathbf{p}}$ and $\widetilde{\mathbf{x}}$:

$$(\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} \widetilde{x}_j \widetilde{p}_{k-j} + \sum_{k=k+1}^{N-1} \widetilde{x}_j \widetilde{p}_{N+k-j}\,.$$

First we take into account that $\widetilde{x}_j = 0$ for $j \geq n$:

$$\blacktriangleright \quad (\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} x_j \widetilde{p}_{k-j} + \sum_{j=k+1}^{n-1} x_j \widetilde{p}_{N+k-j}\,. \tag{4.5.10}$$

Sums whose lower bound is larger than the upper bound are meant to be $= 0$.

Now we restrict ourselves to $k \in \{0,\ldots,n-1\}$, which implies

- that in the first sum of (4.5.10) that $k-j \in \{0,\ldots,n-1\}$,
- and that $j-k \in \{1,\ldots,n-1\}$ in the second sum of (4.5.10), which means $\widetilde{p}_{N+k-j} = p_{n+k-j}$ thanks to (4.5.9).

$$\blacktriangleright \quad (\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} x_j p_{k-j} + \sum_{k=k+1}^{n-1} x_j p_{n+k-j} = (\mathbf{p} *_n \mathbf{x})_k\,, \quad k \in \{0,\ldots,n-1\}\,.$$

This is exploited in the following C++ function, to which the parameter `N` passes N:

C++ code 4.5.11: Resizing periodic convolution

```
2  Eigen::VectorXd pconvN(const Eigen::VectorXd &p, const Eigen::VectorXd &x,
3                         unsigned int N) {
4    const int n = x.size();
5    assert(n == p.size());
6    // Requirement on N to avoid overlap in periodic padding
7    assert(N >= 2 * n);
8    // Periodic padding of vector p
9    Eigen::VectorXd pt{Eigen::VectorXd::Zero(N)};
10   pt.head(n) = p;
11   pt.tail(n) = p;
12   // Zero padding of vector x
13   Eigen::VectorXd xt{Eigen::VectorXd::Zero(N)};
14   xt.head(n) = x;
15   // Periodic convolution of length N
```

```
16    const Eigen::VectorXd y = pconv(pt, xt);
17    return y.head(n);
18  }
```

Get it on GitLab (discreteconvolution.hpp).

Of course, any efficient implementation must be based on DFTs by the FFT algorithm. Thus, we have to replace the call to `pconv()` in Code 4.5.11 by the DFT-based implementation given in [Lecture → Code 4.2.2.1]. We have to comply with the requirement that only DFTs of length $N = 2^\ell$, $\ell \in \mathbb{N}$ are admissible, and also meet the condition $N \geq 2n$. Thus, N has to be chosen as the smallest power of 2 which is $\geq 2n$. The following code chooses that N and merges Code 4.5.11 and [Lecture → Code 4.2.2.1].

C++ code 4.5.12: Fast discrete periodic convolution implemented by FFT-based DFT of length $N = 2^\ell$

```
2   Eigen::VectorXd pconv_fast(const Eigen::VectorXd &p, const Eigen::VectorXd &x) {
3     const int n = x.size();
4     assert(n == p.size());
5     Eigen::VectorXd z = Eigen::VectorXd::Zero(n);
6
7     // TODO: (4-5.b) Compute the discrete periodic convolution of p with x
8     // in an efficient manner.
9     // START
10
11    // N >= 2*n is required to avoid overlap in periodic padding
12    int N = 1;
13    while (N < 2 * n) {
14      N *= 2;
15    }
16    // Note that Eigen's FFT routines expect complex vectors
17    // Periodic padding of vector p
18    Eigen::VectorXcd pt{Eigen::VectorXcd::Zero(N)};
19    pt.head(n) = p.template cast<std::complex<double>>();
20    pt.tail(n) = p.template cast<std::complex<double>>();
21    // Zero padding of vector x
22    Eigen::VectorXcd xt{Eigen::VectorXcd::Zero(N)};
23    xt.head(n) = x.template cast<std::complex<double>>();
24    // Periodic convolution of length N = 2^l realized by FFT
25    Eigen::FFT<double> fft;
26    z = (fft.inv(((fft.fwd(pt)).cwiseProduct(fft.fwd(xt))).eval()))
27            .real()
28            .head(n);
29
30    // END
31    return z;
32  }
```

Get it on GitLab (discreteconvolution.hpp).

Note that this choice of N is important for efficiency, because choosing a larger power of 2 will incur significant extra work for large n.

Remark. An even more efficient implementation could also exploit the fact that $\mathbf{p}, \mathbf{x}$ are *real* vectors, see [Lecture → Section 4.2.4]. However, this further optimization is not expected here.

SOLUTION of (4-6.a):

- The function involves forming the tensor product of two vectors of length n, which created a densely populated $n \times n$-matrix at asymptotic cost of $O(n^2)$ for $n \to \infty$
- The solution of a *triangular* linear system of size n amounts to a simple elimination step with asymptotic cost $O(n^2)$ for $n \to \infty$, see [Lecture → § 2.3.2.15].

Hence the overall asymptotic complexity of `slvtriag()` is $O(n^2)$ for $n \to \infty$.

Remark. The function computes

$$\texttt{slvtriag}(\mathbf{u},\mathbf{v}) = \mathrm{triu}(\mathbf{u}\mathbf{v}^\top)^{-1}\mathbf{u}\ .$$

SOLUTION of (4-6.b):

A call to `slvsymrom()` involves the solution of a dense $n \times n$ linear system of equations by means of LU-decomposition and subsequent eliminations. The most costly step is the computation of the LU-decomposition with cost $O(n^3)$ for $n \to \infty$ [Lecture $\to$ Eq. (2.3.2.10)]. This step determines the asymptotic computational complexity of the function.

Remark. The result of the function is

$$\texttt{slvsymrom}(\mathbf{u}) = (\mathbf{I} + \mathbf{u}\mathbf{u}^\top)^{-1}\mathbf{u} \ .$$

HINT 1 for (4-6.c): Note that the constructor of **Eigen::HouseholderQR<>** computes an economical QR-factorization stored in compressed format. The call `qr.householderQ()*` just applies Householder transformations. ⌟

SOLUTION of (4-6.c):

According to [Lecture → § 3.3.3.37] the computation of the (thin, which is the default in EIGEN) QR-decomposition of $\mathbf{A} \in \mathbb{R}^{n,k}$ incurs an asymptotic computational effort of $O(nk^2)$ for $n,k \to \infty$. Since k is small and fixed, this means $O(n)$ computational cost. Note that the Q-factor is never computed as a dense $n \times n$-matrix, but stored in compressed format as discussed in [Lecture → Rem. 3.3.3.21].

In the last step the Q-factor is multiplied with a matrix $\mathbf{M} := \begin{bmatrix} \mathbf{O} \\ \mathbf{I} \end{bmatrix}$ of size $n \times (n-k)$, resulting in a matrix of the same size. What is going on internally is that k Householder transformations are applied to $\mathbf{M}$, which creates a dense $n \times (n-k)$-matrix with $O(n^2)$ entries. This will take $O(n^2)$ operations for $n \to \infty$ and this determines the overall asymptotic cost of the function.

Remark. If $\mathbf{A}$ has full rank, the function `getcompbas()` returns an orthonormal basis of the orthogonal complement of $\mathcal{R}(\mathbf{A})$ in the columns of the result matrix.

SOLUTION of (4-6.d):

The function executes $2n$ discrete Fourier transforms of length n. Using the FFT algorithm the cost of each will be $O(n \log n)$ [Lecture → § 4.3.0.3], which leads to a total computational effort of $O(n^2 \log n)$ for $n \to \infty$

Remark. The code performs the 2D DFT discussed in [Lecture → Section 4.2.5].

SOLUTION of (4-7.a):

Remark. The formula (4.7.2) is derived as follows

$$\begin{aligned} f_k &= \sum_{j=0}^{n-1} a_j \cos\big(\pi\frac{kj}{n}\big) = \frac{1}{2}\sum_{j=0}^{n-1} a_j\Big(e^{-\pi\imath\frac{kj}{n}} + e^{\pi\imath\frac{kj}{n}}\Big) \\ &= \sum_{j=0}^{n-1} \tfrac{1}{2}a_j e^{-2\pi\imath\frac{kj}{2n}} + \sum_{\ell=0}^{n-1} \tfrac{1}{2}a_{n-1-\ell} e^{2\pi\imath\frac{k(n-1-\ell)}{2n}} \\ &= \sum_{j=0}^{n-1} \tfrac{1}{2}a_j e^{-2\pi\imath\frac{kj}{2n}} + e^{\pi\imath\frac{k(n-1)}{n}} \sum_{\ell=0}^{n-1} \tfrac{1}{2}a_{n-1-\ell} e^{-2\pi\imath\frac{k\ell}{2n}} \end{aligned}$$

Matching this with (4.7.3) we find

$$(\mathbf{x})_j = \begin{cases} \frac{1}{2}a_j & \text{for} \quad j \in \{\, 0,\ldots,n-1 \,\}\,, \\ 0 & \text{for} \quad j \in \{\, n,\ldots,2n-1 \,\}\,, \end{cases} \quad , \quad (\mathbf{y})_j = \begin{cases} \frac{1}{2}a_{n-1-j} & \text{for} \quad j \in \{\, 0,\ldots,n-1 \,\}\,, \\ 0 & \text{for} \quad j \in \{\, n,\ldots,2n-1 \,\}\,, \end{cases} \quad ,$$

$$(\mathbf{D})_{k,k} = \exp\big(\pi\imath k\frac{n-1}{n}\big)\,, \quad \ell = 0,\ldots,n-1\,. \tag{4.7.4}$$

HINT 1 for (4-7.b): The EIGEN helper class **Eigen::FFT<double>** provides the member functions `fwd()` for DFT and `inv()` for inverse DFT. ⌟

HINT 2 for (4-7.b): The member function `reverse()` of **Eigen::VectorXd** reverses the order of the vector components. ⌟

SOLUTION of (4-7.b):

The following code is a straightforward implementation of (**??**) making use of the fact that the DFT is available through the method `fwd()` of an **Eigen::FFT** object. The final loop also performs the multiplication with $\mathbf{D}$, which is a simple component-wise scaling.

Note that (4.7.4) means that

$$(\mathbf{D})_{\ell,\ell} = \left(\exp\left(\pi\imath\frac{(n-1)}{n}\right)\right)^{\ell},$$

such that the scaling factors can be obtained by successive multiplication of 1 with `fac` $= \exp\left(\pi\imath\frac{(n-1)}{n}\right)$.

C++ code 4.7.5: Function `modcostrf()`

```
Eigen::VectorXd modcostrf(const Eigen::VectorXd &a) {
  using Comp = std::complex<double>;
  unsigned int n = a.size();
  Eigen::VectorXd f(n);
  Eigen::FFT<double> fft;
  Eigen::VectorXcd tmp(2 * n);
  tmp << 0.5 * a, Eigen::VectorXcd::Zero(n);
  Eigen::VectorXcd v1 = fft.fwd(tmp);
  tmp << 0.5 * a.reverse(), Eigen::VectorXcd::Zero(n);
  Eigen::VectorXcd v2 = fft.fwd(tmp);
  Comp fac = std::exp(((n - 1) * M_PI * Comp(0, 1)) / (double)n);
  Comp d(1.0, 0.0);
  for (int k = 0; k < n; ++k) {
    f[k] = (v1[k] + d * v2[k]).real();
    d *= fac;
  }
  return f;
}
```

HINT 1 for (4-8.a): In EIGEN the member function `conjugate()` effects complex conjugation of all entries of a matrix/vector (object). ⌟

SOLUTION of (4-8.a):

We simply use $\mathbf{F}_n^{-1} = \frac{1}{n}\overline{\mathbf{F}}_n$ from [Lecture → Lemma 4.2.1.14] and note that

$$\overline{\mathbf{F}}_n \mathbf{c} = \overline{\mathbf{F}_n \overline{\mathbf{c}}}\,,$$

because complex conjugation is an involution.

C++ code 4.8.2: Implementation of `inverseDFT()` Get it on GitLab.

```
2 Eigen::VectorXcd inverseDFT(const Eigen::VectorXcd &c) {
3   const Eigen::VectorXcd::Index n = c.size();
4   Eigen::FFT<double> fft;
5   Eigen::VectorXcd tmp = c.conjugate();
6   Eigen::VectorXcd y = fft.fwd(tmp);
7   return y.conjugate() / n;
8 }
```

SOLUTION of (4-8.b):

The code can be copied from [Lecture → Code 4.2.2.4], because multiplication with a circulant matrix [Lecture → Def. 4.1.4.12] defined by some periodic sequence amounts to a discrete periodic convolution [Lecture → Def. 4.1.4.7].

C++ code 4.8.4: Implementation of `circmatvec()` Get it on GitLab.

```
2 Eigen::VectorXcd circmatvec(const Eigen::VectorXcd &u,
3                             const Eigen::VectorXcd &x) {
4   Eigen::FFT<double> fft;
5   Eigen::VectorXcd tmp = (fft.fwd(u)).cwiseProduct(fft.fwd(x));
6   return fft.inv(tmp);
7 }
```

SOLUTION of (5-1.a):

The idea is to apply the Horner scheme [Lecture → Eq. (5.2.1.6)] to

$$p'(t) = k\texttt{c[0]}t^{k-1} + (k-1)\texttt{c[1]}t^{k-2} + \cdots + 2\texttt{c[k-2]}t + \texttt{c[k-1]}\,, \quad t \in \mathbb{R}\,. \tag{5.1.2}$$

The corresponding extension of [Lecture → Code 5.2.1.7] is straightforward.

C++-code 5.1.3: Implementation of `evaldp()`

```
2  std::pair<double, double> evaldp(const Eigen::VectorXd &c, const double x) {
3    double px, dpx;
4    const int s = c.size();  // degree + 1
5
6    // TODO (5-1.a):  Implement the function evaldp().
7
8    // START
9    // Standard Horner scheme
10   px = c(0);
11   for (int i = 1; i < s; ++i) {
12     px = x * px + c(i);
13   }
14
15   // Horner scheme for derivative
16   dpx = (s - 1) * c(0);
17   for (int i = 1; i < s - 1; ++i) {
18     dpx = x * dpx + (s - i - 1) * c(i);
19   }
20   // END
21
22   return {px, dpx};
23 }
```

Get it on GitLab (eval_deriv.hpp).

SOLUTION of (5-1.b):

The following code uses **std::pow** to evaluate x^k.

C++-code 5.1.4: Solution of (5-1.b): `evaldp_naive()`

```
std::pair<double, double> evaldp_naive(const Eigen::VectorXd &c,
                                       const double x) {
  double px, dpx;
  const int n = c.size();

  // TODO (5-1.b): Implement the function evaldp_naive().

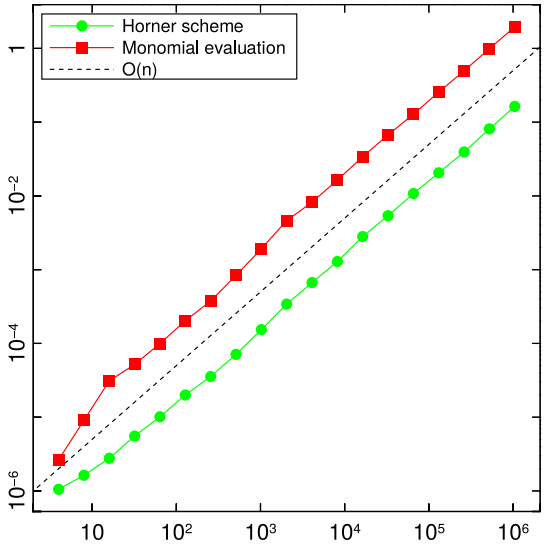
  // START
  px = c[0] * std::pow(x, n - 1);
  for (int i = 1; i < n; ++i) {
    px = px + c[i] * std::pow(x, n - i - 1);
  }

  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);
  }
  // END

  return {px, dpx};
}
```

Get it on GitLab (eval_deriv.hpp).

SOLUTION of (5-1.c):



The answer depends on your implementation. In any case, the code for `evaldp()` from Subproblem (5-1.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 5.1.4 requires $\approx n$ multiplications and additions. However, the implementation also calls the `pow()` function, which may be costly. Beside in Fig. 30 we report comparative runtime measurements for the two implementation. They reveal that **std**`::pow()` probably incurs only constant computational cost!

HINT 1 for (5-1.d): computed values Please recall from [Lecture → Rem. 1.5.3.15] the advice on comparing numerically ⌟

SOLUTION of (5-1.d):

C++-code 5.1.5: Solution of Sub-problem (5-1.d)

```
bool polyTestTime(const unsigned int d) {
  bool ret = false;  // Return value
  double x = 0.123;
  std::pair<double, double> p, p_naive;
  const int repeats = 10;

  std::cout
      << std::setw(10) << "n" << std::setw(25)
      << "Horner scheme:" << std::setw(25) << "Monomial approach:"
      << "\n"
      << " ================================================================\n";

  // TODO (5-1.d): Compare the output of evaldp() and evaldp_naive() for
  // x = 0.123 and tabulate their runtimes for n = 2,4,...,2^d.

  // START
  const double TOL = 1E-8;
  const int max_n = std::pow(2, d);
  Eigen::VectorXd c = Eigen::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;
}
```

Get it on GitLab (eval_deriv.hpp).

HINT 1 for (5-1.e): You may EIGEN's **Array** facilities and componentwise operations to acheive a vectorized implementation. ⌟

SOLUTION of (5-1.e):

Differentiating the recursion formula [Lecture → Eq. (5.2.3.9)] we obtain

$$
\begin{aligned}
p_i(t) &\equiv y_i \quad , \quad p_i'(t) \equiv 0, \qquad i = 0, \ldots, n \, , \\
p_{i_0,\ldots,i_m}(t) &= \frac{(t - t_{i_0}) p_{i_1,\ldots,i_m}(t) - (t - t_{i_m}) p_{i_0,\ldots,i_{m-1}}(t)}{t_{i_m} - t_{i_0}} , \\
p'_{i_0,\ldots,i_m}(t) &= \frac{p_{i_1,\ldots,i_m}(t) + (t - t_{i_0}) p'_{i_1,\ldots,i_m}(t) - p_{i_0,\ldots,i_{m-1}}(t) - (t - t_{i_m}) p'_{i_0,\ldots,i_{m-1}}(t)}{t_{i_m} - t_{i_0}} .
\end{aligned}
$$

C++-code 5.1.6: Example code using Aitken-Neville scheme

```
2  Eigen::VectorXd dipoleval(const Eigen::VectorXd &t, const Eigen::VectorXd &y,
3                            const Eigen::VectorXd &x) {
4    assert(t.size() == y.size() && "t and y must have same size!");
5    Eigen::VectorXd ret;
6    ret.resizeLike(x);  // Return value
7
8    // TODO (5-1.e): Use the Aitken-Neville algorithm to evaluate at x
9    // the derivative of the interpolating polynomial of the data (t,y).
10
11   // START
12   // Non-vectorized version: loop over all evaluation points
13   for (int i = 0; i < x.size(); ++i) {
14     Eigen::VectorXd p(y);
15     Eigen::VectorXd dP = Eigen::VectorXd::Zero(y.size());
16
17     // Aitken-Neville outer loop
18     for (int im = 1; im < y.size(); ++im) {
19       for (int i0 = im - 1; i0 >= 0; --i0) {
20         dP(i0) = (p(i0 + 1) + (x(i) - t(i0)) * dP(i0 + 1) - p(i0) -
21                   (x(i) - t(im)) * dP(i0)) /
22                  (t(im) - t(i0));
23
24         p(i0) = ((x(i) - t(i0)) * p(i0 + 1) - (x(i) - t(im)) * p(i0)) /
25                 (t(im) - t(i0));
26       }
27     }
28
29     ret(i) = dP(0);
30   }
31   // END
32
33   return ret;
34 }
```

Get it on GitLab (eval_deriv.hpp).

SOLUTION of (5-1.f):

C++-code 5.1.7: Implementation of validation function `dipoleval_alt()`

```
2  Eigen::VectorXd dipoleval_alt(const Eigen::VectorXd &t,
3                                const Eigen::VectorXd &y,
4                                const Eigen::VectorXd &x) {
5    assert(t.size() == y.size() && "t and y must have same size!");
6    Eigen::VectorXd ret;  // return value
7    const int n = y.size();
8
9    // TODO (5-1.f): Use polyfit() and polyval() to evaluate at x
10   // the derivative of the interpolating polynomial of the data (t,y).
11
12   // START
13   Eigen::VectorXd P = polyfit(t, y, n - 1).head(n - 1);
14
15   for (int i = 0; i < P.size(); ++i) {
16     P(i) *= n - 1 - i;
17   }
18
19   ret = polyval(P, x);
20   // END
21
22   return ret;
23 }
```

Get it on GitLab (eval_deriv.hpp).

C++-code 5.1.8: Testing framework

```
2  bool testDipolEval() {
3    bool ret = false;
4    Eigen::VectorXd dPx, dPx_alt;
5
6    // TODO (5-1.f): Compare dipoleval() and dipoleval_alt() for the data
7    // given on the problem sheet.
8
9    // START
10   double TOL = 1E-8;
11   const int n = 10;  // polynomial degree
12   Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(100, -1, 4);
13   Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(n + 1, 0, 3);
14   Eigen::VectorXd y = sin(t.array()).matrix();
15
16   dPx = dipoleval(t, y, x);
17   dPx_alt = dipoleval_alt(t, y, x);
18
19   double err = (dPx - dPx_alt).norm();
20
21   if (err < TOL) {
22     ret = true;
23   }
24   // END
25
26   return ret;
```

```
27 }
```

Get it on GitLab (eval_deriv.hpp).

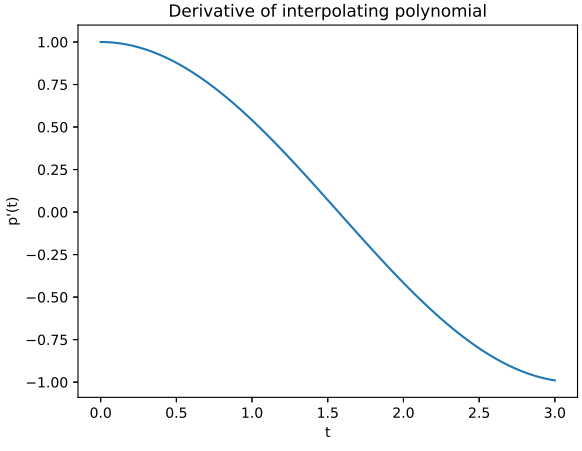
HINT 1 for (5-1.g): Axis labels can be added to the graphics object of MATPLOTLIBCPP by the functions `matplotlibcpp::xlabel()` and `matplotlibcpp::ylabel()`. ⌟

SOLUTION of (5-1.g):

C++-code 5.1.9: Plotting of polynomial interpolant

```
2  void plotPolyInterpolant(const std::string &filename) {
3    plt::figure();
4    // TODO (5-1.g): Plot the derivative of the polynomial that
5    // interpolates the data from (5-1.f).
6    // Hint: matplotlibcpp (here = plt) implements Python's
7    // matplotlib functions such as figure(), plot(), xlabel(), title(),...
8    // You can use main.cpp of the LinearDataFit problem as a reference.
9
10   // START
11   Eigen::VectorXd dPx;
12   const int n = 10;  // polynomial degree
13   Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(100, 0, 3);
14   Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(n + 1, 0, 3);
15   Eigen::VectorXd y = sin(t.array()).matrix();
16
17   dPx = dipoleval(t, y, x);
18
19   plt::plot(x, dPx);
20   plt::title("Derivative of interpolating polynomial");
21   plt::xlabel("t");
22   plt::ylabel("p'(t)");
23   // END
24
25   plt::savefig("./cx_out/" + filename + ".png");
26 }
```

Get it on GitLab (eval_deriv.hpp).



SOLUTION of (5-10.a):

The first step is to compute the auxiliary quantities δ_i, $i = 1,\ldots,n$, and λ_k, $k = 0,\ldots,n$. The δ_i give the lengths of the segments of the polygon, the λ_k the accumulated length as you move along it.

Let us temporarily fix $k \in \{1,\ldots,M\}$. The next step is to determine on which segment the point $\mathbf{p}^* := \mathbf{\Pi}_\Sigma($

$$\ell \in \{1,\ldots,n\}: \quad \lambda_{\ell-1} \le x_k\lambda_n < \lambda_\ell \,.$$

The point $\mathbf{p}^*$ splits the segment $[\mathbf{p}_\ell, \mathbf{p}_{\ell+1}]$ ($\mathbf{p}_{n+1} := \mathbf{p}_1$) with the ratio

$$\rho := \frac{x_k\lambda_n - \lambda_{\ell-1}}{\lambda_\ell - \lambda_{\ell-1}} \in [0,1[\,.$$

Remember from analytic geometry that the line segment connecting two points $\mathbf{a},\mathbf{b} \in \mathbb{R}^d$ is given by

$$[\mathbf{a},\mathbf{b}] = \{\tau\mathbf{a} + (1-\tau)\mathbf{b}: \ 0 \le \tau \le 1\}\,.$$

This motivates the formula

$$\mathbf{p}^* = \rho\mathbf{p}_\ell + (1-\rho)\mathbf{p}_{\ell+1}\,.$$

To convince yourself of its validity, note that

$$\rho = 0 \ \Rightarrow \ \mathbf{p}^* = \mathbf{p}_\ell\,, \quad \rho = 1 \ \Rightarrow \ \mathbf{p}^* = \mathbf{p}_{\ell+1}\,, \quad \rho = \tfrac{1}{2} \ \Rightarrow \ \mathbf{p}^* = \tfrac{1}{2}(\mathbf{p}_\ell + \mathbf{p}_{\ell+1})\,.$$

Your implementation should rely on n evaluations of the function $\mathbf{c}$, and the computational effort should scale linearly in M. To that end it is important to exploit the fact that the sequence (x_k) is sorted. This is done in the nested **for** and **while** loops in Code 5.10.4.

C++ code 5.10.4: Sub-problem (5-10.a): Computation of polygonal interpolant

```
2  std::vector<Eigen::Vector2d> closedPolygonalInterpolant(
3      std::vector<Eigen::Vector2d> &Sigma, const Eigen::VectorXd &x) {
4    // Note: we assume that x is sorted
5    assert(x.minCoeff() >= 0 && x.maxCoeff() <= 1 && "x out of range");
6
7    std::vector<Eigen::Vector2d> res(x.size());
8    unsigned int n = Sigma.size();
9    // extend Sigma with periodic condition
10   Sigma.push_back(Sigma[0]);
11
12   // define variables delta_1, ..., delta_n and lambda_0,...,lambda_n
13   std::vector<Eigen::Vector2d> d(n);
14   std::vector<double> delta(n);
15   std::vector<double> lambda(n + 1);
16
17   lambda[0] = 0.0;
18   for (unsigned int i = 0; i < n; ++i) {
19     d[i] = Sigma[i + 1] - Sigma[i];
20     delta[i] = (d[i]).norm();
21     lambda[i + 1] = lambda[i] + delta[i];
22   }
```

```
23
24    // resume original Sigma value
25    Sigma.pop_back();
26
27    // TODO: (5-10.a)
28    // START
29    unsigned int j = 0;
30    // loop over sides of polygon
31    for (unsigned int i = 1; i <= n; ++i) {
32      // loop over evaluation points x
33      while (j < x.size() && x(j) * lambda[n] <= lambda[i]) {
34        // linear interpolant formula
35        double ratio = (x(j) * lambda[n] - lambda[i - 1]) / delta[i - 1];
36        Eigen::Vector2d coord = (1 - ratio) * Sigma[i - 1] + ratio * Sigma[i];
37
38        res[j] = coord;
39        ++j;
40      }
41      if (j == x.size()) break;
42    }
43    // END
44    return res;
45  }
```

Get it on GitLab (adaptiveintp.hpp).

HINT 1 for (5-10.b): If you have solved Sub-problem (5-10.a) already, you can recycle large parts of your code. ⌟

SOLUTION of (5-10.b):

Your implementation should rely on n evaluations of the function $\mathbf{c}$, and the computational effort should scale linearly in M. For the local representation of $\mathbf{H}_\Sigma$ it is convenient to use the basis polynomials introduced in [Lecture → § 5.3.3.3].

C++ code 5.10.5: Sub-problem (5-10.b): Hermite interpolation of a closed curve

```
2  std::vector<Eigen::Vector2d> closedHermiteInterpolant(
3      std::vector<Eigen::Vector2d> &Sigma, const Eigen::VectorXd &x) {
4    // Note: we assume that x is sorted
5    assert(x.minCoeff() >= 0 && x.maxCoeff() <= 1 && "x out of range");
6    std::vector<Eigen::Vector2d> res(x.size());
7
8    unsigned int n = Sigma.size();
9    // extend Sigma with periodic condition
10   Sigma.push_back(Sigma[0]);
11
12   // define variables delta_1,...,delta_n and lambda_0,...lambda_n
13   std::vector<Eigen::Vector2d> d(n);
14   std::vector<double> delta(n);
15   std::vector<double> lambda(n + 1);
16
17   lambda[0] = 0.0;
18   for (unsigned int i = 0; i < n; ++i) {
19     d[i] = Sigma[i + 1] - Sigma[i];
20     delta[i] = d[i].norm();
21     lambda[i + 1] = lambda[i] + delta[i];
22   }
23
24   // define the slopes as in the definition of the interpolating
25   // closed cubic Hermite curve
26   std::vector<Eigen::Vector2d> slopes(n + 1);
27   // TODO: (5-10.b)
28   // START
29   Eigen::Vector2d s =
30       (delta[n - 1] / delta[0] * d[0] + delta[0] / delta[n - 1] * d[n - 1]) /
31       (delta[n - 1] + delta[0]);
32   slopes[0] = s / s.norm();
33   for (unsigned int i = 1; i < n; ++i) {
34     s = (delta[i] / delta[i - 1] * d[i - 1] + +delta[i - 1] / delta[i] * d[i]) /
35         (delta[i] + delta[i - 1]);
36     slopes[i] = s / s.norm();
37   }
38   slopes[n] = slopes[0];
39
40   Eigen::VectorXd x_scl = x * lambda[n];
41   unsigned int j = 0;
42   // loop over sides of polygon
43   for (unsigned int i = 1; i <= n; ++i) {
44     // loop over evaluation points x
45     while (j < x.size() && x_scl(j) <= lambda[i]) {
46       // Local cubic hermite interpolation
47       double coordx =
48           hermloceval(x_scl(j), lambda[i - 1], lambda[i], Sigma[i - 1](0),
49                       Sigma[i](0), slopes[i - 1](0), slopes[i](0));
```

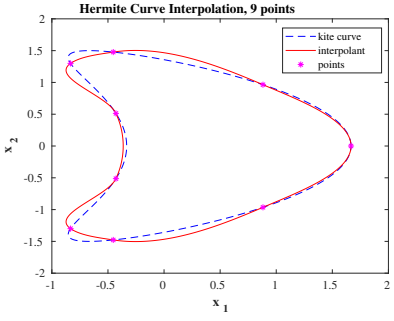
```
      double coordy =
          hermloceval(x_scl(j), lambda[i - 1], lambda[i], Sigma[i - 1](1),
                      Sigma[i](1), slopes[i - 1](1), slopes[i](1));
      // store the next point
      res[j] = {coordx, coordy};
      ++j;
    }
    if (j == x.size()) break;
  }
  // END

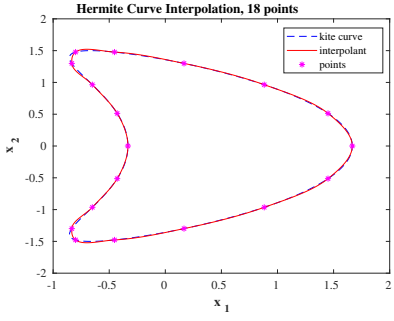
  // resume original Sigma value
  Sigma.pop_back();

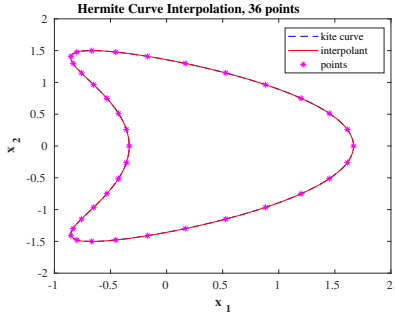
  return res;
}
```

Get it on GitLab (adaptiveintp.hpp).

Some curves produced by cubic Hermite interpolation:







SOLUTION of (5-10.c):

C++ code 5.10.7: Sub-problem (5-10.c): Adaptive Hermite interpolation of closed curve

```
template <typename CurveFunctor>
std::pair<std::vector<Eigen::Vector2d>, std::vector<Eigen::Vector2d>>
    adaptedHermiteInterpolant(
  CurveFunctor &&c, unsigned int nmin,
  const Eigen::VectorXd &x, double tol = 1.0e-3) {
  // clang-format on
  // Note: we assume that x is sorted
  assert(x.minCoeff() >= 0 && x.maxCoeff() <= 1 && "x out of range");
  // will contain the result
  std::vector<Eigen::Vector2d> eval;
  std::vector<Eigen::Vector2d> Sigma;

  Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(nmin + 1, 0, 1);
  // TODO: (5-10.c)
  // START
  // vector of deviations
  Eigen::VectorXd dev;
  do {
    // set current number of points
    unsigned int n = t.size() - 1;
    dev.resize(n);
    Sigma.resize(n + 1);
    // evaluate Sigma for arc-length points t_i
    for (unsigned int i = 0; i <= n; ++i) {
      Sigma[i] = c(t(i));
    }
    // define variables delta_1,...,delta_n and lambda_0,...lambda_n
    Eigen::VectorXd lambda(n + 1);

    Eigen::VectorXd midpt(n);
    lambda[0] = 0.0;
    for (unsigned int i = 0; i < n; ++i) {
      lambda(i + 1) = lambda(i) + (Sigma[i + 1] - Sigma[i]).norm();
      midpt(i) = (lambda(i + 1) + lambda(i)) * 0.5;
    }
    // compute linear and cubic interpolant
    std::vector<Eigen::Vector2d> v =
        closedPolygonalInterpolant(Sigma, midpt / lambda[n]);
    std::vector<Eigen::Vector2d> w =
        closedHermiteInterpolant(Sigma, midpt / lambda[n]);
    // evaluate deviations
    for (unsigned int i = 0; i < n; ++i) {
      dev(i) = (v[i] - w[i]).norm();
    }

    double alpha = dev.mean();
    // append new arc-length values
    std::vector<double> t_temp;
    int num = 0;
    for (unsigned int j = 0; j < n; ++j) {
      if (dev(j) > 0.9 * alpha) {
        t_temp.push_back((t(j) + t(j + 1)) * 0.5);
```

```
53            num++;
54          }
55        }
56        t.conservativeResize(n + 1 + t_temp.size());
57
58        t.tail(t_temp.size()) = Eigen::VectorXd::Map(t_temp.data(), t_temp.size());
59        // sort vector of arc-length values
60        std::sort(t.data(), t.data() + t.size());
61      } while (dev.maxCoeff() > tol);
62
63      eval = closedHermiteInterpolant(Sigma, x);
64      // END
65      return std::make_pair(eval, Sigma);
66    }
```

Get it on GitLab (adaptiveintp.hpp).

SOLUTION of (5-10.d):

C++ code 5.10.8: Sub-problem (5-10.c): Plotting kite-shaped curve

```
void plotKite(const char *filename, double tol = 1.0e-3) {
  plt::figure();
  // TODO: (5-10.d)
  // START
  // preparing call to adaptedHermiteInterpolant
  auto c = [](double t) -> Eigen::Vector2d {
    Eigen::Vector2d ct;
    ct << std::cos(2 * M_PI * t) + 2. / 3. * std::cos(4 * M_PI * t),
        3. / 2. * std::sin(2 * M_PI * t);
    return ct;
  };
  constexpr unsigned int nmin = 6;
  constexpr unsigned int N = 1e5;
  // initializing x, dsicarding the 1 at the end
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N + 1, 0, 1).head(N);

  std::pair<std::vector<Eigen::Vector2d>, std::vector<Eigen::Vector2d>> vec =
      adaptedHermiteInterpolant(c, nmin, x, tol);

  std::vector<Eigen::Vector2d> eval = vec.first;
  std::vector<Eigen::Vector2d> Sigma = vec.second;
  const unsigned int M = Sigma.size();
  Eigen::VectorXd x_hermite(N);
  Eigen::VectorXd y_hermite(N);
  Eigen::VectorXd x_poly(M);
  Eigen::VectorXd y_poly(M);
  // assembling x and y of Hermite Interpolation
  for (unsigned int i = 0; i < N; i++) {
    x_hermite(i) = eval.at(i)(0);
    y_hermite(i) = eval.at(i)(1);
  }
  // assembling x and y of Polygonal Interpolation
  // We do not need to call ClosedPolygonalInterpolate
  // as matplotlibcpp will already connect the points
  // with straight lines for us.
  for (unsigned int i = 0; i < M; i++) {
    x_poly(i) = Sigma.at(i)(0);
    y_poly(i) = Sigma.at(i)(1);
  }
  // making the plot
  plt::title("Adapted Interpolation with tol=" + std::to_string(tol));
  plt::xlabel("x");
  plt::ylabel("y");
  plt::plot(x_poly, y_poly, "r", {{"label", "Polygonal Interpolant"}});
  plt::plot(x_hermite, y_hermite, {{"label", "Hermite Interpolation"}});
  plt::legend();
  // END
  plt::savefig(filename);
}
```

Get it on GitLab (adaptiveintp.hpp).

SOLUTION of (5-11.a):

Since $s \in C^2([-1,2])$ is required for $s \in \mathcal{S}_{3,\mathcal{M}}$ [Lecture $\rightarrow$ Def. 5.4.1.1], at each node we have three continuity conditions for s, the first derivative s', and the second derivative s'':

- At $t_0 = 0$:

$$s(-1) = -1 \quad , \quad s'(-1) = 0 \quad , \quad s''(-1) = 0 \, .$$
$$\blacktriangleright \quad s(t) = \alpha t^3 - 1 \quad \text{for} \quad 0 \le t < 1 \, .$$

  This means $\beta = \gamma = 0$ and $\delta = -1$.

- At $t = 1$

$$s|_{[0,1]}(1) = \alpha - 1 = s|_{[1,2]}(1) = 1 + \sigma + \rho \, ,$$
$$\frac{ds|_{[0,1]}}{dt}(1) = 3\alpha = \frac{ds|_{[1,2]}}{dt}(1) = -3 + 2\sigma + \rho \, ,$$
$$\frac{d^2 s|_{[0,1]}}{dt^2}(1) = 6\alpha = \frac{d^2 s|_{[1,2]}}{dt^2}(1) = -6 + 2\sigma \, .$$

We arrive at the following linear system of equations for the unknown coefficients

$$\begin{bmatrix} 1 & -1 & -1 \\ 3 & -2 & -1 \\ 6 & -2 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \sigma \\ \rho \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ -6 \end{bmatrix} \quad \Rightarrow \quad \begin{array}{c} \alpha = 2 \, , \\ \sigma = 9 \, , \\ \rho = -9 \, . \end{array}$$

SOLUTION of (5-11.b):

- At $t = 0$: $\delta = 2, \gamma = 0, \rho = 0$.
- At $t = 1$: From the continuity conditions

$$\begin{array}{rcl} \sigma + 2 & = & -1 + \alpha + \beta - 1 \\ 3\sigma & = & -3 + 2\alpha + \beta \\ 6\sigma & = & -6 + 2\alpha \end{array} \quad \blacktriangleright \quad \begin{bmatrix} 1 & 1 & -1 \\ 2 & 1 & -3 \\ 2 & 0 & -6 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \sigma \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 6 \end{bmatrix} \quad \Rightarrow \quad \begin{array}{c} \alpha = -9\,, \\ \beta = 9\,, \\ \sigma = -4\,. \end{array}$$

SOLUTION of (5-11.c):

- At $t = 0$: $\delta = 1, \gamma = 0, \beta = 0$
- At $t = 2$:

$$\begin{aligned} 8\alpha + 1 &= -8 + 4\rho + 2\sigma - 7 \\ 12\alpha &= -12 + 4\rho + \sigma \\ 12\alpha &= -12 + 2\rho \end{aligned} \quad \blacktriangleright \quad \begin{bmatrix} 8 & -4 & -2 \\ 12 & -4 & -1 \\ 12 & -2 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \rho \\ \sigma \end{bmatrix} = \begin{bmatrix} -16 \\ -12 \\ -12 \end{bmatrix} \quad \Rightarrow \quad \begin{aligned} \alpha &= -2\,, \\ \rho &= -6\,, \\ \sigma &= 12\,. \end{aligned}$$

SOLUTION of (5-12.a):

If $n = 2$, a polynomial s of degree 3 interpolating (t_0, y_0), (t_1, y_1), and (t_2, y_2) is clearly a not-a-knot cubic spline interpolant according to Def. 5.12.2. By [Lecture $\rightarrow$ Thm. 5.2.2.7] such an interpolating polynomial is not unique, because it would take four interpolation conditions in order to ensure uniqueness.

SOLUTION of (5-12.b):

We write t_j, $j = 0, \ldots, n$, for the $n+1$ ordered knots and y_j, $j = 0, \ldots, n$, for the data values, which correspond to the entries `t_[j]` and `y_[j]` of the data members (of an object of type **NotAKnotCubicSpline**) `t_` and `y_`, respectively. In addition, c_j, $j = 0, \ldots, n$ designates the slope at t_j, which is stored in `c_[j]`. Thus, if $s : [t_0, t_n] \to \mathbb{R}$ stands for the cubic spline interpolant, then

$$s(t_j) = y_j \quad , \quad s'(t_j) = c_j \, , \quad j = 1, \ldots, n \, .$$

According to [Lecture → Eq. (5.4.2.6)] we have the local representation on the knot interval $[t_{j-1}, t_j]$, $j = 1, \ldots, n$:

$$\begin{aligned} s_{|[t_{j-1},t_j]}(t) &= y_{j-1}(1 - 3\tau^2 + 2\tau^3) + y_j(3\tau^2 - 2\tau^3) + \\ &\quad h_j c_{j-1}(\tau - 2\tau^2 + \tau^3) + h_j c_j(-\tau^2 + \tau^3) \, , \end{aligned} \tag{5.12.4}$$

with $h_j := t_j - t_{j-1}$, $\tau := (t - t_{j-1})/h_j$. Since $\frac{d\tau}{dt} = h_j^{-1}$, we find by the chain rule

$$\begin{aligned} s'_{|[t_{j-1},t_j]}(t) &= \frac{1}{h_j}\Big(y_{j-1}(-6\tau + 6\tau^2) + y_j(6\tau - 6\tau^2)\Big) + \\ &\quad c_{j-1}(1 - 4\tau + 3\tau^2) + c_j(-2\tau + 3\tau^2) \, , \\ s''_{|[t_{j-1},t_j]}(t) &= \frac{1}{h_j^2}\big(y_{j-1}(-6 + 12\tau) + y_j(6 - 12\tau)\big) + \\ &\quad \frac{1}{h_j}\big(c_{j-1}(-4 + 6\tau) + c_j(-2 + 6\tau)\big) \, , \\ s^{(3)}_{|[t_{j-1},t_j]}(t) &= \frac{12}{h_j^3}(y_{j-1} - y_j) + \frac{6}{h_j^2}(c_{j-1} + c_j) \, , \end{aligned} \tag{5.12.5}$$

where $s^{(3)}$ is the third derivative of s. Def. 5.12.2 demands the continuity of the third derivative in the knots t_1 and t_{n-1}, that is,

$$s^{(3)}_{|[t_0,t_1]}(t_1) = s^{(3)}_{|[t_1,t_2]}(t_1) \quad , \quad s^{(3)}_{|[t_{n-2},t_{n-1}]}(t_{n-1}) = s^{(3)}_{|[t_{n-1},t_n]}(t_{n-1}) \, .$$

Using the expression found for $s^{(3)}$ we obtain

$$\frac{12}{h_1^3}(y_0 - y_1) + \frac{6}{h_1^2}(c_0 + c_1) = \frac{12}{h_2^3}(y_1 - y_2) + \frac{6}{h_2^2}(c_1 + c_2) \, , \tag{5.12.6}$$

$$\frac{12}{h_{n-1}^3}(y_{n-2} - y_{n-1}) + \frac{6}{h_{n-1}^2}(c_{n-2} + c_{n-1}) = \frac{12}{h_n^3}(y_{n-1} - y_n) + \frac{6}{h_n^2}(c_{n-1} + c_n) \, . \tag{5.12.7}$$

HINT 1 for (5-12.c):



The implementation given in `notaknotcubicspline.hpp` is that for the natural cubic spline interpolant. You have to modify it appropriately.

SOLUTION of (5-12.c):

Of course you start from the available implementation for the natural cubic spline interpolant. You have to remove the two equations induced by the endpoint conditions for a natural cubic spline interpolant:

$$s''(t_0) = 0 \quad , \quad s''(t_n) = 0 \, .$$

Then you have to add the two conditions found in Sub-problem (5-12.b). There we found the relationships (5.12.6) and (5.12.7). We let these equations induce the first and last row of the $(n+1) \times (n+1)$ linear system of equations for the unknown slope vector $[c_0, \ldots, c_n]^\top \in \mathbb{R}^{n+1}$. Starting from the corresponding linear system of equations [Lecture → Eq. (5.4.2.13)] for the natural cubic spline interpolant, we just have to modify the first and last row and get

$$\begin{bmatrix} A_0 & B_0 & C_0 & \cdots & & & \cdots & 0 \\ b_0 & a_1 & b_1 & 0 & \cdots & & \cdots & 0 \\ 0 & b_1 & a_2 & b_2 & & & & \\ & 0 & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & a_{n-2} & b_{n-2} & 0 \\ 0 & \cdots & & \cdots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & \cdots & & & \cdots & C_n & B_n & A_n \end{bmatrix} \begin{bmatrix} c_0 \\ \\ \vdots \\ \vdots \\ \vdots \\ \\ c_n \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{12}{h_1^3}(y_0 - y_1) + \frac{12}{h_2^2}(y_1 - y_2) \\ 3\left(\frac{y_1 - y_0}{h_1^2} + \frac{y_2 - y_1}{h_2^2}\right) \\ \vdots \\ 3\left(\frac{y_{n-1} - y_{n-2}}{h_{n-1}^2} + \frac{y_n - y_{n-1}}{h_n^2}\right) \\ -\frac{12}{h_{n-1}^3}(y_{n-2} - y_{n-1}) + \frac{12}{h_n^3}(y_{n-1} - y_n) \end{bmatrix} , \quad (5.12.8)$$

with $h_j := t_j - t_{j-1}$, $j = 1, \ldots, n$ and matrix entries

$$b_i := \frac{1}{h_{i+1}} \, , \quad a_i := \frac{2}{h_i} + \frac{2}{h_{i+1}} \, , \quad i = 0, 1, \ldots, n-1 \, ,$$
$$A_0 := \frac{6}{h_1^2} \, , \quad B_0 := \frac{6}{h_1^2} - \frac{6}{h_2^2} \, , \quad C_0 := -\frac{6}{h_2^2} \, ,$$
$$A_n := -\frac{6}{h_n^2} \, , \quad B_n := \frac{6}{h_n^2} - \frac{6}{h_{n-1}^2} \, , \quad C_n := \frac{6}{h_{n-1}^2} \, .$$

Of course, the first row of the linear system is directly given by (5.12.6), while the bottom row corresponds to (5.12.7). Knowing this, the values for the matrix entries are clear.

Note that rows 1 through $n-1$ (C++ indexing) of the LSE arise from the continuity conditions for the second derivative at interior nodes. These equations have been derived in [Lecture → § 5.4.2.5].

C++ code 5.12.9: Constructor of NotAKnotCubicSpline

```
NotAKnotCubicSpline::NotAKnotCubicSpline(Eigen::VectorXd t, Eigen::VectorXd y)
    : t_(std::move(t)), y_(std::move(y)), c_(y_.size()) {
  // TODO: (5-12.c) Modify the given constructor of a Natural Cubic Spline into
  // an efficient constructor for Not-a-knot Cubic Spline
  // START
  const long n = t_.size() - 1;
  assert(n > 2);
  assert(n == y_.size() - 1);
  // Sequence (h_i)_{i=1}^n, knots have to be sorted
  auto h = (t_.tail(n) - t_.head(n)).array();
  auto hs = h.pow(2);                                  // h_i^2
  auto b = 1.0 / h;                                    // b_i := h_i^{-1}
  auto a = 2.0 * (b.head(n - 1) + b.tail(n - 1));      // a_i
  // Initialize sparse coefficient matrix for the linear system of equations
  // needed to compute the unknown slopes.  Compared to [Lecture → Eq. (5.4.2.13)] only
  // the first and last row has to be adjusted, see (5.12.8).
  Eigen::SparseMatrix<double> M(n + 1, n + 1);
  M.reserve(Eigen::RowVectorXi::Constant(n + 1, 3));
  // Matrix row arising from the continuity of
  M.insert(0, 0) = 6.0 / hs[0];
  M.insert(0, 1) = (6.0 / hs[0] - 6.0 / hs[1]);
  M.insert(0, 2) = -6.0 / hs[1];
  for (int i = 1; i < n; ++i) {
    M.insert(i, i - 1) = b[i - 1];
    M.insert(i, i) = a[i - 1];
    M.insert(i, i + 1) = b[i];
  }
  // Matrix due to continuity of s^(3) at t_{n-1}
  M.insert(n, n - 2) = 6.0 / hs[n - 2];
  M.insert(n, n - 1) = 6.0 / hs[n - 2] - 6.0 / hs[n - 1];
  M.insert(n, n) = -6.0 / hs[n - 1];
  // Initialize right-hand-side vector, see (5.12.8)
  Eigen::VectorXd rhs(n + 1);
  rhs[0] = -12.0 / (hs[0] * h[0]) * (y_[0] - y_[1]) +
           12.0 / (hs[1] * h[1]) * (y_[1] - y_[2]);
  for (int i = 1; i < n; ++i) {
    rhs[i] =
        3.0 * ((y_[i] - y_[i - 1]) / hs[i - 1] + (y_[i + 1] - y_[i]) / hs[i]);
  }
  rhs[n] = -12.0 / (hs[n - 2] * h[n - 2]) * (y_[n - 2] - y_[n - 1]) +
           12.0 / (hs[n - 1] * h[n - 1]) * (y_[n - 1] - y_[n]);
  // Compute slopes by solving an (almost) tridiagonal linear system
  // with asymptotic effort O(n) for n → ∞.
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(M);
  if (solver.info() != Eigen::Success) {
    throw std::runtime_error("LU factorization failed");
  }
  c_ = solver.solve(rhs);
  if (solver.info() != Eigen::Success) {
    throw std::runtime_error("Elimination failed");
  }
  if (checkC2()) {
```

```
55      std::cout << "C2 continuity satisfied" << std::endl;
56    }
57    // END
58  }
```

Get it on GitLab (notaknotcubicspline.hpp).

Note that *the use of a sparse matrix data type is essential* to allow the sparse elimination solver to exploit the structure of the matrix, *cf.* [Lecture → Section 2.7.3]. It is also essential to tell EIGEN the maximal number of non-zero entries per row before setting matrix entries [Lecture → Code 2.7.2.1].

SOLUTION of (5-12.d):

We use the local representation (5.12.4) of the cubic spline interpolant s and the associated formula

$$\begin{aligned} s'_{|[t_{j-1},t_j]}(t) &= \frac{1}{h_j}\Big(y_{j-1}(-6\tau+6\tau^2)+y_j(6\tau-6\tau^2)\Big)+ \\ &\quad c_{j-1}(1-4\tau+3\tau^2)+c_j(-2\tau+3\tau^2) \\ &= c_{j-1}+\left(\frac{6}{h_j}(y_j-y_{j-1})-4c_{j-1}-2c_j\right)\tau+ \\ &\quad \left(\frac{6}{h_j}(y_{j-1}-y_j)+3c_{j-1}+c_j\right)\tau^2\,, \end{aligned}$$

where $h_j := t_j - t_{j-1}$, $j = 1,\ldots,n$, $\tau := \frac{t-t_{j-1}}{h}$. The implementation of the member function is straightforward then:

C++ code 5.12.10: Member function for the point evaluation of the derivative s'

```
2  double NotAKnotCubicSpline::evalDerivative(double t) const {
3    double derivative = 0;
4    // TODO: (5-12.d) Implement the evaluation of the first derivative
5    // START
6    const int idx = getPtIdx(t);
7    const double h = t_[idx] - t_[idx - 1];
8    const double tau = (t - t_[idx - 1]) / h;
9    const double a0 = c_[idx - 1];
10   const double a1 =
11       6.0 / h * (y_[idx] - y_[idx - 1]) - 4.0 * c_[idx - 1] - 2.0 * c_[idx];
12   const double a2 =
13       6.0 / h * (y_[idx - 1] - y_[idx]) + 3.0 * c_[idx - 1] + 3.0 * c_[idx];
14   derivative = a0 + tau * (a1 + tau * a2);
15   // END
16   return derivative;
17 }
```

Get it on GitLab (notaknotcubicspline.hpp).

Note the use of the Horner scheme [Lecture → Code 5.2.1.7] to evaluate the quadratic polynomial

$$t \mapsto a_0 + a_1 t + a_2 t^2\,.$$

However, any other equivalent formula is acceptable, too.

SOLUTION of (5-13.a):

In this problem we can sometimes appeal to the heuristics that

$$\dim V_i = \dim \mathcal{P}_d(\mathbb{R}) - \sharp\{\text{linear constraints defining } V_i\} \ .$$

From [Lecture → Thm. 5.2.1.2] we know that $\dim \mathcal{P}_d = d+1$.

1. V_1 is defined by a single linear constraint, which means that

$$\dim V_1 = d+1-1 = d \ .$$

2. V_2 is defined by fixing two zeros, which amounts to two linearly independent linear constraints.

$$\blacktriangleright \quad \dim V_2 = \dim \mathcal{P}_d(\mathbb{R}) - 2 = d-1 \ .$$

Remark. In fact we can write

$$V_2 := \left\{ t \mapsto (1-t)^2 q(t),\ q \in \mathcal{P}_{d-2}(\mathbb{R}) \right\} .$$

3. If $p \in \mathcal{P}_d(\mathbb{R})$, then for the third derivative $p^{(3)} \in \mathcal{P}_{d-3}(\mathbb{R})$. Moreover, if a polynomial vanishes on an interval ("has infinitely many zeros"), then it must be zero everywhere. We conclude that $p^{(3)} \equiv 0$. As a consequence $p \in \mathcal{P}_2$.

$$\blacktriangleright \quad \dim V_3 = \begin{cases} 2 & \text{for } d = 1 \ , \\ 3 & \text{for } d \geq 2 \ ,. \end{cases}$$

4. The polynomials in V_4 are *even functions*, which means that they are of the form

$$V_4 \subset \mathcal{P}_n^{\mathrm{even}} := \{ t \mapsto a_0 + a_2 t^2 + a_4 t^4 + \cdots + a_{2n} t^{2n} \} \ , \quad n \in \mathbb{N} \ .$$

Elements of space $\mathcal{P}_n^{\mathrm{even}}$ are defined by $n+1$ parameters, which means $\dim \mathcal{P}_n^{\mathrm{even}} = n+1$.

Note that $n = \frac{d}{2}$ for even d and $n = \frac{d-1}{2}$ for odd d.

$$\blacktriangleright \quad \dim V_4 = \begin{cases} \frac{d}{2}+1 & \text{for even } d \ , \\ \frac{d-1}{2}+1 & \text{for odd } d. \end{cases}$$

SOLUTION of (5-13.b):

- $\mathfrak{B}_1$ is the Lagrangian basis **C** [Lecture $\rightarrow$ Eq. (5.2.2.4)], a cardinal basis for polynomial interpolation in $\{t_0, t_1, \ldots, t_n\}$,
- $\mathfrak{B}_2$ is the Newton basis **B**[Lecture $\rightarrow$ Eq. (5.2.3.23)], used for update-friendly polynomial interpolation
- $\mathfrak{B}_3$ is the monomial basis **A** [Lecture $\rightarrow$ Section 5.2.1].

SOLUTION of (5-13.c):

This is the recursion [Lecture → Eq. (5.2.3.9)] underlying the **Aitken-Neville** scheme:

$$p_{k,\ell}(t) = \frac{\left(t - t_k\right) p_{k+1,\ell}(t) + \left(t_\ell - t\right) p_{k,\ell-1}(t)}{t_\ell - t_k}, \quad t \in \mathbb{R}.$$

The correctness of this recursion is shown by induction, verifying the interpolation conditions.

SOLUTION of (5-14.a):

Since $s \in C^1([-3,3])$ and $s \equiv 0$ on $[2,3]$, we conclude $s(2) = s'(2) = 0$, which means

$$s(t) = \alpha(t-2)^2\,, \quad \alpha \in \mathbb{R} \quad \text{on} \quad [1,2]\,.$$

From the condition $s(1) = 1$ we get $\alpha = 1$, which implies

$$s(t) = 1\cdot t^2 - 4\,t + 4 \quad \text{on} \quad [1,2]\,.$$

As consequence $s'(1) = -2$. Thus the quadratic polynomial p describing s on $[-1,1]$ has to satisfy

$$p(-1) = 1 \quad , \quad p(1) = 1 \quad , \quad p'(1) = -2\,.$$

This uniquely defines the parabola

$$p(t) = 2 - t^2\,, \quad t \in \mathbb{R}\,,$$

from which we infer

$$s(t) = -1\cdot t^2 + 0\,t + 2 \quad \text{on} \quad [-1,1]\,.$$

Remark. Here the solution of the problem can stop, but we have not yet verified that the partial s as found so far can be extended to a spline in $\mathcal{S}_{2,\mathcal{M}}$. We conjecture that s is an even function $s(t) = s(-t)$ and, thus, set

$$s(t) = t^2 + 4t + 4 \quad \text{on} \quad [-2,-1]\,.$$

Symmetry arguments readily demonstrate that this completes the definition of the spline s.

SOLUTION of (5-14.b):

By straightforward computations:

$$\begin{aligned}
&H_1(0)=1\,,\quad H_1(1)=0\,,\quad H_1'(0)=0\,,\quad H_1'(1)=0\,,\quad H_1(\tfrac12)=\tfrac12\ ,\quad H_1'(\tfrac12)=-\tfrac32\,,\\
&H_2(0)=0\,,\quad H_2(1)=1\,,\quad H_2'(0)=0\,,\quad H_2'(1)=0\,,\quad H_2(\tfrac12)=\tfrac12\ ,\quad H_2'(\tfrac12)=\tfrac32\,,\\
&H_3(0)=0\,,\quad H_3(1)=0\,,\quad H_3'(0)=1\,,\quad H_3'(1)=0\,,\quad H_3(\tfrac12)=\tfrac18\ ,\quad H_3'(\tfrac12)=-\tfrac14\,,\\
&H_4(0)=0\,,\quad H_4(1)=0\,,\quad H_4'(0)=0\,,\quad H_4'(1)=1\,,\quad H_4(\tfrac12)=-\tfrac18\,,\quad H_4'(\tfrac12)=-\tfrac14\,.
\end{aligned} \tag{5.14.1}$$

The key insight is that from (5.14.1) and the chain rule we conclude for the spline s

$$s(t_{j-1})=\alpha_j\,,\quad s(t_j)=\beta_j\,,\quad s'(t_{j-1})=\gamma_j\,,\quad s'(t_j)=\delta_j\,,\quad j=1,\dots n\,. \tag{5.14.2}$$

This means that the coefficients have a concrete meaning as the point values and derivative values of the spline in the endpoints of the knot intervals.

The same argument yields, now using the local representation on the knot intervals of $\widetilde{\mathcal{M}}$:

$$s(\widetilde{t}_{\ell-1})=\widetilde{\alpha}_\ell\,,\quad s(\widetilde{t}_\ell)=\widetilde{\beta}_\ell\,,\quad s'(\widetilde{t}_{\ell-1})=\widetilde{\gamma}_\ell\,,\quad s'(\widetilde{t}_\ell)=\widetilde{\delta}_\ell\,,\quad \ell=1,\dots 2n\,. \tag{5.14.3}$$

Next, we have to compute the values of s and s' at midpoints of knot intervals of $\mathcal{M}$. We start with

$$\begin{aligned}
H_1(1/2)&=\tfrac12\ ,\quad H_1'(1/2)=-\tfrac32\,,,\\
H_2(1/2)&=\tfrac12\ ,\quad H_2'(1/2)=\tfrac32\,,\ ,\\
H_3(1/2)&=\tfrac18\ ,\quad H_3'(1/2)=-\tfrac14\,,,\\
H_4(1/2)&=-\tfrac18\,,\quad H_4'(1/2)=-\tfrac14\,,,
\end{aligned} \tag{5.14.4}$$

which implies

$$s(\tfrac12(t_j+t_{j-1}))=\tfrac12(\alpha_j+\beta_j)+\tfrac18(\gamma_j-\delta_j)\,, \tag{5.14.5}$$

$$s'(\tfrac12(t_j+t_{j-1}))=\tfrac32(-\alpha_j+\beta_j)-\tfrac14(\gamma_j+\delta_j)\,. \tag{5.14.6}$$

I. For **even** $\ell=2j$ we have $\widetilde{t}_\ell=t_j$, $j=1,\dots,n$ and the associated knot interval for $\ell, j>0$ is

$$[\widetilde{t}_{\ell-1},\widetilde{t}_\ell]=[\tfrac12(t_j+t_{j-1}),t_j]\,.$$

So its right endpoint is a knot of $\mathcal{M}$, the left endpoint a midpoint of a knot interval of $\mathcal{M}$. We conclude from (5.14.2), (5.14.3), (5.14.5)

$$\begin{aligned}
\widetilde{\alpha}_\ell&=s(\widetilde{t}_{\ell-1})=s(\tfrac12(t_j+t_{j-1}))=\tfrac12(\alpha_j+\beta_j)+\tfrac18 h_j(\gamma_j-\delta_j)\,,\\
\widetilde{\beta}_\ell&=s(\widetilde{t}_\ell)=s(t_j)=\beta_j\,,\\
\widetilde{\gamma}_\ell&=s'(\widetilde{t}_{\ell-1})=s'(\tfrac12(t_j+t_{j-1}))=\tfrac32 h_j^{-1}(-\alpha_j+\beta_j)-\tfrac14(\gamma_j+\delta_j)\,,\\
\widetilde{\delta}_\ell&=s'(\widetilde{t}_\ell)=s'(t_j)=\delta_j\,.
\end{aligned}$$

The factor h_j^{-1} is introduced by applying the chain rule.

II. For **odd** $\ell = 2j-1$, $j = 1, \dots, n$, we consider the $\widetilde{\mathcal{M}}$ knot interval

$$[\widetilde{t}_{\ell-1}, \widetilde{t}_{\ell}] = [t_{j-1}, \tfrac{1}{2}(t_j + t_{j-1})] \, .$$

Now the left endpoint is a midpoint of a knot interval of $\mathcal{M}$, and the right endpoint is a knot of $\mathcal{M}$. As before we conclude

$$\begin{aligned}
\widetilde{\alpha}_\ell &= s(\widetilde{t}_{\ell-1}) = s(t_{j-1}) = \alpha_j \, , \\
\widetilde{\beta}_\ell &= s(\widetilde{t}_{\ell}) = s(\tfrac{1}{2}(t_j + t_{j-1})) = \tfrac{1}{2}(\alpha_j + \beta_j) + \tfrac{1}{8} h_j (\gamma_j - \delta_j) \, , \\
\widetilde{\gamma}_\ell &= s'(\widetilde{t}_{\ell-1}) = s'(t_{j-1}) = \gamma_j \, , \\
\widetilde{\delta}_\ell &= s'(\widetilde{t}_{\ell}) = s'(\tfrac{1}{2}(t_j + t_{j-1})) = \tfrac{3}{2} h_j^{-1} (-\alpha_j + \beta_j) - \tfrac{1}{4}(\gamma_j + \delta_j) \, .
\end{aligned}$$

Summing up

$$\widetilde{\alpha}_\ell = \begin{cases} \tfrac{1}{2}(\alpha_j + \beta_j) + \tfrac{1}{8} h_j (\gamma_j - \delta_j) & \text{for even } \ell \, , \\ \alpha_j & \text{for odd } \ell \, , \end{cases} \qquad \widetilde{\beta}_\ell = \begin{cases} \beta_j & \text{for even } \ell \, , \\ \tfrac{1}{2}(\alpha_j + \beta_j) + \tfrac{1}{8} h_j (\gamma_j - \delta_j) & \text{for odd } \ell \, , \end{cases}$$

$$\widetilde{\gamma}_\ell = \begin{cases} \tfrac{3}{2} h_j^{-1} (-\alpha_j + \beta_j) - \tfrac{1}{4}(\gamma_j + \delta_j) & \text{for even } \ell \, , \\ \gamma_j & \text{for odd } \ell \, , \end{cases} \qquad \widetilde{\delta}_\ell = \begin{cases} \delta_j & \text{for even} \\ \tfrac{3}{2} h_j^{-1} (-\alpha_j + \beta_j) - \tfrac{1}{4}(\gamma_j + \delta_j) & \text{for odd} \end{cases}$$

SOLUTION of (5-15.a):

Since

$$t = t_{j-1} \quad \Rightarrow \quad \tau = 0\,, \quad t = t_j \quad \Rightarrow \quad \tau = 1\,, \tag{5.15.6}$$

we find

$$\mathbf{s}|_{]t_{j-1},t_j[}(t_{j-1}) = \mathbf{y}^{j-1} \quad , \quad \mathbf{s}|_{]t_{j-1},t_j[}(t_j) = \mathbf{y}^j\,, \quad j = 1,\ldots,n\,. \tag{5.15.7}$$

Thus

- the uniqueness of the coefficients $\mathbf{y}^j$ immediately leads to

  $$\mathbf{s}|_{]t_{j-1},t_j[}(t_j) = \mathbf{y}^j = \mathbf{s}|_{]t_j,t_{j+1}[}(t_j)\,, \quad j = 1,\ldots,n-1\,,$$

  which guarantees the continuity of $\mathbf{s}$, and
- the interpolation conditions (5.15.3)/(5.15.4) can easily be enforced by setting

  $$\mathbf{y}^j := \mathbf{p}^j\,, \quad j = 1,\ldots,n \quad , \quad \mathbf{y}^0 := \mathbf{p}^n\,. \tag{5.15.8}$$

As a consequence of (5.15.8) the vector coefficients $\mathbf{y}^j$, $j = 0,\ldots,n$, can be assumed to be known.

SOLUTION of (5-15.b):

We start with the important observation that, assuming the knot sequence $\mathcal{M} := \{0 = t_0 < t_1 < \ldots < t_{n-}$ to be given, the first component of $\mathbf{s}$ depends only on the first components of the points $\mathbf{p}^j$. Thus, for the sake of brevity let us write

$$s(t) := (\mathbf{s}(t))_1 \,, \quad x_j := \left(\mathbf{x}^j\right)_1 \,, \quad j = 1, \ldots, n \,, \quad p_j := \begin{cases} (\mathbf{p}^j)_1 & \text{for } j = 1, \ldots, n \,, \\ (\mathbf{p}_n)_1 & \text{for } j = 0 \,. \end{cases}$$

In light of (5.15.8) we get the local representation

$$s|_{]t_{j-1},t_j[}(t) = p_{j-1}(1-\tau) + x_j h_j \tau(1-\tau) + p_j \tau \,, \quad \tau := \frac{t - t_{j-1}}{h_j} \,, \quad \begin{matrix} t_{j-1} < t < t_j \,, \\ h_j := t_j - t_{j-1} \,. \end{matrix} \tag{5.15.9}$$

As we have seen, the continuity conditions at the knots are implied by this representation. What remains to be satisfied is the *continuity of the first derivative* s' plus the second part of (5.15.4). This amounts to the n conditions

$$s'|_{]t_{j-1},t_j[}(t_j) = s'|_{]t_j,t_{j+1}[}(t_j) \,, \quad j = 1, \ldots, n-1 \,, \tag{5.15.10a}$$

$$s'|_{]t_0,t_1[}(t_0) = s'|_{]t_{n-1},t_n[}(t_n) \,. \tag{5.15.10b}$$

By differentiating the local representation (5.15.9) using the chain rule and $\frac{d\tau}{dt} = h_j^{-1}$, we deduce

$$s'|_{]t_{j-1},t_j[}(t) = -h_j^{-1} p_{j-1} + x_j(1-2\tau) + h_j^{-1} p_j \,, \quad \tau := \frac{t - t_{j-1}}{h_j} \,, \quad t_{j-1} < t < t_j \,. \tag{5.15.11}$$

Owing to (5.15.6) this means that

$$s'|_{]t_{j-1},t_j[}(t_j) = -h_j^{-1} p_{j-1} - x_j + h_j^{-1} p_j \,, \qquad s'|_{]t_{j-1},t_j[}(t_{j-1}) = -h_j^{-1} p_{j-1} + x_j + h_j^{-1} p_j \,, \tag{5.15.12}$$

$$s'|_{]t_j,t_{j+1}[}(t_{j+1}) = -h_{j+1}^{-1} p_j - x_{j+1} + h_{j+1}^{-1} p_{j+1} \,, \qquad s'|_{]t_j,t_{j+1}[}(t_j) = -h_{j+1}^{-1} p_j + x_{j+1} + h_{j+1}^{-1} p_{j+1} \,. \tag{5.15.13}$$

We plug these expressions into (5.15.10) and obtain linear equations

$$\text{(5.15.10b)} \quad \Rightarrow \quad -h_1^{-1} p_0 + x_1 + h_1^{-1} p_1 = -h_n^{-1} p_{n-1} - x_n + h_n^{-1} p_n \,, \tag{5.15.14}$$

$$\text{(5.15.10a)} \quad \Rightarrow \quad -h_j^{-1} p_{j-1} - x_j + h_j^{-1} p_j = -h_{j+1}^{-1} p_j + x_{j+1} + h_{j+1}^{-1} p_{j+1}, \; j = 1, \ldots, n-1 \,. \tag{5.15.15}$$

Arranging the unknowns on the left side we get

$$x_1 + x_n = h_1^{-1} p_0 - h_1^{-1} p_1 - h_n^{-1} p_{n-1} + h_n^{-1} p_n \,, \tag{5.15.16}$$

$$x_j + x_{j+1} = -h_j^{-1} p_{j-1} + h_j^{-1} p_j + h_{j+1}^{-1} p_j - h_{j+1}^{-1} p_{j+1} \,, \quad j = 1, \ldots, n-1 \,. \tag{5.15.17}$$

This is an $n \times n$ linear system of equations for the unknowns $x_1, \ldots, x_n$. Its matrix-vector form is given by

$$\begin{bmatrix} 1 & 0 & \ldots & \ldots & 0 & 1 \\ 1 & 1 & 0 & \ldots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \ldots & \ldots & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \frac{p_n - p_{n-1}}{h_n} - \frac{p_1 - p_0}{h_1} \\ \frac{p_1 - p_0}{h_1} - \frac{p_2 - p_1}{h_2} \\ \frac{p_2 - p_1}{h_2} - \frac{p_3 - p_2}{h_3} \\ \vdots \\ \vdots \\ \frac{p_{n-1} - p_{n-2}}{h_{n-1}} - \frac{p_n - p_{n-1}}{h_n} \end{bmatrix}. \tag{5.15.18}$$

Remove the abbreviations:

$$\begin{bmatrix} 1 & 0 & \ldots & \ldots & 0 & 1 \\ 1 & 1 & 0 & \ldots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \ldots & \ldots & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} (\mathbf{x}^1)_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ (\mathbf{x}^n)_1 \end{bmatrix} = \begin{bmatrix} \frac{(\mathbf{p}^n)_1 - (\mathbf{p}^{n-1})_1}{h_n} - \frac{(\mathbf{p}^1)_1 - (\mathbf{p}^n)_1}{h_1} \\ \frac{(\mathbf{p}^1)_1 - (\mathbf{p}^n)_1}{h_1} - \frac{(\mathbf{p}^2)_1 - (\mathbf{p}^1)_1}{h_2} \\ \frac{(\mathbf{p}^2)_1 - (\mathbf{p}^1)_1}{h_2} - \frac{(\mathbf{p}^3)_1 - (\mathbf{p}^2)_1}{h_3} \\ \vdots \\ \vdots \\ \frac{(\mathbf{p}^{n-1})_1 - (\mathbf{p}^{n-2})_1}{h_{n-1}} - \frac{(\mathbf{p}^n)_1 - (\mathbf{p}^{n-1})_1}{h_n} \end{bmatrix}. \tag{5.15.19}$$

HINT 1 for (5-15.c): You may use EIGEN's built-in sparse solver, which we assume to have optimal complexity. However, efficient initialization of the sparse matrix is mandatory. ⌟

SOLUTION of (5-15.c):

When using EIGEN's **Eigen::SparseMatrix** class, the library must be told about the maximal number of non-zero entries per row in advance of setting matrix entries, see [Lecture → Code 2.7.2.1]. An equivalent alternative is initialization through a triplet format.

C++ code 5.15.21: Constructor of ClosedQuadraticSplineCurve

```
2  ClosedQuadraticSplineCurve::ClosedQuadraticSplineCurve(
3      const Eigen::Matrix2Xd &p)
4      : n_(p.cols()), p_(2, p.cols() + 1), t_(p.cols() + 1), x_(2, p.cols()) {
5    assert((n_ > 2) && "At least three points have to be supplied");
6    assert((n_ % 2 == 1) &&
7           "Number of points must be odd!");  //
8    // Save point coordinates, duplicate endpoints
9    p_.col(0) = p.col(n_ - 1);
10   p_.rightCols(n_) = p;
11   // TODO: (5-15.c) Write the constructor.
12   // START
13   // Compute knot set by accumulating length of polygon segments
14   Eigen::VectorXd lensum{n_};
15   lensum[0] = (p_.col(1) - p_.col(0)).norm();
16   for (unsigned int l = 1; l < n_; ++l) {
17     lensum[l] = lensum[l - 1] + (p_.col(l + 1) - p_.col(l)).norm();
18   }
19   t_[0] = 0.0;
20   for (unsigned int l = 1; l <= n_; ++l) {
21     t_[l] = lensum[l - 1] / lensum[n_ - 1];
22   }
23   // Lengths of knot intervals
24   const Eigen::VectorXd h = t_.tail(n_) - t_.head(n_);
25 // Compute vector coefficients x^j
26 #ifndef SMART_LSE_SOLUTION
27   // Initialize sparse matrix efficiently, see
28   Eigen::SparseMatrix<double> A(n_, n_);
29   A.reserve(Eigen::RowVectorXi::Constant(n_, 2));
30   A.insert(0, 0) = 1.0;
31   A.insert(0, n_ - 1) = 1.0;
32   for (unsigned int k = 1; k < n_; ++k) {
33     A.insert(k, k) = 1.0;
34     A.insert(k, k - 1) = 1.0;
35   }
36   A.makeCompressed();
37   // Compute LU-decomposition
38   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver(A);
39   if (solver.info() != Eigen::Success) {
40     throw std::runtime_error("Matrix factorization failed!");
41   }
42 #endif
43   // Right-hand side vector for linear system of equations
44   Eigen::VectorXd b(n_);
45   // Treat first and second component
46   for (unsigned int d = 0; d <= 1; ++d) {
47     // Compile right-hand side vector
48     const auto c{p_.row(d)};
49     b[0] = (c[n_] - c[n_ - 1]) / h[n_ - 1] - (c[1] - c[0]) / h[0];
```

```
50      for (unsigned int k = 1; k < n_; ++k) {
51        b[k] = (c[k] - c[k - 1]) / h[k - 1] - (c[k + 1] - c[k]) / h[k];
52      }
53  #ifdef SMART_LSE_SOLUTION
54      // Forward elimination in-situ in b
55      for (unsigned int j = 1; j < n_; ++j) {
56        b[j] -= b[j - 1];
57      }
58      // Backward substitution in-situ in b
59      b[n_ - 1] *= 0.5;
60      int sign = 1;
61      for (unsigned int j = 0; j < n_ - 1; ++j) {
62        b[j] -= (sign * b[n_ - 1]);
63        sign *= -1;
64      }
65      x_.row(d) = b.transpose();
66  #else
67      // Solve sparse linear system
68      x_.row(d) = solver.solve(b).transpose();
69      if (solver.info() != Eigen::Success) {
70        throw std::runtime_error("Elimination failed");
71      }
72  #endif
73    }
74    // END
75  }
```

Get it on GitLab (periodicquadraticsplines.hpp).

The status checks on successful execution of the sparse solver are optional.

Remark. Why does, in Line 7, the constructor expect n to be odd? Consider the vector $\mathbf{z} := \left[(-1)^{j-1}\right]_{j=1}^{n}$. Writing $\mathbf{A} \in \mathbb{R}^{n,n}$ for the coefficient matrix of the linear system of equations (5.15.19), it is easy to see that $\mathbf{Az} = \mathbf{0}$, if n is even! In this case the matrix cannot be regular.

Deeper insight is conveyed by studying the LU-decomposition of $\mathbf{A}$:

$$\begin{bmatrix} 1 & 0 & \dots & \dots & 0 & 1 \\ 1 & 1 & 0 & \dots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \dots & \dots & 0 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & \dots & \dots & 0 & 0 \\ 1 & 1 & 0 & \dots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \dots & \dots & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & & \dots & 0 & 1 \\ 0 & 1 & \ddots & & & \vdots & -1 \\ \vdots & \ddots & \ddots & & & & 1 \\ & & & \ddots & \ddots & \vdots & -1 \\ & & & \ddots & \ddots & 0 & \vdots \\ \vdots & & & & \ddots & \ddots & (-1)^{n-2} \\ 0 & \dots & & & \dots & 0 & 1+(-1)^{n-1} \end{bmatrix} =: \mathbf{LU} . \quad (5.15.22)$$

It confirms that $\mathbf{A}$ is regular, if and only if n is odd!

From (5.15.22) we can also get an idea for a very efficient way to solve $\mathbf{Ax} = \mathbf{b}$, compare [Lecture → § 2.3.2.15].

❶ Forward elimination: solve $\mathbf{Lz} = \mathbf{b}$, which, writing $\mathbf{b} = [b_1, \ldots, b_n]$ and $\mathbf{z} = [z_1, \ldots, z_n]$, boils down to

$$z_1 = b_1 \quad , \quad z_j = b_j - z_{j-1} \, , \quad j = 2, \ldots, n \, . \tag{5.15.23}$$

❷ Backward substitution: solve $\mathbf{Ux} = \mathbf{z}$, which means

$$x_n := \tfrac{1}{2} z_n \quad , \quad x_j = z_j + (-1)^j z_n \, , \quad j = 1, \ldots, n-1 \, . \tag{5.15.24}$$

Both, (5.15.23) and (5.15.24) can be realized by means of a single loop and require $O(n)$ computational effort for $n \to \infty$. This smart way of solving (5.15.19) is executed in the above code, if the macro `SMART_LSE_SOLUTION` is defined.

SOLUTION of (5-15.d):

The key to an efficient implementation is to exploit the fact that the `v`-array is sorted. Thus, we can find the knot interval, in which the next v_j is located with a total number of $O(n+N)$ comparisons. Once we have found that $v_k \in [t_{j-1}, t_j[$, we can simply use the formula (5.15.5) to compute $\mathbf{s}(v_k)$.

C++ code 5.15.25: C++ Code for function curve_points()

```
2  Eigen::Matrix2Xd ClosedQuadraticSplineCurve::curve_points(
3      const Eigen::VectorXd &v) const {
4    unsigned int N = v.size();
5    assert(N > 0);
6    // Matrix containing points to be computed
7    Eigen::Matrix2Xd s(2, N);
8    // Lengths of knot intervals
9    const Eigen::VectorXd h = t_.tail(n_) - t_.head(n_);
10   // TODO: (5-15.d) Compute the points on the curve for the sorted
       parameter
11   // values v.
12   // START
13
14   // Run through all the provided parameter values
15   unsigned int knot_idx = 0;
16   for (unsigned int k = 0; k < N; ++k) {
17     assert(((v[k] >= 0.0) && (v[k] < 1.0)) && "Out of range!");
18     if (k > 0) {
19       assert(v[k] >= v[k - 1] && "Parameter values not sorted!");
20     }
21     // Find right knot interval:  knot_idx stores index of knot to the
         right of
22     // current parameter value
23     while ((knot_idx < n_) && (v[k] >= t_[knot_idx])) {
24       knot_idx++;
25     }
26     assert(knot_idx > 0);
27     const double tau = (v[k] - t_[knot_idx - 1]) / h[knot_idx - 1];
28     s.col(k) = ((1.0 - tau) * p_.col(knot_idx - 1)) +
29                (h[knot_idx - 1] * tau * (1 - tau)) * x_.col(knot_idx - 1) +
30                (tau * p_.col(knot_idx));
31   }
32   // END
33   return s;
34 }
```

Get it on GitLab (periodicquadraticsplines.hpp).

The consistency checks for the parameter values `v[k]` are optional.

HINT 1 for (5-15.e): Certainly you know the simple formula

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc \,, \quad a, b, c, d \in \mathbb{R} \,. \tag{5.15.26}$$

SOLUTION of (5-15.e):

The approach to locate the right knot intervals is the same as with `curve_points()` from Sub-problem (5
The $\mathcal{M}$-piecewise constant second derivative is obtained by differentiating (5.15.11) once more using the chain rule:

$$\mathbf{s}''\big|_{]t_{j-1},t_j[}(t) = -2\mathbf{x}^j\,, \quad t_{j-1} < t < t_j\,.$$

C++ code 5.15.27: C++ code for function local_curvatures()

```
Eigen::VectorXd ClosedQuadraticSplineCurve::local_curvatures(
    const Eigen::VectorXd &v) const {
  unsigned int N = v.size();
  assert(N > 0);
  // Vector for storing the computed curvatures
  Eigen::VectorXd c(N);
  // Lengths of knot intervals
  const Eigen::VectorXd h = t_.tail(n_) - t_.head(n_);
  // TODO: (5-15.e) Compute the local curvature at the parameter values
      v.
  // START

  // Same as in curve_points(): Run through all the
  // provided parameter values
  unsigned int knot_idx = 0;
  for (unsigned int k = 0; k < N; ++k) {
    assert(((v[k] >= 0.0) && (v[k] < 1.0)) && "Out of range!");
    if (k > 0) {
      assert(v[k] >= v[k - 1] && "Parameter values not sorted!");
    }
    // Find right knot interval: knot_idx stores index of knot to the
        right of
    // current parameter value
    while ((v[k] >= t_[knot_idx]) && (knot_idx < n_)) {
      knot_idx++;
    }
    assert(knot_idx > 0);
    const double tau = (v[k] - t_[knot_idx - 1]) / h[knot_idx - 1];
    // Derivative vector
    const Eigen::Vector2d ds =
        (p_.col(knot_idx) - p_.col(knot_idx - 1)) / h[knot_idx - 1] +
        (1.0 - 2 * tau) * x_.col(knot_idx - 1);
    // Second derivative vector
    const Eigen::Vector2d dds = -2.0 * x_.col(knot_idx - 1);
    // Evaluate formula for signed curvature
    const double determinant = ds[0] * dds[1] - ds[1] * dds[0];
    c[k] = determinant / ds.norm();
  }
  // END
  return c;
}
```

Get it on GitLab (periodicquadraticsplines.hpp).

SOLUTION of (5-15.f):

SOLUTION of (5-16.a):

From [Lecture → Eq. (5.2.2.4)] we learn

$$L_i(t) := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{t-t_j}{t_i-t_j}\,, \quad i = 0,\ldots,n\,. \qquad \text{[Lecture → Eq. (5.2.2.4)]}$$

This obviously means $L_i \in \mathcal{P}_n$ and that it satisfies the interpolation conditions (5.16.1).

HINT 1 for (5-16.b): You may use the symbol $\prod$ to designate a product and rely on the convention that a product whose upper index is smaller than the lower evaluates to 1. ⌟

SOLUTION of (5-16.b):

The Newton basis for $\mathcal{T} := (t_0, t_1, \ldots, t_n)$ was introduced in [Lecture → § 5.2.3.22], [Lecture → Eq. (5.2.3
and reads

$$\mathcal{N}_{\mathcal{T}} = \left\{ \prod_{j=0}^{i-1} (t - t_i) \right\}_{i=0}^{n} . \tag{5.16.2}$$

SOLUTION of (5-16.c):

- **InterpPolyEval_A**: *Aitken-Neville scheme*, see [Lecture → Section 5.2.3.2].
- **InterpPolyEval_B**: *Barycentric interpolation formula*, see [Lecture → Section 5.2.3.1].
- **InterpPolyEval_C**: *Divided differences*, see [Lecture → Section 5.2.3.4].

HINT 1 for (5-17.a): The algorithm is a variant of the Horner scheme [Lecture → § 5.2.1.5]. ⌟

SOLUTION of (5-17.a):

The algorithm is a variant of the Horner scheme and is discussed in [Lecture → § 5.2.3.33], see [Lecture → Code 5.2.3.34].

C++ code 5.17.2: Function `EvalNewtonForm()` Get it on GitLab.

```
2  double evalNewtonForm(const std::vector<double> &t,
3                        const std::vector<double> &c, double x) {
4    assert((t.size() == c.size() && "Sizes of t and c vectors must agree"));
5    const int n = c.size() - 1;
6    double p = c[n];
7    for (int j = n - 1; j >= 0; --j) {
8      p = (x - t[j]) * p + c[j];
9    }
10   return p;
11 }
```

HINT 1 for (5-17.b): The following function implements the Aitken-Neville algorithm for the evaluation of the polynomial interpolant through the data points (t_i, y_i), $i = 0, \ldots, n$, for the argument x.

```
double ANipoleval(const VectorXd& t, VectorXd y, const double x) {
  for (int i = 0; i < y.size(); ++i) {
    for (int k = i - 1; k >= 0; --k) {
      // Recursion [Lecture → Eq. (5.2.3.9)]
      y[k] = y[k + 1] + (y[k + 1] - y[k])*(x - t[i])/(t[i] - t[k]);
    }
  }
  return y[0];
}
```

SOLUTION of (5-17.b):

What? Extrapolation to *zero*? Well, here we face $n \to \infty$!



Regard the sequence (s_n) as the values of a function $s : [1, \infty[\to \mathbb{R}$ sampled at $\mathbb{N} \subset \mathbb{R}$, and assume that $\lim_{x\to\infty} s(x)$ exists.



$$s^* := \lim_{n\to\infty} s_n = \lim_{x\to\infty} s(x) = \lim_{h\to 0} s(1/h)\,,$$

and we have converted the problem into one of finding a limit $h \to 0$. Of course, we can only evaluate $\psi(h) := s(h^{-1})$ for $h \in \{n^{-1} : n \in \mathbb{N}\}$, which dictates the data points we can use for extrapolation: $(n^{-1}, s_n) \in \mathbb{R} \times \mathbb{R}$, $n \in \mathbb{N}$.

Thus, in the spirit of interpolation to zero we approximate the limit s^* as $p_n(0)$, where $p \in \mathcal{P}_n$ is the degree-n polynomial uniquely defined by the interpolation conditions

$$p(h_i) = s_i\,, \quad h_i = \frac{1}{i}\,, \quad i = 1, \ldots, n+1\,.$$

The code is adapted from [Lecture → Code 5.2.3.19].

C++ code 5.17.4: Function `limitByExtrapolation()` Get it on GitLab.

```
template <class SEQGEN>
double limitByExtrapolation(SEQGEN &&seq, double rtol = 1.0E-6,
                            double atol = 1.0E-8, unsigned int maxdeg = 20) {
  seq.reset();
  std::vector<double> h{1};
  std::vector<double> s{seq.next()};
  // using Aitken-Neville scheme with x = 0, see
  //  [Lecture → Code 5.2.3.10]
  for (unsigned i = 1; i < maxdeg; ++i) {
    // Next data point (h_i, s_i)
    h.push_back(1.0 / (i + 1.0));  // First coordinate of data point
    s.push_back(seq.next());       // Second coordinate of data point
    // Aitken-Neville update
    for (int k = i - 1; k >= 0; --k)
      s[k] = s[k + 1] - (s[k + 1] - s[k]) * h[i] / (h[i] - h[k]);
    // termination of extrapolation when desired tolerance is reached
    const double errest = std::abs(s[1] - s[0]);  // error indicator
    if (errest < rtol * std::abs(s[0]) || errest < atol) {  //
      // Return value extrapolated from largest number of data points
      return s[0];
    }
  }
  throw std::runtime_error("limitByExtrapolation failed to converge!");
  return std::nan("");
}
```

HINT 1 for (5-2.a): Which p will you get in (5.2.1), when setting $y_j = 1$? Recall the uniqueness of Lagrange interpolants asserted in [Lecture $\rightarrow$ Thm. 5.2.2.7]. ⌟

SOLUTION of (5-2.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 $1 = p(x) = \sum_{i=0}^n L_i(x)$.

HINT 1 for (5-2.b): Consider $y_i = t_i$ in (5.2.1) for $i = 0, \ldots, n$. What will be the Lagrange interpolant p in this case? ⌟

SOLUTION of (5-2.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 $x = 0$: $0 = \sum_{i=0}^n L_i(0) t_i^j$.

HINT 1 for (5-2.c): Apply the product rule for differentiation in a nested way to differentiate $\omega(t)$. ⌟

SOLUTION of (5-2.c):

Simply exploiting the product rule for a product of several functions

$$\frac{d}{dx}\left\{ x \mapsto \prod_{j=1}^{n} f_j(x) \right\} = \sum_{i=1}^{n} f_i'(x) \prod_{j\neq i} f_j(x) \ , \tag{5.2.5}$$

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:

$$\begin{aligned} \omega(x) \sum_{i=0}^{n} \frac{y_i}{(x - t_i)\omega'(t_i)} &= \sum_{i=0}^{n} \left[\frac{y_i}{(x - t_i) \prod_{\substack{j=0 \\ j\neq i}}^{n} (t_i - t_j)} \prod_{j=0}^{n} (x - t_j) \right] \\ &= \sum_{i=0}^{n} \left[\frac{y_i}{\prod_{\substack{j=0 \\ j\neq i}}^{n} (t_i - t_j)} \prod_{\substack{j=0 \\ j\neq i}}^{n} (x - t_j) \right] = \sum_{i=0}^{n} \left[y_i \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - t_j}{t_i - t_j} \right] = p(x) \ . \end{aligned}$$

HINT 1 for (5-3.a): Recall how the standard Lagrange polynomial interpolation is linked to the Vandermonde matrix presented in [Lecture → Rem. 5.2.2.10]. ⌟

SOLUTION of (5-3.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 $\mathbf{Va} = \mathbf{b}$, for $\mathbf{a} = [a_0, \ldots, a_{2n+1}]^\top \in \mathbb{R}^{2n+2}$. The first $n+1$ equations of the system arise from the interpolation conditions $p(t_i) = y_i$:

$$y_j = p(t_j) = \sum_{i=0}^{2n+1} a_i t_j^i, \; j = 0, \ldots, n \; .$$

cf. the Vandermonde matrix. Differentiating the polynomial in monomial representation we get $p'(x) = \sum_{i=}^{2n}$
Using the interpolation condition $p'(t_i) = c_i$ for the derivative gives $n+1$ more rows of the linear system:

$$c_j = p'(t_j) = \sum_{i=1}^{2n+1} a_i i t_j^{i-1}, \; j = 0, \ldots, n \; .$$

Hence, we find $\mathbf{b} = [y_0, \ldots, y_n, c_0, \ldots, c_n]^\top$ and the system matrix $\mathbf{V}$ turns out as

$$\mathbf{V} = \begin{bmatrix} 1 & t_0 & t_0^2 & \ddots & t_0^{2n+1} \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & t_n & t_n^2 & \ldots & t_n^{2n+1} \\ 0 & 1 & 2t_0 & \ldots & (2n+1)t_0^{2n} \\ \vdots & \vdots & \vdots & \vdots & \\ 0 & 1 & 2t_n & \ldots & (2n+1)t_n^{2n} \end{bmatrix} \in \mathbb{R}^{2n+2,2n+2} \; .$$

HINT 1 for (5-3.b): Study [Lecture → Ex. 5.2.2.18] and consult [Lecture → Eq. (5.3.3.5)]. ⌟

,

SOLUTION of (5-3.b):

We can find the generalized Lagrange polynomials in monomial form by solving the linear system derived in Sub-problem (5-3.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 (5.3.1) 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

$$L_0(t) = 3\left(\tfrac{1-t}{2}\right)^2 - 2\left(\tfrac{1-t}{2}\right)^3 \quad , \quad L_1(t) = 3\left(\tfrac{1+t}{2}\right)^2 - 2\left(\tfrac{1+t}{2}\right)^3 ,$$
$$K_0(t) = 2\left[\left(\tfrac{1-t}{2}\right)^2 - \left(\tfrac{1-t}{2}\right)^3\right] \quad , \quad K_1(t) = -2\left[\left(\tfrac{1+t}{2}\right)^2 - \left(\tfrac{1+t}{2}\right)^3\right] .$$

Note that $L_0(t) = L_1(-t)$, $K_0(t) = K_1(-t)$, which reflects the symmetry of the interval $[-1,1]$.

SOLUTION of (5-3.c):

1. Any parabola that touches the x-axis in $x = a$ is of the form $x \mapsto c(x-a)^2$ with some $c \in \mathbb{R}$.
2. Every polynomial $p \in \mathcal{P}_n$, $n \geq 2$, with double zero in $t = a$ can be written as

$$p(t) = (t-a)^2 q(t) \quad \text{for some} \quad q \in \mathcal{P}_{n-2} \,.$$

SOLUTION of (5-3.d):

Group the terms of the product accoding to

$$h(t) = g_1(t) \cdot (g_2(t) \cdot \cdots \cdot g_m(t)) \ ,$$

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

$$h'(t) = \sum_{j=1}^{m} \Big\{ g_j'(t) \cdot \prod_{\substack{k=1 \\ k \neq j}}^{m} g_k(t) \Big\} \ . \tag{5.3.2}$$

HINT 1 for (5-3.e): Of course, the insights obtained in Sub-problem (5-3.c) and Sub-problem (5-3.d) should be used. ⌟

HINT 2 for (5-3.e): Note that the polynomials L_i and K_i have double zeros in t_j, $j \neq i$. According to Sub-problem (5-3.c) they can be written as

$$\begin{aligned} L_i(t) &= (\alpha_i t + \delta_i) \prod_{\substack{j=0 \\ j \neq i}}^{n} (t - t_j)^2 \,, \\ K_i(t) &= (\gamma_i t + \beta_i) \prod_{\substack{j=0 \\ j \neq i}}^{n} (t - t_j)^2 \,. \end{aligned} \tag{5.3.3}$$

Then determine α_i, δ_i, γ_i and β_i from (5.3.1) using the interpolation conditions and the result of Sub-problem (5-3.d). ⌟

SOLUTION of (5-3.e):

To begin with we set

$$h_i(t) := \prod_{\substack{j=0,\ldots,n \\ j\neq i}} (t-t_j)^2 \in \mathcal{P}_n \,, t \in \mathbb{R} \,,$$

and apply (5.3.2): for $t \neq t_i$, $i = 0, \ldots, n$,

$$h_i'(t) = 2\sum_{j\neq i}(t-t_j)\prod_{k\neq i,j}(t-t_k)^2 = 2h_i(t)\cdot\sum_{j\neq i}\frac{1}{t-t_j} \quad \blacktriangleright \quad q_i(t) := \frac{h_i'(t)}{h_i(t)} = 2\sum_{j\neq i}\frac{1}{t-t_j} \,.$$

The interpolation conditions (5.3.1) 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 (5.3.3) we find equations for α_i and δ_i

$$\begin{aligned} 1 = L_i(t_i) &= (\alpha_i t_i + \delta_i)h_i(t_i) \,, \\ 0 = L_i'(t_i) &= \alpha_i h_i(t_i) + (\alpha_i t_i + \delta_i)h_i'(t_i) \,, \end{aligned}$$

which leads to the linear system of equations:

$$\begin{bmatrix} t_i h_i(t_i) & h_i(t_i) \\ h_i(t_i) + t_i h_i'(t_i) & h_i'(t_i) \end{bmatrix} \begin{bmatrix} \alpha_i \\ \delta_i \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} .$$

We find the solution

$$\begin{bmatrix} \alpha_i \\ \delta_i \end{bmatrix} = \frac{1}{h_i(t_i)^2}\begin{bmatrix} -h_i'(t_i) \\ h_i(t_i) + t_i h_i'(t_i) \end{bmatrix} = \frac{1}{h_i(t_i)}\begin{bmatrix} -q_i(t_i) \\ 1 + t_i q_i(t_i) \end{bmatrix} .$$

This means

$$L_i(t) = \frac{1}{h_i(t_i)}\big(-q_i(t_i)t + 1 + t_i q_i(t_i)\big)h_i(t) \,. \tag{5.3.4}$$

The coefficients β_i and γ_i for K_i can be computed in a similar way from

$$\begin{aligned} 0 = K_i(t_i) &= (\gamma_i t_i + \beta_i)h(t_i) \,, \\ 1 = K_i'(t_i) &= \gamma_i h(t_i) + (\gamma_i t_i + \beta_i)h_i'(t_i) \,, \end{aligned}$$

which can be recast as a linear system of equations:

$$\begin{bmatrix} t_i h_i(t_i) & h_i(t_i) \\ h_i(t_i) + t_i h_i'(t_i) & h_i'(t_i) \end{bmatrix} \begin{bmatrix} \gamma_i \\ \beta_i \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} .$$

We find the solution

$$\begin{bmatrix} \gamma_i \\ \beta_i \end{bmatrix} = -\frac{1}{h_i(t_i)^2}\begin{bmatrix} h_i'(t_i) & -h_i(t_i) \\ -h_i(t_i) - t_i h_i'(t_i) & t_i h_i(t_i) \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{h_i(t_i)}\begin{bmatrix} 1 \\ -t_i \end{bmatrix} .$$

Written explicitly,

$$K_i(t) = \frac{1}{h_i(t_i)}(t-t_i)h_i(t) \,.$$

SOLUTION of (5-4.a):

Key to efficiency is the *sorting* of the data point according to their first coordinates (the nodes). This paves the way for using *binary search* for locating the relevant node interval for the evaluation.

C++11-code 5.4.1:

```
LinearInterpolant::LinearInterpolant(const Eigen::VectorXd &t,
                                     const Eigen::VectorXd &y) {
  assert(t.size() == y.size() && "Input sizes must match");
  assert(t.size() > 1 && "Must specify at least two nodes");

  // TODO: (5-4.a) Implement the constructor of the class.  t and y may
     not be
  // sorted.
  // START

  // helper lambda to transform two Eigen::VectorXd into one
  // std::vector<std::pair<double, double»
  auto VecToVecPair = [](const double &t_i, const double &y_i) -> pair {
    return std::make_pair(t_i, y_i);
  };

  // perform transformation towards std::vector<std::pair<double,
     double»
  i_points.resize(t.size());
  std::transform(t.data(), t.data() + t.size(), y.data(), i_points.begin(),
                 VecToVecPair);

  // 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);
  // END
}
```

Get it on GitLab (linearinterpolant.hpp).

We leave the binary search to the standard library function **std::**`lower_bound`.

C++11-code 5.4.2:

```
double LinearInterpolant::operator()(double x) {
  double l_x = 0.;
  // TODO: (5-4.a) Implement an efficient evaluation operator.
  // START

  // Lambda for comparison 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
```

```
15    // on the data, provided the data has a random access iterator
16    // http://www.cplusplus.com/reference/iterator/RandomAccessIterator/
17    // this is crucial here, since evaluation operators must be fast
18    // this allows to reduce the complexity,
19    // from linear complexity to a logarithmic
20    auto it = std::lower_bound(i_points.begin(), i_points.end(), x, Compare);
21
22    // Bound checks, if i = 0 and t_0! = x (we are before the first node)
23    // or if x > t_n (we are after the last node).
24    // In such cases return 0 (*it and *(it-1) would be undefined in such
          cases)
25    if ((it == i_points.begin() && it->first != x) || it == i_points.end())
26      return 0;
27
28    // Actually compute the interpolated value, dist_ratio
29    // contains the distance from t_{i-1} to x (as a ratio)
30    double dist_ratio = (x - (it - 1)->first) / (it->first - (it - 1)->first);
31    l_x = (it - 1)->second * (1 - dist_ratio) + it->second * dist_ratio;
32    // END
33    return l_x;
34  }
```

Get it on GitLab (linearinterpolant.hpp).

SOLUTION of (5-4.b):

- The constructor involves sorting a sequence of length n, which requires an effort of $O(n \log n)$ for $n \to \infty$.
- The evaluation operator has to find the node interval $[t_i, t_{i+1}]$ containing x. This can be done with a binary search at an asymptotic cost of $O(\log n)$ for $n \to \infty$.

HINT 1 for (5-5.a): Recall Euler's formula [Lecture → Eq. (5.6.1.2)] expressing trigonometric functions in terms of the complex exponential function. ⌟

SOLUTION of (5-5.a):

Let $S_k(t) := \sin(2\pi kt)$, $C_k(t) := \cos(2\pi kt)$ and $B_k(t) := \exp(2\pi\imath kt)$. Notice $B_0 \equiv C_0 \equiv 1$. Then, by the definition of the trigonometric functions through **Euler's formula**

$$S_k(t) = \frac{1}{2\imath}B_{-k}(t) - \frac{1}{2\imath}B_k\,,$$
$$C_k(t) = \frac{1}{2}B_{-k}(t) + \frac{1}{2}B_k\,.$$

We get the matrix entries by writing the basis elements of $\mathcal{B}_{\mathrm{Re}}$ in terms of the B_k.

$$\begin{aligned}
C_0(t) &= B_0(t)\,,\\
S_1(t) &= \tfrac{1}{2\imath}(\exp(2\pi\imath t) - \exp(-2\pi\imath t)) = \tfrac{1}{2\imath}(B_1(t) - B_{-1}(t))\,,\\
C_1(t) &= \tfrac{1}{2}(\exp(2\pi\imath t) + \exp(-2\pi\imath t)) = \tfrac{1}{2}(B_1(t) + B_{-1}(t))\,,\\
&\vdots\\
S_n(t) &= \tfrac{1}{2\imath}(\exp(2\pi\imath nt) - \exp(-2\pi\imath nt)) = \tfrac{1}{2\imath}(B_n(t) - B_{-n}(t))\,,\\
C_n(t) &= \tfrac{1}{2}(\exp(2\pi\imath nt) + \exp(-2\pi\imath nt)) = \tfrac{1}{2}(B_n(t) + B_{-n}(t))\,.
\end{aligned}$$

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 $2k+1$, $k = 1,\ldots,n$. Further B_{k-n} is associated with matrix row $k+1$, $k = 0,\ldots,2n$. We end up with the basis transformation matrix

$$\mathbf{S} = \begin{bmatrix}
0 & 0 & 0 & \ldots & \frac{\imath}{2} & \frac{1}{2}\\
0 & 0 & 0 & \ldots & 0 & 0\\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\
0 & \frac{\imath}{2} & \frac{1}{2} & \ldots & 0 & 0\\
1 & 0 & 0 & \ldots & 0 & 0\\
0 & -\frac{\imath}{2} & \frac{1}{2} & \ldots & 0 & 0\\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\
0 & 0 & 0 & \ldots & 0 & 0\\
0 & 0 & 0 & \ldots & -\frac{\imath}{2} & \frac{1}{2}
\end{bmatrix} \in \mathbb{C}^{2n+1,2n+1}\,.$$

HINT 1 for (5-5.b): All you need to do is to study the action of S_τ on basis functions. ⌟

SOLUTION of (5-5.b):

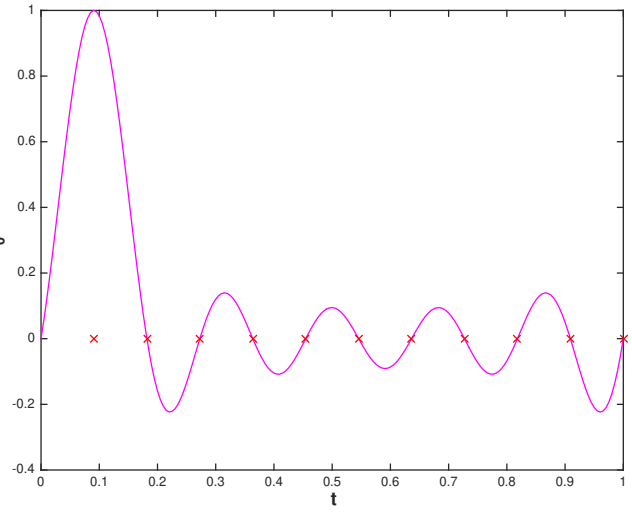
Consider $S_\tau B_k = \exp(2\pi\imath kt - 2\pi\imath k\tau) = B_k \exp(-2\pi\imath k\tau)$. If $p = \sum_{k=-n}^{n} c_k B_k(t)$ (expressed in the basis $\mathcal{B}_{exp}$), then

$$S_\tau p = \sum_{k=-n}^{n} c_k S_\tau B_k(t) = \sum_{k=-n}^{n} c_k \exp(-2\pi\imath k\tau) B_k(t).$$

The new coefficient vector is therefore $\tilde{\mathbf{c}} = [c_k \exp(-2\pi\imath k\tau)]_{k=-n}^{n}$.

SOLUTION of (5-5.c):

We have to feed unit vectors as `y`-argument to `trigpolyvalequid` and `M=1000`. Note that the function uses the equidistant points $t_j = \frac{j}{2n+1}$, $j = 0,\ldots,2n$, which does not match the node set $\mathcal{T}_n$ from (5.5.3). 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.



C++11-code 5.5.5: Plotting of the cardinal basis function b_0.

```
void plot_basis(unsigned int n) {
  // mesh size
  constexpr unsigned int M = 1e3;
  plt::figure();

  // TODO: (5-5.c) use the function trigpolyvalequid from
  // trigipcardfn.hpp to plot the cardinal basis function b_0(t) in function
  // of t for n = 5
  // START
  // basis vector e_1
  Eigen::ArrayXd e = Eigen::ArrayXd::Zero(2 * n + 1);
  e(0) = 1;
  Eigen::VectorXd y;
  trigpolyvalequid(e, M, y);

  Eigen::ArrayXd t = Eigen::ArrayXd::LinSpaced(M, 0, 1);
  Eigen::ArrayXd zer = Eigen::ArrayXd::Zero(t.size());

  // Shift function right a bit
  Eigen::ArrayXd y_shift(M);
  const unsigned int h = M / (2 * n + 1);
  y_shift << y.tail(h), y.head(M - h);

  // plotting functions using matplotlib
  plt::title("b_0(t)");
  plt::xlabel("t");
  plt::ylabel("y");
  plt::plot(t, y_shift, "r", {{"label", "b_0(t)"}});
  plt::plot(t, zer, "--");
  plt::legend();
  // END
  plt::savefig("./cx_out/basis_function.png");
}
```

Get it on GitLab (trigipcardfn.hpp).

HINT 1 for (5-5.d): Verify (5.5.4) for $S_\delta b_k$ ⌟

SOLUTION of (5-5.d):

We have (use periodicity and equidistance of the points):

$$S_\delta b_k(t_j) = b_k(t_j - \frac{1}{2n+1}) = b_k(\frac{(j-1)+1/2}{2n+1}) = b_k(t_{j-1}) = \delta_{k,j-1} = \delta_{k+1,j} = b_{k+1}(t_j).$$

The claim follows from uniqueness of interpolating trigonometric polynomials.

HINT 1 for (5-5.e): Remember the Fourier matrix [Lecture → Eq. (4.2.1.13)]. ⌟

HINT 2 for (5-5.e): Examine [Lecture → Eq. (5.6.3.3)]. However, take into account that the interpolation nodes for this problem are slightly different from those used in those formulas. ⌟

SOLUTION of (5-5.e):

The trigonometric interpolation problem with respect to the basis $\mathcal{B}_{\exp} = \{B_{-n}, \ldots, B_n\}$ from (5.5.2) leads to the linear system of equations

$$\sum_{k=-n}^{n} c_k B_k(t_j) = y_j \,, \quad j = 0, \ldots, 2n \,,$$

for given values $y_j \in \mathbb{C}$. The related interpolation matrix is

$$\mathbf{A} := \begin{bmatrix} B_{-n}(t_0) & \ldots & B_n(t_0) \\ \vdots & & \vdots \\ B_{-n}(t_{2n}) & \ldots & B_n(t_{2n}) \end{bmatrix} \in \mathbb{C}^{2n+1,2n+1} \,.$$

The indexing of the exponential basis functions as defined in (5.5.2) runs from $-n, \ldots, n$ and has to be shifted to match C++ *indexing* of matrix entries:

$$\begin{aligned} (\mathbf{A})_{j,\ell} = B_{\ell-n}(t_j) &= \exp(2\pi\imath \frac{(j+1/2)(\ell-n)}{2n+1}) = \exp(2\pi\imath \frac{j\ell + \frac{1}{2}\ell - jn - \frac{1}{2}n}{2n+1}) \\ &= \underbrace{\exp(2\pi\imath \frac{j\ell}{2n+1})}_{=\left(\overline{\mathbf{F}}_{2n+1}\right)_{j,\ell}} \exp(\pi\imath \frac{\ell}{2n+1}) \exp(-2\pi\imath \frac{jn}{2n+1}) \exp(-\pi\imath \frac{1}{2n+1}) \,, \end{aligned}$$

where $\mathbf{F}_{2n+1} \in \mathbb{C}^{2n+1,2n+1}$ is the Fourier matrix. So we find that the interpolation matrix agrees with a Fourier matrix to which some row and column scaling is applied:

$$\mathbf{A} = \exp(-\pi\imath \frac{n}{2n+1})\, \mathbf{D}_1 \overline{\mathbf{F}}_{2n+1} \mathbf{D}_2 \,, \tag{5.5.7}$$

$$\mathbf{D}_1 := \operatorname{diag}\left(\left[e^{-2\pi\imath \frac{jn}{2n+1}} \right]_{j=0,\ldots,2n} \right) \quad , \quad \mathbf{D}_2 := \operatorname{diag}\left(\left[e^{\pi\imath \frac{\ell}{2n+1}} \right]_{\ell=0,\ldots,2n} \right) .$$

HINT 1 for (5-5.f): Recall [Lecture → Lemma 4.2.1.14].

SOLUTION of (5-5.f):

We exploit the factorization (5.5.7), remember from linear algebra $(\mathbf{XY})^{-1} = \mathbf{Y}^{-1}\mathbf{X}$ for all regular square matrices $\mathbf{X}$, $\mathbf{Y}$, and use the fact that $\exp(\imath\xi)^{-1} = \exp(-\imath\xi)$ for all $\xi \in \mathbb{R}$, and get

$$\begin{aligned}\mathbf{A}^{-1} &= e^{\pi\imath\frac{n}{2n+1}}\,\mathbf{D}_2^{-1}\left(\overline{\mathbf{F}}_{2n+1}\right)^{-1}\mathbf{D}_1^{-1}\\ &= e^{\pi\imath\frac{n}{2n+1}}\,\operatorname{diag}\left(\left[e^{-\pi\imath\frac{\ell}{2n+1}}\right]_{\ell=0,\ldots,2n}\right)\,\frac{1}{2n+1}\mathbf{F}_{2n+1}\,\operatorname{diag}\left(\left[e^{2\pi\imath\frac{jn}{2n+1}}\right]_{j=0,\ldots,2n}\right), \qquad (5.5.8)\end{aligned}$$

where we also used

Lemma [Lecture → Lemma 4.2.1.14]. Properties of Fourier matrices

The scaled ***Fourier-matrix*** $\frac{1}{\sqrt{n}}\mathbf{F}_n$ *is unitary (→ [Lecture → Def. 6.3.1.2]):* $\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^{\mathrm{H}} = \frac{1}{n}\overline{\mathbf{F}}_n$.

Based on

$$\mathbf{F}_{2n+1} = \left[\exp(-2\pi\imath\frac{j\ell}{2n+1})\right]_{j,\ell=0}^{2n} \in \mathbb{C}^{2n+1,2n+1},$$

we can recast the formula for $\mathbf{A}^{-1}$ as

$$\begin{aligned}\mathbf{A}^{-1} &= \frac{1}{2n+1}\operatorname{diag}\left(\left[e^{\pi\imath\frac{n-j}{2n+1}}\right]_{j=0,\ldots,2n}\right)\left[e^{-2\pi\imath\frac{j\ell}{2n+1}}\right]_{j,\ell=0}^{2n}\operatorname{diag}\left(\left[e^{2\pi\imath\frac{\ell n}{2n+1}}\right]_{\ell=0,\ldots,2n}\right)\\ &= \frac{1}{2n+1}\left[e^{-2\pi\imath\frac{\frac{1}{2}(n-j)-j\ell+\ell n}{2n+1}}\right]_{j,\ell=0}^{2n} = \frac{1}{2n+1}\mathbf{A}^{\mathrm{H}},\end{aligned}$$

by matching with

$$(\mathbf{A})_{j,\ell} = \exp(2\pi\imath\frac{(j+1/2)(\ell-n)}{2n+1}).$$

HINT 1 for (5-5.g): Use that b_k has the coefficient vector $\mathbf{X}\mathbf{e}_{k+1}$ with respect to $\mathcal{B}_{\exp}$, where $\mathbf{e}_j \in \mathbb{C}^{2n+1}$, $j = 1, \ldots, 2n+1$, is the j-th unit vector, $\mathbf{X}$ the matrix found in (5-5.f). ⌟

SOLUTION of (5-5.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:

$$\begin{aligned} b_k(t) = \sum_{j=-n}^{n} c_j B_j(t) &= \sum_{j=-n}^{n} \frac{1}{2n+1}(\exp(2\pi\imath(k+1)\frac{j}{2n+1})\exp(-\pi\imath\frac{j}{2n+1})\exp(-2\pi j\imath t) \\ &= \frac{1}{2n+1}\sum_{j=-n}^{n}(\exp(2\pi\imath(k+1/2)\frac{j}{2n+1} - 2\pi j\imath t) \end{aligned}$$

Using Euler's formula

$$(n+\tfrac{1}{2})b_k(t) = \tfrac{1}{2} + \sum_{j=1}^{n}\cos(2\pi j(t-t_k))\,, \quad t \in \mathbb{R}\,. \tag{5.5.9}$$

This can be simplified further using the geometric sum formula and a few manipulations:

$$\begin{aligned} &\tfrac{1}{2} + \sum_{j=1}^{n}\cos(2\pi j(t-t_k)) \\ =&Re(\tfrac{1}{2} + \sum_{j=1}^{n}\exp(2\pi\imath j(t-t_k))) \\ =&Re(-\tfrac{1}{2} + \frac{1-\exp(2\pi\imath(n+1)(t-t_k))}{1-\exp(2\pi\imath(t-t_k))}) \\ =&Re(-\tfrac{1}{2} + \frac{\exp(-\pi\imath(t-t_k)) - \exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))}) \\ =&Re(-\tfrac{1}{2}\frac{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k)) - 2\exp(-\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))} \\ &\quad - \frac{\exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))}) \\ =&Re(\tfrac{1}{2}\frac{\exp(-\pi\imath(t-t_k)) + \exp(\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))} - \frac{\exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))}) \\ =&\tfrac{1}{2}Re(\frac{\cos(\pi(t-t_k))}{-\sin(\pi(t-t_k))}) - Re(\frac{\exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(t-t_k))}{\imath\sin(\pi(t-t_k))}) \\ =&\frac{\sin(2\pi(n+1)(t-t_k) - \pi(t-t_k))}{\sin(\pi(t-t_k))} \\ =&\frac{\sin(2\pi(n+1/2)(t-t_k))}{\sin(\pi(t-t_k))} \end{aligned}$$

Hence:

$$(n+\tfrac{1}{2})b_k(t) = \frac{\sin(2\pi(n+\tfrac{1}{2})t)}{2\sin(\pi(t-t_k))}\,, \quad t \in \mathbb{R}\,. \tag{5.5.10}$$

SOLUTION of (5-5.h):

The following code is not optimal. A better, but more complicated implementation could rely on FFT.

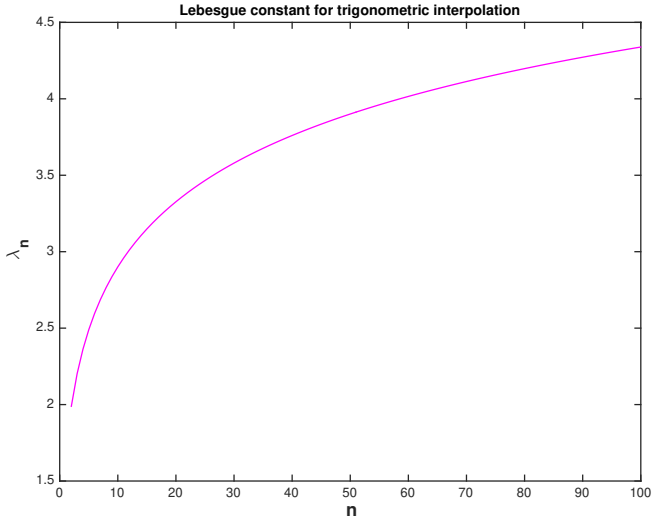
C++11-code 5.5.12: Computation of Lebesgue constant λ for trigonometric interpolation.

```
double triglpL(std::size_t n) {
  double lambda = 0.;
  // TODO: (5-5.h) write a function that approximately computes the
  // Lebesgue constant λ(n)
  // START
  // Nodes where to evaluate
  Eigen::ArrayXd t = Eigen::ArrayXd::LinSpaced(1e4, 0, 1);
  // Will contain value of funtion at all nodes
  Eigen::ArrayXd s = Eigen::ArrayXd::Zero(t.size());
  // Sum over all basis functions
  for (unsigned int k = 0; k <= 2 * n; ++k) {
    // Compute the b_k using formula (6.5.8)
    Eigen::ArrayXd b = Eigen::ArrayXd::Zero(t.size());
    const double tk = (k + 0.5) / (2 * n + 1.);
    b = sin(M_PI * (2 * n + 1) * (t - tk)) / sin(M_PI * (t - tk));

    s += b.cwiseAbs();
  }
  // Find max and rescale
  lambda = s.maxCoeff() / (2 * n + 1);
  // END
  return lambda;
}
```

Get it on GitLab (trigipcardfn.hpp).

Graph of $n \mapsto \lambda(n)$



SOLUTION of (5-5.i):

The asymptotic behavior of $\lambda(n)$ is $O(\log n)$, see E. W. CHENEY AND T. J. RIVLIN, *A note on some Lebesgue constants*, Rocky Mountain J. Math., 6 (1976), pp. 435–439.

HINT 1 for (5-6.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 (5-6.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.

SOLUTION of (5-6.j):

C++ code 5.6.13: Implementation of `plotquadspline()`

```
void plotquadspline(const std::string &filename) {
  Eigen::VectorXd mesh = Eigen::VectorXd::LinSpaced(9, .1, .9);
  auto f = [](Eigen::VectorXd t) { return (2 * M_PI * t).array().sin().exp(); };

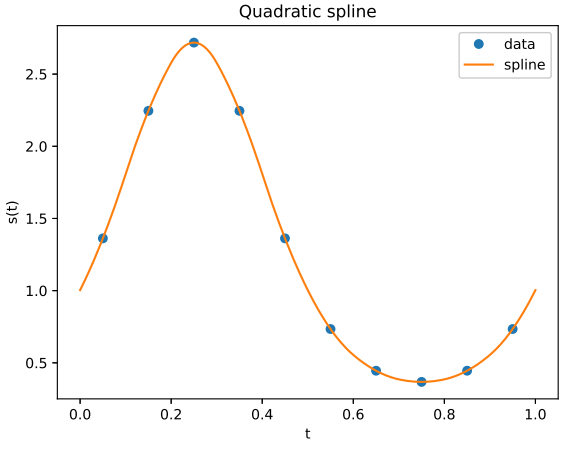
  plt::figure();
  // TODO: (5-6.j) plot the quadratic spline for the function f based on
  // the intervals defined in t. Plot also the data (interpolation) points.

  // START
  // The interpolation points are the midpoints of the intervals
  Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(10, .05, .95);
  Eigen::VectorXd y = f(t);
  // plot data points
  plt::plot(t, y, "o", {{"label", "data"}});

  // std::cout << y <<std::endl;
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(200, 0, 1);
  Eigen::VectorXd spline_val = quadspline(mesh, y, x);
  // std::cout << spline_val <<std::endl;

  plt::plot(x, spline_val, {{"label", "spline"}});
  plt::legend();
  plt::title("Quadratic spline");
  plt::xlabel("t");
  plt::ylabel("s(t)");
  // END
  plt::savefig("./cx_out/" + filename + ".png");
}
```

Get it on GitLab (quadsplines.hpp).



◁ plot of the quadratic spline s.

HINT 1 for (5-6.k): We expect an asymptotic dependence of the error on n of the form $E_n = O(n^{\alpha})$. ⌟

SOLUTION of (5-6.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

◁ Tabulated spline approximation errors in the maximum norm. For larger values of n we get $E_{2j} : E_{2j+1} \approx 8$, which indicates and asymptotic behavior of the error according to the law

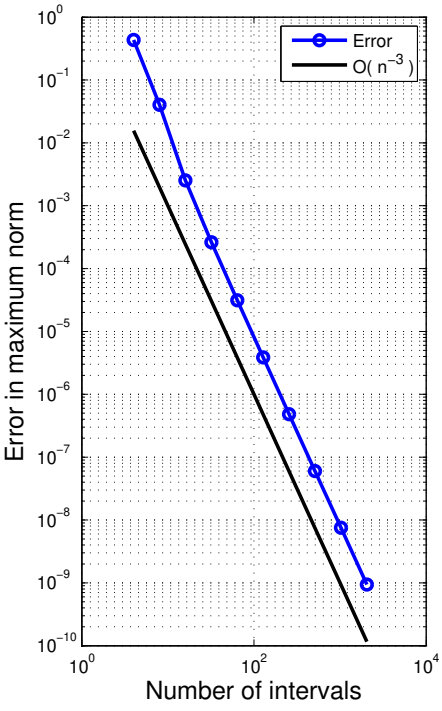
$$E_n = O(n^{-3}) \ .$$

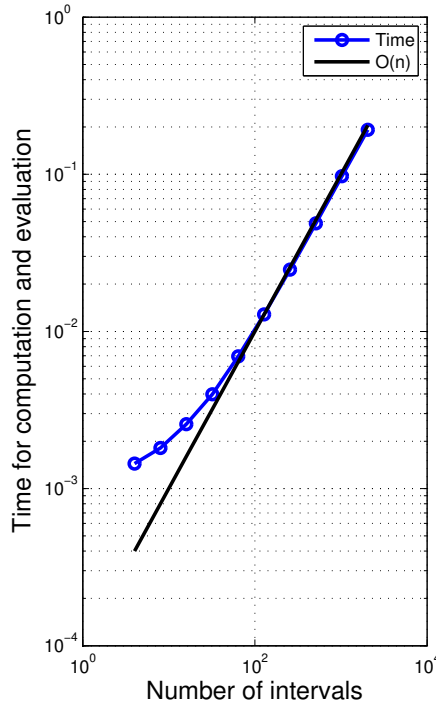
C++ code 5.6.14: Implementation of `qsq_error()`

```
std::vector<double> qsp_error(unsigned int q) {
  std::vector<double> Err;
  constexpr unsigned int N = 10000;
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N, 0, 1);
  auto f = [](Eigen::VectorXd t) { return (2 * M_PI * t).array().sin().exp(); };
  // TODO: (5-6.k) compute L^infty errors for all n = 2, 4, ..., 2^q
  // START
  const unsigned int n_max = std::pow(2, q);
  Eigen::VectorXd f_exact = f(x);

  for (unsigned int n = 2; n <= n_max; n <<= 1) {
    Eigen::VectorXd t =
        Eigen::VectorXd::LinSpaced(n - 1, 1.0 / n, 1.0 - 1.0 / n);
    // The interpolation points are the midpoints of the intervals
    Eigen::VectorXd interp =
        Eigen::VectorXd::LinSpaced(n, .5 / n, 1.0 - .5 / n);
    Eigen::VectorXd y = f(interp);
    Eigen::VectorXd f_spline = quadspline(t, y, x);
    // compute max error
    Err.push_back((f_spline - f_exact).cwiseAbs().maxCoeff());
  }
  // END
  return Err;
}
```

Get it on GitLab (quadsplines.hpp).





◁ 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.

HINT 1 for (5-6.b): 1-periodicity means that $s(0) = s(1)$ and $s'(0) = s'(1)$. ⌟

SOLUTION of (5-6.b):

As in [Lecture → Section 5.4.1] we resort to a counting argument:

- 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 $2n - n = n$.
- Alternatively, we know from [Lecture → Thm. 5.4.1.2] that $\dim \mathcal{S}_{2,\mathcal{M}} = n+2$. The extra constraints $s(0) = s(1)$ and $s'(0) = s'(1)$ absorb two degrees of freedom, which leaves us with n:

$$\dim\{s \in \mathcal{S}_{2,\mathcal{M}} : s(0) = s(1),\ s'(0) = s'(1)\} = n \ .$$

SOLUTION of (5-6.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 C^0-continuity for free by fixing the coefficients d_j. However the derivatives do not necessarily match, which means that the representation (5.6.2) does not guarantee the continuity of the derivative.

HINT 1 for (5-6.d): By [Lecture → Def. 5.4.1.1] we know $\mathcal{S}_{2,\mathcal{M}} \subset C^1([0,1])$, which provides linear constraints at the nodes, analogous to [Lecture → Eq. (5.4.2.7)] for cubic splines. Also take into account 1-periodicity. ⌟

SOLUTION of (5-6.d):

Taking into account the interpolation conditions (5.6.1), we can plug $t = \frac{1}{2}(t_j + t_{j-1})$ into (5.6.2) and set the values equal to y_j. We obtain $\tau = 1/2$ and the following conditions:

$$\frac{1}{4}d_j + c_j + \frac{1}{4}d_{j-1} = y_j, \quad j = 1, ..., n. \tag{5.6.3}$$

We obtain conditions on d_j by matching the derivatives at the interfaces. The derivative of the quadratic spline can be computed from (5.6.2), 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 $j = 1, \ldots, n$, if we define $t_{n+1} = t_1 + 1$ and $c_{n+1} = c_1$). Simplifying for d_j we obtain:

$$d_j = \frac{2\frac{c_j}{\Delta_j} + 2\frac{c_{j+1}}{\Delta_{j+1}}}{\frac{1}{\Delta_j} + \frac{1}{\Delta_{j+1}}} = 2\frac{c_j\Delta_{j+1} + c_{j+1}\Delta_j}{\Delta_j + \Delta_{j+1}} = 2\frac{c_j(t_{j+1} - t_j) + c_{j+1}(t_j - t_{j-1})}{t_{j+1} - t_{j-1}}, \tag{5.6.4}$$

for $j = 1, \ldots, n$. Plugging this expression into (5.6.3), we get the following system of equations:

$$\frac{1}{2}\frac{c_{j+1}(t_j - t_{j-1}) + c_j(t_{j+1} - t_j)}{t_{j+1} - t_{j-1}} + c_j + \frac{1}{2}\frac{c_j(t_{j-1} - t_{j-2}) + c_{j-1}(t_j - t_{j-1})}{t_j - t_{j-2}} = y_j, \qquad j = 1, \ldots, n, \tag{5.6.5}$$

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

$$\underbrace{\left\{\frac{1}{2}\frac{t_j - t_{j-1}}{t_j - t_{j-2}}\right\}}_{=:A_{j-1}} c_{j-1} + \underbrace{\left\{\frac{1}{2}\frac{t_{j+1} - t_j}{t_{j+1} - t_{j-1}} + 1 + \frac{1}{2}\frac{t_{j-1} - t_{j-2}}{t_j - t_{j-2}}\right\}}_{=:B_j} c_j + \underbrace{\left\{\frac{1}{2}\frac{t_j - t_{j-1}}{t_{j+1} - t_{j-1}}\right\}}_{=:C_{j+1}} c_{j+1} = y_j, \tag{5.6.6}$$

for $j = 1, \ldots, n$. If we have an equidistant knot set with $t_j - t_{j-1} = 1/n$, this simplifies to

$$\frac{1}{4}c_{j-1} + \frac{3}{2}c_j + \frac{1}{4}c_{j+1} = y_j, \quad j = 1, \ldots, n. \tag{5.6.7}$$

The system of equations in matrix form looks as follows:

$$\begin{bmatrix} B_1 & C_2 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & A_0 \\ A_1 & B_2 & C_3 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_3 & C_4 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & A_{n-2} & B_{n-1} & C_n \\ C_{n+1} & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & A_{n-1} & B_n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}. \tag{5.6.8}$$

SOLUTION of (5-6.e):

Denote by $\mathbf{M}$ the system matrix from (5.6.8). Then

$$\mathbf{M} = \begin{bmatrix} B_1 - 1 & C_2 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ A_1 & B_2 & C_3 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_3 & C_4 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & A_{n-2} & B_{n-1} & C_n \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & A_{n-1} & B_n - A_0 C_{n+1} \end{bmatrix} + \mathbf{u}\mathbf{v}^\top ,$$

$$\text{with} \quad \mathbf{u} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ C_{n+1} \end{bmatrix} \in \mathbb{R}^n \quad , \quad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ A_0 \end{bmatrix} \in \mathbb{R}^n .$$

SOLUTION of (5-6.f):

The linear system to be solved features a system matrix which is a rank-1 perturbation 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. 3.3.3.26]. 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.

Remark. 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.

Key is the use of the Sherman-Morrison-Woodbury formula from [Lecture → Lemma 2.6.0.21]. We use it in the form of 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 (5-6.e). Besides computing $\mathbf{M}^{-1}\mathbf{y}$ and $\mathbf{M}^{-1}\mathbf{u}$ with effort $O(n)$ it merely involves cheap vector operations.

SOLUTION of (5-6.g):

We use an auxiliary function that, among others, computes the length of knot intervals:

C++ code 5.6.9: Implementation of `increments()`

```
std::pair<Eigen::VectorXd, Eigen::VectorXd> increments(
    const Eigen::VectorXd &t) {
  Eigen::VectorXd dt, ddt;
  // TODO: (5-6.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
  Eigen::VectorXd ext_t(n + 3);
  ext_t << t(n - 2) - 1, 0, t, 1, 1 + t(0);
  // Increments in t
  dt = ext_t.tail(n + 2).array() - ext_t.head(n + 2).array();
  ddt = ext_t.tail(n + 1).array() - ext_t.head(n + 1).array();
  // END

  return std::make_pair(dt, ddt);
}
```

Get it on GitLab (quadsplines.hpp).

Use EIGEN's built-in *sparse solver* to solve (5.6.8). It will achieve optimal asymptotic computational complexity $O(n)$ for $n \to \infty$. The use of a sparse solver is mandatory.

C++ code 5.6.10: Implementation of `compute_c()`

```
Eigen::VectorXd compute_c(const Eigen::VectorXd &t, const Eigen::VectorXd &y) {
  const unsigned int n = y.size();  // number of intervals
  Eigen::VectorXd c(n);
  // TODO: (5-6.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<Eigen::VectorXd, Eigen::VectorXd> p = increments(t);
  Eigen::VectorXd dt = p.first;
  Eigen::VectorXd ddt = p.second;

  // Note: A_0 = A_n and C_1 = C_{n+1}
  Eigen::VectorXd A = dt.tail(n).cwiseQuotient(2 * ddt.tail(n));
  Eigen::VectorXd C = dt.head(n).cwiseQuotient(2 * ddt.head(n));

  std::vector<Eigen::Triplet<double>> triplets(3 * n);

  // Build matrix as triplets
  for (unsigned int j = 0; j < n - 1; ++j) {
```

```
24      triplets.push_back(Eigen::Triplet<double>(j, j, A(j) + C(j) + 1));
25      triplets.push_back(Eigen::Triplet<double>(j, j + 1, C(j + 1)));
26      triplets.push_back(Eigen::Triplet<double>(j + 1, j, A(j)));
27    }
28    triplets.push_back(
29        Eigen::Triplet<double>(n - 1, n - 1, A(n - 1) + C(n - 1) + 1));
30    triplets.push_back(Eigen::Triplet<double>(n - 1, 0, C(0)));
31    triplets.push_back(Eigen::Triplet<double>(0, n - 1, A(n - 1)));
32
33    Eigen::SparseMatrix<double> M(n, n);
34    M.setFromTriplets(triplets.begin(), triplets.end());
35    M.makeCompressed();
36    // solve linear system
37    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
38    solver.compute(M);
39    c = solver.solve(y);
40    // END
41    return c;
42  }
```

Get it on GitLab (quadsplines.hpp).

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].

SOLUTION of (5-6.h):

We can simply rely on the formula (5.6.4).

C++ code 5.6.11: Implementation of `compute_d()`

```
2  Eigen::VectorXd compute_d(const Eigen::VectorXd &c, const Eigen::VectorXd &t) {
3    const unsigned int n = c.size();  // number of intervals
4    Eigen::VectorXd d_ext(n + 1);
5    // TODO: (5-6.h) compute coefficients d_j for j = 0...n
6    // Hint: periodic conditions give d_0 = d_n
7    // START
8    std::pair<Eigen::VectorXd, Eigen::VectorXd> p = increments(t);
9    Eigen::VectorXd dt = p.first;
10   Eigen::VectorXd ddt = p.second;
11
12   // extend c periodically for vectorization
13   Eigen::VectorXd c_ext(c.size() + 1);
14   c_ext << c, c(0);
15
16   // coefficients d_j for j = 1...n
17   Eigen::VectorXd d =
18       c.cwiseProduct(dt.tail(n)) + c_ext.tail(n).cwiseProduct(dt.segment(1, n));
19   d = 2 * d.cwiseQuotient(ddt.tail(n));
20   // extend d by periodic condition
21   d_ext << d(n - 1), d;
22   // END
23   return d_ext;
24 }
```

Get it on GitLab (quadsplines.hpp).

SOLUTION of (5-6.i):

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 (5.6.2).

C++ code 5.6.12: Implementation of `quadspline()`

```
Eigen::VectorXd quadspline(const Eigen::VectorXd &t, const Eigen::VectorXd &y,
                           const Eigen::VectorXd &x) {
  Eigen::VectorXd fval(x.size());

  // TODO: (5-6.i) 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
  Eigen::VectorXd c = compute_c(t, y);
  const unsigned int n = y.size();  // number of intervals

  std::pair<Eigen::VectorXd, Eigen::VectorXd> p = increments(t);
  Eigen::VectorXd dt = p.first;
  Eigen::VectorXd ddt = p.second;

  Eigen::VectorXd d_ext = compute_d(c, t);
  // extend t for vectorization
  Eigen::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)) {
      const double tau = (x(i) - t_ext(j - 1)) / dt(j);
      const 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;
}
```

Get it on GitLab (quadsplines.hpp).

In fact, this implementation leaves ample room for improvement. For instance, a binary search for the right knot interval using the C++ library function **std::**`lower_boundf()` could reduce the complexity to $O(n \log n)$ for $n \to \infty$.

HINT 1 for (5-7.a): Recall the considerations in the beginning of [Lecture → Section 5.3.3.2]. ⌟

SOLUTION of (5-7.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 $\left(\mathbf{y}^{(j)}\right)_1 = \cdots = \left(\mathbf{y}^{(j)}\right)_j = 0$ and $\left(\mathbf{y}^{(j)}\right)_{j+1} = \cdots = \left(\mathbf{y}^{(j)}\right)_{n=1} = 1$. 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 $k = 1, \ldots, n-1$, belong to one of these two intervals, therefore $\left(I_{\mathcal{T}}(\mathbf{y}^{(j)})\right)'(t_k)$

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.

SOLUTION of (5-7.b):

Every Cartesian coordinate vector of $\mathbb{R}^{n+1}$ can be written as a linear combination of the $\mathbf{y}^{(j)}$s:

$$\mathbf{e}^{(j)} := (\underbrace{0,\ldots,0}_{j},1,0,\ldots,0)^T = \mathbf{y}^{(j)} - \mathbf{y}^{(j+1)}, \quad \forall\, j = 0,\ldots,n-1, \quad \mathbf{e}^{(n)} := (\underbrace{0,\ldots,0}_{n},1)^T = \mathbf{y}^{(n)} ,$$

$$\Rightarrow \quad \mathbb{R}^{n+1} = \text{span}\{\mathbf{e}^{(j)},\ j = 0,\ldots,n\} \subseteq \text{span}\{\mathbf{y}^{(j)},\ j = 0,\ldots,n\} \subseteq \mathbb{R}^{n+1} ,$$

$$\Rightarrow \quad \text{span}\{\mathbf{y}^{(j)},\ j = 0,\ldots,n\} = \mathbb{R}^{n+1} .$$

SOLUTION of (5-7.c):

Thanks to the result of Sub-problem (5-7.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

$$\big(I_{\mathcal{T}}(\mathbf{y})\big)'(t_k) \overset{\text{linearity of } I_{\mathcal{T}}}{=} \sum_{j=0}^{n} c_j \big(I_{\mathcal{T}}(\mathbf{y}^{(j)})\big)'(t_k) \overset{\text{Sub-problem (5-7.a)}}{=} 0\,, \quad k = 1, \ldots, n-1\,.$$

HINT 1 for (5-8.a): Remember the power series for sin:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{\ell=0}^{\infty} \frac{(-1)^\ell x^{2\ell+1}}{(2\ell+1)!}\,, \quad x \in \mathbb{R}\,. \tag{5.8.3}$$

SOLUTION of (5-8.a):

Using the power series for $\sin$ (5.8.3) 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

$$\sum_{\ell=L+1}^{\infty} \frac{(-1)^\ell \pi^{2\ell+1} k^{-2\ell}}{(2\ell+1)!} \le C k^{-2(L+1)} \quad \text{for} \quad C > 0 \quad \text{independent of} \quad k \ .$$

SOLUTION of (5-8.b):

C++ code 5.8.4: Implementation of `extrapolate_to_pi()`

```
2  double extrapolate_to_pi(const unsigned int k) {
3    double pi = 0.0;
4
5    // TODO (5-8.b): Use the Aitken-Neville scheme to approximate
6    // pi by extrapolation.
7    // Hint: You can use the constant M_PI and Eigen's sin()
8    // when calculating s_j for j = 2,...,k.
9
10   // START
11
12   // Data for interpolation.
13   // Note: We are using eigen arrays because they allow for coefficient-wise
14   // operations while eigen matrices/vectors are used for linear algebra
15   // operations.
16   Eigen::ArrayXd J = Eigen::ArrayXd::LinSpaced(k - 1, 2, k);
17   Eigen::ArrayXd T = 1.0 / J;   // t_j = 1/j
18   Eigen::ArrayXd S = J * sin(M_PI * T);
19
20   // Aitken-Neville scheme for evaluating the interpolating
21   // polynomial of (t_j,s_j) at x = 0.
22   for (int l = 0; l < S.size(); l++) {
23     for (int m = l - 1; m >= 0; m--) {
24       S(m) = S(m + 1) + (S(m + 1) - S(m)) * (0.0 - T(l)) / (T(l) - T(m));
25     }
26   }
27   // Now S(0) has contributions from all data points.
28   pi = S(0);
29   // END
30
31   return pi;
32 }
```

Get it on GitLab (extrapolpi.hpp).

SOLUTION of (5-8.c):

C++ code 5.8.5: Implementation of `plotExtrapolationError()`

```
void plotExtrapolationError(const unsigned int kmax) {
  plt::figure();
  // TODO (5-8.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 and 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
  Eigen::VectorXd K = Eigen::VectorXd::LinSpaced(kmax - 1, 2, kmax);
  Eigen::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;
  }

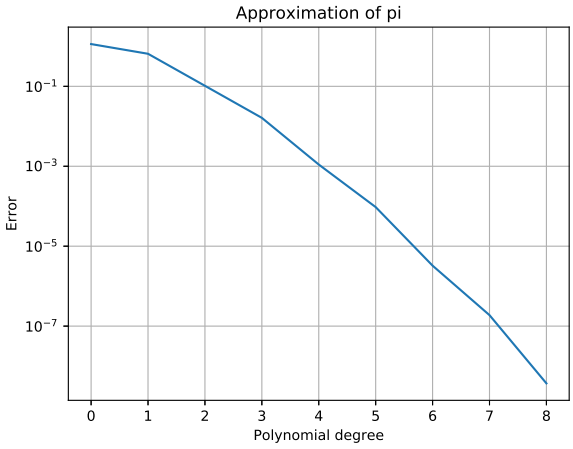
  // Number of data points is k-1, so polynomial degree is k-2.
  Eigen::VectorXd Deg = K - 2 * Eigen::VectorXd::Ones(kmax - 1);
  plt::semilogy(Deg, Err);
  plt::title("Approximation of pi");
  plt::xlabel("Polynomial degree");
  plt::ylabel("Error");
  plt::grid();
  // END

  std::string path = "./cx_out/pi_error_" + std::to_string(kmax) + ".png";
  plt::savefig(path);
}
```

Get it on GitLab (extrapolpi.hpp).

The tabulated errors should be:

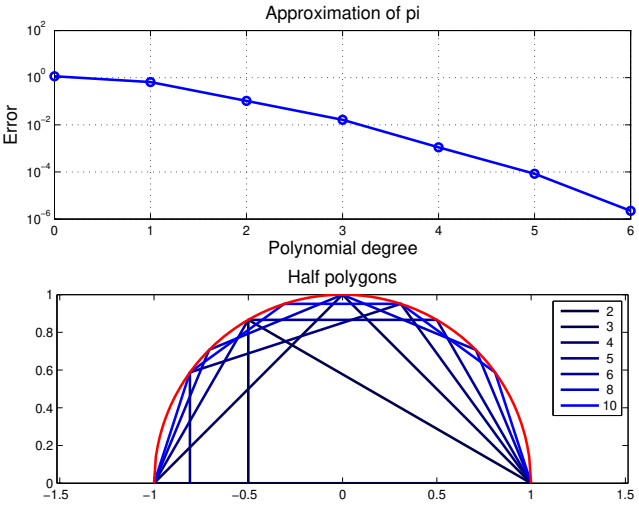
```
 num. data points, approximation of pi, error:
1                2.00000000           1.1415926
2                3.79422863           0.6526359
3                3.24473110           0.1031384
4                3.12532464           0.0162680
5                3.14048823           0.0011044
6                3.14167629           0.0000836
7                3.14159488           0.0000022
```



Linear-logarithmic plot of error in the approximation of π by extrapolation.

HINT 1 for (5-8.d): Study [Lecture → § 6.2.2.9] to learn how to determine the type of convergence from error data. ⌟

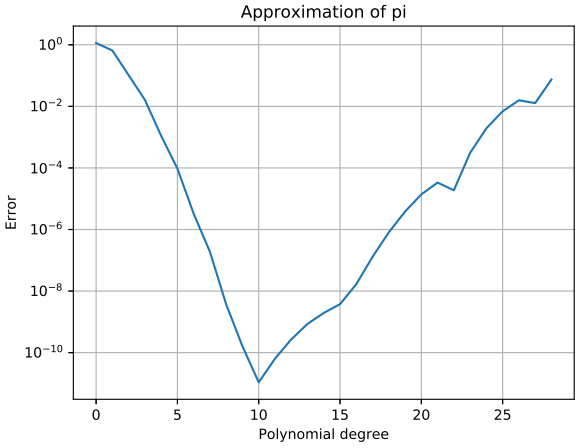
SOLUTION of (5-8.d):



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.

SOLUTION of (5-8.e):



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.

SOLUTION of (5-9.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:



$$\dim V_0 = \dim V_{-1} - (n-1) = 2n+1 \ .$$

- Another continuity condition for the first derivative is imposed in the case of the space V_1. This introduces another $n-1$ constraints.



$$\dim V_1 = \dim V_0 - (n-1) = n+2 \ .$$

- Further $n-1$ continuity conditions have to take into account in the case of V_2:



$$\dim V_2 = \dim V_1 - (n-1) = 3 = \dim \mathcal{P}_2 \ .$$

  In fact, we have $V_2 = \mathcal{P}_2$!

SOLUTION of (5-9.b):

Spline spaces according to [Lecture → Def. 5.4.1.1] are piecewise polynomial spaces with maximal smoothness that do not collapse to spaces of simple polynomials:



$$V_1 = \mathcal{S}_{2,\mathcal{M}} \, .$$

SOLUTION of (5-9.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).

SOLUTION of (5-9.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 $b = 2$.

SOLUTION of (5-9.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.

HINT 1 for (6-1.a): Look for a suitable estimate in [Lecture → Section 6.6.1].

SOLUTION of (6-1.a):

Equation [Lecture → Eq. (6.6.1.9)] says (we write $f'' := f^{(2)}$ for the second derivative):

$$\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)} \leq \frac{1}{2n^2} \|f''\|_{L^\infty(I)} \,, \tag{6.1.2}$$

because the meshwidth is $h = 1/n$. Thus, the convergence is quadratic, i.e. algebraic with order 2:

$$\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)} = O(n^{-2}) \quad \text{for} \quad n \to \infty \,. \tag{6.1.3}$$

SOLUTION of (6-1.j):

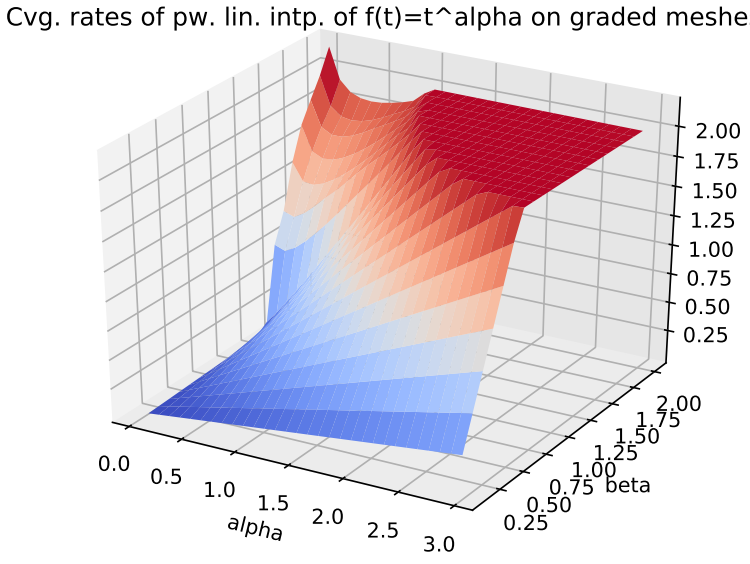
C++ code 6.1.10: Implementation of `testcvgGradedMesh()`

```cpp
void testcvgGradedMesh() {
  plt::figure();
  // TODO: (6-1.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
  constexpr unsigned int n_alphas = 30;
  Eigen::VectorXd alpha = Eigen::VectorXd::LinSpaced(n_alphas, 0.05, 2.95);
  constexpr unsigned int n_betas = 20;
  Eigen::VectorXd beta = Eigen::VectorXd::LinSpaced(n_betas, 0.1, 2.0);

  // ConvRates(i,j) = convergence rate using beta(i) and alpha(j).
  Eigen::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).
  Eigen::MatrixXd alphaGrid = alpha.replicate(1, n_betas).transpose();
  Eigen::MatrixXd betaGrid = beta.replicate(1, n_alphas);

  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");
  // END
  plt::savefig("./cx_out/alphabeta.png");
}
```

Get it on GitLab (gradedmeshes.hpp).



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 (6-1.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.

HINT 1 for (6-1.b): Notice that I is a closed interval and check the continuity of the derivatives in the endpoints of I. ⌟

SOLUTION of (6-1.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.

SOLUTION of (6-1.c):

For efficient implementation it is essential to make use of the fact that the the nodes passed in `t` are sorted.

C++ code 6.1.4: Implementation of `pwlintpMaxError()`

```
2  template <typename FUNCTION>
3  double pwlintpMaxError(FUNCTION &&f, const Eigen::VectorXd &t) {
4    constexpr unsigned int N =
5        1e5;              // Sampling resolution for approximating L-infinty norm.
6    double maxerr = 0.0;  // L-infinity error so far.
7
8    // TODO: (6-1.c) Approximate the maximum norm of (f - If) on [0,1],
9    // where If is the piecewise linear interpolation of f on the mesh t.
10   // START
11   // We iterate over the sampling points x = 0, 1/N, 2/N, ... 1.
12   // For each x we need to interpolate f in the correct mesh interval.
13   // We are currently in the interval [tj, tj1] = [0,t(1)].
14   unsigned int j = 0;
15   double tj = t[j], tj1 = t[j + 1];
16   double fj = f(tj), fj1 = f(tj1);
17   for (unsigned int i = 0; i <= N; ++i) {
18     // We are currently in the mesh interval [tj, tj1].
19     double x = (1.0 * i) / N;
20     // If x is outside the current mesh interval,
21     // then find the interval containing x.
22     while (x > tj1) {  // x >= tj always satisfied.
23       tj = tj1;
24       fj = fj1;
25       j++;
26       tj1 = t[j + 1];
27       fj1 = f(tj1);
28     }
29     // Now we have tj <= x <= tj1.
30
31     // Evaluate linear interpolant [tj,tj1] at x.
32     double lfx = fj + (fj1 - fj) / (tj1 - tj) * (x - tj);
33     // Update L-infinity error on [0,x]
34     double err = std::abs(f(x) - lfx);
35     if (err > maxerr) {
36       maxerr = err;
37     }
38   }
39   // END
40   return maxerr;
41 }
```

Get it on GitLab (gradedmeshes.hpp).

SOLUTION of (6-1.d):

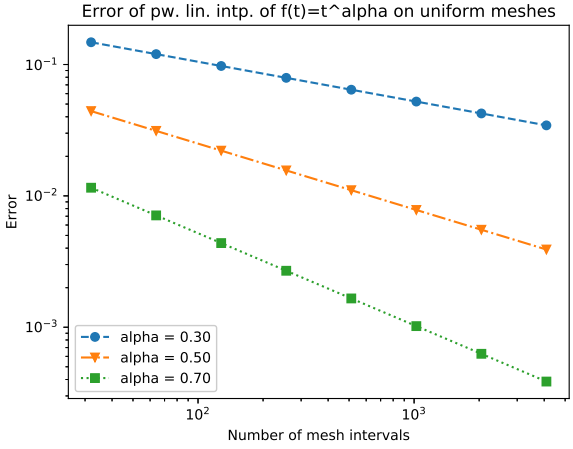
C++ code 6.1.5: Implementation of `cvgplotEquidistantMesh()`

```
void cvgplotEquidistantMesh(const Eigen::VectorXd &alpha) {
  const unsigned int n_alphas = alpha.size();
  plt::figure();
  // TODO: (6-1.d) Create log-log plots of the maximum norm
  // errors (obtained by pwlintMaxError()) with
  // number of mesh intervals 32, 64, ..., 4096.
  // START
  constexpr unsigned int kmin = 5, kmax = 12;
  const unsigned int n_meshes = kmax - kmin + 1;
  // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
  Eigen::ArrayXd N =
      Eigen::pow(2, Eigen::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.
  Eigen::MatrixXd Errors(n_meshes, n_alphas);
  for (unsigned int i = 0; i < n_meshes; ++i) {
    const unsigned int n = N(i);  // The mesh has #M = (n+1) = (N(i)+1) nodes.
    Eigen::VectorXd mesh = Eigen::VectorXd::LinSpaced(n + 1, 0.0, 1.0);
    for (unsigned int j = 0; j < n_alphas; ++j) {
      // Compute L-infinity error for f_j(t) = t^alpha(j).
      const 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.
  constexpr unsigned 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 (unsigned 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");
}
```

Get it on GitLab (gradedmeshes.hpp).



◁ The data points $(n, \mathrm{errornorm(n)})$ almost exactly lie on straight lines in the doubly logaritmic plot, *cf.* [Lecture → § 6.2.2.9].



algebraic convergence in n^{-1}

We conjecture an asymptotic behavior $\|f - I_{\mathcal{M}} f\|_{L^\infty(I)} = O(n^{-\beta})$ with $\beta = \beta(\alpha)$ an increasing function of α, because β can be read off the slope of the lines in the plot.

HINT 1 for (6-1.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 (6-1.e):

C++ code 6.1.6: Implementation of `cvgrateEquidistantMesh()`

```
Eigen::VectorXd cvgrateEquidistantMesh(const Eigen::VectorXd &alpha) {
  const unsigned int n_alphas = alpha.size();
  Eigen::VectorXd Rates(n_alphas);

  // TODO: (6-1.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
  constexpr unsigned int kmin = 5, kmax = 12;
  const unsigned int n_meshes = kmax - kmin + 1;
  // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
  Eigen::ArrayXd N =
      Eigen::pow(2, Eigen::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.
  Eigen::MatrixXd Errors(n_meshes, n_alphas);
  for (unsigned int i = 0; i < n_meshes; ++i) {
    const unsigned int n = N(i);  // The mesh has #M = (n+1) = (N(i)+1) nodes.
    Eigen::VectorXd mesh = Eigen::VectorXd::LinSpaced(n + 1, 0.0, 1.0);
    for (unsigned int j = 0; j < n_alphas; ++j) {
      // Compute L-infinity error for f_j(t) = t^alpha(j).
      const 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 (unsigned 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.
    Eigen::VectorXd coeff =
        polyfit(N.array().log(), Errors.col(j).array().log(), 1);
    Rates(j) = -coeff[1];
  }
  // END
  return Rates;
}
```

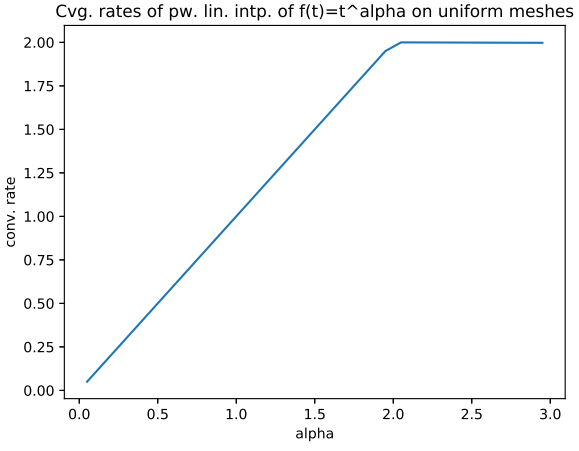
Get it on GitLab (gradedmeshes.hpp).

SOLUTION of (6-1.f):

C++ code 6.1.7: Implementation of `testcvgEquidistantMesh()`

```
2  void testcvgEquidistantMesh() {
3    plt::figure();
4    // TODO: (6-1.f) Plot and tabulate the convergence rates of pw.
5    // lin. intp. of t^α using equidistant meshes on [0,1],
6    // for alpha = 0.05, 0.15, 0.25 ..., 2.95.
7    // START
8    constexpr unsigned int n_alphas = 30;
9    Eigen::VectorXd alpha = Eigen::VectorXd::LinSpaced(n_alphas, 0.05, 2.95);
10
11   Eigen::VectorXd ConvRates = cvgrateEquidistantMesh(alpha);
12   std::cout << std::setw(10) << "alpha" << std::setw(20) << "Convergence rate"
13             << std::endl;
14   for (unsigned int j = 0; j < n_alphas; ++j) {
15     std::cout << std::setw(10) << alpha(j) << std::setw(20) << ConvRates(j)
16               << std::endl;
17   }
18
19   plt::plot(alpha, ConvRates);
20   plt::xlabel("alpha");
21   plt::ylabel("conv. rate");
22   plt::title("Cvg. rates of pw. lin. intp. of f(t)=t^alpha on uniform meshes");
23   // END
24   plt::savefig("./cx_out/cvgRateEquidistant.png");
25 }
```

Get it on GitLab (gradedmeshes.hpp).



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.

HINT 1 for (6-1.g): Recall [Lecture → Thm. 6.2.2.15] and apply it in the piecewise linear setting. ⌟

SOLUTION of (6-1.g):

The error representation [Lecture $\rightarrow$ Eq. (6.5.2.27)] in the linear case ($n = 1$) for some $\tau_t \in (t_j, t_{j+1})$ reads as:

$$\forall\, t \in (t_j, t_{j+1}) \qquad \left| f(t) - \left(\mathrm{I}_{\mathcal{M}} f\right)(t) \right| = \frac{1}{2} |f''(\tau_t)\ (t - t_j)(t - t_{j+1})|$$
$$\leq \frac{1}{8} |f''(\tau_t)|\ (t_{j+1} - t_j)^2 = \frac{1}{8n^2} |f''(\tau_t)|$$

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.

HINT 1 for (6-1.h): Use the insight gained in Sub-problem (6-1.g) to simplify the problem. ⌟

SOLUTION of (6-1.h):

From (6-1.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:

$$\varphi(t) = f(t) - \big(\mathrm{I}_{\mathcal{M}} f\big)(t) = t^{\alpha} - t\,\frac{1}{n^{\alpha-1}} \quad , \quad (\varphi(0) = \varphi(1/n) = 0) \, ,$$

$$\varphi'(t) = \alpha t^{\alpha-1} - \frac{1}{n^{\alpha-1}} \, ,$$

$$\blacktriangleright \qquad \varphi'(t^*) = 0 \quad \text{if} \quad t^* = \frac{1}{n}\alpha^{-1/(\alpha-1)} \, .$$

This means

$$\max_{t\in(0,1/n)} |\varphi(t)| = |\varphi(t^*)| = \left| \frac{\alpha^{-\alpha/(\alpha-1)}}{n^{\alpha}} - \frac{\alpha^{-1/(\alpha-1)}}{n^{\alpha}} \right|$$

$$= \frac{1}{n^{\alpha}} \left| \alpha^{-\alpha/(\alpha-1)} - \alpha^{-1/(\alpha-1)} \right| = \mathcal{O}(n^{-\alpha}) = \mathcal{O}(h^{\alpha}) \, .$$

The order of convergence in $h = 1/n$ is equal to the parameter α, as we could have guessed from Fig. 52.

SOLUTION of (6-1.i):

C++ code 6.1.9: Implementation of `cvgrateGradedMesh()`

```
Eigen::MatrixXd cvgrateGradedMesh(const Eigen::VectorXd &alpha,
                                  const Eigen::VectorXd &beta) {
  const unsigned int n_alphas = alpha.size();
  const unsigned int n_betas = beta.size();
  Eigen::MatrixXd Rates(n_betas, n_alphas);

  // TODO: (6-1.i) Fill in the entries of Rates.
  // START
  constexpr unsigned int kmin = 5, kmax = 12;
  const unsigned int n_meshes = kmax - kmin + 1;
  // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
  Eigen::ArrayXd N =
      Eigen::pow(2, Eigen::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).
  Eigen::MatrixXd Errors(n_meshes, n_alphas * n_betas);
  for (unsigned int i = 0; i < n_meshes; ++i) {
    const unsigned int n = N(i);  // The mesh has #M = (n+1) = (N(i)+1) nodes.
    Eigen::VectorXd mesh = Eigen::VectorXd::LinSpaced(n + 1, 0.0, 1.0);
    for (unsigned int k = 0; k < n_betas; ++k) {
      // Transform the mesh using the grading parameter beta.
      Eigen::VectorXd gradedMesh = pow(mesh.array(), beta(k)).matrix();
      for (unsigned int j = 0; j < n_alphas; ++j) {
        // Compute L-infinity error for f_j(t) = t^alpha(j).
        const double alphaj = alpha(j);
        auto fj = [alphaj](double t) { return std::pow(t, alphaj); };
        const unsigned int ind = k * n_alphas + j;
        Errors(i, ind) =
            pwlintpMaxError<std::function<double(double)>>(fj, gradedMesh);
      }
    }
  }

  for (unsigned int k = 0; k < n_betas; ++k) {
    for (unsigned 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.
      const unsigned int ind = k * n_alphas + j;
      Eigen::VectorXd coeff =
          polyfit(N.array().log(), Errors.col(ind).array().log(), 1);
      Rates(k, j) = -coeff[1];
    }
  }
  // END
  return Rates;
}
```

Get it on GitLab (gradedmeshes.hpp).

SOLUTION of (6-2.a):

In general a linear function cannot interpolate three data points.

HINT 1 for (6-2.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 $\rightarrow$ Fig. 150]. ⌟

SOLUTION of (6-2.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. (5.1.0.22)] will vanish and the matrix cannot be invertible.

HINT 1 for (6-2.c): Since we have to distribute $n+1$ nodes on n intervals, there will be a single interval that contains 2 nodes. ⌟

SOLUTION of (6-2.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.

HINT 1 for (6-2.d): While you check the condition of Sub-problem (6-2.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 (6.2.1). 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 (6-2.d):

C++11-code 6.2.2: Solution of (6-2.d)

```
Eigen::VectorXd tentBasCoeff(const Eigen::VectorXd& x, const Eigen::VectorXd& t,
                             const Eigen::VectorXd& y) {
  // Initialization
  const std::size_t n = t.size();
  Eigen::VectorXd s = Eigen::VectorXd::Zero(n);

  // TODO: (6-2.d) Return the interpolant evaluated at the knots x.
  // START

  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.

  // Check condition of subproblem c
  std::size_t i = 0;
  std::size_t k = 0;
  for (std::size_t j = 0; j < (n - 1); ++j) {
    bool intervalOK = false;
    while (i < n) {
      const 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]);

```

```
54    s(x_indices[k]) = gamma * x(x_indices[k]) + beta;
55    s(x_indices[k + 1]) = gamma * x(x_indices[k + 1]) + beta;
56
57    // Find intercept, slope and value s at the lower bound x
58    // for all intervals on the left of interval with 2 nodes k
59    for (int j = k - 1; j >= 0; --j) {
60      gamma = (s(x_indices[j + 1]) - y(t_indices[j])) /
61              (x(x_indices[j + 1]) - t(t_indices[j]));
62      beta = y(t_indices[j]) - gamma * t(t_indices[j]);
63
64      s(x_indices[j]) = gamma * x(x_indices[j]) + beta;
65    }
66
67    // Find intercept, slope and value s at the upper bound x
68    // for all intervals on the right of interval with 2 nodes k
69    for (int j = k + 2; j < n; ++j) {
70      gamma = (y(t_indices[j]) - s(x_indices[j - 1])) /
71              (t(t_indices[j]) - x(x_indices[j - 1]));
72      beta = s(x_indices[j - 1]) - gamma * x(x_indices[j - 1]);
73
74      s(x_indices[j]) = gamma * x(x_indices[j]) + beta;
75    }
76    // END
77
78    return s;
79  }
```

Get it on GitLab (interleaveLIP.hpp).

HINT 1 for (6-2.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 (6-2.e): The function `tentBasCoeff` can be reused as a private method of the class. ⌟

SOLUTION of (6-2.e):

C++11-code 6.2.3: Full declaration of class `PwLinIP`

```
2  class PwLinIP {
3   public:
4    /**
5     * \brief Construct a new PwLinIP object
6     *
7     * \param x Vector of knots
8     * \param t Vector of nodes
9     * \param y Vector of values of interpolant in nodes
10    */
11   PwLinIP(const Eigen::VectorXd& x, const Eigen::VectorXd& t,
12           const Eigen::VectorXd& y);
13
14   /**
15    * \brief evaluate interpolant at arg.
16    *
17    * \param arg argument to evaluate at
18    * \return double
19    */
20   double operator()(double arg) const;
21
22  private:
23   Eigen::VectorXd x_;
24   Eigen::VectorXd t_;
25   Eigen::VectorXd y_;
26   Eigen::VectorXd s_;
27 };
```

Get it on GitLab (interleaveLIP.hpp).

C++11-code 6.2.4: Implementation of `PwLinIP::PwLinIP` (constructor)

```
2  PwLinIP::PwLinIP(const Eigen::VectorXd& x, const Eigen::VectorXd& t,
3                   const Eigen::VectorXd& y) {
4    assert(t.size() == y.size() && t.size() == x.size() &&
5           "x, t, y must have same size!");
6
7    const std::size_t n = t.size();
8    x_.resize(n);
9    t_.resize(n);
10   y_.resize(n);
11
12   // TODO: (6-2.e) Fill the member variables of the class.
13   // START
14   auto x_indices = order(x);
15   for (std::size_t i = 0; i < n; ++i) {
16     x_(i) = x[x_indices[i]];
17   }
18
19   auto t_indices = order(t);
20   for (std::size_t i = 0; i < n; ++i) {
21     t_(i) = t[t_indices[i]];
22     y_(i) = y[t_indices[i]];
```

```
  }

  s_ = tentBasCoeff(x_, t_, y_);
  // END
}
```

Get it on GitLab (interleaveLIP.hpp).

C++11-code 6.2.5: Implementation of `PwLinIP::operator()`

```
double PwLinIP::operator()(double arg) const {
  double ret_val = 0.;

  // TODO: (6-2.e) Implement the evaluation operator efficiently!
  // START
  if (arg < x_(0) || arg > x_(x_.size() - 1)) {
    ret_val = 0;
  } else {
    std::vector<double> x(
        x_.size() -
        1);  // Exclude x_0 from search of interval including arg
    Eigen::VectorXd::Map(&x.front(), x_.size() - 1) = x_.tail(x_.size() - 1);
    const std::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

    const double gamma = (s_(j) - s_(j - 1)) / (x_(j) - x_(j - 1));
    const double beta = s_(j - 1) - gamma * x_(j - 1);

    ret_val = gamma * arg + beta;
  }
  // END

  return ret_val;
}
```

Get it on GitLab (interleaveLIP.hpp).

SOLUTION of (6-2.f):

C++11-code 6.2.7: Solution of (6-2.f)

```
void plotCardinalBasisFunctions() {
  // Initialization
  constexpr std::size_t n = 11;

  // Nodes
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(n, 0, 10);
  Eigen::VectorXd t(n);
  t(0) = 0;
  t.tail(n - 1).setLinSpaced(n - 1, 0.5, 9.5);

  plt::figure();
  // Plot colors
  std::vector<std::string> C = {"b", "r", "m"};

  // TODO: (6-2.f) Plot cardinal basis functions for interpolation.
  // START

  // Loop over basis functions
  for (unsigned int i = 0; i < 3; ++i) {
    // Basis function e_i
    Eigen::VectorXd y = Eigen::VectorXd::Zero(n);
    y(i) = 1;

    // Create interpoland with values y
    PwLinIP cardinalBasis(x, t, y);

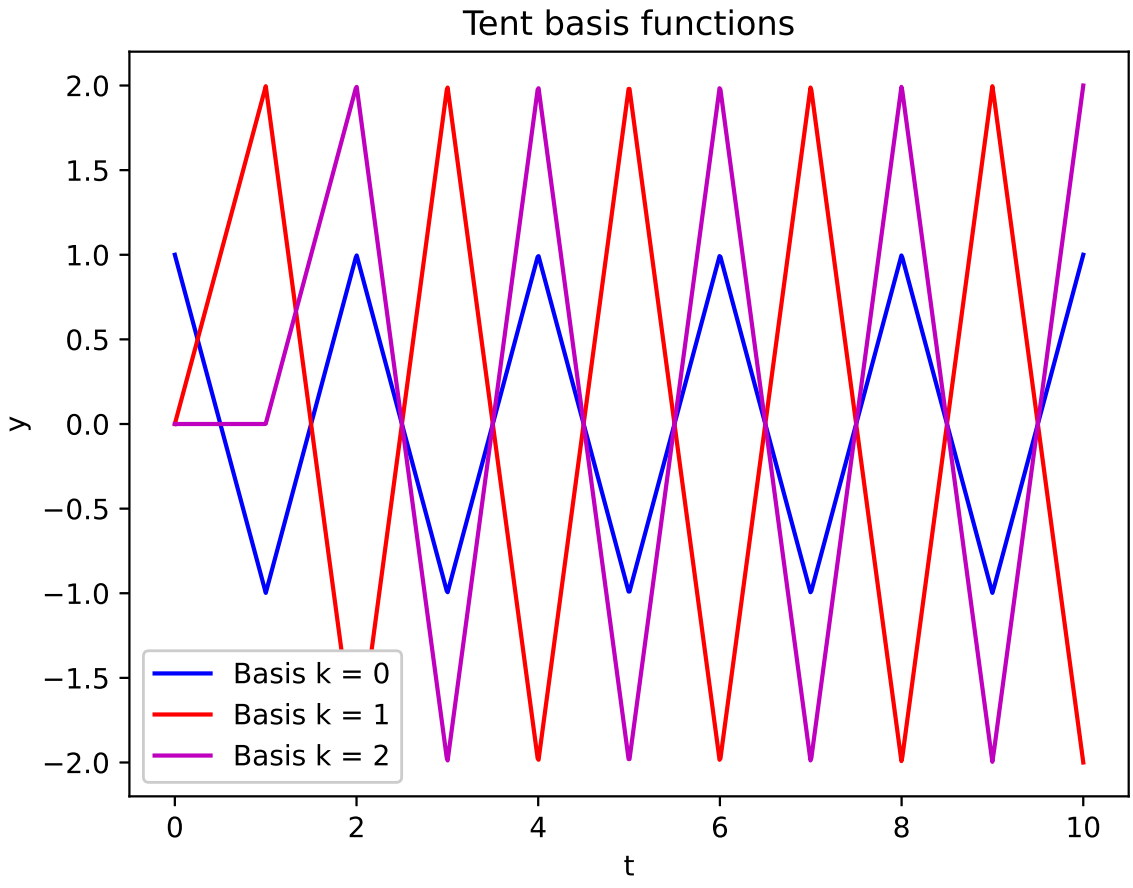
    // Plotting values
    constexpr unsigned int N = 1000;
    Eigen::VectorXd xval = Eigen::VectorXd::LinSpaced(N, 0, 10);
    Eigen::VectorXd s(N);
    for (unsigned int j = 0; j < N; ++j) {
      s(j) = cardinalBasis(xval(j));
    }

    plt::plot(xval, s, C[i], {{"label", "Basis k = " + std::to_string(i)}});
  }

  plt::xlabel("t");
  plt::ylabel("y");
  plt::legend();
  plt::title("Tent basis functions");
  // END

  plt::savefig("cx_out/tent_basis_functions.png");
}
```

Get it on GitLab (interleaveLIP.hpp).


Tent basis functions
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
0
2
4
6
8
10
t
Basis k = 0
Basis k = 1
Basis k = 2

HINT 1 for (6-3.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 (6-3.a):

See Code 6.3.6 below. For this part of the exercise you can ignore the argument `err_vs_step_no`.

SOLUTION of (6-3.b):

C++ code 6.3.6: Implementation of function `adaptivepolyintp()`

```
template <class Function>
Eigen::VectorXd adaptivepolyintp(Function&& f, double a, double b, double tol,
                                 unsigned int N,
                                 std::vector<double>* errortab = nullptr) {
  // Generate sampling points and evaluate f there
  Eigen::VectorXd sampling_points = Eigen::VectorXd::LinSpaced(N, a, b),
                  fvals_at_sampling_points = sampling_points.unaryExpr(f);

  // TODO: (6-3.a) Implement the greedy algorithm for adaptive interpolation.
  // Ignore the errortab part of this function for now.
  // TODO: (6-3.b) save the error in errortab
  // START
  // Approximate max|f(x)|
  const 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 (std::size_t i = 0; i < N; ++i) {
    // *** Step 1: interpolate with current nodes
    //   need to convert std::vector to
    //   Eigen::VectorXd to use the function intpolyval
    Eigen::Map<Eigen::VectorXd> te(t.data(), t.size());
    Eigen::Map<Eigen::VectorXd> ye(y.data(), y.size());
    Eigen::VectorXd intpolyvals_at_sampling_points =
        intpolyval(te, ye, sampling_points);

    // *** Step 2: find node where error is the largest
    Eigen::VectorXd err =
        (fvals_at_sampling_points - intpolyvals_at_sampling_points).cwiseAbs();

    // We use an Eigen "Visitor"
    //
       https://eigen.tuxfamily.org/dox/group__TutorialReductionsVisitorsBroadcasting.htm
    int idx = 0;
    const double 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));
    // (6-3.b): save error
    if (errortab != nullptr) {
      errortab->push_back(max);
    }
  }
  std::cerr << "Desired accuracy could not be reached." << std::endl;
```

```
53    // END
54    return sampling_points;  // return all sampling points
55  }
```

Get it on GitLab (adaptivepolyintp.hpp).

SOLUTION of (6-3.c):

C++ code 6.3.7: Function `plotInterpolationError()`

```
void plotInterpolationError() {
  // 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); };

  // Test interval
  constexpr double a = 0, b = 1;
  // Get interpolation nodes
  constexpr unsigned int N = 1000;  // no. of sampling points
  constexpr double tol = 1e-6;      // tolerance

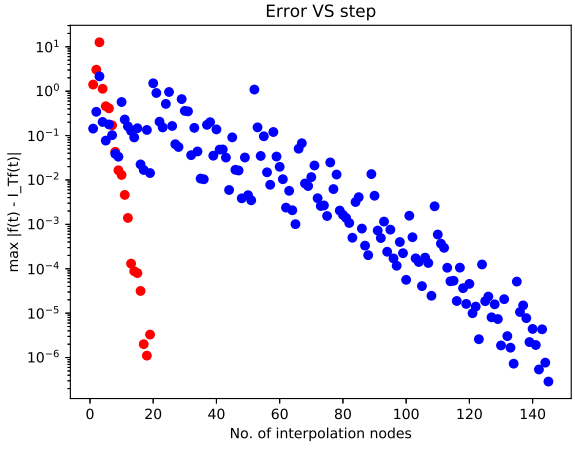
  plt::figure();

  // TODO: (6-3.c) generate the plots of error vs number of nodes for f1, f2
  // START
  Eigen::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);
  Eigen::VectorXd n1 = Eigen::VectorXd::LinSpaced(ef1.size(), 1, ef1.size());
  Eigen::VectorXd n2 = Eigen::VectorXd::LinSpaced(ef2.size(), 1, ef2.size());
  // Plot
  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", "f1(t) = sin(e^{2t})"}});
  plt::semilogy(n2, ef2, "bo", {{"label", "f2(t) = sqrtt/(1+16t^2)"}});
  plt::legend();
  // END

  plt::savefig("./cx_out/intperrplot.png");
}
```

Get it on GitLab (adaptivepolyintp.hpp).



- $\hat{=} f_1(t) := \sin(e^{2t})$
- $\hat{=} f_2(t) = \frac{\sqrt{t}}{1+16t^2}$

The error data points are approximately located on straight lines in the linear-logaritmic error plots.

Exponential convergence

HINT 1 for (6-4.a): Perform integration by substitution: $t = \sin s$. ⌟

SOLUTION of (6-4.a):

Obviously the weight function is positive so that we can drop the $|\cdot|$. Then a direct computation of the integral by means of the substitution

$$t = \sin s \quad \Rightarrow \quad \mathrm{d}t = \cos s \, \mathrm{d}s \,, \quad -\frac{\pi}{2} \leq s \leq \frac{\pi}{2} \,,$$

yields

$$\int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} \, \mathrm{d}t = \int_{-\frac{1}{2}\pi}^{\frac{1}{2}\pi} \frac{\cos s}{|\cos s|} \, \mathrm{d}s = \pi \,.$$

HINT 1 for (6-4.j): Note that (6.4.15) is a Chebychev expansion. Then use the result of Sub-problem (6-4.i) to express the coefficients of the Chebychev expansions of L_j. ⌟

SOLUTION of (6-4.j):

We have already seen that $\{T_l/\lambda_l : l = 0,\dots,n\}$ is an $(\cdot,\cdot)_{n+1}$-ONB of $\mathcal{P}_n$. Exploiting the defining property of Lagrange polynomials

$$L_j(\xi_\ell^{(n+1)}) = \delta_{k,\ell}\,, \quad k,\ell \in \{0,\dots,n\}\,,$$

we can thus 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 (6.4.11).

HINT 1 for (6-4.b): Recall the trigonometric identity $2\cos(x)\cos(y) = \cos(x+y) + \cos(x-y)$. ⌟

SOLUTION of (6-4.b):

For $k,l = 0,\ldots,n$ with $k \neq l$, by using the substitution

$$t = \cos s \quad \Leftrightarrow \quad s = \arccos t \quad \Rightarrow \quad \mathrm{d}s = -\frac{1}{\sqrt{1-t^2}}\,\mathrm{d}t\,,$$

and simple trigonometric identities we readily compute:

$$\begin{aligned}
(T_k, T_l)_w &= \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} \cos(k\,\arccos t)\cos(l\,\arccos t)\,dt \\
&= \int_0^{\pi} \cos(ks)\cos(ls)\,ds \\
&= \frac{1}{2}\int_0^{\pi} \cos((k+l)s) + \cos((k-l)s)\,ds \\
&= \frac{1}{2}\left\{ [\sin((k+l)s)/(k+l)]_0^{\pi} + [\sin((k-l)s)/(k-l)]_0^{\pi} \right\} = 0\,,
\end{aligned}$$

since $k+l \neq 0$ and $k-l \neq 0$.

SOLUTION of (6-4.c):

Bi-linearity, symmetry, and non-negativity should be evident for $(\cdot,\cdot)_{n+1}$. Positive definiteness we conclude from

$$(p,p)_{n+1} = \sum_{j=0}^{n} |p(\xi_j^{(n+1)})|^2 = 0 \quad \Rightarrow \quad p(\xi_j^{(n+1)}) = 0 \quad \forall j = 0,\ldots,n \quad \Rightarrow \quad p \equiv 0\,,$$

because by [Lecture $\rightarrow$ Thm. 5.2.2.7] a polynomial of degree $\leq n$ that vanishes in $n+1$ distinct points must be zero.

HINT 1 for (6-4.d): [Lecture → Code 6.2.3.6] demonstrates the efficient evaluation of Chebychev polynomials based on their 3-term recurrence formula from [Lecture → Thm. 6.2.3.4]. ⌟

HINT 2 for (6-4.d): Revisit and heed [Lecture → Rem. 1.5.3.15] to devise a “numerically safe” test. ⌟

SOLUTION of (6-4.d):

As a consequence of [Lecture → Thm. 6.2.3.4], we already know that $\{T_0, \ldots, T_n\}$ is a basis for $\mathcal{P}_n$. We check the orthogonality with the C++ code given in Code 6.4.6.

C++ code 6.4.6: Code [Lecture → Code 6.2.3.6]

```
2  // Computes the values of the Chebychev polynomials T_0,...,T_d
3  // at points passed in x using the 3-term recursion
4  // (??).  The values T_k(x_j), are returned in
5  // row k+1 of V.
6  void chebpolmult(const unsigned int d, const RowVectorXd& x, MatrixXd& V) {
7    const unsigned int n = x.size();
8    V = MatrixXd::Ones(d + 1, n);  // T_0 ≡ 1
9    if (d == 0) return;
10   V.block(1, 0, 1, n) = x;  // T_1(x) = x
11   if (d == 1) return;
12   for (unsigned int k = 1; k < d; ++k) {
13     const RowVectorXd p = V.block(k, 0, 1, n);       // p = T_k
14     const RowVectorXd q = V.block(k - 1, 0, 1, n);  // q = T_{k-1}
15     V.block(k + 1, 0, 1, n) =
16         2 * x.cwiseProduct(p) - q;  // 3-term recursion
17   }
18 }
```

C++ code 6.4.7: Solution of (6-4.d)

```
2  bool checkDiscreteOrthogonality(unsigned int n) {
3    bool return_val = false;
4    constexpr double tol =
5        std::numeric_limits<double>::epsilon();  // machine epsilon
6
7    // TODO: (6-4.d) Check the orthogonality property of the Chebychev
         polynomials
8    // up to order n.
9    // START
10   Eigen::MatrixXd V;
11   // generate Chebychev nodes
12   Eigen::RowVectorXd x = Eigen::RowVectorXd::LinSpaced(n + 1, 0, n);
13   x = x.unaryExpr(
14       [n](double x) { return std::cos(M_PI * (2. * x + 1.) / 2. / (n + 1.)); });
15   chebpolmult(n, x, V);  // compute values of polynomials
16
17   // check for orthogonality
18   // note that the second loop starts at k + 1 because this inner
         product is
19   // (anti-)symmetric
20   return_val = true;
21   for (unsigned int k = 0; k < n + 1; ++k) {
22     for (unsigned int l = k + 1; l < n + 1; ++l) {
23       return_val &= (std::abs(V.row(k).dot(V.row(l))) < tol * 100.);
24     }
25   }
26   // END
27
```

```
28     return return_val;
29 }
```

Get it on GitLab (chebpolyproperties.hpp).

HINT 1 for (6-4.e): Use the relationship of trigonometric functions and the complex exponential together with the summation formula for geometric sums. ⌟

SOLUTION of (6-4.e):

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 (6.4.1) we have:

$$\begin{aligned}(T_k, T_l)_{n+1} &= \sum_{j=0}^{n} T_k\left(\xi_j^{(n)}\right) T_l\left(\xi_j^{(n)}\right) \\ &= \sum_{j=0}^{n} \cos\left(k \frac{2j+1}{2(n+1)}\,\pi\right) \cos\left(l \frac{2j+1}{2(n+1)}\,\pi\right) \\ &= \frac{1}{2} \sum_{j=0}^{n} \left[\cos\left((k+l)\frac{2j+1}{2(n+1)}\,\pi\right) + \cos\left((k-l)\frac{2j+1}{2(n+1)}\,\pi\right)\right].\end{aligned}$$

It is now enough to show that:

$$\sum_{j=0}^{n} \cos\left(m \frac{2j+1}{2(n+1)}\,\pi\right) = 0 \quad \forall m \in \mathbb{Z} \setminus \{0\}\,. \tag{6.4.8}$$

In order to verify this, observe that:

$$\sum_{j=0}^{n} \cos\left(m \frac{2j+1}{2(n+1)}\,\pi\right) = \mathrm{Re}\left(\sum_{j=0}^{n} e^{\imath m \frac{2j+1}{2(n+1)}\,\pi}\right) = \mathrm{Re}\left(e^{\imath m \frac{1}{2(n+1)}\,\pi} \sum_{j=0}^{n} e^{\imath m \frac{j}{n+1}\,\pi}\right).$$

Finally, by the standard formula for the geometric sum we have:

$$\begin{aligned}e^{\imath m \frac{1}{2(n+1)}\,\pi} \sum_{j=0}^{n} e^{\imath m \frac{j}{n+1}\,\pi} &= e^{\imath m \frac{1}{2(n+1)}\,\pi} \frac{1 - e^{\imath m \frac{n+1}{n+1}\,\pi}}{1 - e^{\imath m \frac{1}{n+1}\,\pi}} \\ &= \frac{1 - e^{\imath m \pi}}{e^{-\imath m \frac{1}{2(n+1)}\,\pi} - e^{\imath m \frac{1}{2(n+1)}\,\pi}} = -\frac{1 - e^{\imath m \pi}}{2} \frac{1}{\imath \sin\left(m \frac{1}{2(n+1)}\,\pi\right)} \in \imath \cdot \mathbb{R}\,.\end{aligned}$$

This implies that

$$\mathrm{Re}\left(e^{\imath m \frac{1}{2(n+1)}\,\pi} \sum_{j=0}^{n} e^{\imath m \frac{j}{n+1}\,\pi}\right) = 0\,,$$

as desired.

HINT 1 for (6-4.f): The task boils down to determining the coefficients α_j. Use Thm. 6.4.5 and a slight extension of [Lecture → Cor. 6.3.1.15].

Corollary [Lecture → Cor. 6.3.1.15]. ONB representation of best approximant

If $\{b_1, \ldots, b_N\}$ is an orthonormal basis [Lecture → Def. 6.3.1.14] of $V \subset X$, then the best approximant $q := \operatorname{argmin}_{p \in V} \|f - p\|_X$ of $f \in X$ has the representation

$$q = \sum_{j=1}^{N} (f, b_j)_X b_j \,. \qquad \text{[Lecture → Eq. (6.3.1.16)]}$$

SOLUTION of (6-4.f):

In view of the theorem, the family $\{T_0, \ldots, T_m\}$ is an orthogonal basis of $\mathcal{P}_m$ with respect to the inner product ($\rightarrow$ Sub-problem (6-4.c)) $(\ ,\ )_{n+1}$. By ((6-4.e)) and (6.4.8) we have for $k \in \{0, \ldots, n\}$

$$\lambda_k^2 := \|T_k\|_{n+1}^2 = (T_k, T_k)_{n+1} = \begin{cases} \frac{1}{2}\sum_{j=0}^{n}(\cos(0) + \cos(0)) = n+1 & \text{if } k = 0\ , \\ \frac{1}{2}\sum_{j=0}^{n}\cos(0) = (n+1)/2 & \text{otherwise.} \end{cases} \tag{6.4.11}$$

Thus the family $\{T_k/\lambda_k : k = 0, \ldots, m\}$ is an orthonormal basis of $\mathcal{P}_m$ with respect to the inner product $(\ ,\ )_{n+1}$. Hence, we apply [Lecture $\rightarrow$ Cor. 6.3.1.15] with

$$(\cdot,\cdot)_X := (\cdot,\cdot)_{n+1} \quad \text{and} \quad \{b_1, \ldots, b_N\} := \left\{\frac{T_0}{\lambda_0}, \ldots, \frac{T_m}{\lambda_m}\right\} \ bcom$$

and get

$$q_m = \sum_{j=0}^{m} (f, T_j/\lambda_j)_{n+1} \frac{T_j}{\lambda_j} = \sum_{j=0}^{m} \alpha_j T_j\ ,$$

$$\blacktriangleright \qquad \alpha_j := \frac{1}{n+1}\begin{cases} (f, T_0)_{n+1} & \text{if } j = 0\ , \\ 2(f, T_j)_{n+1} & \text{for } j = 1, \ldots, m\ . \end{cases} \tag{6.4.12}$$

SOLUTION of (6-4.g):

If $m = n$ we can achieve $\|f - q_n\|_{n+1} = 0$ by choosing q_n as the Chebychev interpolant of f, that is,

$$q_n \in \mathcal{P}_n: \quad q_n(\xi_j^{(n+1)}) = f(\xi_j^{(n+1)}) \quad \forall j = 0, \ldots, n \, .$$

SOLUTION of (6-4.h):

Of course, we rely on (6.4.12), which can be implemented directly.

C++11-code 6.4.13: Solution of (6-4.h)

```
2  template <typename Function>
3  Eigen::VectorXd bestpolchebnodes(const Function& f, unsigned int n,
4                                   unsigned int m) {
5    Eigen::VectorXd alpha = Eigen::VectorXd::Zero(m + 1);
6
7    // TODO: (6-4.h) Compute the vector of coefficients such that the
         Chebychev
8    // polynomials of up to order m interpolate the function f.
9    // START
10
11   // generate Chebychev nodes
12   Eigen::RowVectorXd x = Eigen::RowVectorXd::LinSpaced(n + 1, 0, n);
13   x = x.unaryExpr(
14       [n](double x) { return std::cos(M_PI * (2. * x + 1.) / 2. / (n + 1.)); });
15
16   // evaluate function
17   Eigen::VectorXd f_eval = x.unaryExpr(f);
18
19   // evaluate Chebychev polynomials
20   Eigen::MatrixXd V;
21   chebpolmult(m, x, V);
22
23   // find coefficients using previously determined formula in subproblem
         (f)
24   alpha = 2. * V * f_eval / (n + 1.);
25   alpha(0) /= 2.;
26   // END
27
28   return alpha;
29 }
```

Get it on GitLab (chebpolyproperties.hpp).

HINT 1 for (6-4.i): Again, [Lecture → Code 6.2.3.6] is useful for evaluating Chebychev polynomials. ⌟

SOLUTION of (6-4.i):

C++11-code 6.4.14: Solution of (6-4.i)

```
std::vector<double> testBestPolyChebNodes(unsigned int n) {
  std::vector<double> errors(n + 1, 0.);
  auto f = [](double x) {
    return 1. / (std::pow(5. * x, 2) + 1.);
  };  // function to test on
  const Eigen::VectorXd X =
      Eigen::VectorXd::LinSpaced(1e6, -1, 1);  // evaluation grid

  // TODO: (6-4.i) Test the function from the previous subproblem by
     tabulating
  // the error norms.
  // START
  std::cout << " m  L^inf error" << std::endl;

  for (unsigned int m = 0; m < n + 1; ++m) {
    Eigen::VectorXd alpha = bestpolchebnodes(f, m, m);

    Eigen::MatrixXd V;
    chebpolmult(m, X, V);

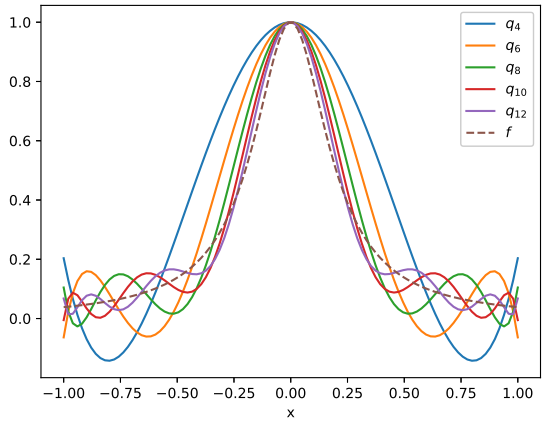
    Eigen::VectorXd qm = V.transpose() * alpha;
    Eigen::VectorXd f_eval = X.unaryExpr(f);

    double err_max = (f_eval - qm).lpNorm<Eigen::Infinity>();

    // tabulate errors
    std::cout << std::setw(2) << m << std::setw(2) << " "
              << std::setprecision(10) << err_max << std::endl;
  }
  // END

  return errors;
}
```

Get it on GitLab (chebpolyproperties.hpp).



◁ plot for $n = 20$ and different q_m.

m	$\|f - q_m\|_{L^\infty(]-1,1[)}$
0	0.9615384615
1	0.9259259259
2	0.600597751
3	0.75030012
4	0.4020169354
5	0.5559113388
6	0.2642275018
7	0.3917402846
8	0.170835638
9	0.2691783353
10	0.1091535109
11	0.182758282
12	0.06921571321
13	0.123397672
14	0.04660234663
15	0.08310704777
16	0.03261358361
17	0.0559074779
18	0.02249229017
19	0.03759032888
20	0.01533373519

◁ Tabulated error norms for $n = 20$.

SOLUTION of (6-5.a):

The ellipse is approximated by a polygon, whose length is considered an approximation of the length of the ellipse. Recall that $|w-z|$ gives the distance of two complex numbers $w,z \in \mathbb{C}$ considered as points in the plane $\mathbb{C} \cong \mathbb{R}^2$.

C++11-code 6.5.2: Code for `lengthEllipse`.

```
2  double lengthEllipse(double rho, unsigned int N) {
3    double length = 0.0;
4    // TODO: (6-5.a) Approximate the length of ellipse given ρ and N
5    // equidistant points (in θ)
6    // START
7    const complex i(0, 1);  // imaginary unit
8    // Define γ_p
9    auto f = [rho, i](double theta) { return cos(theta - i * log(rho)); };
10   Eigen::VectorXd eval_points = Eigen::VectorXd::LinSpaced(N + 1, 0, 2 * M_PI);
11   // Calculating distance between last and current point on ellipse for
       all
12   // points
13   complex last = f(eval_points(0));
14   for (unsigned int i = 0; i < N; ++i) {
15     complex current = f(eval_points(i + 1));
16     length += std::abs(current - last);
17     last = current;
18   }
19
20   // Alternative method using an approximation over the integral of the
       length
21   // of γ'_p.
22   // df = |γ'_p(θ)|
23   double length_alt = 0.0;
24   auto df = [rho](long double theta) {
25     return 0.5 * sqrt(rho * rho + 1 / (rho * rho) - 2 * cos(2 * theta));
26   };
27   double dtheta = 2 * M_PI / N;
28   double theta_i = 0;
29   for (unsigned int i = 0; i < N; ++i) {
30     length_alt += df(theta_i);
31     theta_i += dtheta;
32   }
33   length_alt *= dtheta;
34   // END
35   return length;
36 }
```

Get it on GitLab (chebyApprox.hpp).

HINT 1 for (6-5.b): Consult [Lecture → Rem. 6.2.2.67].

SOLUTION of (6-5.b):

f is analytic in

$$D := \mathbb{C} \setminus \{\frac{2}{3}\pi ik - \frac{1}{3}\pi i \mid k \in \mathbb{Z}\} \ .$$

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 $\rightarrow$ Thm. 6.2.2.68], f is analytic in

$$\mathbb{C} \setminus \{z \in \mathbb{C} \mid g(z) = 0\} =: \mathbb{C} \setminus S \ ,$$

where $S \subset \mathbb{C}$ is the "domain of singularity". To find S let $z := a + ib$, $a, b \in \mathbb{R}$ and compute

$$\begin{aligned} -1 = \exp(z) = \exp(a+ib) = \exp(a)\exp(ib) \quad &\Leftrightarrow \quad a = 0,\ b \in 2\pi\mathbb{Z} + \pi \\ -1 = \exp(z) \quad &\Leftrightarrow \quad z \in i(2\pi\mathbb{Z} + \pi) \\ -1 = \exp(-3z) \quad &\Leftrightarrow \quad z \in \frac{i(2\pi\mathbb{Z} - \pi)}{3} \ . \end{aligned}$$

HINT 1 for (6-5.c): In [Lecture → Rem. 6.2.3.26] you can find alternative formulas for γ_ρ. ⌟

SOLUTION of (6-5.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:

$$\cos(\pi/2 - i\log(\rho_{\max})) = 1/3\pi i$$
$$\Updownarrow$$
$$\sinh(\log(\rho_{\max})) = \pi/3 \quad \Leftrightarrow \quad \rho_{\max} = \exp(\sinh^{-1}(\pi/3)) \ .$$

SOLUTION of (6-5.d):

C++11-code 6.5.6: Code for `bestBound`.

```
std::pair<double, double> bestBound() {
  const double maxRho = exp(asinh(M_PI / 3.));
  constexpr unsigned int N = 1000;
  double m = 0.0, r = 0.0;  // min and rho for output
  // TODO: (6-5.d) Compute (6.5.4) with d from (6.5.5) and
  // |γ_ρ| from (6-5.a)
  // START
  // 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));
  };

  m = std::numeric_limits<double>::max();  // save min in here
  // discretize rhos and remove first (open interval!)
  Eigen::VectorXd rhos = Eigen::VectorXd::LinSpaced(N + 1, 1, maxRho).head(N);
  r = rhos(0);  // save argmin here

  // discretize the ellipse
  Eigen::VectorXd thetas = Eigen::VectorXd::LinSpaced(N, 0, 2 * M_PI);

  // loop over all rhos and get compute expression
  for (unsigned int k = 0; k < rhos.size(); ++k) {
    const double rho = rhos(k);  // current rho
    // get values on the ellipse
    Eigen::VectorXcd z(thetas.size());
    for (unsigned int j = 0; j < thetas.size(); ++j) {
      z(j) = gamma(rho, thetas(j));
    }

    // sup of f of γ_ρ
    const Eigen::VectorXcd fz = z.unaryExpr(f);
    const double M = lengthEllipse(rho, N) * fz.cwiseAbs().maxCoeff() *
                     M_1_PI;  // 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
  }
  // END
  return std::make_pair(m, r);
}
```

Get it on GitLab (chebyApprox.hpp).

SOLUTION of (6-5.e):

Let M be the approximation of the minimum computed in Sub-problem (6-5.d) and ρ the associated ellipse parameter. Then we deduce from [Lecture $\rightarrow$ Eq. (6.2.3.28)] that

$$\|f - L_n f\|_{L^\infty([-1,1])} \lesssim \frac{M}{\rho^{n+1} - 1} \, .$$

From Sub-problem (6-5.d) we learned that $M \approx 10.0753$ and $\rho \approx 1.91355$. Applying this to the above bound, we find that

$$\|f - L_n f\|_{L^\infty([-1,1])} \lesssim \frac{10.0753}{1.91355^{n+1} - 1} \, . \tag{6.5.7}$$

HINT 1 for (6-5.f): The function `intpolyval` has the signature:

```
Eigen::VectorXd intpolyval(const Eigen::VectorXd& t, const
   Eigen::VectorXd& y, const Eigen::VectorXd& x);
```

Where (t, y) is the interpolation data and p is used to save the values of the interpolant evaluated in x. ⌟

SOLUTION of (6-5.f):

C++11-code 6.5.8: Code for `compareErrorAndBound`.

```
std::pair<std::vector<double>, std::vector<double>> compareErrorAndBound(
    unsigned int n_max) {
  std::vector<double> upperBound, errInf;
  constexpr unsigned N = 10000;

  // TODO: (6-5.f) Tabulate the L-inf error and the bounds.
  // START
  auto m_rho = bestBound();
  const double m = m_rho.first;
  const double rho = m_rho.second;

  // Test for degrees 4 to n_max
  Eigen::VectorXd degrees = Eigen::VectorXd::LinSpaced(n_max - 3, 4, n_max);

  for (unsigned int j = 0; j < degrees.size(); ++j) {
    upperBound.emplace_back(m / (std::pow(rho, degrees(j) + 1) - 1));
  }

  // Compute actual error for any degree
  auto f = [](double x) { return 1. / (1 + std::exp(-3 * x)); };

  for (unsigned int l = 0; l < degrees.size(); ++l) {
    const int n = degrees(l);
    // Cheby nodes and value of f at the nodes
    Eigen::ArrayXd k = Eigen::ArrayXd::LinSpaced(n + 1, 0, n);
    Eigen::VectorXd t = ((2 * k + 1) / (2 * (n + 1)) * M_PI).cos().matrix(),
                    y = t.unaryExpr(f);
    // Evaluate error at a couple of points
    Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N, -1, 1),
                    F = x.unaryExpr(f);

    // Use barycentric formula (in intpolyval.hpp)
    Eigen::VectorXd LnF = intpolyval(t, y, x);
    errInf.push_back((F - LnF).cwiseAbs().maxCoeff());
  }
  // END
  return {errInf, upperBound};
}
```

Get it on GitLab (chebyApprox.hpp).

Remark. Algorithms related to Chebychev interpolation can usually be implemented very efficiently by relying on FFT, see [Lecture → Section 6.2.3.3]. In this problem, the focus is not on efficient implementation and so we do not pursue this.

Degree n	$\|f - L_n f\|_{L^\infty([-1,1])}$	Bound (6.5.7)
4	0.0122079	0.408628
5	0.00263694	0.209489
6	0.00202698	0.1084
7	0.000423312	0.0563594
8	0.000330751	0.0293744
9	6.79868e-05	0.0153294
10	5.3577e-05	0.00800517
11	1.09199e-05	0.00418183
12	8.64874e-06	0.00218495
13	1.75395e-06	0.00114171
14	1.3936e-06	0.000596614
15	2.8172e-07	0.000311775
16	2.24322e-07	0.000162928
17	4.52499e-08	8.51438e-05
18	3.60853e-08	4.4495e-05
19	7.26804e-09	2.32526e-05
20	5.80245e-09	1.21515e-05

The actual interpolation error is still far below the bound, which is hardly surprising, because the path integral has been estimated fairly crudely.

SOLUTION of (6-5.g):

Let $\Phi : [-1,1] \to [-a,a]$ the unique affine mapping of the two intervals to each other, which means $\Phi(t) = at$, which means $(\Phi^* f)(t) = f(\frac{t}{a})$. Thus, the rescaled function $\Phi^* f$ will have a *different domain of analyticity* D_a compared to f. In fact, its domain of analyticity will be $D_a = D/a$.

Thus, 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. $\rho_{\max}$ becomes smaller). This will result in a larger bound.

SOLUTION of (6-6.a):

Jackson's theorem reads:

Theorem 6.6.1. 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

$$\|\cdot\|_{\infty,-1,1} \quad \text{or} \quad \|\cdot\|_{\infty,[-1,1]} \quad \text{or} \quad \|\cdot\|_{\infty,]-1,1[}.$$

HINT 1 for (6-6.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) $\forall \mathbf{x} \in X$: $\mathbf{x} \neq 0 \Leftrightarrow \|\mathbf{x}\| > 0$ (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 (6-6.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 (6.6.2) 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

$$E_n(\lambda f) = |\lambda| E_n(f) \quad \forall \lambda \in \mathbb{R} \,. \tag{6.6.3}$$

- (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):

$$E_n(f) = E_n(f-g+g) \overset{\text{(i)}}{\leq} E_n(f-g) + E_n(g) \,.$$

HINT 1 for (6-7.a): Use a local generalized cardinal basis representation, cf. [Lecture → § 5.3.3.3] and [Lecture → Eq. (5.3.3.4)]. Also invoke [Lecture → Thm. 6.6.2.3]. ⌟

SOLUTION of (6-7.a):

Let s be the piecewise cubic polynomial interpolant of f. Using the notations from [Lecture → Eq. (5.3.3.5)]

$$\begin{aligned} H_1(t) &:= \phi(\frac{x_i - t}{h_i}) \,, \\ H_2(t) &:= \phi(\frac{t - x_{i-1}}{h_i}) \,, \\ H_3(t) &:= -h_i\psi(\frac{x_i - t}{h_i}) \,, \\ H_4(t) &:= h_i\psi(\frac{t - x_{i-1}}{h_i}) \,, \end{aligned} \quad , \quad \begin{aligned} h_i &:= x_i - x_{i-1} \,, \quad i = 1, \ldots, m \,, \\ \phi(\tau) &:= 3\tau^2 - 2\tau^3 \,, \\ \psi(\tau) &:= \tau^3 - \tau^2 \,, \end{aligned}$$

we can rewrite $s|_{[x_{i-1},x_i]}$ using the local representation with cardinal basis polynomials: for $x_{i-1} \le t \le x_i$, $i = 1, \ldots, m$,

$$\begin{aligned} s(t) &= y_{i-1}H_1(t) + y_iH_2(t) + c_{i-1}H_3(t) + c_iH_4(t) \\ &= y_{i-1}H_1(t) + y_iH_2(t) + (f'(t_{i-1}) + \delta_{i-1})H_3(t) + (f'(t_i) + \delta_i)H_4(t) \\ &= y_{i-1}H_1(t) + y_iH_2(t) + f'(t_{i-1})H_3(t) + f'(t_i)H_4(t) + \delta_{i-1}H_3(t) + \delta_iH_4(t) \,. \end{aligned}$$

Hence, if we denote by $\tilde{s}$ the Hermite interpolant with exact slopes, taking the maximum over all mesh cells and using the estimate from [Lecture → Thm. 6.6.2.3]

$$\begin{aligned} \|f - s\|_{L^\infty([a,b])} &\le \|f - \tilde{s} + \tilde{s} - s\|_{L^\infty([a,b])} \le \|f - \tilde{s}\|_{L^\infty([a,b])} + \|\tilde{s} - s\|_{L^\infty([a,b])} \\ &\le O(h_{\mathcal{M}}^4) + \max_i \|\delta_{i-1}H_3(t) + \delta_iH_4(t)\|_{L^\infty([t_{i-1},t_i])} \\ &= O(h_{\mathcal{M}}^4) + O(h_{\mathcal{M}}^{\beta+1}) = O(h_{\mathcal{M}}^{\min(4,\beta+1)}) \,, \end{aligned}$$

since

$$\|H_3(t)\|_{L^\infty([x_{i-1},x_i])} = \|H_4(t)\|_{L^\infty([x_{i-1},x_i])} = O(h) \,.$$

To see this one easily computes that the functions H_3, H_4 attain their maximum at $t = x_i + \frac{2}{3}h_i$ resp. minimum at $t = x_i + \frac{1}{3}h_i$, with value $h_i((\frac{2}{3})^3 - (\frac{2}{3})^2)$. See also [Lecture → Fig. 162].

SOLUTION of (6-7.b):

C++ code 6.7.4: Sub-problem (6-7.b): Computation of cubic Hermite interpolant

```
template <typename Function>
Eigen::VectorXd fppchip(Function &&f, const Eigen::VectorXd &t,
                        const Eigen::VectorXd &x) {
  Eigen::VectorXd ret{x.size()};
  // TODO: (6-7.b) Compute ret.
  // START
  Eigen::VectorXd y = t.unaryExpr(f);
  unsigned int i_star = 0;
  double tmp, h_tmp, t1, t2, y1, y2, a1, a2, a3;
  for (unsigned int j = 0; j < x.size();) {
    // Stores the current interval index and some temporary variable
    // Find the interval for x_j while assuming that x values are sorted
    // and within [t_min, t_max]
    if (t(i_star) <= x(j) && x(j) <= t(i_star + 1)) {
      t1 = t(i_star);
      t2 = t(i_star + 1);
      h_tmp = t2 - t1;  // h\_tmp = h\_ for equidistant grid
      // values of interpolant
      y1 = y(i_star);
      y2 = y(i_star + 1);

      a1 = y2 - y1;
      a2 = a1;
      a3 = -a1 - a2;

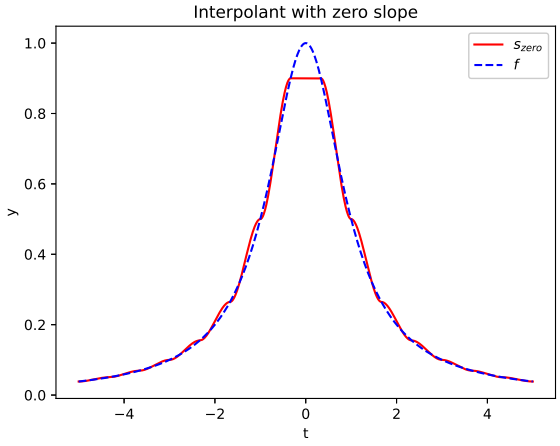
      // Compute s(x(j))
      tmp = (x(j) - t1) / h_tmp;
      ret(j) = y1 + (a1 + (a2 + a3 * tmp) * (tmp - 1.)) * tmp;

      ++j;
    } else {
      // Otherwise, go to next interval
      ++i_star;
      // Terminate if outside any interval
      if (i_star >= x.size() - 1) break;
      continue;
    }
  }

  // END
  return ret;
}
```

Get it on GitLab (pchi.hpp).

SOLUTION of (6-7.c):



◁ Flat-point piecewise cubic interpolant of the Runge function $f(t) = \dfrac{1}{1+t^2}$ in $[-5,5]$.

Obviously this interpolant provides a pathetic approximation.

C++ code 6.7.5: Sub-problem (6-7.c): Convergence plot for flat-point piecewise cubic interpolant

```
2  std::vector<double> fppchipConvergence() {
3    auto f = [](double x) { return 1. / (1. + x * x); };
4
5    constexpr double a = 5;  // Interval bounds will be (-a,a)
6    constexpr unsigned int M =
7        1000;  // Number of points in which to evaluate the interpolant
8
9    // Precompute values at which evaluate f
10   Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(M, -a, a);
11   Eigen::VectorXd fx = x.unaryExpr(f);
12
13   // Store error and number of nodes
14   std::vector<double> N_nodes, err_zero;
15
16   // TODO: (6-7.c) Compute err\_zero.
17   // START
18   for (unsigned int i = 4; i <= 512; i = i << 1) {
19     // Define subintervals and evaluate f there (find pairs (t,y))
20     Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(i, -a, a);
21
22     // compute reconstructed slopes
23     Eigen::VectorXd s_zero_x = fppchip(f, t, x);
24
25     // Compute infinity norm of error
26     err_zero.push_back((s_zero_x - fx).lpNorm<Eigen::Infinity>());
27     N_nodes.push_back(i);
28
29     if (i == 16) {
30       plt::figure();
31       plt::title("Interpolant with zero slope");
32       plt::xlabel("t");
33       plt::ylabel("y");
34       plt::plot(x, s_zero_x, "r", {{"label", "$s_{zero}$"}});
35       plt::plot(x, fx, "b--", {{"label", "$f$"}});
36       plt::legend();
```

```
37          plt::savefig("./cx_out/p_zero.png");
38        }
39      }
40      // END
41
42      return err_zero;
43  }
```

Get it on GitLab (pchi.hpp).

No. of nodes	error norm
4	0.735269
8	0.337813
16	0.099975
32	0.0253299
64	0.0100745
128	0.00494491
256	0.00244481
512	0.00119703

The table shows the corresponding error for each node of the flat-point piecewise cubic interpolation.

Doubling the numbers of mesh nodes, that is, halving the mesh width, leads to a reduction of the maximum norm of the interpolation error by a factor of 2. Thus, we observe asymptotic algebraic h-convergence of $\|f - s\|_{L^\infty([a.b])}$ with rate 1.

SOLUTION of (6-7.d):

According to Sub-problem (6-7.a), since $s'(x_j) = f'(x_j) - f'(x_j)$, i.e. $|\delta_j| = O(1)$, $\beta = 0$, the rate of asymptotic algebraic convergence is limited to $O(h)$.

SOLUTION of (6-7.e):

C++ code 6.7.7: Sub-problem (6-7.e): Definition of Hermite interpolant-class

```
class CubicHermiteInterpolant {
 public:
  /*!
   * \brief Construct the slopes from the data.
   * Use finite-differences or setting s'(x_j) = 0.
   * \param[in] t Vector of nodes (assumed equidistant and sorted).
   * \param[in] y Vector of values at nodes t.
   * \param[in] s Flag to set if you want to reconstruct or set the slopes to
   *zero.
   */
  template <typename Function>
  CubicHermiteInterpolant(Function &&f, const Eigen::VectorXd &t);
  virtual ~CubicHermiteInterpolant() = default;

  /*!
   * \brief Evaluate the intepolant at the nodes x.
   * The input is assumed sorted, unique and inside the interval
   * \param[in] x The vector of points x_i where to compute s(x_i).
   * \return Values of interpolant at x (vector)
   */
  Eigen::VectorXd eval(const Eigen::VectorXd &x) const;

 private:
  // Provided nodes and values (t,y) to compute spline,
  // Eigen vectors, c contains slopes
  // All have the same size n
  Eigen::VectorXd t_, y_, c_;
  // Difference t(i) - t(i-1)
  double h_;
  // Size of t, y and c.
  unsigned int n_;
};
```

Get it on GitLab (pchi.hpp).

C++ code 6.7.8: Sub-problem (6-7.e): Constructor of Hermite interpolant-class

```
template <typename Function>
CubicHermiteInterpolant::CubicHermiteInterpolant(Function &&f,
                                                 const Eigen::VectorXd &t)
    : t_(t), y_(t.unaryExpr(f)), c_(t.size()) {
  // Sanity check
  n_ = t_.size();
  assert(n_ == y_.size() && "t and y must have same dimension.");
  assert(n_ >= 3 && "Need at least three nodes.");
  h_ = t_(1) - t_(0);

  // Reconstruction of the slope,
  // TODO: (6-7.e) Implement reconstruction of slopes.
  // START
  // First order alternative:
  c_(0) = (-1 * y_(2) + 4 * y_(1) - 3 * y_(0)) / 2 / h_;
```

```
  for (int i = 1; i < n_ - 1; ++i) {
    c_(i) = (y_(i + 1) - y_(i - 1)) / 2 / h_;
  }

  // First order alternative:
  c_(n_ - 1) = (3 * y_(n_ - 1) - 4 * y_(n_ - 2) + 1 * y_(n_ - 3)) / 2 / h_;
  // END
}
```

Get it on GitLab (pchi.hpp).

C++ code 6.7.9: Sub-problem (6-7.e): Implementation of constructor of evaluation operator

```
Eigen::VectorXd CubicHermiteInterpolant::eval(const Eigen::VectorXd &x) const {
  Eigen::VectorXd ret(x.size());

  // TODO: (6-7.e) Implement evaluation function.
  // START
  unsigned int i_star = 0;
  double tmp, h_tmp, t1, t2, y1, y2, c1, c2, a1, a2, a3;
  for (unsigned int j = 0; j < x.size();) {
    // Stores the current interval index and some temporary variable
    // Find the interval for x_j while assuming that x values are sorted
    // and within [t_min, t_max]
    if (t_(i_star) <= x(j) && x(j) <= t_(i_star + 1)) {
      t1 = t_(i_star);
      t2 = t_(i_star + 1);
      h_tmp = t2 - t1;  // h_tmp = h_ for equidistant grid
      y1 = y_(i_star);
      y2 = y_(i_star + 1);
      c1 = c_(i_star);
      c2 = c_(i_star + 1);
      a1 = y2 - y1;
      a2 = a1 - h_tmp * c1;
      a3 = h_tmp * c2 - a1 - a2;

      // Compute s(x(j))
      tmp = (x(j) - t1) / h_tmp;
      ret(j) = y1 + (a1 + (a2 + a3 * tmp) * (tmp - 1.)) * tmp;

      ++j;
    } else {
      // Otherwise, go to next interval
      ++i_star;
      // Terminate if outside any interval
      if (i_star >= t_.size() - 1) break;
      continue;
    }
  }
  // END

  return ret;
}
```

Get it on GitLab (pchi.hpp).

SOLUTION of (6-7.f):

C++ code 6.7.10: Sub-problem (6-7.f): Convergence of Hermite interpolation with reconstructed slopes

```
std::vector<double> rspchipConververgence() {
  auto f = [](double x) { return 1. / (1. + x * x); };

  constexpr double a = 5;  // Interval bounds will be (-a,a)
  constexpr unsigned int M =
      1000;  // Number of points in which to evaluate the interpolant

  // Precompute values at which evaluate f
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(M, -a, a);
  Eigen::VectorXd fx = x.unaryExpr(f);

  // Store error and number of nodes
  std::vector<double> N_nodes, err_reconstr;

  // TODO: (6-7.f) Compute err\_reconstr.
  // START
  for (unsigned int i = 4; i <= 512; i = i << 1) {
    // Define subintervals and evaluate f there (find pairs (t,y))
    Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(i, -a, a);

    // Construct PCHI with zero and reconstructed slopes
    CubicHermiteInterpolant s_reconstr(f, t);

    // Evaluate interpolant
    Eigen::VectorXd s_reconstr_x = s_reconstr.eval(x);

    // Compute infinity norm of error
    err_reconstr.push_back((s_reconstr_x - fx).lpNorm<Eigen::Infinity>());
    N_nodes.push_back(i);

    if (i == 16) {
      plt::figure();
      plt::title("Interpolant with reconstructed slope");
      plt::xlabel("t");
      plt::ylabel("y");
      plt::plot(x, s_reconstr_x, "r", {{"label", "$s_{reconstr}$"}});
      plt::plot(x, fx, "b--", {{"label", "$f$"}});
      plt::legend();
      plt::savefig("./cx_out/p_reconstr.png");
    }
  }
  // END

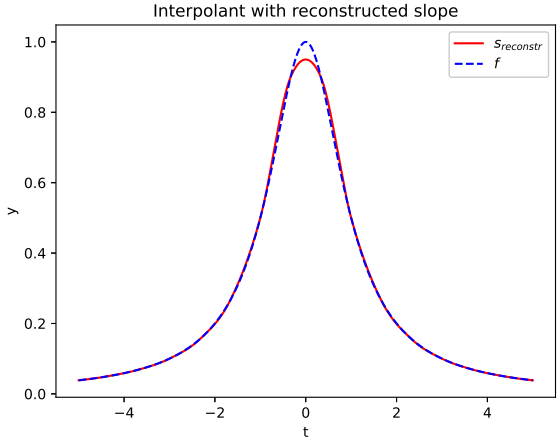
  return err_reconstr;
}
```

Get it on GitLab (pchi.hpp).

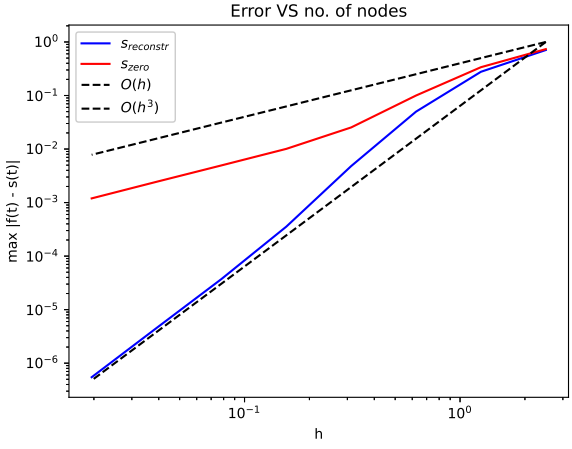
No. of nodes	error norm
4	0.706989
8	0.277399
16	0.0499862
32	0.00480491
64	0.000355225
128	3.72701×10^{-5}
256	4.53776×10^{-6}
512	5.46835×10^{-7}

The table shows the corresponding error for each node of the Hermite interpolation with reconstructed slopes.

Halving the meshwidth engenders a reduction of the maximum norm of the interpolation error by roughly a factor of 8. Thus, in this case we observe algebraic h-convergence of $\|t - s\|_{L^\infty([a,b])}$ with rate 3 as $h_{\mathcal{M}} \to 0$.



◁ Cubic Hermite interpolant of the Runge function $f(t) = \dfrac{1}{1+t^2}$ in $[-5,5]$.



◁ Convergence plot of the two cubic interpolation methods.

HINT 1 for (6-7.g): Use Taylor expansion [Lecture → Eq. (1.5.4.28)] of f in x_j to express $f(x_{j\pm1})$ and $f(x_{j\pm2})$ as a "perturbation" of $f(x_j)$. ⌟

SOLUTION of (6-7.g):

For the sake of simplicity this solution assumes an equidistant mesh with meshwidth $h = x_j - x_{j-1}$, $j = 1, \ldots, m$. The same approach works for more general meshes, but incurs distracting notational complexity. Throughout we write $s := \mathsf{I}_H f$.

First, we show that the approximation $s'(x_j) = f'(x_j) + O(h^2)$. This follows from Taylor expansion:

$$f(x) = f(x_j) + f'(x_j)(x - x_j) + f''(x_j)(x - x_j)^2/2 + O(|x - x_j|^3) \quad \text{as} \quad x \to x_j \,. \tag{6.7.11}$$

Using $x = x_{j-1}$ and $x = x_{j+1}$ (and $h = x_{j+1} - x_j$):

$$\begin{aligned} \frac{f(x_{j+1}) - f(x_j)}{h} &= f'(x_j) + f''(x_j)h/2 + O(h^2) \,, \\ \frac{f(x_{j-1}) - f(x_j)}{h} &= -f'(x_j) + f''(x_j)h/2 + O(h^2) \,. \end{aligned}$$

Subtracting the second equation from the first equation we get

$$\frac{f(x_{j+1}) - f(x_{j-1})}{h} = 2f'(x_j) + O(h^2) \,.$$

For the one-sided difference we expand at $x = x_{j+2}$ and $x = x_{j+1}$ ($h = x_{j+1} - x_j = x_{j+2} - x_{j+1}$):

$$\begin{aligned} \frac{f(x_{j+1}) - f(x_j)}{h} &= f'(x_j) + f''(x_j)h/2 + O(h^2) \,, \\ \frac{f(x_{j+2}) - f(x_j)}{2h} &= f'(x_j) + f''(x_j)h + O(h^2) \,. \end{aligned} \tag{6.7.12}$$

In light of the definition of the slope in Def. 6.7.6 this yields for the slopes at interior mesh nodes

$$|s'(x_j) - f'(x_j)| = O(h^2) \quad \text{as} \quad h \to 0 \quad \text{for} \quad j = 1, \ldots, m-1 \,. \tag{6.7.13}$$

Subtracting the first equation of (6.7.12) from half of the second equation:

$$\begin{aligned} \frac{f(x_{j+2}) - f(x_j)}{4h} - \frac{f(x_{j+1}) - f(x_j)}{h} &= f'(x_j)(1/2 - 1) + O(h^2) \,, \\ \frac{f(x_{j+1}) - f(x_j) - 4f(x_{j+1}) + 4f(x_j)}{4h} &= f'(x_j)(1/2 - 1) + O(h^2) \,, \\ \frac{-f(x_{j+2}) + 4f(x_{j+1}) - 3f(x_j)}{2h} &= f'(x_j) + O(h^2) \,. \end{aligned}$$

We apply this to x_0 and x_m, which gives (6.7.13) also for $j = 1, m$.

Finally we combine (6.7.13) with the result of Sub-problem (6-7.a) and find $\|f - s\|_{L^\infty([a,b])} = O(h^3)$ for $h \to 0$. The rate of convergence is limited to 3.

SOLUTION of (6-8.a):

C++ code 6.8.2: Implementation of a function providing the monomial coefficients of a Chebychev expansion

```
Eigen::VectorXd chebexpToMonom(const Eigen::VectorXd &a) {
  const int n = a.size() -1; // degree of polynomial
  assert(n >= 0);
  // Vector for returning monomial coefficients
  Eigen::VectorXd r{ Eigen::VectorXd::Zero(n+1) };
  r[0] = a[0]; // T_0 ≡ 1
  if (n > 0) {
    r[1] = a[1];  // T_1(t) = t
    if (n > 1) {
      // two vectors temporarily storing the monomial coefficients
      // of Chebychev polynomials of two consecutive degrees
      std::array<Eigen::VectorXd, 2> c
         { Eigen::VectorXd::Zero(n+1), Eigen::VectorXd::Zero(n+1) };
      c[0][0] = 1.0; c[1][1] = 1.0; // For T_0 and T_1
      // Run through all Chebychev polynomials
      for (int j=2; j<=n; ++j) {
        // Update monomial coefficients by 3-term recursion
        c[j%2][0] *= -1.0;
        for (int k=1; k <= j; ++k) {
          // Formula for 3-term recursion
          c[j%2][k] = 2*c[(j+1)%2][k-1] - c[j%2][k];
        }
        // Add monomial coefficients of contribution of T_j
        r += a[j]*c[j%2];
      }}}
  return r;
}
```

SOLUTION of (6-9.a):

We write $P := \mathbb{C} \setminus D \subset \mathbb{C}$ for the complement of the domain of analyticity (the "domain of singularity") of f. Multiplication with b does not have any impact on analyticity. By Thm. 6.9.2 we have

$$\begin{aligned} P &= \{z \in \mathbb{C} : \ a + \sin(2\pi z) \in \mathbb{R}_0^-\} \\ &= \{z \in \mathbb{C} : \ \mathrm{Im}(a + \sin(2\pi z)) = 0 \quad \wedge \quad \mathrm{Re}(a + \sin(2\pi z)) \le 0\} \ . \end{aligned}$$

Next, we use (6.9.1) with $z = x + \imath y$, $x, y \in \mathbb{R}$:

$$\sin(2\pi z) = \sin(2\pi x)\cosh(2\pi y) + \imath \cos(2\pi x)\sinh(2\pi y) \ ,$$

which means for $z = x + \imath y$

$$\mathrm{Im}(a + \sin(2\pi z)) = \cos(2\pi x)\sinh(2\pi y) \quad , \quad \mathrm{Re}(a + \sin(2\pi z)) = a + \sin(2\pi x)\cosh(2\pi y) \ .$$

We first examine $\mathrm{Im}(a + \sin(2\pi z)) = 0 \Leftrightarrow \cos(2\pi x)\sinh(2\pi y) = 0$:

$$\mathrm{Im}(a + \sin(2\pi z)) = 0 \quad \Leftrightarrow \quad y = 0 \quad \vee \quad x \in \tfrac{1}{2}\mathbb{Z} + \tfrac{1}{4} \ .$$

The case $y = 0$ need not be considered any further, because, since $a > 1$,

$$y = 0 \quad \Rightarrow \quad \mathrm{Re}(a + \sin(2\pi z)) = a + \sin(2\pi x) > 0 \ .$$

The other case $x \in \frac{1}{2}\mathbb{Z} + \frac{1}{4}$ means

$$x = (2k+1)\tfrac{1}{4} \ , \quad k \in \mathbb{Z} \ , \quad \Rightarrow \quad \sin(2\pi x) = (-1)^k \ .$$

Hence, $\mathrm{Re}(a + \sin(2\pi z)) \le 0$ is possible only if

$$x = (4k+3)\frac{1}{4} \ , \quad k \in \mathbb{Z} \ , \quad a - \cosh(2\pi y) \le 0 \quad \Leftrightarrow \quad |y| \ge \frac{1}{2\pi}\cosh^{-1}(a) \ .$$

Summing up we have found

$$D = \mathbb{C} \setminus \{z \in \mathbb{C} : \ \mathrm{Re}(z) \in \mathbb{Z} + \tfrac{3}{4} \ , \ |\mathrm{Im}(z)| \ge \zeta, \ \zeta > 0, \ \cosh(2\pi\zeta) = a\} \ .$$

The presence of the parameter b has no impact on the domain of analyticity.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \mathbb{Z} + \frac{3}{4} \ , \\ \mathrm{Im}(z) \in \{y \in \mathbb{R} : \ |y| \ge q, \ q > 0, \ \cosh(2\pi q) = a\} \end{array} \right\} \ .$$

SOLUTION of (6-9.b):

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \mathbb{Z}\,, \\ \mathrm{Im}(z) \in \{ y \in \mathbb{R} : \ |y| \geq q,\ \cosh(2\pi q) = b \} \end{array} \right\} .$$

SOLUTION of (6-9.c):

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \mathbb{Z} + \frac{1}{4}\,, \\ \mathrm{Im}(z) \in \{ y \in \mathbb{R} : \, |y| \geq q, \, \cosh(2\pi q) = b \} \end{array} \right\} .$$

HINT 1 for (7-1.a): Use as exact value of the integral $I \approx 0.870267525725852642$. ⌟

SOLUTION of (7-1.a):

In our implementation we defined an auxiliary function that approximates the integral for a given function and quadrature rule according to [Lecture → Def. 7.2.0.1].

C++-code 7.1.2: Implementation of the auxiliary function `integrate()`

```
template <class Function>
double integrate(const QuadRule& qr, const Function& f) {
  double I = 0;

  // TODO: (7-1.a) optional:  compute the integral with this helper function
  // START

  // Compute weighted sum of function values according to
  // the concept of a generic quadrature formula
  for (unsigned int i = 0; i < qr.weights_.size(); ++i) {
    I += qr.weights_(i) * f(qr.nodes_(i));
  }

  // END
  return I;
}
```

C++-code 7.1.3: Implementation of the function `gaussConv()`

```
template <class Function>
double gaussConv(const Function& fh, const double I_ex, const unsigned int N) {
  std::vector<double> evals,  // used to save no. of quad nodes
      error,                  // used to save the error
      evals_ref;  // used to plot reference line of order of convergence

  double I_N = 0.;  // best approx.  of integral
  plt::figure();

  // TODO: (7-1.a) write code that plots the no.  of nodes versus the error
  // using matplotlibcpp and returns the best approximation
  // START

  // Build integrand
  auto f = [fh](double x) { return std::asin(x) * fh(x); };

  double I;
  for (unsigned int n = 1; n <= N; ++n) {
    QuadRule qr = gaussquad(n);  // Create quadrature rule

    I = integrate(qr, f);  // Compute integral

    evals.push_back(n);  // Save no.  of quadrature nodes
    evals_ref.push_back(
        std::pow(n, -3));  // used for plotting to show convergence rate
    error.push_back(std::abs(I - I_ex));  // Save error
  }
  // save best approx.  of integral
```

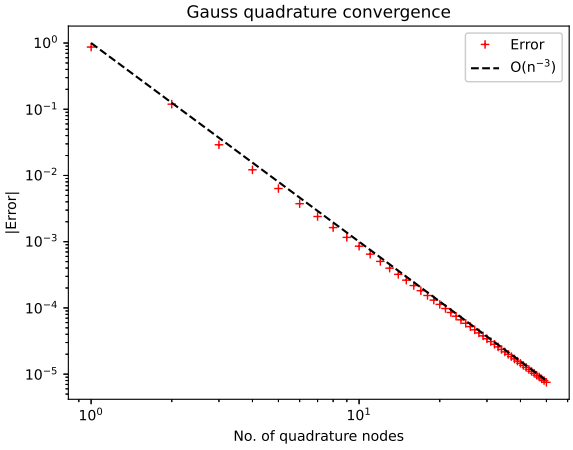
```
30    I_N = I;
31
32    // Create convergence plots
33    plt::title("Gauss quadrature convergence");
34    plt::loglog(evals, error, " +r", {{"label", "Error"}});  // plot error
35    plt::loglog(evals, evals_ref, "k--",
36                {{"label", "O(n^-3)"}}); // reference line
37    plt::xlabel("No. of quadrature nodes");
38    plt::ylabel("|Error|");
39    plt::legend("best");
40    // END
41
42    plt::savefig("./cx_out/GaussConv.png");
43    // return the best approximation
44    return I_N;
45  }
```

Get it on GitLab (gaussConvergence.hpp).

The convergence plot is included in the solution of the next sub-problem.

HINT 1 for (7-1.b): Types of convergence have been introduced in [Lecture → Def. 6.2.2.7] for approximation errors. They apply to quadrature errors as well. ⌟

SOLUTION of (7-1.b):



◁ *Doubly logarithmic* plot of the quadrature error.

Using other axes scales yields plots from which the type of convergence is not immediately obvious.

The plot indicates algebraic convergence (being a straight line in log-log mode), with a rate of about 3:

$\texttt{error} = O(n^{-3})$ for $n \to \infty$.

HINT 1 for (7-1.c): Combine the results of [Lecture → § 7.4.3.2], [Lecture → Lemma 7.4.3.1], and [Lecture → Rem. 6.2.3.26]. ⌟

HINT 2 for (7-1.c): Use the transformation $t = \sin(x)$. ⌟

SOLUTION of (7-1.c):

With the change of variable $t = \sin(x)$, $\mathrm{d}t = \cos x \mathrm{d}x$, we obtain

$$I = \int_{-1}^{1} \arcsin(t)\, f(t)\, \mathrm{d}t = \int_{-\pi/2}^{\pi/2} x\, f(\sin(x)) \cos(x)\, \mathrm{d}x \ . \tag{7.1.4}$$

The change of variable has to provide an analytic integrand on the integration interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ The integrand is analytic, provided f has this property on $[-1,1]$, because the trigonometric functions $\sin$ and $\cos$ are entire functions, that is, analytic on $\mathbb{C}$. Then invoke [Lecture $\to$ Thm. 6.2.2.68].

If the integrand possesses an analytic extension beyond $[-\frac{\pi}{2}, \frac{\pi}{2}]$ we can appeal to the results of [Lecture $\to$ Rem. 6.2.3.26] and [Lecture $\to$ Thm. 7.4.3.3] to conclude exponential convergence of the quadrature errors.

SOLUTION of (7-1.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:

$$\begin{cases} \Phi : [-1,1] & \to [a,b] \\ \hat{x} & \to a + \frac{b-a}{2}(\hat{x}+1) = \frac{b-a}{2}\hat{x} + \frac{a+b}{2} \end{cases} \tag{7.1.5}$$

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:

$$\int_a^b g(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} g\left(\frac{b-a}{2}\hat{x}+\frac{a+b}{2}\right)d\hat{x} \approx \frac{b-a}{2}\sum_{i=1}^{n}\hat{w}_i f\left(\frac{b-a}{2}\hat{x}+\frac{a+b}{2}\right) \tag{7.1.6}$$

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:

$$w_i = \frac{b-a}{2}\hat{w}_i \tag{7.1.7}$$

In our example, this means that $w_i = \frac{\pi}{2}\hat{w}_i$.

C++-code 7.1.8: Implementation of `gaussConvCV()`

```
2  template <class Function>
3  double gaussConvCV(const Function& f, const double I_ex, const unsigned int N) {
4    std::vector<double> evals,  // Used to save no.  of quad nodes
5        error;                  // Used to save the error
6
7    double I_N = 0.;  // best approx.  of integral
8    plt::figure();
9
10   // TODO: (7-1.d) complete to perform Gauss-Legendre quadrature with
         the
11   // variable transform of the integral derived in 8-1.c)
12   // write code that plots the no.  of nodes versus the error
13   // using matplotlibcpp and returns the best approximation
14   // START
15
16   // Lambda function providing transformed integrand
17   auto g = [f](double x) { return x * f(std::sin(x)) * std::cos(x); };
18
19   double I;
20   for (unsigned int n = 1; n <= N; ++n) {
21     QuadRule qr = gaussquad(n);  // Obtain quadrature rule
22
23     // Transform nodes and weights to new interval
24     Eigen::VectorXd w = qr.weights_ * M_PI / 2;
25     Eigen::VectorXd c =
26         (-M_PI / 2 + M_PI / 2 * (qr.nodes_.array() + 1)).matrix();
27
```

```
    // Evaluate g at quadrature nodes c
    Eigen::VectorXd gc = c.unaryExpr(g);

    // Same as I = \sum_{i=1}^{n} w_i g(c_i)
    I = w.dot(gc);

    evals.push_back(n);                    // Save no.  of quadrature nodes
    error.push_back(std::abs(I - I_ex));  // Save error
  }

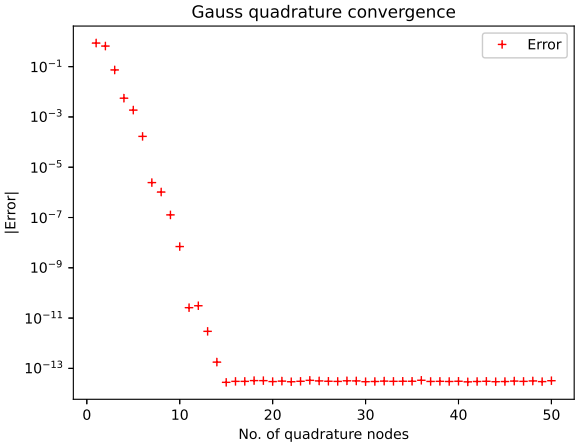
  // save approximation of integral computed with highest-order rule
  I_N = I;

  // Create convergence plots
  plt::title("Gauss quadrature convergence");
  plt::semilogy(
      evals, error, " +r",
      {{"label", "Error"}});  // plot error as lin-log plot to see
          convergence
  plt::xlabel("No. of quadrature nodes");
  plt::ylabel("|Error|");
  plt::legend("best");
  // END

  plt::savefig("./cx_out/GaussConvCV.png");
  // return the best approximation
  return I_N;
}
```

Get it on GitLab (gaussConvergence.hpp).

SOLUTION of (7-1.e):



◁ *Linear-logarithmic plot* of the quadrature error of Gauss-Legendre quadrature rules applied to the transformed integral as a function of the number of quadrature nodes.

The plot indicates exponential convergence (being a straight line in lin-log scale).

SOLUTION of (7-1.f):

In Sub-problem (7-1.b) we found merely asymptotic algebraic convergence of the quadrature error of Gauss-Legendre quadrature in terms of the number of quadrature nodes, whereas in Sub-problem (7-1.d) we observe exponential convergence.

The reason is that the function $t \mapsto \arcsin t$ merely belongs to $C^0([-1,1])$, but its derivative $t \mapsto \frac{1}{\sqrt{1-t^2}}$ has singularities in the points $t = \pm 1$. Hence, it and the entire integrand in (7.1.1) has finite smoothness only and *cannot be extended analytically* beyond the interval $[-1,1]$, which rules out exponential convergence.

Conversely, in Sub-problem (7-1.d) the change of variable provides a perfectly smooth integrand: $x \mapsto x\cos(x)\sinh(\sin x)$ that possesses an analytic extension to some complex neighborhood of $[-\frac{\pi}{2}, \frac{\pi}{2}]$. This makes it possible to apply the estimated of [Lecture → Thm. 7.4.3.3] and [Lecture → Eq. (6.2.3.28)] and conclude exponential convergence in the number of quadrature nodes.

HINT 1 for (7-10.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 (7-10.a):

By [Lecture → Lemma 7.4.2.17], 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:

$$P_{n-1}(\xi_l^n) P_{n-1}(\xi_{l+1}^n) \geq 0 \,. \tag{7.10.1}$$

Recall that, by construction, we have the following orthogonality property:

$$\int_{-1}^{1} P_{n-1} q \, dt = 0, \qquad q \in \mathcal{P}_{n-2} \,. \tag{7.10.2}$$

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 (7.10.2) and the fact that Gaussian quadrature is exact on P_{2n-1} we obtain:

$$0 = \int_{-1}^{1} P_{n-1} q \, dt \;\; = \sum_{j=1}^{n} w_j P_{n-1}(\xi_j^n) q(\xi_j^n) = w_l P_{n-1}(\xi_l^n) q(\xi_l^n) + w_{l+1} P_{n-1}(\xi_{l+1}^n) q(\xi_{l+1}^n) \,.$$

Hence, both terms of the right hand side must have different sign, namely:

$$[w_l P_{n-1}(\xi_l^n) q(\xi_l^n)] \big[w_{l+1} P_{n-1}(\xi_{l+1}^n) q(\xi_{l+1}^n) \big] \leq 0 \,.$$

On the other hand, by [Lecture → Lemma 7.4.3.1] 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 (7.10.1), 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 (7.10.2) 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. (7.4.2.21)]. 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. (7.4.2.21)] 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:

$$(-1)^{n+1-j} P_{n+1}(\xi_j^n) > 0, \qquad j = 1, \dots, n \,. \tag{7.10.3}$$

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 (7.10.3) yields the result for $n+1$.

SOLUTION of (7-10.b):

Differentiating [Lecture → Eq. (7.4.2.21)] 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. (7.4.2.21)], gives the desired recursions.

SOLUTION of (7-10.c):

C++ code 7.10.4: Suggested solution for `legvals()`

```
void legvals(const Eigen::VectorXd& x, Eigen::MatrixXd& Lx,
             Eigen::MatrixXd& DLx) {
  const unsigned int n = Lx.cols() - 1;
  const unsigned int N = x.size();

  // TODO: (7-10.c) Fill matrices Lx and DLx with values of the legendre
  // polynomials at x, i.e.  {P_k(x_j)}_{jk} and {P'_k(x_j)}_{jk} START
  assert(n >= 2);
  for (unsigned int j = 0; j < N; ++j) {
    Lx(j, 0) = 1.;
    Lx(j, 1) = x(j);
    DLx(j, 0) = 0;
    DLx(j, 1) = 1.;
    for (unsigned int k = 2; k <= n; ++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);
    }
  }
  // END
}
```

Get it on GitLab (legendre.hpp).

HINT 1 for (7-10.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 (7-10.d):

The proposed implementation uses the function `Pkx`, which you can find below in Code 7.10.6.

C++-code 7.10.5: Suggested solution for `gaussPts`

```
Eigen::MatrixXd gaussPts(const unsigned int n, const double rtol = 1e-10,
                         const double atol = 1e-12) {
  Eigen::MatrixXd zeros = Eigen::MatrixXd::Zero(n, n);

  // TODO: (7-10.d) Find the zeros of the legendre polynomials using the secant
  // method.  START
  double x0, x1, f0, f1, s;
  for (unsigned int k = 1; k <= n; ++k) {
    for (unsigned int j = 1; j <= k; ++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 (unsigned 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;
        }
      }
    }
  }
  // END
  return zeros;
}
```

Get it on GitLab (legendre.hpp).

C++-code 7.10.6: Implementation of `Pkx`

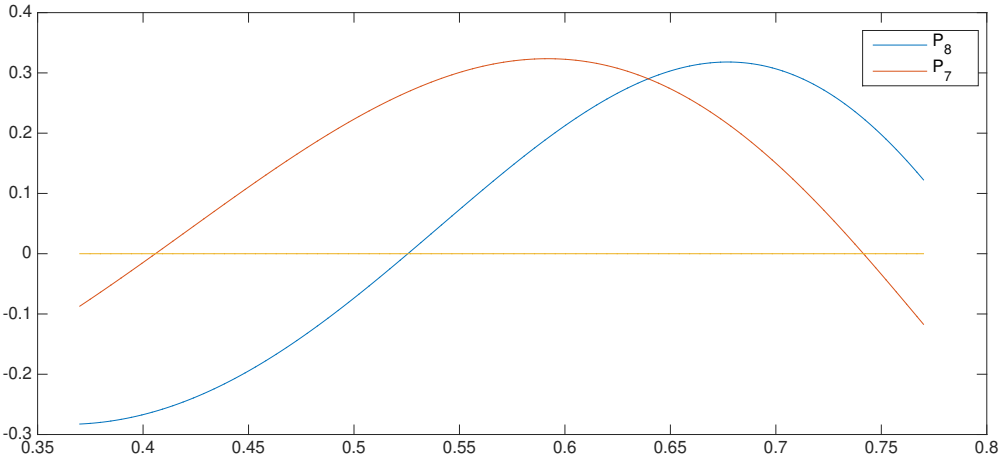
```
double Pnx(const double x, const int n) {
  Eigen::VectorXd Px(n + 1);

  // TODO: (7-10.d, optional) Evaluate P_n(x) for a scalar x and integer
  // n.  START
  assert(n >= 1);
```

```
8     Px(0) = 1.;
9     Px(1) = x;
10    for (unsigned int k = 2; k <= n; ++k) {
11      Px(k) = (2 * k - 1.) / k * x * Px(k - 1) - (k - 1.) / k * Px(k - 2);
12    }
13    // END
14    return Px(n);
15  }
```

Get it on GitLab (legendre.hpp).

HINT 1 for (7-10.e): Fig. 68 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).



SOLUTION of (7-10.e):

See file `legendre.cpp` for the implementation. The zeros of $P_k, k = 1, \ldots, 7$ are all correctly computed, but the zero $\zeta_6^8 \approx 0.525532$ 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. 68, 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.

SOLUTION of (7-10.f):

You can find the complete implementation in the file `legendre.cpp`.

C++-code 7.10.7: Suggested solution for `gausZerosLegendrets_regulaFalsi` (corrected version)

```
Eigen::MatrixXd gaussPts_regulaFalsi(const unsigned int n,
                                     const double rtol = 1e-10,
                                     const double atol = 1e-12) {
  Eigen::MatrixXd zeros = Eigen::MatrixXd::Zero(n, n);

  // TODO: (7-10.f) Find the zeros of the legendre polynomials using the secant
  // method with regula falsi.  START
  double x0, x1, f0, f1, s;
  for (unsigned int k = 1; k <= n; ++k) {
    for (unsigned int j = 1; j <= k; ++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 (unsigned int i = 0; i < 1e4; ++i) {
        f1 = Pnx(x1, k);
        s = f1 * (x1 - x0) / (f1 - f0);
        if (Pnx(x1 - s, k) * f1 < 0) {  // 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;
        }
      }
    }
  }
  // END
  return zeros;
}
```

Get it on GitLab (legendre.hpp).

SOLUTION of (7-11.a):

We have to implement the formula from [Lecture → Eq. (7.5.0.17)]:

$$\int_a^b f(t)\,\mathrm{d}t \approx T_n(f) := h\left(\tfrac{1}{2}f(a) + \sum_{k=1}^{n-2} f(kh) + \tfrac{1}{2}f(b)\right), \quad h := \frac{b-a}{n-1},$$

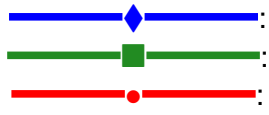
where the sum is empty for $n = 2$.

- left box: `0.5 * h`,
- middle box: `0.5 * h`,
- right box: `h`.

SOLUTION of (7-11.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.

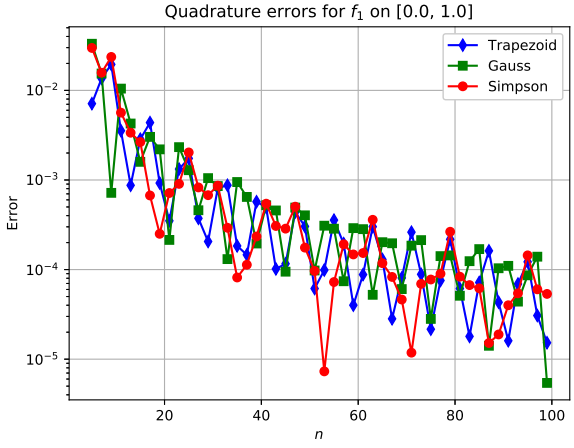


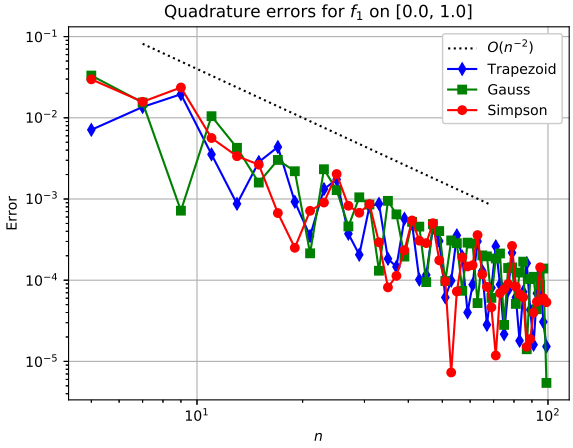
: ✔ TR ✔ SR ✔ GR

: ✔ TR ✔ SR ✔ GR

: ✔ TR ✔ SR ✔ GR

The actual association of rules and error curves as as follows:





SOLUTION of (7-11.c):

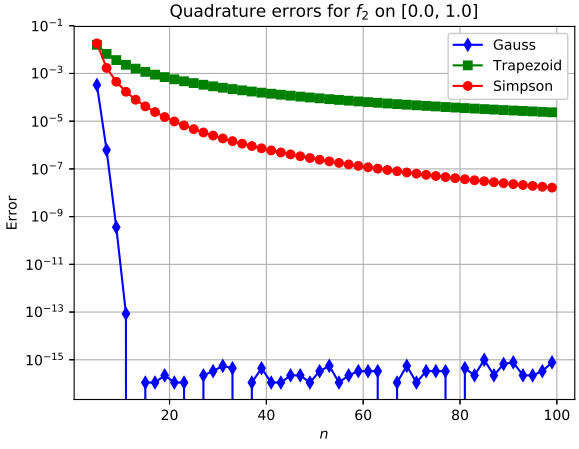
The integrand belongs to $C^\infty(\mathbb{R})$, it is even analytic.

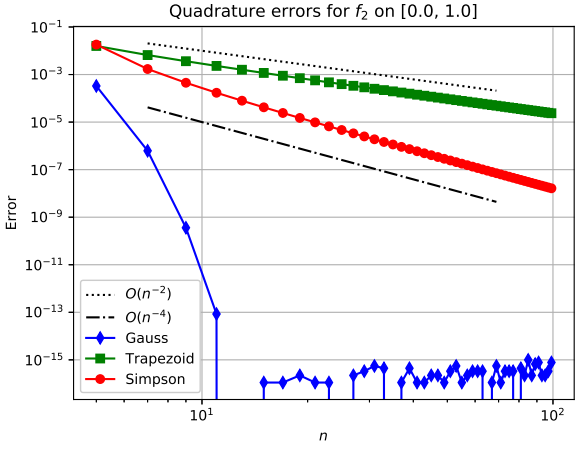
Therefore the the composite quadrature rules can achieve their maximal rate of (algebraic) convergence, 2 for the equidistant trapezoidal rules, and 4 for the equidistant Simpson rules, see [Lecture → Exp. 7.5.0.12].

Gauss quadrature will display exponential convergence as in [Lecture → Ex. 7.4.3.9], evident from aligned error points in a lin-log scale.



The actual association of rules and error curves as as follows:

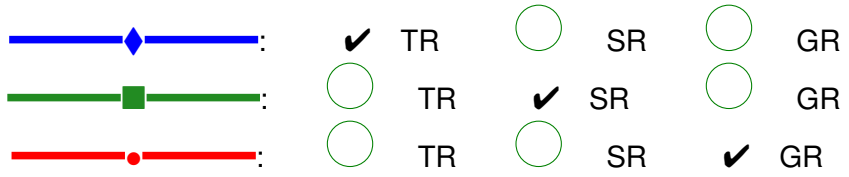




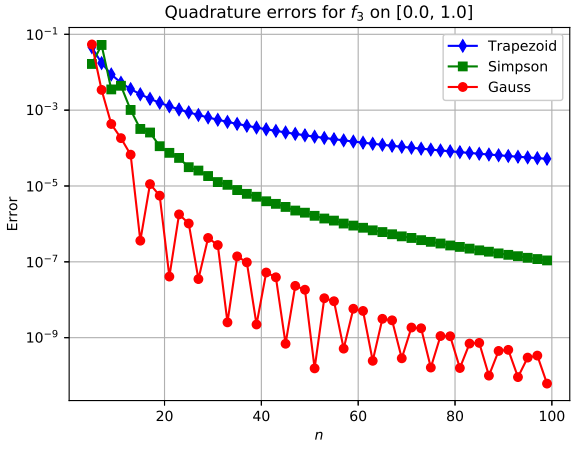
SOLUTION of (7-11.d):

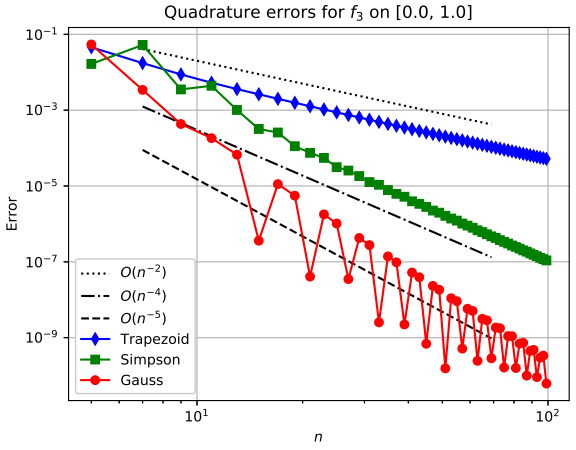
The integrand is not C^∞, but only $\in C^4$ and piecewise smooth.

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.



The true association of rules and error curves as as follows:





SOLUTION of (7-11.e):

(i) **true**, see [Lecture → Thm. 7.4.1.6].

(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 7.3].

(iii) **true**, because the proof of [Lecture → Lemma 7.4.3.1] 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 $\mathbb{Z}$.

SOLUTION of (7-12.a):

From Line 4 of Code 7.12.2 we read off that the $n+1$ quadrature nodes are

$$c_j = \cos\left(j\frac{\pi}{n}\right), \quad j = 0, \ldots, n \ .$$

Remark. The nodes are the sites of the extrema of the Chebyshev polynomial of degree n.

SOLUTION of (7-12.b):

It is important to realize that `m = n/2` is an integer division: $m = \lfloor \frac{n}{2} \rfloor$.

According to we have

$$w_j = \frac{a_j}{n}\left(1 - \sum_{\ell=1}^{m} \frac{b_\ell}{4\ell^2 - 1} \cos\left(2\ell j \frac{\pi}{n}\right)\right), \quad j = 0, \ldots, n \ , \tag{7.12.3}$$

where

$$m := \begin{cases} n/2 & \text{for even } n \ , \\ n-1/2 & \text{for odd } n \ , \end{cases}$$

$$a_j := \begin{cases} 1 & \text{for } j = 0, n \ , \\ 2 & \text{else,} \end{cases} \quad j = 0, \ldots, n \ ,$$

$$b_\ell := \begin{cases} 1 & \text{for } \ell = \frac{n}{2} \ , \\ 2 & \text{else,} \end{cases} \quad \ell = 1, \ldots, m \ .$$

A collapsed sum is supposed to evaluate to zero. There are several ways to write this formula, for instance, one could separately state

$$w_0 = w_n = \frac{1}{n}\left(1 - \sum_{\ell=1}^{m} \frac{b_\ell}{4\ell^2 - 1}\right) .$$

One could also distinguish the cases of even and odd n.

SOLUTION of (7-12.c):

The order of a quadrature rule is one plus the largest degree d of uni-variate polynomials that are still integrated exactly by the quadrature rule. Thanks to linearity, we need to check this for the elements of a basis of $\mathcal{P}_d$ only and we opt for the convenient basis of monomials $\{t \mapsto t^k, k = 0, \ldots, d\}$, for which we find the exact integral values

$$\int_{-1}^{1} t^k \,\mathrm{d}t = \begin{cases} 2/k+1 & \text{for even} \quad k\,, \\ 0 & \text{for odd} \quad k\,, \end{cases} \qquad k \in \mathbb{N}_0\,. \tag{7.12.4}$$

Applied to the monomials a generic $n+1$-point quadrature formula with nodes/weights c_j/w_j, $j = 0, \ldots, n$ yields

$$Q(\{t \mapsto t^k\}) = \sum_{j=0}^{n} w_j c_j^k\,, \quad k \in \mathbb{N}_0\,. \tag{7.12.5}$$

This is the complete implementation of the function:

C++ code 7.12.6: Implementation of `determineOrderCCF()`

```
2  unsigned int determineOrderCCF(unsigned int n) {
3    const CCFQuadRule ccfqr(n);
4    unsigned int d =
5        0;          // Degree of monomial, whose integral value is being
           tested
6    double quaderr;  // Quadrature error
7    double I_ex;     // Exact integral value
8    double I_qr;     // Value computed with CCF quadrature rule
9    // Predefined fixed tolerances
10   const double abstol = std::numeric_limits<double>::epsilon() * n * 2;
11   const double reltol = std::numeric_limits<double>::epsilon() * n * 20;
12   // Main quadrature loop
13   do {
14     // Compute powers of nodes:  c_j^d
15     const Eigen::ArrayXd pdvals = ccfqr.nodes().array().pow(d);
16     // Evaluate quadrature formula:  Implicit summation via dot product
17     I_qr = ccfqr.weights().dot(pdvals.matrix());
18     // Exact integral value according to (7.12.4)
19     I_ex = (d % 2 == 0) ? (2.0 / (d + 1)) : 0.0;
20     quaderr = std::abs(I_qr - I_ex);
21     d = d + 1;
22   } while ((quaderr <= abstol) ||
23            (quaderr <= reltol * I_ex));  //
24   return d - 1;
25 }
```

Get it on GitLab (clenshawcurtisfejer.hpp).

Of course, testing `I_ex == I_qr` is pointless, because both values have been computed numerically and will certainly be affected by roundoff. This is why we have to test for absolute and relative closeness in Line 23 of the code.

Note that any other basis of $\mathcal{P}_k$ could also be used, for instance Legendre polynomials.

SOLUTION of (7-12.d):

Of course, we aim to compute the weights w_j from (7.12.3) by means of the discrete Fourier transform (DFT) [Lecture → Def. 4.2.1.18]. To that end we rewrite the formula. Using the notations from Subproblem (7-12.b), we abbreviate

$$z_\ell := \frac{b_\ell}{4\ell^2 - 1} \in \mathbb{R} \quad \ell = 1, \ldots, m \quad , \quad z_0 := 1 \, ,$$

which permits us to express

$$w_j = \frac{a_j}{n}(1 - \widetilde{w}_j) \, , \quad \widetilde{w}_j := \sum_{\ell=1}^{m} z_\ell \cos(2\ell j \frac{\pi}{n}) \, , \quad j = 0, \ldots, n-1 \, . \tag{7.12.7}$$

Considering only $j \leq n-1$ is sufficient, since $w_n = w_0$. Thanks to Euler's formula $\cos x = \frac{1}{2}(e^{\imath x} + e^{-\imath x})$ = and the fact that $z_\ell \in \mathbb{R}$, we can conclude

$$\widetilde{w}_j = \mathrm{Re}\left\{ \sum_{\ell=1}^{m} z_\ell e^{-\frac{2\pi\imath}{n}\ell j} \right\} \, , \quad j = 0, \ldots, n-1 \, . \tag{7.12.8}$$

Recall from [Lecture → Def. 4.2.1.18] that the DFT amounts to a matrix×vector multilication with the **Fourier matrix**

$$\mathbf{F}_n := \left[\exp(-\frac{2\pi\imath}{n}\ell j) \right]_{\ell,j=0}^{n-1} \in \mathbb{C}^{n,n} \, . \qquad \text{[Lecture → Eq. (4.2.1.13)]}$$

Thus (7.12.8) means that

$$\left[\widetilde{w}_j\right]_{j=0}^{n-1} = \mathrm{Re}\{\mathbf{F}_n \mathbf{z}\} = \mathrm{Re}\{\mathsf{DFT}_n(\mathbf{z})\} \quad \text{with} \quad \mathbf{z} := \begin{bmatrix} 0 \\ z_1 \\ \vdots \\ z_m \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^n \, . \tag{7.12.9}$$

From the definition of the Fourier matrix it is clear that

$$\mathbf{F}_n \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \, ,$$

which means that (C++ indexing!)

$$\left[w_j\right]_{j=0}^{n-1} = \left[\frac{a_j}{n}(1 - \widetilde{w}_j)\right]_{j=0}^{n-1} = \frac{1}{n} \, \mathrm{diag}\left(\left[a_j\right]_{j=0}^{n-1} \right) \left(\mathrm{Re}\{\mathsf{DFT}_n(\mathbf{z}')\} \right)_{0:n-1} \, ,$$

with $\mathbf{z}' = \begin{bmatrix} 1 & -z_1 & \cdots & -z_m & 0 & \cdots & 0 \end{bmatrix}^\top \in \mathbb{R}^m$. This formula is implemented in the following code.

C++ code 7.12.10: FFT-based implementation of constructor of CCFQuadRule_Fast

```
CCFQuadRule_Fast::CCFQuadRule_Fast(unsigned int n) : weights_(n + 1) {
  // TODO: (7-12.d) Provide an implementation with asymptotic complexity of O(n
  // log(n))
  // START
  assert(n > 0);
  // The nodes c_j = cos(jπ/n), j = 0,...,n
  nodes_ = (Eigen::ArrayXd::LinSpaced(n + 1, 0, n) * M_PI / n).cos().matrix();
  // Length of sum:  m = n/2 for even n, m = (n-1)/2 for odd
  // n
  const unsigned int m = n / 2;
  // Initialize vector z to be multiplied with Fourier matrix
  Eigen::VectorXcd z{Eigen::VectorXcd::Zero(n)};
  z[0] = 1.0;
  for (unsigned int l = 1; l <= m; ++l) z[l] = -2.0 / (4.0 * l * l - 1.0);
  if (n % 2 == 0) z[m] /= 2.0;
  // FFT-based multiplication with Fourier matrix
  Eigen::FFT<double> fft;
  // Eigen's built-in FFT crashes for vectors of length = 1
  // This is a work around
  if (n == 1)
    weights_[0] = 2.0;
  else
    weights_.head(n) = 2.0 * (fft.fwd(z)).real() / n;
  weights_[0] /= 2.0;
  weights_[n] = weights_[0];
  // END
}
```

Get it on GitLab (clenshawcurtisfejer.hpp).

HINT 1 for (7-13.a): The function $t \mapsto \arccos t$ is available in C++ as **std**`::acos()`, if the header `cmath` is included. ⌟

HINT 2 for (7-13.a): The integral cannot be evaluated in closed form. The idea is to use an approximate value provided by "overkill numerical quadrature" with many more quadrature nodes than used for the smallest error taken into account for gauging the convergence behavior.

Such a *reference value* is $I = 1.7576137811123187$. ⌟

SOLUTION of (7-13.a):

Types of convergence have been introduced in [Lecture → Def. 6.2.2.7] for approximation errors. They apply to quadrature errors as well.

Due to the singularity of $t \mapsto \arccos t$ at $t = \pm 1$, we cannot expect exponential convergence in the number of quadrature nodes. So, taking the cue from [Lecture → § 6.2.2.9], we first try to detect algebraic convergence:

$$\epsilon_i \approx C n_i^{-p} \quad \Leftrightarrow \quad \log \epsilon_i \approx \log C - p \log n_i \,, \tag{7.13.2}$$

where $(\epsilon_i)_{i=1}^L$ is a sequence of quadrature errors, $(n_i)_{i=1}^L \in \mathbb{N}$ is the corresponding sequence of numbers of quadrature nodes and $p > 0$ is the rate of algebraic convergence. It is convenient to opt for a geometric growth of n_i, for instance, $n_i = m_0 2^i$, $m_0 \in \mathbb{N}$, because in this case

$$\log \epsilon_i \approx \log C - p \log m_0 - pi \log 2 \,, \quad i = 1, \ldots, L \,.$$

Then an estimate for the rate p can be obtained as

$$p \approx \frac{\log \epsilon_{i-1} - \log \epsilon_i}{\log 2} \,. \tag{7.13.3}$$

This is another quantity, one should tabulate.

Alternatively, using $n_i = m_0 2^i$, $m_0 \in \mathbb{N}$, the error behavior like (7.13.2) also manifests itself through

$$\frac{\epsilon_i}{\epsilon_{i+1}} \approx \frac{C n_i^{-p}}{C n_{i+1}^{-p}} = 2^p \,.$$

Hence, one may increase the number of quadrature points by a constant factor in a sequence of tests and then watch the ratio of two successive errors. If that ratio attains about the same value throughout the sequence, we can also conclude algebraic convergence. Thus the values $\epsilon_{i-1} : \epsilon_i$ could also be included in the table.

C++ code 7.13.4: Convenience struct `cvgdata`

```
2  struct cvgdata {
3    long n_{0};          // number of quadrature points
4    double val_{0.0};    // approximate value
5    double err_{-1.0};   // quadrature error
6    double rate_{-1.0};  // estimate for rate of algebraic convergence
7  };
```

Get it on GitLab (arccosquad.hpp).

C++ code 7.13.5: Code for `testConvGaussQuad()`

```
2  void testConvGaussQuad() {
3    // Table header
4    std::cout << std::setw(5) << "n" << std::setw(20) << " I_n " << std::setw(30)
5              << " Error " << std::setw(10) << (" rate\n " + std::string(65, '-'))
6              << std::endl;
7
```

```
  // Value obtained by overkill quadrature as reference value
  constexpr double ref_val = 1.7576137811123187;

  // TODO: (7-13.a) Print a table that allows you to predict the asymptotic
  // behaviour of Gauss-Legendre numerical quadrature when approximating I(f).
  // START
  // The integrand as lambda funtion
  auto integrand = [](double t) {
    return std::acos(t) * 1.0 / (1.0 + std::exp(t));
  };

  // For recording errors etc.
  std::vector<cvgdata> data;
  // We suspect algebraic convergence.  Accordingly use doubling of number of
  // quadrature nodes
  constexpr unsigned int L = 8;  // Use at most 2^L quadrature nodes
  unsigned int n = 2;
  for (unsigned int l = 1; l <= L; ++l, n *= 2) {
    const QuadRule qr{gaussquad(n)};
    // No transformation required:  the integral is on [-1,1] anyway.
    double I = 0.0;  // Summation variable for quadrature formula
    for (unsigned int j = 0; j < n; ++j) {
      I += qr.weights_[j] * integrand(qr.nodes_[j]);
    }
    data.push_back({qr.weights_.size(), I, 0.0, 0.0});
  }
  for (unsigned int l = 0; l < data.size(); ++l) {
    data[l].err_ = std::abs(data[l].val_ - ref_val);
    if (l > 0) {
      // Estimate for rate of algebraic convergence
      data[l].rate_ =
          (std::log(data[l - 1].err_) - std::log(data[l].err_)) / std::log(2.0);
    }
  }

  // Output table
  for (unsigned int l = 0; l < data.size(); ++l) {
    const cvgdata& item{data[l]};
    std::cout << std::setw(5) << item.n_ << std::setw(20) << std::fixed
              << std::setprecision(16) << item.val_ << std::setw(30)
              << std::scientific << std::setprecision(16) << item.err_
              << std::setw(10) << std::fixed << std::setprecision(2)
              << item.rate_ << std::endl;
  }
  // END
}
```

Get it on GitLab (arccosquad.hpp).

The code produces the following output:

n	I_n	Error	rate
2	1.7436937792198774	1.3920001892441247e−02	0.00
4	1.7553382175123289	2.2755635999898161e−03	2.61
8	1.7572962271476178	3.1755396470090069e−04	2.84
16	1.7575711396897615	4.2641422557165853e−05	2.90
32	1.7576082279627416	5.5531495770644312e−06	2.94
64	1.7576130715900262	7.0952229247467358e−07	2.97
128	1.7576136914136691	8.9698649619052162e−08	2.98
256	1.7576137698376213	1.1274697397922750e−08	2.99
512	1.7576137797013049	1.4110137502854059e−09	3.00
1024	1.7576137809381109	1.7420775932919241e−10	3.02
2048	1.7576137810929557	1.9362955683277505e−11	3.17

The data provide clear evidence for algebraic convergence with rate ≈ 3 .

SOLUTION of (7-13.b):

We use the transformation

$$s = \arccos t \quad \Leftrightarrow \quad t = \cos s \, ,$$

for which

$$\mathrm{d}t = -\sin s \, \mathrm{d}s \, .$$

The new integration bounds will be $\pi = \arccos(-1)$ and $0 = \arccos(1)$. Thus, we arrive at

$$I(f) = -\int_{\pi}^{0} s \, f(\cos s) \sin s \, \mathrm{d}s \quad . \tag{7.13.6}$$

Since $s \mapsto \cos s$, $s \mapsto \sin s$, are entire functions, they do not affect the smoothness of the integrand:

$$f \in C^{\infty}([-1,1]) \quad \Rightarrow \quad s \mapsto s \, f(\cos s) \sin s \in C^{\infty}([0,\pi]) \, .$$

SOLUTION of (7-13.c):

Of course, we now apply Gauss-Legendre quadrature to the transformed integral (7.13.6). If f is analytic in a neigborhood of $[-1,1]$, then $s \mapsto sf(\cos s)\sin s$ will be analytic in a neighborhood of $[0,\pi]$, because $s \mapsto \cos s$, $s \mapsto \sin s$, are entire functions, that is, analytic on all of $\mathbb{C}$.

Note that the weights and nodes of the Gauss-Legendre quadrature rules have to be transformed to the interval $[0,\pi]$ as explained in [Lecture → Rem. 7.2.0.4].

C++ code 7.13.7: Code for `arccosWeightedQuad()`

```
2  template <typename FUNCTION>
3  double arccosWeightedQuad(FUNCTION&& f, unsigned int n) {
4    double I = 0.0;  // For accumulating quadrature result
5    // TODO: (7-13.c) Approximate I(f) with exponential convergence in n.
6    // START
7    // Obtain Gauss-Legendre quadrature rule
8    QuadRule qr{gaussquad(n)};
9    // The transformed integral covers the interval [0,π], which entails
10   // a transformation of the quadrature nodes and weights, see
11   // [Lecture → Rem. 7.2.0.4]
12   qr.weights_ *= 0.5 * M_PI;
13   qr.nodes_ = 0.5 * M_PI * (1.0 + qr.nodes_.array());
14   // Straightforward implementation of quadrature formula
15   for (unsigned int j = 0; j < n; ++j) {
16     const double c = qr.nodes_[j];
17     const double fval = f(std::cos(c));
18     I += qr.weights_[j] * fval * std::sin(c) * c;
19   }
20   // END
21   return I;
22 }
```

Get it on GitLab (arccosquad.hpp).

SOLUTION of (7-13.d):

Note that $t \mapsto \frac{1}{1+\exp(t)}$ is analytic in a neighborhood of $[-1,1]$, because its domain of analyticity is

$$\mathbb{C} \setminus (2\mathbb{Z}+1)\imath \,.$$

Thus, this time we expect exponential convergence, that is, using the notation from the solution of Subproblem (7-13.a) we expect

$$\epsilon_i \approx C\exp(-qn_i) \quad \text{for some} \quad q > 0 \,. \tag{7.13.8}$$

In this case it is advantageous to let n_i increase linearly, for instance, $n_i = i$, because this implies

$$\frac{\epsilon_{i+1}}{\epsilon_i} \approx \exp(-q) < 1 \,.$$

In the case of exponential convergence we should observe a decay of the approximation error by a constant factor when raising n by 1. Thus tabulating the ratio of consecutive errors is highly recommended.

C++ code 7.13.9: Code for `testConvTrfGaussQuad()`

```
void testConvTrfGaussQuad() {
  // Table header
  std::cout << std::setw(5) << "n" << std::setw(20) << " I_n " << std::setw(30)
            << " Error " << std::setw(10)
            << (" decay ratio\n " + std::string(65, '-')) << std::endl;

  // Value obtained by overkill quadrature as reference value
  constexpr double ref_val = 1.7576137811123187;

  // TODO: (7-13.d) Print a table that allows you to predict the asymptotic
  // behaviour of arccosWeightedQuad().
  // START
  // The integrand (/arccos(t)) as lambda funtion
  auto f = [](double t) { return 1.0 / (1.0 + std::exp(t)); };

  // For recording errors etc.
  std::vector<cvgdata> data;
  // We expect exponential convergence.  Increment number of quadrature nodes by
  // one in each step
  constexpr unsigned int L = 12;  // Use a most L quadrature nodes
  for (unsigned int n = 2; n <= L; ++n) {
    data.push_back({n, arccosWeightedQuad(f, n), 0.0, 0.0});
  }
  for (unsigned int l = 0; l < data.size(); ++l) {
    data[l].err_ = std::abs(data[l].val_ - ref_val);
    if (l > 0) {
      // Quotient of successive error values
      data[l].rate_ = data[l].err_ / data[l - 1].err_;
    }
  }

  // Output table
  for (unsigned int l = 0; l < data.size(); ++l) {
```

```
    const cvgdata& item{data[l]};
    std::cout << std::setw(5) << item.n_ << std::setw(20) << std::fixed
              << std::setprecision(16) << item.val_ << std::setw(30)
              << std::scientific << std::setprecision(16) << item.err_
              << std::setw(10) << std::fixed << std::setprecision(2)
              << item.rate_ << std::endl;
  }
  // END
}
```

Get it on GitLab (arccosquad.hpp).

The code produces the following output:

n	I_n	Error	decay ratio
2	1.8492385346839764	9.1624753571657669e−02	0.00
3	1.7329033484157947	2.4710432696523954e−02	0.27
4	1.7605214393475945	2.9076582352758340e−03	0.12
5	1.7572524700920542	3.6131102026448758e−04	0.12
6	1.7576617761797750	4.7995067456341189e−05	0.13
7	1.7576074242880906	6.3568242281153431e−06	0.13
8	1.7576146283917027	8.4727938398643232e−07	0.13
9	1.7576136678083938	1.1330392490904728e−07	0.13
10	1.7576137962618741	1.5149555387949931e−08	0.13
11	1.7576137790901427	2.0221759822192098e−09	0.13
12	1.7576137813858141	2.7349544851063001e−10	0.14

The ratio of successive approximation errors clearly stabilizes at a value $\approx 0.14 < 1$, which hints at exponential convergence :

$$|I(f) - \texttt{arccosWeightedQuad(f,n)}| = O(0.14^n) \quad \text{for} \quad n \to \infty .$$

SOLUTION of (7-14.a):

Definition *cf.* [Lecture → Def. 7.4.1.1]. Order of a quadrature rule

The **order** of a quadrature rule $Q_n : C^0([a,b]) \to \mathbb{R}$ is defined as

$$\text{order}(Q_n) := \max \left\{ m \in \mathbb{N}_0 : \quad Q_n(\ p\ ) = \int_a^b p(t)\ \mathrm{d}t \quad \forall p \in \mathcal{P}_{m-1} \right\} .$$

HINT 1 for (7-14.b):

- The C++ operator `%` gives the remainder of integer division: `p % q` $\hat{=} p \mod q$.
- **std**`::pow(x,n)` computes x^n for $n \in \mathbb{N}_0$, when x is a floating-point type, and n a cardinal number. By specification **std**`::pow(0.0,0)== 1.0`.

⌟

SOLUTION of (7-14.b):

Writing $Q(f)$ for the evaluation of the quadrature formula for a function f, we have to find the largest $d \in \mathbb{N}_0$ such that

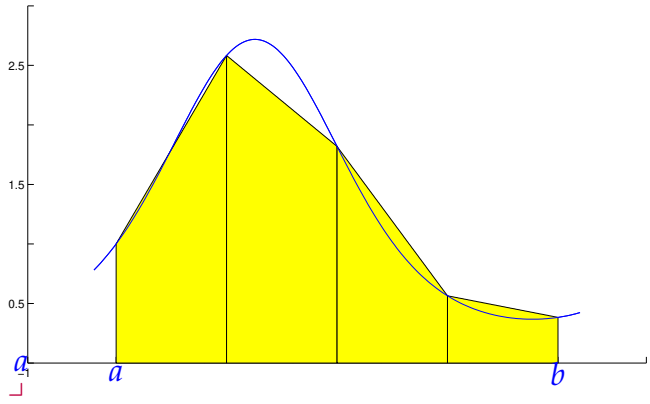
$$Q(\{t \mapsto t^d\}) = \int_{-1}^{1} t^d \mathrm{d}t = \begin{cases} 2/d+1 & \text{for even } d\,, \\ 0 & \text{for odd } d\,. \end{cases}$$

C++ code 7.14.1: Implementation of function `checkQuadOrder()`

```
unsigned int checkQuadOrder(const Eigen::VectorXd &c, const Eigen::VectorXd &w,
                            const double tol = 1.0E-10) {
  const unsigned long N = c.size();  // Number of quadrature points
  assert(N == w.size());             // Numbers of nodes and weights must agree
  // Check for exactness for polynomials of degree d
  // Note that the highest possible order is 2N!
  for (unsigned int d = 0; d < 2 * N; ++d) {
    // j-loop performs the evaluation of the quadrature formula
    // for the monomial t ↦ t^d.
    double s = 0.0;
    for (int j = 0; j < N; ++j) {
      s += w[j] * std::pow(c[j], d);
    }
    // Exact value:  2/(d+1) for even degree, zero else.
    double val = (d % 2 == 0) ? 2.0 / (d + 1) : 0.0;
    // Safe test for equality of computed floating point numbers
    if (std::abs(s - val) > tol) {
      return d;
    }
  }
  // The quadrature rule has the highest possible order
  return 2 * N;
}
```

Get it on GitLab (checkquadorder.cpp).

HINT 1 for (7-14.c):



Recall that the trapezoidal rule approximates the integral $\int_a^b f(t)\,\mathrm{d}t$ by the area under an interpolating polygonal line.

SOLUTION of (7-14.c):

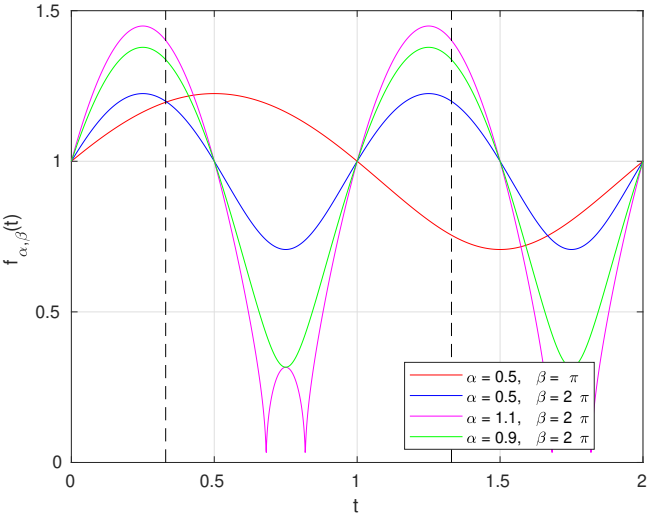
C++ code 7.14.2: Implementation of function `trapezoidalRule()`

```
2  template <typename FUNCTOR>
3  double trapezoidalRule(FUNCTOR &&f, double a, double b, unsigned int N) {
4    assert(a < b);
5    assert(N >= 2);
6    // Spacing of the N equidistant quadrature nodes
7    const double h = (b - a) / (N - 1);
8    // Variable containing location of current quadrature node
9    double t = a + h;
10   // Left endpoint has weight 0.5*h
11   double s = 0.5 * f(a);
12   // Summing contributions of N-2 interior quadrature nodes with weight
       h
13   for (int j = 1; j < N - 1; ++j) {
14     s += f(t);
15     t += h;
16   }
17   // Right endpoint has weight 0.5*h
18   s += 0.5 * f(t);
19   // Scaling with h is performed in the end.  Can also be done when
       accumulating
20   // value of quadrature formula.
21   return h * s;
22 }
```

Get it on GitLab (checkquadorder.cpp).

SOLUTION of (7-14.d):

This sub-problem is closely related to [Lecture → Exp. 7.5.0.16] and [Lecture → § 7.5.0.18]. See there for in-depth explanations.



◁ The four functions from the $f_{\alpha,\beta}$ family, for whom plots of the quadrature errors have been provided.

The colors used correspond to the colors of the associated error graphs.

In the sequel, n stands for the number of quadrature nodes.

- (1) $\alpha = 0.9$, $\beta = 2\pi$: the function is 1-periodic and analytic in a strip along the real axis. We expect exponential asymptotic convergence for $n \to \infty$, reflected by aligned error points in a linear-logarithmic plot Fig. 83. This expectation is matched by graphs B or D. Since B is already claimed, just D remains.
- (2) $\alpha = 0.5$, $\beta = \pi$: the function is smooth, but **not** 1-periodic. Thus we can just expect regular $O(n^{-2})$ asymptotic convergence of the trapezoidal rule for $n \to \infty$. By exclusion we conclude that this must correspond to graph A .
- (3) $\alpha = 1.1$, $\beta = 2\pi$: The function has root-type singularities where $1 + 1.1\sin(2\pi t) = 0$. Hence for this non-smooth integrand we cannot expect full $O(n^{-2})$ algebraic convergence, let alone exponential convergence. This function will result in the slowest observed convergence of the quadrature error, hence graph C .
- (4) $\alpha = 0.5$, $\beta = 2\pi$: Same as for (1) but with a wider strip of analyticity along the real axis, hence faster exponential convergence than (1), graph B .

The best way to approach this sub-problem is to rank the different functions according to the expected speed of convergence of the quadrature quadrature error. Appealing to the properties of the functions discussed above we obtain:

$$(3) \quad < \quad (2) \quad < \quad (1) \quad < (4)\,,$$

where "$<$" means "expected to converge to zero more slowly than". A corresponding ranking of the error graphs is evident:

$$(C) \quad < \quad (A) \quad < \quad (D) \quad < \quad (B)\,.$$

This yields the right association of functions and graphs.

SOLUTION of (7-15.a):

Obviously ψ_j is a 1-periodic function. To emphasize that we now consider ψ_j as a function of a complex argument we write

$$\psi_j(w) := e^{-2\pi\imath jw} F\left(\frac{1 - re^{2\pi\imath w}}{\tau}\right), \quad w \in \mathbb{C}\,.$$

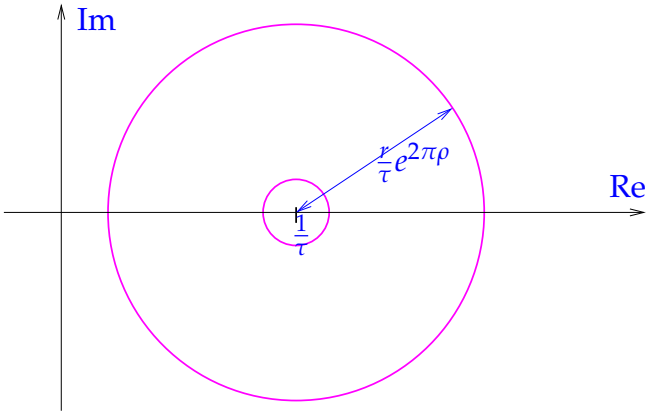
For every $j \in \mathbb{Z}$ the function $w \mapsto e^{2\pi\imath jw}$ is an entire function (analytic in all of $\mathbb{C}$), so that it does not affect the domain of analyticity of ψ_j. We can safely ignore this forefactor. In particular, the domain of analyticity of ψ_j does not depend on j.

Owing to Ass. 7.15.3, ψ_j is analytic in the pre-image of $\mathbb{C}^+$ under the map $w \mapsto \frac{1-e^{2\pi\imath w}}{\tau}$. To begin with, recall that

$$w \mapsto e^{2\pi\imath w} = e^{-2\pi \operatorname{Im} w}(\cos(2\pi \operatorname{Re} w) + \imath \sin(2\pi \operatorname{Re} w))$$

maps the strip S_ρ to an annulus:

$$\{w \mapsto e^{2\pi\imath w}\}(S_\rho) = T_\rho := \{z \in \mathbb{C} : e^{-2\pi\rho} < |z| < e^{2\pi\rho}\}\,. \tag{7.15.6}$$



Hence, $w \mapsto \frac{1-e^{2\pi\imath w}}{\tau}$ maps S_ρ to an annulus shifted by $\frac{1}{\tau}$ and scaled by $\frac{r}{\tau}$.

◁ Domain $\dfrac{1 - rT_\rho}{\tau}$: image of strip.

This means that

$$\tfrac{1}{\tau} - \tfrac{r}{\tau} T_\rho \subset \mathbb{C}^+ \quad \Rightarrow \quad \psi_j \text{ is analytic in } S_\rho\,.$$

Geometric consideration reveal, see Fig. 86,

$$\tfrac{1}{\tau} - \tfrac{r}{\tau} T_\rho \subset \mathbb{C}^+ \quad \Leftrightarrow \quad 1 - re^{2\pi\rho} > 0 \quad \Leftrightarrow \quad \rho < -\frac{1}{2\pi}\log r\,.$$

The largest possible width of the strip is $\frac{1}{2\pi}\log 1/r$.

SOLUTION of (7-15.b):

Starting from

$$w_j^\tau = r^{-j} \int_0^1 \psi_j(\varphi)\,\mathrm{d}\varphi\,, \quad \psi_j(\varphi) := e^{-2\pi\imath j\varphi} F\left(\frac{1 - re^{2\pi\imath\varphi}}{\tau}\right).$$

and the definition of the equidistant trapezoidal quadrature rule

$$\int_a^b f(t)\,\mathrm{d}t \approx T_m(f) := h\left(\tfrac{1}{2}f(a) + \sum_{k=1}^{m-1} f(kh) + \tfrac{1}{2}f(b)\right), \quad h := \frac{b-a}{m}, \quad \text{[Lecture} \rightarrow \text{Eq. (7.5.0.17)]}$$

applied on $[0,1]$ with $m = N+1$ and taking into account the 1-periodicity of the integrand we arrive at

$$w_j^\tau \approx w_j^{\tau,N} = \frac{1}{(N+1)r^j} \sum_{k=0}^{N} \psi_j\left(\frac{k}{N+1}\right) = \frac{1}{(N+1)r^j} \sum_{k=0}^{N} e^{-2\pi\imath j\frac{k}{N+1}} F\left(\frac{1 - re^{2\pi\imath\frac{k}{N+1}}}{\tau}\right).$$

SOLUTION of (7-15.c):

In Sub-problem (7-15.b) we derived the formula

$$w_j^{\tau,N} = \frac{1}{(N+1)r^j} \sum_{k=0}^{N} e^{-2\pi\imath j \frac{k}{N+1}} F\left(\frac{1 - re^{2\pi\imath \frac{k}{N+1}}}{\tau}\right) .$$

Let us introduce the vector

$$\mathbf{v} = \left[F\left(\frac{1 - re^{2\pi\imath \frac{k}{N+1}}}{\tau}\right) \right]_{k=0}^{N} . \tag{7.15.7}$$

Its components can be obtained by sampling F on a circle in $\mathbb{C}^+$ ($r < 1$). Next recall the discrete Fourier transform from [Lecture → Def. 4.2.1.18]:

Definition [Lecture → Def. 4.2.1.18]. Discrete Fourier transform

The linear map $\mathrm{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathrm{DFT}_n(\mathbf{y}) := \mathbf{F}_n\mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called **discrete Fourier transform** (DFT), i.e. for $[c_0, \ldots, c_{n-1}] := \mathrm{DFT}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} = \sum_{j=0}^{n-1} y_j \exp\left(-2\pi\imath \frac{kj}{n}\right) \quad , \quad k = 0, \ldots, n-1 . \qquad \text{[Lecture → Eq. (4.2.1.19)]}$$

Matching (7.15.7) and [Lecture → Eq. (4.2.1.19)] we find

$$\left[w_j^{\tau,N}\right]_{j=0}^{N} = \mathbf{D} \cdot \mathrm{DFT}_{N+1}(\mathbf{v}) , \quad \mathbf{D} = \frac{1}{N+1} \operatorname{diag}(1, r^{-1}, r^{-2}, \ldots, r^{-N}) . \tag{7.15.8}$$

For the implementation one has to take into account how DFT is done by means of EIGEN's built-in helper class **Eigen::FFT**, see [Lecture → Code 4.2.1.22].

C++ code 7.15.9: Computation of convolution quadrature weights

```
2  template <typename FFUNCTOR>
3  Eigen::VectorXcd compute_cq_weights(FFUNCTOR&& F, unsigned int N, double tau,
4                                      double r = 0.0) {
5    using Comp = std::complex<double>;
6    assert(N >= 1 && "At least two quadrature points have to be used!");
7    // Default choice for circle radius for integration contour
8    if (r == 0.0) {
9      r = std::pow(std::numeric_limits<double>::epsilon(), 0.5 / (N + 1));
10   }
11   Eigen::VectorXcd w(N + 1);
12   const Comp iu = 2.0 * Comp{0.0, 1.0} * M_PI;  // 2*pi*i
13   // TODO: (7-15.c) Compute the approximate convolution quadrature
         weights by
14   // means of the equidistant trapezoidal rule.
15   // START
16   Eigen::VectorXcd v(N + 1);
17   // Sampling of F on a circle in the complex plane, see (7.15.7)
18   const Comp expfac = iu / (static_cast<double>(N + 1));
19   for (unsigned int k = 0; k <= N; ++k) {
```

```
    const Comp z =
        (Comp(1.0, 0.0) - r * std::exp(expfac * static_cast<double>(k))) / tau;
    v[k] = F(z);
  }
  // Compute DFT of the vector v
  Eigen::FFT<double> fft;
  w = fft.fwd(v);  // DFT in EIGEN
  // Finally perform the scaling of the components
  double fac = N + 1;
  for (unsigned int k = 0; k <= N; ++k) {
    w[k] /= fac;
    fac *= r;
  }
  // END
  return w;
}
```

Get it on GitLab (convolutionquadrature.hpp).

SOLUTION of (7-15.d):

In Sub-problem (7-15.a) we have seen that the integrand in

$$w_j^\tau = r^{-j} \int_0^1 \psi_j(\varphi)\,\mathrm{d}\varphi$$

is 1-periodic and possesses an analytic extension to a strip in the complex plane parallel to the real axis. Therefore the quadrature error will *converge exponentially* to zero as $N \to \infty$:

$$\exists q \in [0,1[: \quad |w_j^\tau - w_j^{\tau,N}| = O(q^N) \quad \text{for} \quad N \to \infty\,. \tag{7.15.10}$$

The reason for this behavior is that the $(N+1)$-point equidistant trapezoidal rule integrates trigonometric polynomials of degree $2N$ exactly, see [Lecture → Lemma 7.5.0.22]. Then the approximation results of [Lecture → Section 6.5.3] can be applied.

Remark. The following lemma from the course

Lemma [Lecture → Lemma 7.5.0.22]. Exact quadrature by equidistant trapezoidal rule

The n-point equidistant trapezoidal quadrature rule

$$\int_a^b f(t)\,\mathrm{d}t \approx T_n(f) := h\left(\tfrac{1}{2}f(a) + \sum_{k=1}^{n-1} f(a+kh) + \tfrac{1}{2}f(b)\right)\,, \quad h := \frac{b-a}{n}\,.$$

[Lecture → Eq. (7.5.0.17)]

is exact for trigonometric polynomials (→ [Lecture → Section 6.5.1]) of degree $\leq 2n-2$.

together with

Theorem [Lecture → Thm. 6.5.3.14]. Exponential convergence of trigonometric interpolation for analytic interpolands

If $f : \mathbb{R} \to \mathbb{C}$ is 1-periodic and possesses an analytic extension to the strip

$$\bar{S} := \{z \in \mathbb{C}:\ -\eta \leq \operatorname{Im} z \leq \eta\}\,, \quad \textit{for some } \eta > 0\,,$$

then there is $C_\eta > 0$ depending only on η such that

$$\|f - \mathsf{T}_n f\|_* \leq C_\eta e^{-\pi\eta n}\|f\|_{L^\infty(\bar{S})}\,, \quad n \in \mathbb{N}\,, \quad (\,* = L^2(]0,1[), L^\infty(]0,1[)\,)\,.$$

[Lecture → Eq. (6.5.3.15)]

tells us how the width ρ of the strip of analyticity determines the number q in (7.15.10):

$$q = \exp(-\pi\rho)\,.$$

SOLUTION of (7-16.a):

C++ code 7.16.2: Constructor of QuadFormula Get it on GitLab.

```
2  QuadFormula::QuadFormula(const QuadFormulaRef& qf, double a, double b)
3      : a_(a), b_(b), nodes_(qf.nodes()), weights_(qf.weights()) {
4    const double len = b - a;
5    const unsigned int n_pts = nodes_.size();
6    assert((weights_.size() == n_pts) && "Size mismatch");
7    for (unsigned int j = 0; j < n_pts; ++j) {
8      // Affine transformation of quadrature nodes, see [Lecture →
         Rem. 7.2.0.4].
9      nodes_[j] = a + (nodes_[j] + 1.0) * len / 2;
10     // Rescaling of weights
11     weights_[j] *= len / 2;
12   }
13 }
```

HINT 1 for (7-16.b): The order of a p-point quadrature formula on $[-1,1]$ with nodes $c_j \in [-1,1]$ and weights $w_j \in \mathbb{R}$, $j \in \{1,\ldots,p\}$ is *at least* $q \in \mathbb{N}$, if

$$\sum_{\ell=1}^{p} w_\ell c_\ell^k = \int_{-1}^{1} \xi^k \,\mathrm{d}\xi = \begin{cases} \frac{2}{k+1} & \text{, if } \quad k \quad \text{is even,} \\ 0 & \text{, if } \quad k \quad \text{is odd,} \end{cases} \quad \forall k \in \{0,\ldots,q-1\}\,.$$

⌟

SOLUTION of (7-16.b):

Note that a distinction between relative and absolute tolerance does not make sense, because we are comparing an error to a fixed value.

C++ code 7.16.4: Complete implementation of `checkQuadOrder()` Get it on GitLab.

```
unsigned int checkQuadOrder(const QuadFormulaRef& qf,
                            const double tol = 1.0E-10) {
  const std::vector<double>& c = qf.nodes();
  const std::vector<double>& w = qf.weights();
  const unsigned int n_pts = c.size();
  assert((w.size() == n_pts) && "Size mismatch");

  // Values of monomials at quadrature points
  std::vector<double> monom_vals(n_pts, 1.0);
  // Check for exactness for polynomials of degree d
  // Note that the highest possible order is 2N!
  for (unsigned int d = 0; d < n_pts; ++d) {
    // j-loop performs the evaluation of the quadrature formula
    // for the monomial t ↦ t^{2d} (even polynomial)
    double s = 0.0;
    for (unsigned int j = 0; j < n_pts; ++j) {
      s += w[j] * monom_vals[j];
    }
    // Exact value: 2/(d+1) for even degree, zero else.
    double val = 2.0 / (2 * d + 1);
    // Safe test for equality of computed floating point numbers
    if (std::abs(s - val) > tol) {
      return 2 * d;
    }
    // Now test for odd-degree monomials, for which the integral has to
    // evaluate to zero
    for (unsigned int j = 0; j < n_pts; ++j) {
      monom_vals[j] *= c[j];
    }
    s = 0.0;
    for (unsigned int j = 0; j < n_pts; ++j) {
      s += w[j] * monom_vals[j];
    }
    // Safe test for equality with zero
    if (std::abs(s) > tol) {
      return 2 * d + 1;
    }
    for (unsigned int j = 0; j < n_pts; ++j) {
      monom_vals[j] *= c[j];
    }
  }
  // The quadrature rule has the highest possible order
  return 2 * n_pts;
}
```

SOLUTION of (7-2.a):

The quadrature is exact for every polynomial $p(x) \in \mathcal{P}^n$ if and only if it is exact for $1, x, x^2, \ldots, x^n$. 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.

HINT 1 for (7-2.b): Use the fundamental theorem of calculus and your result from the previous exercise.

SOLUTION of (7-2.b):

By the fundamental theorem of calculus we know that

$$z(2) - z(1) = \int_1^2 z'(x)dx, \tag{7.2.3}$$

hence, applying the quadrature rule and the transformation $x \mapsto x + 1$, we obtain:

$$z(2) = \int_0^1 z'(x+1)dx + z(1) \approx \frac{11}{27} \cdot z'(1) + \frac{1}{18} \cdot z''(1) + \frac{16}{27} \cdot z'\left(\frac{7}{4}\right) + z(1). \tag{7.2.4}$$

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':

$$z(2) = \int_1^2 z'(x)dx + z(1) = \frac{1}{2}\log(x^2+1)|_1^2 + 1 = 1.45815\ldots$$

HINT 1 for (7-3.a): You will probably remember that $\arctan' = g$, where the prime tags a derivative. ⌟

SOLUTION of (7-3.a):

An antiderivative of g is atan. Since the function g is even, we have

$$E_T = 2\int_b^\infty g(t)dt = \lim_{x\to\infty} 2\operatorname{atan}(x) - 2\operatorname{atan}(b) = \pi - 2\operatorname{atan}(b) \overset{!}{<} 10^{-6}\,, \tag{7.3.2}$$

i.e. $b > \tan\big((\pi - 10^{-6})/2\big) = \cot(10^{-6}/2) \approx 2\cdot 10^6$.

SOLUTION of (7-3.b):

A good choice of b requires a detailed knowledge about the decay of f, which may not be available for f defined implicitly.

SOLUTION of (7-3.c):

We have:

$$\frac{dt}{ds} = -\left(1+\cot^2(s)\right) = -\left(1+t^2\right)$$

$$\int_{-\infty}^{\infty} f(t)\,dt = -\int_{\pi}^{0} f(\cot(s))\left(1+\cot^2(s)\right)ds = \int_{0}^{\pi} \frac{f(\cot(s))}{\sin^2(s)}\,ds\ , \tag{7.3.3}$$

because $\sin^2(\theta) = \frac{1}{1+\cot^2(\theta)}$.

HINT 1 for (7-3.d): Use identities involving squares of trigonometric functions, see, for instance the Pythagorean trigonometric identities online. ⌟

SOLUTION of (7-3.d):

Replacing the function $g(t)$, we get:

$$\int_0^{\pi} \frac{1}{1+\cot^2(s)} \frac{1}{\sin^2(s)} ds = \int_0^{\pi} \frac{1}{\sin^2(s)+\cos^2(s)} ds = \int_0^{\pi} ds = \pi$$

HINT 1 for (7-3.e): A lambda function with signature (**double**) -> **double** automatically satisfies the requirement for f. ⌟

SOLUTION of (7-3.e):

We give an implementation based on an auxiliary function `quad()` that applies a quadrature rule on $[-1,1]$ to approximate an integral over $[a,b]$. Of course you need not employ such auxiliary functions.

C++-code 7.3.4: Implementation of transformed quadrature rule

```
2  template <class Function>
3  double quad(Function &&f, const Eigen::VectorXd &w, const Eigen::VectorXd &x,
4              double a, double b) {
5    double I = 0.;
6    // TODO: (7-3.e) implement transformation of quadrature rule
7    // START
8    for (unsigned int i = 0; i < w.size(); ++i) {
9      I += f((x(i) + 1) * (b - a) / 2 + a) * w(i);
10   }
11   I *= (b - a) / 2.;
12   // END
13   return I;
14 }
```

Get it on GitLab (quadinf.hpp).

C++-code 7.3.5: Implementation of `quadinf()`

```
2  template <class Function>
3  double quadinf(const int n, Function &&f) {
4    double I = 0.;
5    // TODO: (7-3.e) define the function quadinf that integrates the input
6    // lambda f over the real axis.  First, transform the integrand with a
       change
7    // of variables and then use an n point Gauss quadrature.
8    // Hint 1:  lambda functions can take parameters inside the []
       brackets
9    // Hint 2:  you may write an auxiliary function to compute the
       quadrature over
10   // a bounded interval, see function quad above.
11
12   // START
13   Eigen::VectorXd w, x;
14
15   // Compute nodes and weights of Gauss quadrature rule
16   // using Golub-Welsh algorithm
17   golubwelsh(n, w, x);
18
19   //!  NOTE: no function cot available in c++, need to resort to
       trigonometric
20   //!  identities.  Both lines below are valid, the first computes three
21   //!  trigonometric functions, but the second is prone to cancellation
22   auto ftilde = [&f](double x) {
23     return f(std::cos(x) / std::sin(x)) / pow(std::sin(x), 2);
24   };
25   /* auto ftilde = [&f] (double x) { double cot = std::tan(PI_HALF - x); return
26    * f(cot) * (1. + pow(cot,2)); }; */
27
28   // integrate ftilde from 0 to PI using Gauss quadrature rules
29   I = quad(ftilde, w, x, 0, PI);
```

```
30    // END
31    return I;
32  }
```

Get it on GitLab (quadinf.hpp).

SOLUTION of (7-3.f):

C++-code 7.3.6: Implementation of `cvgQuadInf()`

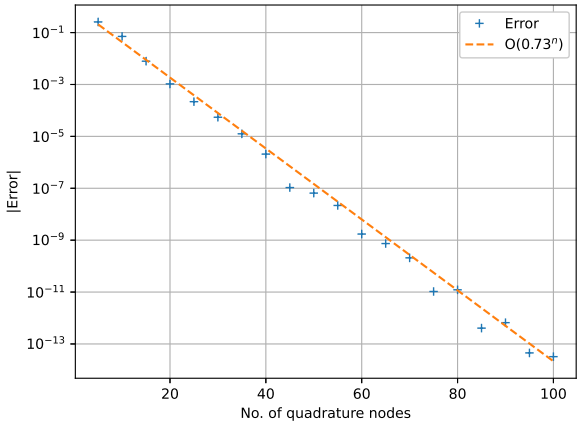
```
void cvgQuadInf(void) {
  // Number of max Gauss pts.
  constexpr int N = 100;
  plt::figure();
  // TODO: (7-3.f) plot convergence errors against number of integration nodes
  // START
  // variables for the plots
  std::vector<int> num_nodes;
  std::vector<double> err;
  std::vector<double> cvg_rate;

  // Integrand and exact integral
  auto f = [](double t) { return std::exp(-std::pow((t - 1), 2)); };
  const double I_ex = std::sqrt(PI);

  for (unsigned int n = 5; n <= N; n += 5) {
    // Approximated value of integral
    const double QS = quadinf(n, f);
    // plot data
    num_nodes.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_nodes, err, "+", {{"label", "Error"}});
  plt::semilogy(num_nodes, cvg_rate, "--", {{"label", "O(0.73^n)"}});
  plt::legend("best");
  plt::xlabel("No. of quadrature nodes");
  plt::ylabel("|Error|");

  plt::grid("True");
  // END
  plt::savefig("./cx_out/convergence.png");
}
```

Get it on GitLab (quadinf.hpp).



According to [Lecture → § 6.2.2.9] 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.

Appealing to [Lecture → Def. 6.2.2.7] and recalling [Lecture → § 6.2.2.9], the plot provides empiric evidence of exponential convergence.

SOLUTION of (7-4.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.

SOLUTION of (7-4.b):

We transform the quadrature rule according to the formulas derived in [Lecture → Rem. 7.2.0.4].

C++-code 7.4.2: Implementation of `evalquad()`.

```
2  template <class Function>
3  double evalquad(const double a, const double b, Function &&f,
4                  const QuadRule &Q) {
5    double I = 0.;
6    // TODO: (7-4.b) Use Q to approximate the integral of f over [a,b]
7    // START
8    // Loop over all nodes/weights pairs
9    for (unsigned int i = 0; i < Q.weights_.size(); ++i) {
10     I += f((Q.nodes_(i) + 1.) * (b - a) / 2. + a) * Q.weights_(i);
11   }
12   I = I * (b - a) / 2.;
13   // END
14   return I;
15 }
```

Get it on GitLab (laserquad.hpp).

HINT 1 for (7-4.c): Write (7.4.3) explicitly as a double integral as you were asked to do in Subproblem (7-4.a). Take special care to correctly find the intervals of integration. ⌟

HINT 2 for (7-4.c): C++ lambda functions are very useful for this kind of implementation, see [Lecture → Section 0.3.3]. ⌟

SOLUTION of (7-4.c):

As we have seen in Sub-problem (7-4.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:

$$g(y) := \int_0^{1-y} f_y(x)dx \approx \frac{1-y}{2} \sum_{i=0}^{n} w_i f_y\left(\frac{1-y}{2}(x_i+1)\right) =: \mathcal{I}(y), \tag{7.4.4}$$

the integral of which can be approximated, using a nested Gauss quadrature:

$$\int_\Delta f(x,y)\,dx\,dy = \int_0^1 \int_0^{1-y} f_y(x)\,dx\,dy = \int_0^1 g(y)dy \approx \frac{1}{2} \sum_{j=1}^{n} w_j \mathcal{I}\left(\frac{y_j+1}{2}\right). \tag{7.4.5}$$

C++-code 7.4.6: Implementation of `gaussquadtriangle()`.

```
2  template <class Function>
3  double gaussquadtriangle(const Function &f, const unsigned N) {
4    double I = 0.;
5    // TODO: (7-4.c) Use N-node gaussquad() to integrate f over
6    // the triangle 0<=x,y, x+y<=1.
7    // START
8    // Get nodes/weights for integral over dx and dy
9    QuadRule Q = gaussquad(N);
10
11   // We define the function g as the function of y
12   // g(y) := ∫_0^{1-y} f(x,y)dx.
13   auto g = [&f, &Q](double y) {
14     return evalquad(
15         0, 1 - y, [&f, &y](double x) { return f(x, y); }, Q);
16   };
17   // We integrate the function g over y from 0 to 1 to get
18   // ∫_0^1 g(y)dy := ∫_0^1 ∫_0^{1-y} f(x,y)dxdy.
19   I = evalquad(0, 1, g, Q);
20   // END
21   return I;
22 }
```

Get it on GitLab (laserquad.hpp).

This problem can also be solved using loops (so no nested lambda function). This more straightforward approach is listed next.

C++-code 7.4.7: Loop-based implementation of `gaussquadtriangle_loop()`.

```cpp
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);
  // Integration over y from 0 to 1 of g(y) := ∫_0^{1-y} I(x,y)dx
  double I = 0;
  constexpr double a = 0., b = 1.;
  for (unsigned int i = 0; i < Q.weights_.size(); ++i) {
    // Find out the y at which we are
    const 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.;
}
```

Get it on GitLab (laserquad.hpp).

If you choose that implementation, also have a look at the nested-lambda version listed in Code 7.4.6 and try to understand how it works.

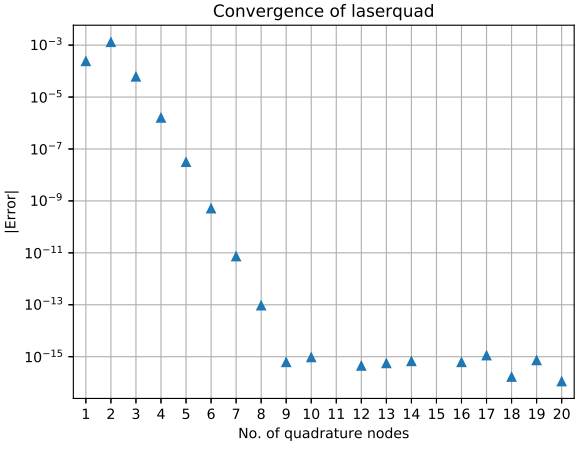
HINT 1 for (7-4.d): [Lecture → § 6.2.2.9] explains how to distinguish different kinds of convergence from error plots. ⌟

SOLUTION of (7-4.d):

N	I_approx	error
1	0.365808	0.000238736
2	0.367346	0.00129915
3	0.365986	6.05452e-05
4	0.366048	1.5884e-06
5	0.366047	3.11447e-08
6	0.366047	5.07046e-10
7	0.366047	7.22744e-12
8	0.366047	9.32032e-14
9	0.366047	6.10623e-16
10	0.366047	9.4369e-16
11	0.366047	0
12	0.366047	4.44089e-16
13	0.366047	5.55112e-16
14	0.366047	6.66134e-16
15	0.366047	0
16	0.366047	6.10623e-16
17	0.366047	1.11022e-15
18	0.366047	1.66533e-16
19	0.366047	7.21645e-16
20	0.366047	1.11022e-16

The table shows the corresponding approximated value of I and the error w.r.t. number of quadrature nodes N.

Why does the decrease of the quadrature error stall at $N = 9$? Well, by $N = 9$ we have reached an error of about the same size as the machine precision $\texttt{EPS} \approx 10^{-16}$, see [Lecture → Ex. 1.5.3.12]. Then roundoff error interfere with any further improvement of accuracy. Thus, what we see as errors for $N > 9$ is essentially "random" roundoff errors.



We observe that for $N < 9$ 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. 6.2.2.7] and [Lecture → § 6.2.2.9].
We expect from theoretical considerations that the convergence is exponential, because we deal with an perfectly smooth (analytic) integrand, see [Lecture → § 7.4.3.2] and [Lecture → Ex. 7.4.3.9].

C++-code 7.4.8: Implementation of `convtest2DQuad()`.

```
2  void convtest2DQuad(unsigned int nmax = 20) {
3    // "Exact" integral
4    constexpr double I_ex = 0.366046550000405;
5    plt::figure();
6    // TODO: (7-4.d) Tabulate the error of gaussquadtriangle() for
7    // a laser beam intensity, using n=1,2,3,...,nmax nodes.
8    // START
9    // Parameters
10   constexpr double alpha = 1., p = 0., q = 0.;
11   // Laser beam intensity
```

```
  auto I = [](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 int n = 1; n <= nmax; n++) {
    const 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::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");
  // END
  plt::savefig("./cx_out/convergence.png");
}
```

Get it on GitLab (laserquad.hpp).

HINT 1 for (7-5.a): Notice the different axis scales in the plots. ⌟

SOLUTION of (7-5.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.

SOLUTION of (7-5.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.

SOLUTION of (7-6.j):

The full code can be found in `quadU.cpp`.

C++ code 7.6.5: Computation of convergence of `quadU`.

```
void testQuadU(unsigned int nmax = 20) {
  // "Exact" value of integral
  constexpr double exact = 0.483296828976607;
  // Integrand
  auto f = [](double t) { return 1. / (2. + std::exp(3. * t)); };

  plt::figure();

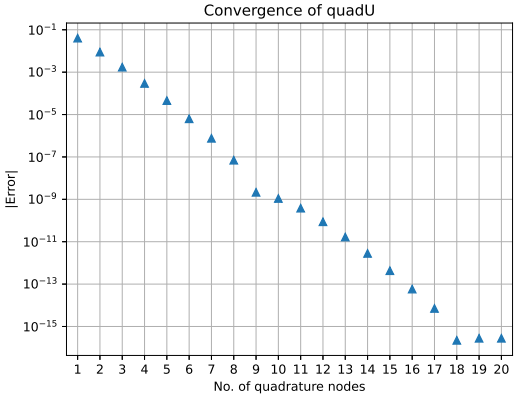
  // TODO: (7-6.j) Tabulate and plot the quadrature error.
  // START
  // memory allocation for plot
  std::vector<double> err(nmax);
  std::vector<int> num_pts(nmax);

  std::cout << std::setw(5) << "n" << std::setw(20) << "Error" << std::setw(20)
            << "q" << std::endl;

  for (unsigned int n = 1; n <= nmax; ++n) {
    // Compute error
    err[n - 1] = std::abs(quadU(f, n) - exact);
    num_pts[n - 1] = n;

    if (n > 1) {
      // Print table
      std::cout << std::setw(5) << n << std::setw(20) << err[n - 1]
                << std::setw(20) << err[n - 1] / err[n - 2] << std::endl;
    }
  }
  // Error plot rendered by matplotlibcpp
  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 quadU");
  plt::grid("True");
  // END
  plt::savefig("./cx_out/convergence.png");
}
```

Get it on GitLab (quadU.hpp).



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

SOLUTION of (7-6.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.

SOLUTION of (7-6.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 $k = 0, \ldots, n$ 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. $\square$

SOLUTION of (7-6.d):

With the substitution $t = \cos s$ we obtain

$$\begin{aligned}\int_{-1}^{1} \sqrt{1-t^2}U_m(t)U_n(t)\,dt &= \int_{-1}^{1} \sqrt{1-t^2}\frac{\sin((m+1)\arccos t)\sin((n+1)\arccos t)}{\sin^2(\arccos t)}\,dt \\ &= \int_0^{\pi} \sin s \frac{\sin((m+1)s)\sin((n+1)s)}{\sin^2 s}\sin s\,ds \\ &= \int_0^{\pi} \sin((m+1)s)\sin((n+1)s)\,ds \\ &= \frac{1}{2}\int_0^{\pi} \cos((m-n)s) - \cos((m+n+2)s)\,ds.\end{aligned}$$

SOLUTION of (7-6.e):

From the definition of U_n we immediately find that the zeros are given by

$$\xi_j^n = \cos\left(\frac{j}{n+1}\pi\right), \qquad j = 1, \ldots, n. \tag{7.6.1}$$

SOLUTION of (7-6.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 (7.6.1) 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 $\operatorname{Re}(1/(a+ib)) = a/(a^2+b^2)$. Thus

$$\operatorname{Re}\left(\frac{1}{1 - e^{ikx}}\right) = \operatorname{Re}\left(\frac{1}{1 - \cos(kx) - i\sin(kx)}\right) = \frac{1 - \cos(kx)}{(1 - \cos(kx))^2 + \sin(kx)^2} = \frac{1}{2}$$

Arguing in a similar way we have $\operatorname{Re}\,(1 - e^{i(k+2)x})^{-1} = 1/2$. 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 (7-6.d), since $U_0(t) = 1$.

HINT 1 for (7-6.g): Refer to [Lecture → Thm. 7.4.2.11].

SOLUTION of (7-6.g):

The conclusion follows by applying the same argument given in [Lecture $\rightarrow$ Thm. 7.4.2.11] with the weighted L^2 scalar product with weight w defined above.

HINT 1 for (7-6.h): See [Lecture → § 7.4.3.2].

SOLUTION of (7-6.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 $\rightarrow$ § 7.4.3.2], we obtain the exponential decay, as desired.

SOLUTION of (7-6.i):

The full code can be found in `quadU.cpp`.

C++ code 7.6.4: Solution for `quadU`.

```
2  template <typename Function>
3  double quadU(const Function& f, const unsigned int n) {
4    // Value of integral
5    double I = 0;
6
7    // TODO: (7-6.i) Integrate f using weighted Gauss quadrature
8    // START
9    for (unsigned int j = 0; j < n; ++j) {
10     // Weight
11     const double w =
12         M_PI / (n + 1) * std::pow(std::sin((j + 1) * M_PI / (n + 1)), 2);
13
14     // Node
15     const double xi = std::cos((j + 1.) / (n + 1) * M_PI);
16
17     I += w * f(xi);
18   }
19   // END
20   return I;
21 }
```

Get it on GitLab (quadU.hpp).

HINT 1 for (7-7.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 (7-7.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 (7-7.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

$$I(f) = \int_{-1}^1 f_1((t+1)/2)\, ds + \int_{-1}^1 f_2((t+1)/2)\, ds. \tag{7.7.2}$$

Observe that we never get close to the singularity of the root terms.

SOLUTION of (7-7.b):

Applying Gauss quadrature to these integrals in (7.7.2), 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

$$w_j^n = \begin{cases} \tilde{w}_j^n \sqrt{2 - c_j^n} & \text{if } j = 1, \dots, n, \\ \tilde{w}_{j-n}^n \frac{2 - c_j^n}{\sqrt{c_j^n}} & \text{if } j = n+1, \dots, 2n. \end{cases}$$

SOLUTION of (7-7.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 $\mathcal{P}_{-1} = \emptyset$.

SOLUTION of (7-8.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 $l, j = 1, \ldots, n$.

SOLUTION of (7-8.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 $l = 1, \ldots, n$ 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 $l = 1, \ldots, n-1$ 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 7.8.2: Implementation of `comp_g_gausspts`.

```
2  template <typename Function>
3  Eigen::VectorXd comp_g_gausspts(Function f, unsigned int n) {
4    Eigen::VectorXd g = Eigen::VectorXd::Zero(n);
5
6    // TODO: (7-8.b) Compute (g_n(ξ_l^n))_{l=1}^n with optimal complexity
7    // O(n).  START
8    Eigen::VectorXd w(n), xi(n);
9    gaussrule(n, w, xi);  // Compute Gauss nodes and weights relative to
         [-1,1]
10
11   Eigen::VectorXd p(n), q(n), expxi(n), fxi(n);
12   w = w / 2.;                                  // Rescale the weights to
         [0,1]
13   xi = (xi + Eigen::MatrixXd::Ones(n, 1)) / 2.;  // Rescale the nodes to [0,1]
14
15   // Evaluate at points needed later on
16   for (unsigned int l = 0; l < n; l++) {
17     fxi(l) = f(xi(l));
18     expxi(l) = std::exp(xi(l));
19   }
20
21   // Initialize the Eigen::VectorXds p and q; base cases for recursive
         formulas
22   p(0) = 0;
23   q(n - 1) = w(n - 1) * expxi(n - 1) * fxi(n - 1);
24
25   // Fill in the Eigen::VectorXds p and q (O(n) complexity)
26   for (unsigned int l = 0; l < n - 1; l++) {
27     p(l + 1) = p(l) + w(l) / expxi(l) * fxi(l);
28     q(n - 2 - l) =
29         q(n - 1 - l) + w(n - 2 - l) * expxi(n - 2 - l) * fxi(n - 2 - l);
30   }
31
32   // Finally, constructing the output (O(n) complexity)
```

```
33    g = (p.array() * expxi.array() + q.array() / expxi.array()).matrix();
34    // END
35    return g;
36  }
```

Get it on GitLab (gauss_pts.hpp).

SOLUTION of (7-8.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 7.8.3: Testing code.

```
2  double testCompGGaussPts() {
3    double g_val = 0;
4
5    // TODO: (7-8.c) Compute g(ξ_11^21) for f(y) = e^{-|0.5-y|}.
6    // START
7    constexpr unsigned int n = 21;
8    auto f = [](double y) { return std::exp(-std::abs(.5 - y)); };
9    const Eigen::VectorXd g = comp_g_gausspts(f, n);
10   g_val = g((n + 1) / 2 - 1);
11   // END
12   return g_val;
13 }
```

Get it on GitLab (gauss_pts.hpp).

SOLUTION of (7-9.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.

HINT 1 for (7-9.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 (7-9.c): You may use $\sqrt{1-t^2} = \sqrt{1-t}\sqrt{1+t}$. ⌟

SOLUTION of (7-9.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 7.9.2: Implementation of `quadsingint`.

```
template <class Function>
double quadsingint(const Function& f, const unsigned n) {
  double I = 0.;

  // TODO: (7-9.c) Compute ∫_{-1}^{1} √(1-t^2) f(t) dt using
  // Gauss-Legendre quadrature. Ensure asymptotic exponential convergence.
  // START
  // Method 1: s = √(1 ± t)
  // Will use same node twice
  QuadRule Q = gauleg(n);

  // quadrature done on interval [0, 1], i.e. nodes and weights need to be
  // shifted
  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));
  }

```

```
  // Method 2: t = cos(s)
  // We are actually using twice the number of nodes than Method 1
  //  QuadRule Q = gauleg(2 * n);
  //
  // quadrature done on interval [-pi/2, pi/2], i.e. nodes and weights need to
  // be shifted
  //  for(unsigned i = 0; i < 2 * n; ++i) {
  // Evaluate 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;
  //  }
  // END
  return I;
}
```

Get it on GitLab (quadsingint.hpp).

SOLUTION of (7-9.d):

Using Method 1:

Let (x_j, w_j), $j = 1, \ldots, n$ 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 $l \in 1, \ldots, 2n$ in $[-1, 1]$ as follows:

$$c_{2j-i} = (-1)^i (1 - x_j^2), \qquad j = 1, \ldots, n, \ i = 0, 1.$$

The weights $\tilde{w}_l, l = 1, \ldots, 2n$, become:

$$\tilde{w}_{2j-i} = 2 w_j x_j^2 \sqrt{2 - x_j^2}, \qquad j = 1, \ldots, n, \ i = 0, 1.$$

Using Method 2:

Let (x_j, w_j), $j = 1, \ldots, n$ 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:

$$c_j = \sin(x_j \pi / 2), \qquad j = 1, \ldots, n$$

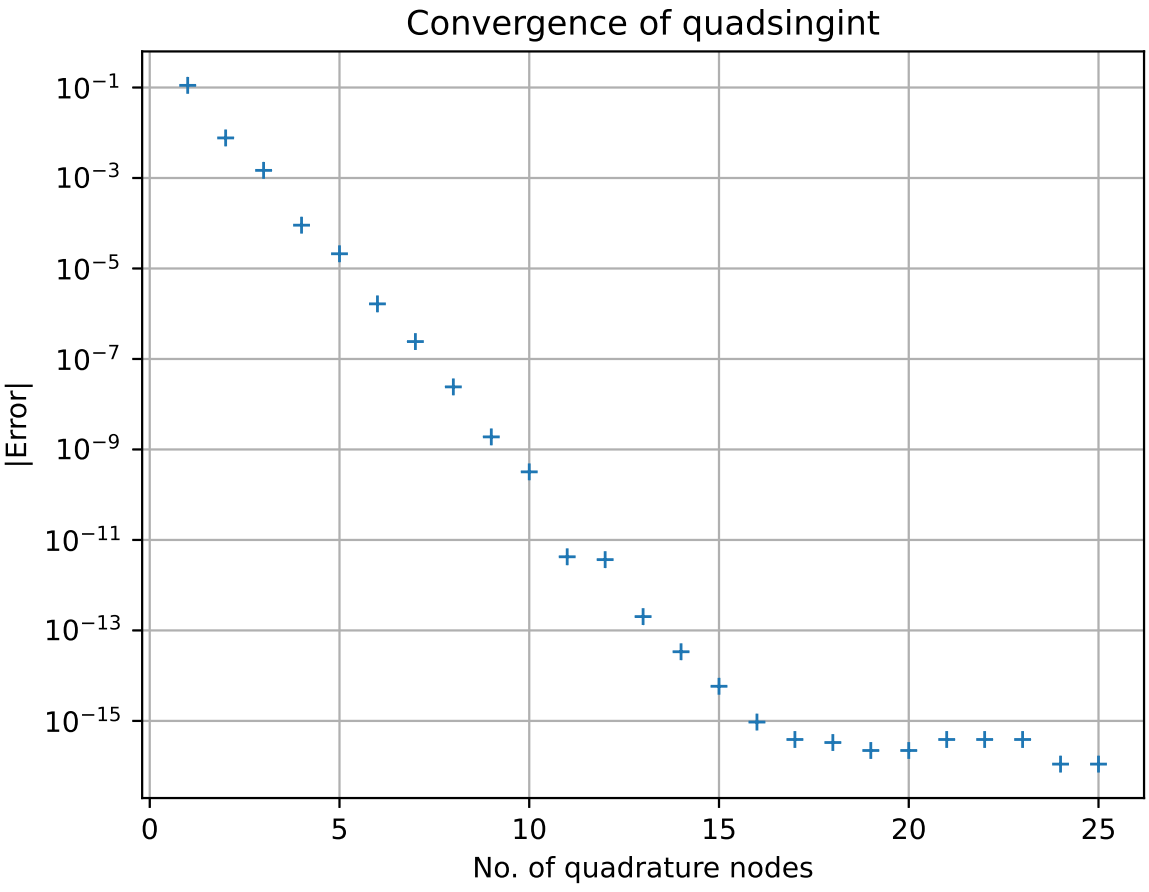
The weights $\tilde{w}_j, j = 1, \ldots, n$, become:

$$\tilde{w}_j = w_j \cos^2(x_j \pi / 2) \pi / 2.$$

SOLUTION of (7-9.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 7.9.3: Implementation of convergence test.

```
void tabAndPlotQuadErr() {
  // Max num of Gauss points to use (in each direction)
  constexpr unsigned max_N = 25;
  // "Exact" integral
  constexpr double I_ex = 0.483296828976607;

  auto f = [](double t) { return 1. / (2. + std::exp(3. * t)); };

  plt::figure();
  // TODO: (7-9.e) Tabulate and plot quadrature error for the given function.
  // START
  std::vector<int> num_nodes(max_N);  // number of nodes on x axis
  std::vector<double> err(max_N);     // error on y axis

  // Table header
  std::cout << std::setw(3) << "N" << std::setw(15) << "I_approx"
            << std::setw(15) << "error" << std::endl;

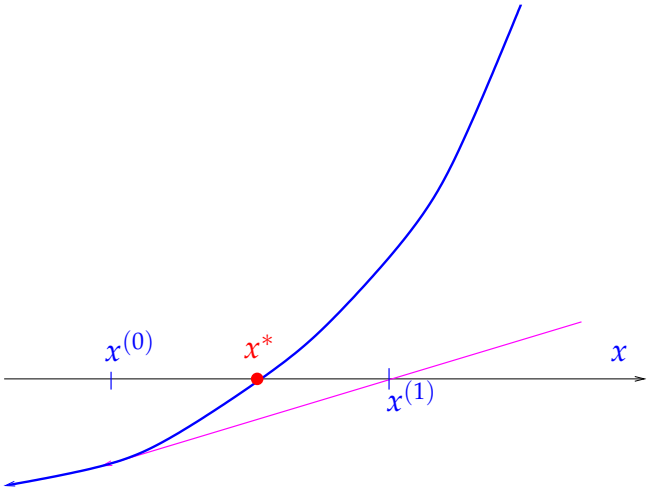
  for (unsigned N = 1; N <= max_N; ++N) {
    const double I_approx = quadsingint(f, N);
    num_nodes[N - 1] = N;
    err[N - 1] = std::abs(I_ex - I_approx);

    std::cout << std::setw(3)
              << N
```

```
                // Value of integral
                << std::setw(15)
                << I_approx
                // Error
                << std::setw(15) << err[N - 1] << std::endl;
    }
    plt::semilogy(num_nodes, err, "+", {{"label", "Error"}});
    plt::xlabel("No. of quadrature nodes");
    plt::ylabel("|Error|");
    plt::title("Convergence of quadsingint");
    plt::grid("True");
    // END
    plt::savefig("./cx_out/convergence.png");
}
```

Get it on GitLab (quadsingint.hpp).

SOLUTION of (8-1.a):

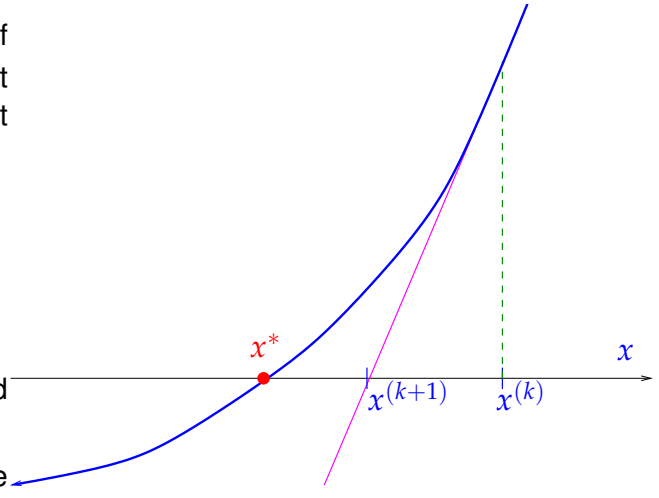


If $x^{(0)} < x^*$, where $F(x^*) = 0$, then $x^{(1)} > x^*$ and $F(x^{(1)}) > 0$.
Therefore, we may assume $x^{(1)} > x^*$ from now.

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

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} < x^{(k)} .$$



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:

$$x^+ = x^+ - \frac{F(x^+)}{F'(x^+)} \quad \blacktriangleright \quad F(x^+) = 0 .$$

SOLUTION of (8-10.a):

The fixed-point iteration $\mathbf{x}^{(k)} \to \mathbf{x}^{(k+1)}$, $k \in \mathbb{N}$, is defined by

$$\mathbf{x}^{(k+1)} \in \mathbb{R}^n: \quad \mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k+1)} = \boldsymbol{v} \quad \Leftrightarrow \quad \mathbf{x}^{(k+1)} = \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b} \,. \tag{8.10.3}$$

HINT 1 for (8-10.b): For the sake of efficiency you have to use sparse matrix data types, see [Lecture → Section 2.7.2]. ⌟

SOLUTION of (8-10.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 8.10.4: Iterate computation.

```
2  Eigen::VectorXd fixed_point_step(const Eigen::VectorXd &xk,
3                                   const Eigen::VectorXd &b) {
4    Eigen::VectorXd x_new = Eigen::VectorXd::Zero(b.size());
5    // TODO: (8-10.b) implement one step of the fixed point iteration
6    // START
7    // Define matrix by fixed point x^k
8    const unsigned int n = xk.size();
9    const double nrm = xk.norm();
10   Eigen::SparseMatrix<double> A(n, n);
11   A.reserve(3 * n);
12   for (unsigned int i = 0; i < n; ++i) {
13     if (i > 0) A.insert(i, i - 1) = 1;
14     A.insert(i, i) = 3 + nrm;
15     if (i < n - 1) A.insert(i, i + 1) = 1;
16   }
17   // solve linear system
18   Eigen::SparseLU<Eigen::SparseMatrix<double>> Ax_lu(A);
19   x_new = Ax_lu.solve(b);
20   // END
21   return x_new;
22 }
```

Get it on GitLab (quasilin.hpp).

SOLUTION of (8-10.c):

C++11-code 8.10.5: Solution of fixed point.

```
Eigen::VectorXd solveQuasiLinSystem(double rtol, double atol,
                                    Eigen::VectorXd &b) {
  const int n = b.size();

  Eigen::VectorXd x = b;
  // TODO: (8-10.c) apply the iterative method fixed_point_step to solve
  // the quasi-linear system
  // START
  Eigen::VectorXd x_new(n);
  constexpr unsigned int max_itr = 100;
  for (unsigned int itr = 0; itr < max_itr; ++itr) {
    x_new = fixed_point_step(x, b);

    const double err = (x - x_new).norm();
    x = x_new;

    // Check if Fixed point method is successful
    if (err < atol || err < rtol * x_new.norm()) {
      break;
    }
  }
  // END
  return x;
}
```

Get it on GitLab (quasilin.hpp).

SOLUTION of (8-10.d):

The Newton iteration, as provided in the lecture document in [Lecture → Ex. 8.5.1.30], reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2} \right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \,. \tag{8.10.7}$$

SOLUTION of (8-10.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.

HINT 1 for (8-10.f): If you did not manage to solve Sub-problem (8-10.e), use directly the Newton's method formula. ⌟

SOLUTION of (8-10.f):

C++11-code 8.10.8: Newton step.

```
Eigen::VectorXd newton_step(const Eigen::VectorXd &x,
                            const Eigen::VectorXd &b) {
  Eigen::VectorXd x_new = Eigen::VectorXd::Zero(b.size());
  // TODO: (8-10.f) implement one Newton step exploiting the
  // Sherman-Morrison-Woodbury formula.
  // START
  // Define matrix by fixed point x^k
  const unsigned int n = x.size();
  const 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;
}
```

Get it on GitLab (quasilin.hpp).

SOLUTION of (8-10.g):

C++11-code 8.10.9: Solve Newton.

```
Eigen::VectorXd solveQLSystem_Newton(double rtol, double atol,
                                     Eigen::VectorXd &b) {
  const unsigned int n = b.size();

  Eigen::VectorXd x = b;
  // TODO: (8-10.c) apply the iterative method newton_step to solve
  // the quasi-linear system
  // START
  Eigen::VectorXd x_new(n);
  constexpr unsigned int max_itr = 100;
  for (unsigned int itr = 0; itr < max_itr; ++itr) {
    x_new = newton_step(x, b);

    const double err = (x - x_new).norm();
    x = x_new;

    // Check if Newton method is successful
    if (err < atol || err < rtol * x_new.norm()) {
      break;
    }
  }
  // END
  return x;
}
```

Get it on GitLab (quasilin.hpp).

HINT 1 for (8-11.a): 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. ⌟

HINT 2 for (8-11.a): Assume general initial values $\Phi_A(0) = A_0$, $\Phi_B(0) = B_0$, and verify your solution by plugging it into the system (8.11.1). ⌟

SOLUTION of (8-11.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 (8.11.1) 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

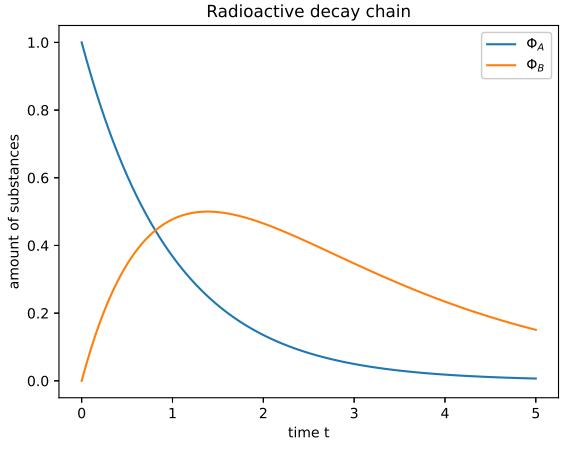
$$y(t) = e^{ct} y(0) + \int_0^t e^{c(t-\tau)} g(\tau) \, d\tau \, .$$

Thus, the solution of the second equation is

$$\begin{aligned}
\Phi_B(t) &= e^{-\lambda_2 t} B_0 + \int_0^t e^{-\lambda_2 (t-\tau)} \lambda_1 e^{-\lambda_1 \tau} A_0 \, d\tau \\
&= e^{-\lambda_2 t} B_0 + \lambda_1 e^{-\lambda_2 t} A_0 \int_0^t e^{(\lambda_2 - \lambda_1)\tau} \, d\tau \\
&= e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} \left(e^{(\lambda_2 - \lambda_1) t} - 1 \right) A_0 \\
&= e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(e^{-\lambda_1 t} - e^{-\lambda_2 t} \right) A_0 \, .
\end{aligned}$$

It satisfies the initial condition $\Phi_B(0) = B_0$, we test if it is a solution of the ODE (8.11.1):

$$\begin{aligned}
\Phi_B'(t) &= -\lambda_2 e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(-\lambda_1 e^{-\lambda_1 t} + \lambda_2 e^{-\lambda_2 t} \right) A_0 \\
&= -\lambda_2 e^{-\lambda_2 t} B_0 + \lambda_2 \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} A_0 - \lambda_1 \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_1 t} A_0 \\
&= -\lambda_2 \Phi_B(t) + \lambda_2 \frac{\lambda_1}{\lambda_2 - \lambda_1} \Phi_A(t) - \lambda_1 \frac{\lambda_1}{\lambda_2 - \lambda_1} \Phi_A(t) \\
&= -\lambda_2 \Phi_B(t) + \lambda_1 \Phi_A(t) \, .
\end{aligned}$$



◁ Amounts $t \mapsto \Phi_A(t)$ and $t \mapsto \Phi_B(t)$ for
- $A_0 = 1$, $B_0 = 0$,
- $\lambda_1 = 1$, $\lambda_2 = \frac{1}{2}$.

SOLUTION of (8-11.b):

The N components of the function which describes the overdetermined system are given by

$$F_i(\mathbf{x}) \quad = \quad \Phi_B(t_i) - m_i \quad = \quad e^{-\lambda_2 t_i} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i} \right) A_0 - m_i, \qquad i = 1, \ldots, N.$$

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:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{F}(\mathbf{x})\|_2^2 = \underset{\mathbf{x}}{\operatorname{argmin}} \sum_{i=1}^{N} |F_i(\mathbf{x})|^2.$$

SOLUTION of (8-11.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, $i = 1, \ldots,$ of $\mathbf{F}$ is given by

$$\mathbf{grad}\, F_i(\mathbf{x}) = \left[\frac{\partial F_i}{\partial A_0}, \frac{\partial F_i}{\partial B_0}, \frac{\partial F_i}{\partial \lambda_1}, \frac{\partial F_i}{\partial \lambda_2}\right]$$

$$= \begin{bmatrix} \frac{\lambda_1}{\lambda_2-\lambda_1}\left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i}\right) \\ e^{-\lambda_2 t_i} \\ \frac{\lambda_2}{(\lambda_2-\lambda_1)^2}\left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i}\right) A_0 - \frac{\lambda_1 t_i}{\lambda_2-\lambda_1} e^{-\lambda_1 t_i} A_0 \\ -t_i e^{-\lambda_2 t_i} B_0 - \frac{\lambda_1}{(\lambda_2-\lambda_1)^2}\left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i}\right) A_0 + \frac{\lambda_1 t_i}{\lambda_2-\lambda_1} e^{-\lambda_2 t_i} A_0 \end{bmatrix} \in \mathbb{R}^4.$$

The Jacobian $\mathrm{D}\,\mathbf{F}$ is the collection of these vectors:

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \begin{pmatrix} \mathbf{grad}\, F_1(\mathbf{x})^T \\ \vdots \\ \mathbf{grad}\, F_N(\mathbf{x})^T \end{pmatrix} \quad \in \mathbb{R}^{N,4}\,.$$

The Gauss-Newton iteration reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{s}\,, \quad \text{where } \mathbf{s} \text{is the least-squares solution of} \quad \mathbf{DF}(\mathbf{x}^{(k)})\,\mathbf{s} = \mathbf{F}(\mathbf{x}^{(k)})\,,$$

which corresponds to the statement that the increment $\mathbf{s}$ is the solution of the minimization problem

$$\mathbf{s} = \operatorname*{argmin}_{\mathbf{a}\in\mathbb{R}^4} \|\mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{DF}(\mathbf{x}^{(k)})\,\mathbf{a}\|_2\,.$$

HINT 1 for (8-11.d): Consult [Lecture → Code 3.3.4.2] to see EIGEN's "generic linear least squares solver" in action. ⌟

HINT 2 for (8-11.d): Use the generated plot of your fitted Φ_B along with the measured data to spot errors in your implementation. ⌟

SOLUTION of (8-11.d):

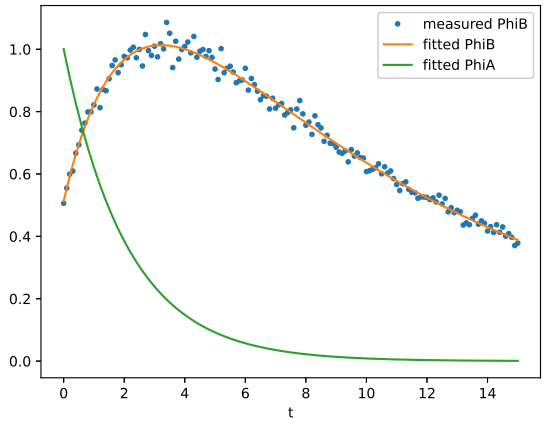
The implementation of $\mathbf{F}$ and $D\mathbf{F}$ is straightforward. QR decomposition is used to solve each Least Squares Problem.

C++ code 8.11.2: Gauss-Newton iteration core loop

```
template <typename F, typename DF>
std::vector<double> GaussNewton(Eigen::Vector4d& x, F&& f, DF&& df,
                                const double tol = 1e-14) {
  std::vector<double> gn_update;

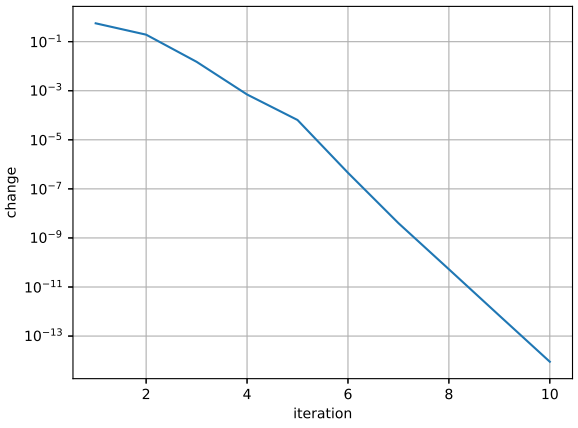
  // TODO: (8-11.d) Implement the Newton iteration for the given Least
      squares
  // problem.  Put in gn_update the norms of the iteration steps for
      further
  // investigation.
  // START
  do {
    Eigen::Vector4d s = df(x).colPivHouseholderQr().solve(f(x));
    x -= s;
    gn_update.push_back(s.lpNorm<Eigen::Infinity>());
  } while (gn_update.back() > tol);
  // END
  return gn_update;
}
```

Get it on GitLab (radioactive.hpp).



◁ The solution of the system of ODEs (8.11.1) for the estimated parameters matches the given data points rather well.

This observation provides a first validation of the code.



$\|\mathbf{s}\|_2$	rate	est. order
5.6022e-01	0	0
1.9307e-01	3.4463e-01	0
1.5121e-02	7.8320e-02	2.3909e+00
7.0427e-04	4.6575e-02	1.2041e+00
6.4211e-05	9.1174e-02	7.8096e-01
4.5257e-07	7.0481e-03	2.0689e+00
4.0132e-09	8.8676e-03	9.5365e-01
5.2940e-11	1.3191e-02	9.1595e-01
6.8268e-13	1.2895e-02	1.0052e+00
8.7962e-15	1.2885e-02	1.0002e+00

From both the plot and the tabulated data we conclude linear convergence .

SOLUTION of (8-12.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

$$\begin{aligned} \text{eq. (8.12.1)} \quad &\Leftrightarrow \quad x^2 - 2m_1 x + m_1^2 + y^2 - 2m_2 y + m_2^2 - r^2 = 0 \\ &\Leftrightarrow \quad -2xm_1 - 2ym_2 - c = -x^2 - y^2 \ . \end{aligned}$$

Thus, for the data (x_i, y_i), $i = 1, \ldots, N$, we have the $N \times 3$ linear system

$$\begin{pmatrix} -2x_1 & -2y_1 & -1 \\ \vdots & \vdots & \vdots \\ -2x_N & -2y_N & -1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ c \end{pmatrix} = \begin{pmatrix} -x_1^2 - y_1^2 \\ \vdots \\ -x_N^2 - y_N^2 \end{pmatrix} \ .$$

SOLUTION of (8-12.j):

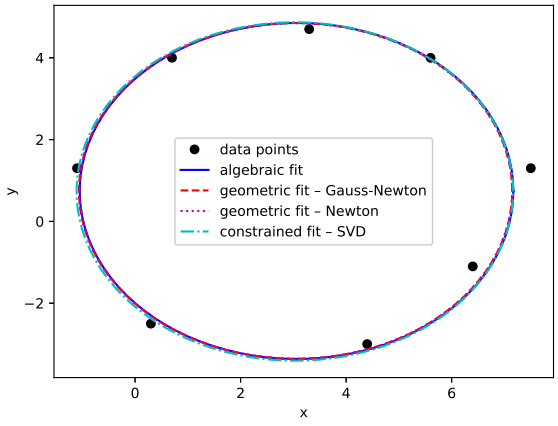
The centers of the circles computed with linear L.S., Gauss-Newton method, Newton method and non-linear constrained minimization are

```
(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.



The various fitted circles do not differ much.

This is due to the fact that the given points almost lie on a circle.

SOLUTION of (8-12.b):

C++11-code 8.12.2: Fitting circle using algebraic method.

```
Eigen::Vector3d circl_alg_fit(const Eigen::VectorXd& x,
                              const Eigen::VectorXd& y) {
  Eigen::Vector3d z = Eigen::VectorXd::Zero(3);
  assert(x.size() == y.size() && "Size mismatch!");
  const unsigned int n = x.size();

  // TODO: (8-12.b) Fit the circle in the least squares sense.
  // START
  Eigen::MatrixXd A(n, 3);
  A << -2. * x, -2. * y, -Eigen::VectorXd::Ones(n);
  // use Eigen's arrays which provide componentwise operations
  Eigen::VectorXd b = -x.array() * x.array() - y.array() * y.array();

  z = lsqHHR(A, b);  // lsqSVD(A, b); lsqNRM(A, b);
  z(2) = std::sqrt(z(2) + z(0) * z(0) + z(1) * z(1));
  // END

  return z;
}
```

Get it on GitLab (circle_approx.hpp).

Estimates: $[m_1, m_2, r]^\top \in \mathbb{R}^3 = [3.0603, 0.743607, 4.10914]$

SOLUTION of (8-12.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 $i = 1, \ldots, N$, 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 $\|\mathbf{F}(\mathbf{z})\|^2 = \sum_{i=1}^{n} d_i^2$.

SOLUTION of (8-12.d):

First of all we compute a few derivatives that will be useful later on:

$$\partial_{m_1} R_i = \frac{-(x_i - m_1)}{R_i}, \quad \partial_{m_1} \frac{1}{R_i} = \frac{x_i - m_1}{R_i^3}, \tag{8.12.3}$$

$$\partial_{m_2} R_i = \frac{-(y_i - m_2)}{R_i}, \quad \partial_{m_2} \frac{1}{R_i}, = \frac{y_i - m_2}{R_i^3}, \tag{8.12.4}$$

$$\partial^2_{m_1,m_1} R_i = \frac{1}{R_i} - \frac{(x_i - m_1)^2}{R_i^3}, \; \partial^2_{m_2,m_2} R_i = \frac{1}{R_i} - \frac{(y_i - m_2)^2}{R_i^3}, \tag{8.12.5}$$

$$\partial^2_{m_1,m_2} R_i = \frac{-(x_i - m_1)(y_i - m_2)}{R_i^3}\,. \tag{8.12.6}$$

Thus, the gradients of the components of $\mathbf{F}$, assembled in the Jacobian $\mathbf{DF}$, are

$$\mathbf{grad}\, F_i(\mathbf{z}) = \begin{bmatrix} \dfrac{-(x_i - m_1)}{R_i}, & \dfrac{-(y_i - m_2)}{R_i}, & -1 \end{bmatrix}^T, \qquad \mathrm{D}\,\mathbf{F}(\mathbf{z}) = \begin{bmatrix} \mathbf{grad}\, F_1(\mathbf{z})^T \\ \vdots \\ \mathbf{grad}\, F_N(\mathbf{z})^T \end{bmatrix} \in \mathbb{R}^{N,3}.$$

Using the formulas in [Lecture → Section 8.7.1], 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

$$\begin{aligned} \mathrm{H}\,\Phi(\mathbf{z}) &= \underbrace{\mathbf{DF}(\mathbf{z})^T}_{\in \mathbb{R}^{3,N}} \cdot \underbrace{\mathbf{DF}(\mathbf{z})}_{\in \mathbb{R}^{N,3}} + \sum_{i=1}^{N} \underbrace{F_i(\mathbf{z})}_{\in \mathbb{R}} \underbrace{\mathbf{H}F_i(\mathbf{x})}_{\in \mathbb{R}^{3,3}} \\ &= \sum_{i=1}^{N} \begin{bmatrix} \frac{x_i - m_1}{R_i} \\ \frac{y_i - m_2}{R_i} \\ 1 \end{bmatrix} \begin{bmatrix} \frac{x_i - m_1}{R_i}, & \frac{y_i - m_2}{R_i}, & 1 \end{bmatrix} + \sum_{i=1}^{N} (R_i - r) \begin{bmatrix} \frac{1}{R_i} - \frac{(x_i - m_1)^2}{R_i^3} & \frac{-(x_i - m_1)(x_i - m_2)}{R_i^3} & 0 \\ \frac{-(x_i - m_1)(x_i - m_2)}{R_i^3} & \frac{1}{R_i} - \frac{(y_i - m_2)^2}{R_i^3} & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} N + r\sum_{i=1}^{N} \left(\frac{(x_i - m_1)^2}{R_i^3} - \frac{1}{R_i} \right) & r\sum_{i=1}^{N} \frac{(x_i - m_1)(y_i - m_2)}{R_i^3} & \sum_{i=1}^{N} \frac{(x_i - m_1)}{R_i} \\ r\sum_{i=1}^{N} \frac{(x_i - m_1)(y_i - m_2)}{R_i^3} & N + r\sum_{i=1}^{N} \left(\frac{(y_i - m_2)^2}{R_i^3} - \frac{1}{R_i} \right) & \sum_{i=1}^{N} \frac{(y_i - m_2)}{R_i} \\ \sum_{i=1}^{N} \frac{(x_i - m_1)}{R_i} & \sum_{i=1}^{N} \frac{(y_i - m_2)}{R_i} & N \end{bmatrix}. \end{aligned}$$

SOLUTION of (8-12.e):

From [Lecture → Eq. (8.7.2.1)] we learn that the Gauss-Newton iteration 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++ code 8.12.7: Fitting circle using geometric method and Gauss-Newton

```
2  void circl_geo_fit_GN(const Eigen::VectorXd& x, const Eigen::VectorXd& y,
3                        Eigen::Vector3d& z, std::vector<double>* err = nullptr) {
4    assert(x.size() == y.size() && "Size mismatch!");
5    const unsigned int n = x.size();
6    constexpr double tol = 1e-14;
7
8    // TODO: (8-12.e) Use the Gauss-Newton method to compute the geometric
       fit.
9    // START
10   // the radius
11   auto R = [&x, &y](const Eigen::Vector3d& z) -> Eigen::ArrayXd {
12     return sqrt((x.array() - z(0)) * (x.array() - z(0)) +
13                 (y.array() - z(1)) * (y.array() - z(1)));
14   };
15
16   auto F = [&R](const Eigen::Vector3d& z) -> Eigen::VectorXd {
17     return R(z) - z(2);
18   };
19
20   auto DF = [&x, &y, &R, &n](const Eigen::Vector3d& z) -> Eigen::MatrixX3d {
21     Eigen::MatrixXd ret(n, 3);
22     ret << -(x.array() - z(0)) / R(z), -(y.array() - z(1)) / R(z),
23         -Eigen::VectorXd::Ones(n);
24     return ret;
25   };
26
27   // Gauss-Newton method with correction-based stopping criterion
28   // we keep track of the error-norms in err
29   double err_norm;
30   do {
31     Eigen::Vector3d s =
32         lsqHHR(DF(z), F(z));  // lsqSVD(DF(z), F(z)); lsqNRM(DF(z), F(z));
33     z -= s;
34     err_norm = s.lpNorm<Eigen::Infinity>();
35     if (err) {
36       err->push_back(err_norm);
37     }
38   } while (err_norm > tol);
39   // END
40 }
```

Get it on GitLab (circle_approx.hpp).

SOLUTION of (8-12.f):

The iteration of the Newton method [Lecture → Eq. (8.7.1.6)] reads

$$\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} - \mathbf{H}\Phi(\mathbf{z}^{(k)})^{-1}\,\mathbf{grad}\,\Phi(\mathbf{z}^{(k)}).$$

C++11-code 8.12.8: Fitting circle using distance approach and Newton's method

```
void circl_geo_fit_N(const Eigen::VectorXd& x, const Eigen::VectorXd& y,
                     Eigen::Vector3d& z, std::vector<double>* err = nullptr) {
  assert(x.size() == y.size() && "Size mismatch!");
  const unsigned int n = x.size();
  constexpr double tol = 1e-14;

  // TODO: (8-12.f) Use the Newton method to compute the geometric fit.
  // START
  auto gradPhi = [&n](const Eigen::Vector3d& z, const Eigen::ArrayXd& xm,
                      const Eigen::ArrayXd& ym,
                      const Eigen::ArrayXd& R) -> Eigen::Vector3d {
    Eigen::Vector3d ret;
    ret << xm.matrix().dot((z(2) / R - 1).matrix()),
        ym.matrix().dot((z(2) / R - 1).matrix()), n * z(2) - R.sum();
    return ret;
  };

  auto HessianPhi = [&n](const Eigen::Vector3d& z, const Eigen::ArrayXd& xm,
                         const Eigen::ArrayXd& ym,
                         const Eigen::ArrayXd& R) -> Eigen::Matrix3d {
    const double sum_x = (xm / R).sum();
    const double sum_y = (ym / R).sum();
    const double mixed_sum = z(2) * (xm * ym / pow(R, 3)).sum();
    const double inv_sum = z(2) * (1 / R).sum();
    Eigen::Matrix3d ret;
    ret << n - inv_sum + z(2) * (pow(xm, 2) / pow(R, 3)).sum(), mixed_sum,
        sum_x, mixed_sum, n - inv_sum + z(2) * (pow(ym, 2) / pow(R, 3)).sum(),
        sum_y, sum_x, sum_y, n;
    return ret;
  };

  // Newton method with correction-based stopping criterion
  // we keep track of the error-norms in err
  double err_norm;
  do {
    Eigen::ArrayXd xm = x.array() - z(0);
    Eigen::ArrayXd ym = y.array() - z(1);
    Eigen::ArrayXd R = sqrt(xm * xm + ym * ym);
    Eigen::Vector3d s = lsqHHR(
        HessianPhi(z, xm, ym, R),
        gradPhi(
            z, xm, ym,
            R));  // lsqSVD(HessianPhi(z, xm, ym, R), gradPhi(z, xm, ym,
                  R));
                    // lsqNRM(HessianPhi(z, xm, ym, R), gradPhi(z, xm, ym,
                       R));
    z -= s;
    err_norm = s.lpNorm<Eigen::Infinity>();
    if (err) {
```

```
49          err->push_back(err_norm);
50        }
51      } while (err_norm > tol);
52      // END
53  }
```

Get it on GitLab (circle_approx.hpp).

SOLUTION of (8-12.g):

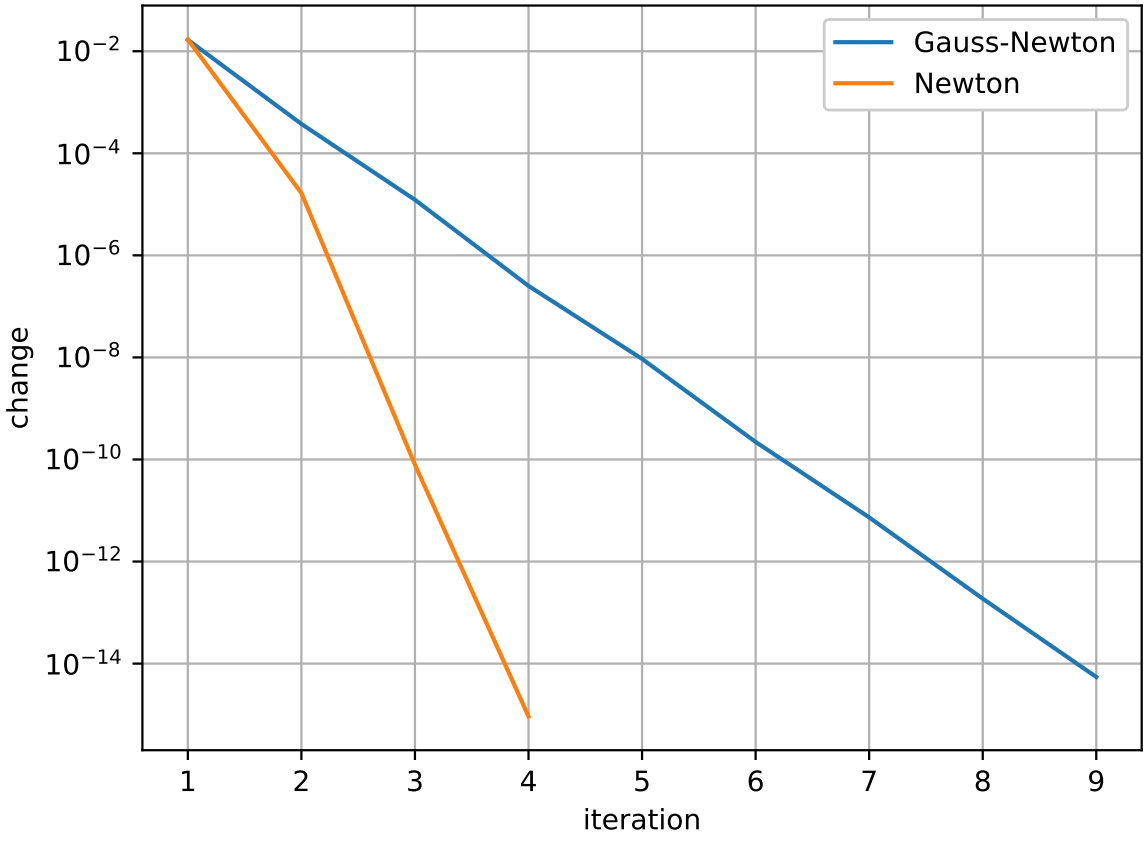
The norm of the iteration corrections, their rates and the order of convergence, for the Gauss-Newton method are:

```
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.



SOLUTION of (8-12.h):

Defining $\mathbf{x} = [x_1, x_2]^\top$, 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 the circle parameters

$$m_1 = \frac{-b_1}{2a}, \quad m_2 = \frac{-b_2}{2a} \quad \text{and} \quad r = \sqrt{\frac{b_1^2 + b_2^2}{4a^2} - \frac{c}{a}}\,.$$

Here, we have to assume $c/a < (b_1^2 + b_2^2)/(4a^2)$).

SOLUTION of (8-12.i):

The least squares linear system to be solved is just the collection of the equations (8.12.9) 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.

C++11-code 8.12.10: Fitting circle using SVD.

```
2  Eigen::Vector3d circl_svd_fit(const Eigen::VectorXd& x,
3                                const Eigen::VectorXd& y) {
4    const unsigned int N = x.size();
5    assert(N == y.size() && "Size mismatch!");
6    Eigen::Vector3d ret = Eigen::VectorXd::Zero(3);
7
8    // TODO: (8-12.i) Fit the circle using the constrained method and SVD.
9    // START
10   const Eigen::ArrayXd x_ = x.array();
11   const Eigen::ArrayXd y_ = y.array();
12   Eigen::MatrixXd A(x.size(), 4);
13   A << (x_.square() + y_.square()).matrix(), x, y,
14       Eigen::VectorXd::Ones(x.size());
15
16   // we need to tell Eigen to compute V
17   Eigen::JacobiSVD<Eigen::MatrixXd> svd(A, Eigen::ComputeThinV);
18   Eigen::MatrixXd V = svd.matrixV();
19
20   // formulas from (8-12.h)
21   ret(0) = -V(1, 3) / (2. * V(0, 3));
22   ret(1) = -V(2, 3) / (2. * V(0, 3));
23   ret(2) = std::sqrt((V(1, 3) * V(1, 3) + V(2, 3) * V(2, 3)) /
24                      (4. * V(0, 3) * V(0, 3)) -
25                  V(3, 3) / V(0, 3));
26   // END
27   return ret;
28 }
```

Get it on GitLab (circle_approx.hpp).

SOLUTION of (8-13.a):

We start with the obvious implication

$$\boldsymbol{x} \in R_{\mathbf{d}} \cap \partial \mathcal{L}_c \quad \Leftrightarrow \quad \exists \xi > 0\colon \quad \boldsymbol{x} = \xi \mathbf{d} \quad \text{and} \quad f(\xi \mathbf{d}) = c\,.$$

Hence, we have to solve the zero-finding problem for a real-valued function $F : \mathbb{R}_0^+ \to \mathbb{R}$

$$\text{find} \quad \xi > 0: \quad F(\xi) := f(\xi \mathbf{d}) - c = 0\,. \tag{8.13.4}$$

By the chain rule

$$F'(\xi) = \mathbf{grad}\, f(\xi \mathbf{d})^\top \mathbf{d}\,, \quad \xi > 0$$

The Newton iteration [Lecture → Eq. (8.4.2.1)] for (8.13.4) is the recursion

$$\xi^{(k+1)} = \xi^{(k)} - \frac{F(\xi^{(k)})}{F'(\xi^{(k)})} = \xi^{(k)} - \frac{f(\xi^{(k)}\mathbf{d}) - c}{\mathbf{grad}\, f(\xi^{(k)}\mathbf{d})^\top \mathbf{d}}\,, \quad k \geq 0\,. \tag{8.13.5}$$

As initial guess we choose some $\xi^{(0)} > 0$.

Note that the denominator in (8.13.5) cannot vanish, because $\mathbf{d} \neq \mathbf{0}$ and f is strictly convex, which means that the derivative of $\xi \mapsto f(\xi \mathbf{d})$ is strictly increasing on $[0, \infty[$.

SOLUTION of (8-13.b):

According to [Lecture → Eq. (8.4.2.30)] the two-point iteration for the secant method applied to (8.13.4) reads:

$$\begin{aligned} \xi^{(k+1)} &= \xi^{(k)} - \frac{\xi^{(k)} - \xi^{(k-1)}}{F(\xi^{(k)}) - F(\xi^{(k-1)})} F(\xi^{(k)}) \\ &= \frac{\xi^{(k)} - \xi^{(k-1)}}{f(\xi^{(k)}\mathbf{d}) - f(\xi^{(k-1)}\mathbf{d})} \big(f(\xi^{(k)}\mathbf{d}) - c\big) \ . \end{aligned} \tag{8.13.6}$$

SOLUTION of (8-13.c):

The implementation of (8.13.6) is straightforward. The problem statements asks us to use $\xi^{(0)} = \frac{\|\mathbf{x}_0\|_2}{\|\mathbf{d}\|_2}$ and $\xi^{(1)} = \frac{\|\mathbf{x}_1\|_2}{\|\mathbf{d}\|_2}$. An efficient implementation uses only a single f-evaluation per step.

Concerning the correction-based termination criterion see [Lecture → Code 8.4.2.31].

C++ code 8.13.7: Sub-problem (8-13.c): The function `pointLevelSet()`

```
template <typename Functor>
Eigen::Vector2d pointLevelSet(Functor&& f, const Eigen::Vector2d& d, double c,
                              const Eigen::Vector2d& x0, double rtol = 1e-10,
                              double atol = 1e-16) {
  Eigen::Vector2d intersect = Eigen::VectorXd::Zero(2);

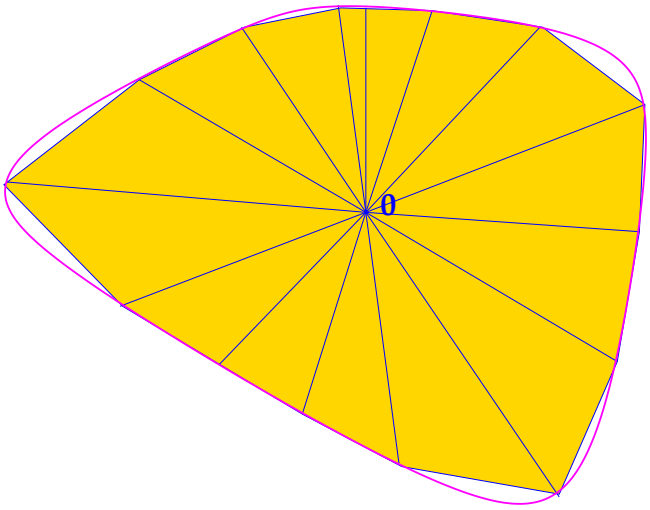
  // TODO: (8-13.c) Use the secant method to find the intersection.
  // START
  // the function whose zero has to be found
  auto F = [&f, &d, &c](double t) { return f(t * d) - c; };

  double upd = 1.;
  // initial guesses for the secant method
  double t0 = x0.norm() / d.norm();
  double t1 = 1.1 * t0;

  double fo = F(t0);
  // check correction based termination criterion
  while (std::abs(upd) > std::max(atol, rtol * std::min(t0, t1))) {
    const double fn = F(t1);
    // secant update
    upd = fn * (t1 - t0) / (fn - fo);
    // new iteration points
    t0 = t1;
    t1 = t1 - upd;
    fo = fn;
  }
  // compute intersection of the boundary of the level set with the ray
  intersect = t1 * d;
  // END
  return intersect;
}
```

Get it on GitLab (levelset.hpp).

HINT 1 for (8-13.d): A convex polygonal domain P with $\mathbf{0} \in P$ can be partitioned into (non-overlapping) triangles with one vertex in $\mathbf{0}$ and the other two vertices coinciding with corners of P.



◁ Approximating the area enclosed by a convex level set boundary by the sum of areas of triangles having the same angle $\frac{2\pi}{n}$ at vertex $\mathbf{0}$.

The area of a triangle $\triangle ABC$ with angle α at vertex A is

$$|\triangle ABC| = \tfrac{1}{2}\overline{AB} \cdot \overline{AC} \sin\alpha \ .$$

⌟

HINT 2 for (8-13.d): It is convenient to use the function `pointLevelSet()` to find points on $\partial\mathcal{L}_c$. ⌟

SOLUTION of (8-13.d):

C++ code 8.13.8: Sub-problem (8-13.d): The function `areaLevelSet()`

```
2  template <typename Functor>
3  double areaLevelSet(Functor&& f, unsigned int n, double c) {
4    assert(n > 2 && "n too small");
5    double area = 0.;
6
7    // define initial guesses
8    Eigen::Vector2d x0 = {1, 0};
9
10   // TODO: (8-13.d) Compute the "Archimedean approximation" of the area of the
11   // level set.
12   // START
13   // constant angle
14   const double sin_ct = std::sin(2. * M_PI / n) * 0.5;
15   Eigen::VectorXd side(n + 1);
16   for (unsigned int j = 0; j < n; ++j) {
17     Eigen::Vector2d d;
18     d << std::cos(2. * M_PI * j / n), std::sin(2. * M_PI * j / n);
19     // compute length of archimedean radii
20     side(j) = pointLevelSet(f, d, c, x0).norm();
21   }
22   // periodic extension for vectorization
23   side(n) = side(0);
24   // multiply pairs of adjacent sides
25   area = (side.tail(n).cwiseProduct(side.head(n))).sum() * sin_ct;
26   // END
27   return area;
28 }
```

Get it on GitLab (levelset.hpp).

SOLUTION of (8-13.e):

n	error in area
5	0.7322814883
10	0.2020820848
20	0.0539834484
40	0.0136526713
80	0.0034265044

We observe a decrease of the error by roughly a factor of four, when doubling number n of evaluation points. Thus, we conclude *algebraic convergence* with rate 2:

$$\text{error} = O((1/n)^2) \quad \text{for} \quad n \to \infty \,.$$

HINT 1 for (8-14.a): From [Lecture → Section 3.4.4.2] recall the use of the singular value decomposition (SVD) to solve low-rank best approximation problems. ⌟

SOLUTION of (8-14.a):

If $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ is the singular-value decomposition of $\mathbf{M}$, we know from [Lecture → Thm. 3.4.4.19] that $(\mathbf{U})_{:,1}(\mathbf{\Sigma})_{1,1}(\mathbf{V})_{:,1}^\top$ is a rank-1 matrix closest to $\mathbf{M}$ in Frobenius norm.

If $\mathbf{M} = \mathbf{M}^\top$ is positive semi-definite, then $\mathbf{U} = \mathbf{V}$ in its SVD, which is an immediate consequence of [Lecture → Lemma 3.4.1.6]. As a consequence, the closest rank-1 matrix is also symmetric! This means that

$$\mathbf{M} = \mathbf{M}^\top \quad \blacktriangleright \quad (\mathbf{U})_{:,1}\sqrt{(\mathbf{\Sigma})_{1,1}} \in \operatorname*{argmin}_{\mathbf{z}\in\mathbb{R}^n} \left\| \mathbf{M} - \mathbf{z}\mathbf{z}^\top \right\|_F .$$

Implementation in EIGEN is straightforward, see [Lecture → Code 3.4.2.1].

C++ code 8.14.3: Sub-problem (8-14.a): The function `symRankOneBestApproxSym()`

```
2  Eigen::VectorXd symRankOneBestApproxSym(const Eigen::MatrixXd &M) {
3    // ensure that M is symmetric
4    assert(M.cols() == M.rows() && M == M.transpose() && "M is not symmetric");
5    // Vector to be used to return result
6    Eigen::VectorXd z = Eigen::VectorXd::Zero(M.cols());
7    // TODO: (8-14.a)
8    //  solve equ. (8.14.1) for a symmetric matrix with the aid of singular value
9    //  decomposition (SVD)
10   // START
11   // Compute economical SVD of M
12   Eigen::JacobiSVD<Eigen::MatrixXd> svd(
13       M, Eigen::ComputeThinU | Eigen::ComputeThinV);
14   // Extract first column of u and first singular value
15   Eigen::VectorXd first_U = svd.matrixU().col(0);
16   Eigen::VectorXd sv = svd.singularValues();
17   // Scaling
18   z = first_U * std::sqrt(sv[0]);
19   // END
20   return z;
21 }
```

Get it on GitLab (symrank1.hpp).

There is no difference between economical and full SVD for a square matrix.

HINT 1 for (8-14.b): You can exploit that the Kronecker product is a bilinear operation and use the general product rule. ⌟

SOLUTION of (8-14.b):

① We can find the Jacobian of $\mathbf{\Phi}$ by a perturbation approach. Using that the Kronecker product is a bi-linear operation we get

$$\mathbf{\Phi}(\mathbf{x}+\mathbf{h}) - \mathbf{\Phi}(\mathbf{x}) = -\mathbf{x}\otimes\mathbf{h} - \mathbf{h}\otimes\mathbf{x} - \mathbf{h}\otimes\mathbf{h} \overset{!}{=} \mathsf{D}\,\mathbf{\Phi}(\mathbf{x})\mathbf{h} + O(\|\mathbf{h}\|^2) \,,$$

$$\blacktriangleright \qquad \mathsf{D}\,\mathbf{\Phi}(\mathbf{x})\mathbf{h} = -\mathbf{x}\otimes\mathbf{h} - \mathbf{h}\otimes\mathbf{x} \,. \tag{8.14.5}$$

② The same result follows from the general product rule [Lecture → Eq. (8.5.1.17)].

- Product rule: $F : D \subset V \mapsto W$, $G : D \subset V \mapsto U$ sufficiently smooth, $b : W \times U \mapsto Z$ *bilinear*, *ie.*, linear in each argument:

$$\begin{aligned} & T(\mathbf{x}) = b(F(\mathbf{x}), G(\mathbf{x})) \\ \Rightarrow \quad & \mathsf{D}\,T(\mathbf{x})\mathbf{h} = b(\mathsf{D}\,F(\mathbf{x})\mathbf{h}, G(\mathbf{x})) + b(F(\mathbf{x}), \mathsf{D}\,G(\mathbf{x})\mathbf{h}) \,, \\ & \mathbf{h} \in V, \mathbf{x} \in D \,. \end{aligned} \tag{8.14.6}$$

We apply it with

- $D = V = W = U = \mathbb{R}^n$,
- $F \equiv G = \mathsf{Id}$ (identity operators, for which $\mathsf{D}\,F = \mathsf{D}\,G = \mathsf{Id}$),
- $b(\mathbf{x},\mathbf{y}) = -\mathbf{x}\otimes\mathbf{y}$ and $Z = \mathbb{R}^{n^2}$,

such that (8.14.6) gives us

$$\mathsf{D}\,T(\mathbf{x})\mathbf{h} = b(\mathbf{h},\mathbf{x}) + b(\mathbf{x},\mathbf{h}) = -\mathbf{h}\otimes\mathbf{x} - \mathbf{x}\otimes\mathbf{h} \,.$$

HINT 1 for (8-14.c): You may use the following properties of the Kronecker product:

$$(\mathbf{A}\otimes\mathbf{B})(\mathbf{C}\otimes\mathbf{D}) = (\mathbf{AC})\otimes(\mathbf{BD})\,, \tag{8.14.7}$$

$$(\mathbf{A}\otimes\mathbf{B})^\top = \mathbf{A}^\top\otimes\mathbf{B}^\top\,, \tag{8.14.8}$$

$$\mathbf{x}\otimes\mathbf{y} = (\mathbf{x}\otimes\mathbf{I}_n)\mathbf{y} = (\mathbf{I}_n\otimes\mathbf{y})\mathbf{x}\,, \tag{8.14.9}$$

for all matrices $\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}$ of suitable sizes and vectors $\mathbf{x},\mathbf{y}\in\mathbb{R}^n$. ⌟

SOLUTION of (8-14.c):

① The most concise derivation starts from (8.14.5) and uses (8.14.9):

$$\begin{aligned} \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})\mathbf{h} &= -\mathbf{x}\otimes\mathbf{h} - \mathbf{h}\otimes\mathbf{x} \\ &= -(\mathbf{x}\otimes\mathbf{I}_n)([1]\otimes\mathbf{h}) - (\mathbf{I}_n\otimes\mathbf{x})(\mathbf{h}\otimes[1]) \\ &= -(\mathbf{x}\otimes\mathbf{I}_n + \mathbf{I}_n\otimes\mathbf{x})\mathbf{h}\ . \end{aligned}$$

The Jacobian is just the matrix representation of the linear mapping $\mathbf{h}\mapsto \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})\mathbf{h}$ in standard bases, so that

$$\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}) = -\mathbf{x}\otimes\mathbf{I}_n - \mathbf{I}_n\otimes\mathbf{x} \in \mathbb{R}^{n^2,n}\ . \tag{8.14.10}$$

Recalling the definition of the Kronecker product we can explicitly write

$$-\mathbf{x}\otimes\mathbf{I}_n - \mathbf{I}_n\otimes\mathbf{x} = -\begin{bmatrix} x_1 & & & & \\ & \ddots & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & x_1 \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ x_n & & & & \\ & \ddots & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & \ddots \\ & & & & x_n \end{bmatrix} - \begin{bmatrix} x_1 & & & \\ \vdots & & & \\ \vdots & & & \\ x_n & & & \\ & x_1 & & \\ & \vdots & & \\ & \vdots & & \\ & x_n & & \\ & & \vdots & \\ & & \vdots & \\ & & & x_1 \\ & & & \vdots \\ & & & \vdots \\ & & & x_n \end{bmatrix},$$

$$\blacktriangleright \quad \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)}) = -\begin{bmatrix} x_1 + x_1 & & & & & & \\ x_2 & \ddots & & & & & \\ \vdots & & \ddots & & & & \\ \vdots & & & \ddots & & & \\ x_{n-1} & & & & \ddots & & \\ x_n & & & & & x_1 & \\ \vdots & & & & \vdots & & \\ \vdots & & & & \vdots & & \\ x_n & & & & & x_1 & \\ & \ddots & & & & x_2 & \\ & & \ddots & & & \vdots & \\ & & & \ddots & & \vdots & \\ & & & & \ddots & x_{n-1} & \\ & & & & & x_n + x_n & \end{bmatrix} \in \mathbb{R}^{n^2,n}\,.$$

② We can do direct componentwise differentiation of $(x_i := (\mathbf{x})_i)$

$$\boldsymbol{\Phi}(\mathbf{x}) = \mathrm{vec}(\mathbf{M}) + \Big[x_1^2, x_1x_2, \ldots, x_1x_n, x_2x_1, x_2^2, x_2x_3, \\ \ldots, x_2x_n, x_3x_1, \ldots, \ldots, x_nx_1, \ldots, x_n^2\Big]^\top\,,$$

which yields, e.g.,

$$\frac{\partial \boldsymbol{\Phi}}{\partial x_1}(\mathbf{x}) = [2x_1, x_2, x_3, \ldots, x_n, x_2, 0, \ldots, 0, x_3, 0, \ldots, \ldots, x_n, 0, \ldots, 0]\,,$$
$$\frac{\partial \boldsymbol{\Phi}}{\partial x_2}(\mathbf{x}) = [0, x_1, 0, \ldots, 0, x_1, 2x_2, x_3, \ldots, x_n, 0, x_3, 0, \ldots, \ldots, 0, x_n, 0, \ldots, 0]\,,$$
$$\vdots$$

In the case $n = 3$ we obtain

$$\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)}) = -\begin{bmatrix} 2x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & 0 & x_1 \\ x_2 & x_1 & 0 \\ 0 & 2x_2 & 0 \\ 0 & x_3 & x_2 \\ x_3 & 0 & x_1 \\ 0 & x_3 & x_2 \\ 0 & 0 & 2x_3 \end{bmatrix}\,.$$

HINT 1 for (8-14.d): The system matrix $\mathbf{T}$ of the normal equations is a symmetric rank-1 matrix added to a scaled identity:

$$\mathbf{T} = \alpha\mathbf{I} + \mathbf{v}\mathbf{v}^\top , \quad \alpha \in \mathbb{R} , \quad \mathbf{v} \in \mathbb{R}^n , \tag{8.14.12}$$

α and $\mathbf{v}$ to be determined. ⌟

SOLUTION of (8-14.d):

The normal equations for (8.14.11) read

$$\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b} \quad \blacktriangleright \quad \mathbf{T} = \mathbf{A}^\top \mathbf{A} \,.$$

Here, we have $\mathbf{A} = \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})$ and, thus, we have to compute $\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})^\top \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})$ for $\mathbf{x} = \mathbf{x}^{(k)}$. For the sake of simplicity we write $\mathbf{x}$ in place of $\mathbf{x}^{(k)}$.

Approach ①: Use (8.14.10) together with (8.14.8) and, then, (8.14.9),

$$\begin{aligned}
\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})^\top \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}) &= (\mathbf{x}\otimes\mathbf{I}_n + \mathbf{I}_n \otimes \mathbf{x})^\top (\mathbf{x}\otimes\mathbf{I}_n + \mathbf{I}_n \otimes \mathbf{x}) \\
&= (\mathbf{x}\otimes\mathbf{I}_n)^\top (\mathbf{x}\otimes\mathbf{I}_n) + (\mathbf{x}\otimes\mathbf{I}_n)^\top (\mathbf{I}_n\otimes\mathbf{x}) + \\
&\quad (\mathbf{I}_n\otimes\mathbf{x})^\top (\mathbf{x}\otimes\mathbf{I}_n) + (\mathbf{I}_n\otimes\mathbf{x})^\top (\mathbf{I}_n\otimes\mathbf{x}) \\
&= (\mathbf{x}^\top\otimes\mathbf{I}_n)(\mathbf{x}\otimes\mathbf{I}_n) + (\mathbf{x}^\top\otimes\mathbf{I}_n)(\mathbf{I}_n\otimes\mathbf{x}) + \\
&\quad (\mathbf{I}_n\otimes\mathbf{x}^\top)(\mathbf{x}\otimes\mathbf{I}_n) + (\mathbf{I}_n\otimes\mathbf{x}^\top)(\mathbf{I}_n\otimes\mathbf{x}) \\
&= (\mathbf{x}^\top\mathbf{x})\otimes\mathbf{I}_n + \mathbf{x}^\top\otimes\mathbf{x} + \mathbf{x}^\top\otimes\mathbf{x} + \mathbf{I}_n\otimes(\mathbf{x}^\top\mathbf{x}) \\
&= \|\mathbf{x}\|^2\mathbf{I}_n + \mathbf{x}\mathbf{x}^\top + \mathbf{x}\mathbf{x}^\top + \|\mathbf{x}\|^2\mathbf{I}_n \,.
\end{aligned}$$

$$\blacktriangleright \quad \mathbf{T} = \boxed{2\|\mathbf{x}\|^2\mathbf{I}_n + 2\mathbf{x}\mathbf{x}^\top} \,.$$

Approach ②: From Sub-problem (8-14.b) we know that

$$\mathbf{A}\mathbf{h} = -(\mathbf{x}\otimes\mathbf{h} + \mathbf{h}\otimes\mathbf{x}) \,, \quad \mathbf{x} := \mathbf{x}^{(k)} \,, \quad \mathbf{h} \in \mathbb{R}^n \,.$$

The normal equations associated with (8.14.11) are [Lecture → Eq. (3.1.2.2)]

$$\mathbf{T}\mathbf{z} = \mathbf{A}^\top \mathbf{b} \quad \text{with} \quad \mathbf{T} := \mathbf{A}^\top \mathbf{A} \,.$$

The rules for manipulating the Kronecker product (8.14.7) and (8.14.8) give us

$$\begin{aligned}
\mathbf{h}^\top \mathbf{T}\,\mathbf{h} &= (\mathbf{A}\mathbf{h})^\top(\mathbf{A}\mathbf{h}) = (\mathbf{x}\otimes\mathbf{h} + \mathbf{h}\otimes\mathbf{x})^\top(\mathbf{x}\otimes\mathbf{h} + \mathbf{h}\otimes\mathbf{x}) \\
&= (\mathbf{x}^\top\otimes\mathbf{h}^\top)(\mathbf{x}\otimes\mathbf{h}) + (\mathbf{x}^\top\otimes\mathbf{h}^\top)(\mathbf{h}\otimes\mathbf{x}) + \\
&\quad (\mathbf{h}^\top\otimes\mathbf{x}^\top)(\mathbf{x}\otimes\mathbf{h}) + (\mathbf{h}^\top\otimes\mathbf{x}^\top)(\mathbf{h}\otimes\mathbf{x}) \\
&= (\mathbf{x}^\top\mathbf{x})(\mathbf{h}^\top\mathbf{h}) + (\mathbf{x}^\top\mathbf{h})(\mathbf{h}^\top\mathbf{x}) + (\mathbf{h}^\top\mathbf{x})(\mathbf{x}^\top\mathbf{h}) + (\mathbf{h}^\top\mathbf{h})(\mathbf{x}^\top\mathbf{x}) \\
&= 2(\mathbf{x}^\top\mathbf{x})\mathbf{h}^\top\mathbf{h} + 2\mathbf{h}^\top(\mathbf{x}\mathbf{x}^\top)\mathbf{h} \\
&= \mathbf{h}^\top\Big(\boxed{2\|\mathbf{x}\|^2\mathbf{I}_n + 2\mathbf{x}\mathbf{x}^\top}\Big)\mathbf{h} \quad \forall \mathbf{h} \in \mathbb{R}^n \,.
\end{aligned}$$

This establishes the highlighted formula for $\mathbf{T}$.

HINT 1 for (8-14.e): Exploit the Sherman-Morrison-Woodbury formula or the inverse of a matrix under low-rank modification:

Lemma [Lecture → Lemma 2.6.0.21]. Sherman-Morrison-Woodbury formula

For regular $\mathbf{A} \in \mathbb{K}^{n,n}$, *and* $\mathbf{U}, \mathbf{V} \in \mathbb{K}^{n,k}$, $n, k \in \mathbb{N}$, $k \leq n$, *holds*

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I}_k + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^H\mathbf{A}^{-1} ,$$

if $\mathbf{I}_k + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U}$ *is regular.*

HINT 2 for (8-14.e): If you failed to compute $\mathbf{T}$, you may use the general formula from (8.14.12). ⌟

SOLUTION of (8-14.e):

We apply the Sherman-Morrison-Woodbury formula [Lecture → Lemma 2.6.0.21] for $k = 1$ with $\mathbf{A} = 2\|\mathbf{x}\|^2$, $\mathbf{U} = 2\mathbf{x} \in \mathbb{R}^{n,1}$, $\mathbf{V} = \mathbf{x} \in \mathbb{R}^{n,1}$, and get

$$\mathbf{T}^{-1} = \frac{1}{2\|\mathbf{x}\|^2}\left(\mathbf{I} - \frac{1}{2}\frac{\mathbf{x}\mathbf{x}^\top}{\|\mathbf{x}\|^2}\right) . \tag{8.14.13}$$

Hence, solving a linear system of equations with system matrix $\mathbf{T}$ can be accomplished by carrying out a few simple vector operations.

HINT 1 for (8-14.f): The reshaping of vectors into matrices via EIGEN's **Eigen::Map** wrapper class may be useful, see [Lecture → Code 1.2.3.7]. ⌟

SOLUTION of (8-14.f):

The considerations follow [Lecture → Ex. 1.4.3.8].

Writing $v_j := (\mathbf{v})_j$, by the formulas for the Kronecker product we have

$$\mathbf{v}^\top \otimes \mathbf{I}_n = [v_1\mathbf{I}_n\ ,\ v_2\mathbf{I}_n\ ,\ \ldots\ ,\ v_n\mathbf{I}_n] \in \mathbb{R}^{n,n^2}\ ,$$
$$\mathbf{I}_n \otimes \mathbf{v}^\top = \begin{bmatrix} \mathbf{v}^\top & & & \\ & \mathbf{v}^\top & & \\ & & \mathbf{v}^\top & \\ & & & \mathbf{v}^\top \end{bmatrix} \in \mathbb{R}^{n,n^2}\ .$$

Next, chop up $\mathbf{b} \in \mathbb{R}^{n^2}$ into n sections of length n:

$$\mathbf{b} = \begin{bmatrix} \mathbf{b}^1 \\ \vdots \\ \mathbf{b}^n \end{bmatrix}\ ,\quad \mathbf{b}^\ell \in \mathbb{R}^n\ ,\quad \ell \in \{1,\ldots,n\}\ ,$$

and introduce the matrix with those parts as columns:

$$\mathbf{B} = \left[\mathbf{b}^1, \mathbf{b}^2, \ldots, \mathbf{b}^n\right] \in \mathbb{R}^{n,n}\ .$$

In the code this matrix can be formed by means of **Eigen::Map**.

By the rules of matrix×vector multiplication:

$$(\mathbf{v}^\top \otimes \mathbf{I}_n)\mathbf{b} = [v_1\mathbf{I}_n, v_2\mathbf{I}_n, \ldots, v_n\mathbf{I}_n] \begin{bmatrix} \mathbf{b}^1 \\ \vdots \\ \mathbf{b}^n \end{bmatrix} = v_1\mathbf{b}^1 + \cdots + v_n\mathbf{b}^n = \mathbf{B}\mathbf{v}\ ,$$
$$(\mathbf{I}_n \otimes \mathbf{v}^\top)\mathbf{b} = \begin{bmatrix} \mathbf{v}^\top & & & \\ & \mathbf{v}^\top & & \\ & & \mathbf{v}^\top & \\ & & & \mathbf{v}^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}^1 \\ \vdots \\ \mathbf{b}^n \end{bmatrix} = \begin{bmatrix} \mathbf{v}^\top\mathbf{b}^1 \\ \mathbf{v}^\top\mathbf{b}^2 \\ \vdots \\ \mathbf{v}^\top\mathbf{b}^n \end{bmatrix} = \mathbf{B}^\top\mathbf{v}\ .$$

In the implementation you should not create a temporary matrix to hold $\mathbf{B}$; rather use **Eigen::Map** twice.

C++ code 8.14.15: Sub-problem (8-14.f): The function `computeKronProdVecMult()`

```
2  Eigen::VectorXd computeKronProdVecMult(const Eigen::VectorXd &v,
3                                         const Eigen::VectorXd &b) {
4    const unsigned int n = v.size();
5    assert(b.size() == n * n && "Size mismatch");
6    // Vector to be used to return result
7    Eigen::VectorXd res = Eigen::VectorXd::Zero(n);
8    // TODO: (8-14.f)
9    //  evaluate the expression (8.14.14) efficiently with the aid of Eigen::map
10   // START
11   // Raw data array for argument vector b
12   const double *data = b.data();
13   // Twice reinterpretation of data, no copy
```

```
14    res = Eigen::MatrixXd::Map(data, n, n) * v +
15          Eigen::MatrixXd::Map(data, n, n).transpose() * v;
16    // END
17    return res;
18  }
```

Get it on GitLab (symrank1.hpp).

HINT 1 for (8-14.g): Take the cue from [Lecture → Code 8.7.2.2]. ⌟

HINT 2 for (8-14.g): It may be convenient to call `computeKronProdVecMult()` at one point. ⌟

HINT 3 for (8-14.g): **If and only if** you failed to find $\mathbf{T}^{-1}$ in Sub-problem (8-14.e) and the vector $\mathbf{b}$ of (8.14.11), then use

- $\mathbf{T}^{-1} = \mathrm{diag}(1, 2, \ldots, n)$,
- $\mathbf{b} = \mathrm{vec}(\mathbf{M})$,
- $\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}) = [x_1\mathbf{I}_n, x_2\mathbf{I}_n, \ldots, x_n\mathbf{I}_n]^\top$.

SOLUTION of (8-14.g):

Using (8.14.13) in the k-th step of the iteration, the solution of the normal equations is

$$\mathbf{s} = \frac{1}{2\|\mathbf{x}^{(k)}\|^2}\left(\mathbf{I} - \frac{1}{2}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|^2}\right) \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})^\top(\mathrm{vec}(\mathbf{M}) - \mathbf{x}^{(k)} \otimes \mathbf{x}^{(k)}) \,. \tag{8.14.16}$$

From the result of (8.14.10) and (8.14.7) we infer

$$\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})^\top = -\mathbf{x}^\top \otimes \mathbf{I}_n - \mathbf{I}_n \otimes \mathbf{x}^\top \quad \text{with} \quad \mathbf{x} := \mathbf{x}^{(k)} \,.$$

Thus, after you have computed $\mathbf{b} := \mathrm{vec}(\mathbf{M}) - \mathbf{x}^{(k)} \otimes \mathbf{x}^{(k)}$ you can call `computeKronProdVecMult()` to obtain $\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})^\top\mathbf{b}$. Everything else in (8.14.16) is straightforward.

The vector $\mathbf{s}$ then supplies the correction in the current Gauss-Newton step.

C++ code 8.14.17: Sub-problem (8-14.g): The function `symmRankOneApprox()`

```
Eigen::VectorXd symmRankOneApprox(const Eigen::MatrixXd &M,
                                  const double rtol = 1e-6,
                                  const double atol = 1e-8) {
  // ensure M is square
  assert(M.cols() == M.rows() && "Matrix must be square");

  const unsigned int n = M.cols();
  // Initial guess: Column of symmetric part of M with largest norm
  auto Msym = 0.5 * (M + M.transpose());
  Eigen::Index j, dummy;
  Msym.colwise().norm().maxCoeff(&dummy, &j);
  // Set initial guess. This vector is also used to return the result
  Eigen::VectorXd z{Msym.col(j)};
  // TODO: (8-14.g)
  //  solve equ. (8.14.1) using Gauss-Newton iteration with tolerances rtol and
  //  atol
  // START
  // Gauss-Newton correction vector
  Eigen::VectorXd s(n);
  // Direct access to the array storing the entries of M
  const double *dataM = M.data();

  std::cout << " Correction s to current newton step:\n";
  // Main Gauss-Newton loop with correction-based termination
  do {
    // Compute b = vec(M) - kron(z,z)
    const Eigen::MatrixXd Z = z * z.transpose();
    const double *dataz = Z.data();
    const Eigen::VectorXd b = Eigen::MatrixXd::Map(dataM, n * n, 1) -
                              Eigen::MatrixXd::Map(dataz, n * n, 1);
    // Solve normal equations
    const double normalize = 2 * z.squaredNorm();
    Eigen::VectorXd kronMult = computeKronProdVecMult(z, b);
    s = (kronMult - z.dot(kronMult) / normalize * z) / normalize;
    // Apply Gauss-Newton update
    z = z + s;
    std::cout << "|s| = " << s.norm() << std::endl;
  } while (s.norm() > std::max(atol, rtol * z.norm()));
```

```
40    std::cout << "\n";
41    // END
42    return z;
43  }
```

Get it on GitLab (symrank1.hpp).

Remark. Write tr for the **trace** of a square matrix, that is, the sum of its diagonal entries. Then

$$\mathbb{R}^{n,n} \times \mathbb{R}^{n,n} \to \mathbb{R} \quad , \quad (\mathbf{X}, \mathbf{Y}) \mapsto (\mathbf{X}, \mathbf{Y})_F := \operatorname{tr}(\mathbf{X}^\top \mathbf{Y}) \, ,$$

defines an *inner product* on the matrix space $\mathbb{R}^{n,n}$. The Frobenius norm is the associated norm! By straightforward computations we find

$$\operatorname{tr}(\mathbf{AB}) = \operatorname{tr}(\mathbf{BA}) \quad \forall \mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n} \, .$$

As a consequence, symmetric and skew-symmetric matrices are $(\cdot,\cdot)_F$-orthogonal

$$\mathbf{X} = \mathbf{X}^\top \, , \quad \mathbf{Y} = -\mathbf{Y}^\top \quad \Rightarrow \quad (\mathbf{X}, \mathbf{Y})_F = 0 \, ,$$

because

$$(\mathbf{X}, \mathbf{Y})_F = \operatorname{tr}(\mathbf{X}^\top \mathbf{Y}) = -\operatorname{tr}(\mathbf{X}\mathbf{Y}^\top) = -\operatorname{tr}(\mathbf{Y}^\top \mathbf{X}) = -(\mathbf{Y}, \mathbf{X})_F \, .$$

Thus, by Pythagoras' theorem we have

$$\begin{aligned} \left\| \mathbf{M} - \mathbf{x}\mathbf{x}^\top \right\|_F^2 &= \left\| \tfrac{1}{2}(\mathbf{M} + \mathbf{M}^\top) + \tfrac{1}{2}(\mathbf{M} - \mathbf{M}^\top) - \mathbf{x}\mathbf{x}^\top \right\|_F^2 \\ &= \left\| \tfrac{1}{2}(\mathbf{M} + \mathbf{M}^\top) - \mathbf{x}\mathbf{x}^\top \right\|_F^2 + \left\| \tfrac{1}{2}(\mathbf{M} - \mathbf{M}^\top) \right\|_F^2 \, . \end{aligned}$$

Hence the symmetric rank-1 best approximation of the symmetric part $\frac{1}{2}(\mathbf{M} + \mathbf{M}^\top)$ is the symmetric rank-1 best approximation also for the general matrix $\mathbf{M}$!

As a consequence the initial guess `symRankOneBestApproxSym`$(1/2(\mathbf{M} + \mathbf{M}^\top))$ is already the final solution in `symRankOneApprox()` and the Gauss-Newton iteration does have any effect! What a waste of time that we programmed it before plunging into deep mathematical considerations. Yet, this is how it usually works in computational mathematics: With hindsight one could have avoided a lot of coding by clever mathematical analysis, but only the computational results suggested that analysis.

SOLUTION of (8-15.a):

I) F can directly be inferred from (8.15.1). We have n equations with 4 unknowns.

$$F : \left\{ \begin{array}{ccc} \mathbb{R}^4 & \to & \mathbb{R}^n \\ \mathbf{x} := \begin{bmatrix} \mathbf{p}^* \\ t^* \end{bmatrix} & \mapsto & F(\mathbf{x}) := \left[\left\| \mathbf{q}^j - \mathbf{p}^* \right\|_2 - c(t_j - t^*) \right]_{j=1}^n \end{array} \right. \tag{8.15.2}$$

Of course, the unknowns in the vector $\mathbf{x}$ can be swapped.

II) Since $t_j > t^*$ an *equivalent* non-linear system of equations can be obtained by squaring (8.15.1):

$$\left\| \mathbf{q}^j - \mathbf{p}^* \right\|_2^2 = c^2 (t_j - t^*)^2 \,, \quad j = 1, \ldots, n \,. \tag{8.15.3}$$

In this case we would get

$$F : \left\{ \begin{array}{ccc} \mathbb{R}^4 & \to & \mathbb{R}^n \\ \mathbf{x} := \begin{bmatrix} \mathbf{p}^* \\ t^* \end{bmatrix} & \mapsto & F(\mathbf{x}) := \left[\left\| \mathbf{q}^j - \mathbf{p}^* \right\|_2^2 - c^2 (t_j - t^*)^2 \right]_{j=1}^n \end{array} \right. \tag{8.15.4}$$

SOLUTION of (8-15.b):

Let us write for the independent variables, *cf.* (8.15.2),

$$\mathbf{x} = \begin{bmatrix} \mathbf{p} \\ t \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} . \tag{8.15.5}$$

Of course, any other arrangement of independent variables is also possible.

The two approaches from Sub-problem (8-15.a), (8.15.2) and (8.15.4) lead to different Jacobians, of course.

I) From [Lecture → Ex. 8.5.1.20] we recall that the Euclidean norm function $g : \mathbb{R}^d \to \mathbb{R}$, $g(\mathbf{x}) := \|\mathbf{x}\|_2$, is differentiable for $\mathbf{x} \neq \mathbf{0}$ with derivative $\mathrm{D}\, g(\mathbf{x}) = \mathbf{x}^\top / \|\mathbf{x}\|_2$, see [Lecture → Eq. (8.5.1.21)].

Setting $\mathbf{q}^j = \left[q_1^j, q_2^j, q_3^j\right]$, we rewrite

$$\begin{aligned} F(\mathbf{x}) = F\left(\begin{bmatrix} \mathbf{p} \\ t \end{bmatrix}\right) &= \left[\left\|\mathbf{q}^j - \mathbf{p}\right\|_2 - c(t_j - t)\right]_{j=1}^n \\ &= \left[\sqrt{(q_1^j - x_1)^2 + (q_2^j - x_2)^2 + (q_3^j - x_3)^2} - c(t_j - x_4)\right]_{j=1}^n . \end{aligned}$$

We directly compute the columns of the Jacobian as partial derivatives of the vector $\mathbf{x} \mapsto F(\mathbf{x})$ with respect to x_i:

$$\frac{\partial F}{\partial x_i}(\mathbf{x}) = \left[\frac{x_i - q_i^j}{\sqrt{(q_1^j - x_1)^2 + (q_2^j - x_2)^2 + (q_3^j - x_3)^2}}\right]_{j=1}^n , \quad i = 1,2,3 \quad , \quad \frac{\partial F}{\partial x_4}(\mathbf{x}) = [c]_{j=1}^n .$$

This yields the Jacobian, here written in compact form

$$\mathrm{D}\, F\left(\begin{bmatrix} \mathbf{p} \\ t \end{bmatrix}\right) = \begin{bmatrix} -\dfrac{(\mathbf{q}^1 - \mathbf{p})^\top}{\|\mathbf{q}^1 - \mathbf{p}\|_2} & c \\ \vdots & \vdots \\ -\dfrac{(\mathbf{q}^n - \mathbf{p})^\top}{\|\mathbf{q}^n - \mathbf{p}\|_2} & c \end{bmatrix} \in \mathbb{R}^{n,4} . \tag{8.15.6}$$

It is defined for all $[\mathbf{p}, t]^\top \in \mathbb{R}^4$ with $\mathbf{p} \neq \mathbf{q}^j$ for all $j = 1, \ldots, n$.

II) In the case (8.15.4) we have

$$\begin{aligned} F(\mathbf{x}) = F\left(\begin{bmatrix} \mathbf{p} \\ t \end{bmatrix}\right) &= \left[\left\|\mathbf{q}^j - \mathbf{p}\right\|_2^2 - c^2(t_j - t)^2\right]_{j=1}^n \\ &= \left[(q_1^j - x_1)^2 + (q_2^j - x_2)^2 + (q_3^j - x_3)^2 - c^2(t_j - x_4)^2\right]_{j=1}^n . \end{aligned}$$

This expression is easily differentiated:

$$\frac{\partial F}{\partial x_i}(\mathbf{x}) = \left[-2(x_i - q_i^j)\right]_{j=1}^n , \quad i = 1,2,3 \quad , \quad \frac{\partial F}{\partial x_4}(\mathbf{x}) = \left[2c^2(t_j - x_4)\right]_{j=1}^n .$$

Returning to the original independent variables, we end up with the Jacobian matrix

$$\mathrm{D}\,F\left(\begin{bmatrix}\mathbf{p}\\ t\end{bmatrix}\right) = 2\begin{bmatrix}-(\mathbf{q}^1-\mathbf{p})^\top & c^2(t_1-t)\\ \vdots & \vdots\\ -(\mathbf{q}^n-\mathbf{p})^\top & c^2(t_j-t)\end{bmatrix} \in \mathbb{R}^{n,4}\,, \tag{8.15.7}$$

with is defined for all $[\mathbf{p},t]^\top \in \mathbb{R}^4$.

SOLUTION of (8-15.c):

You may just use a copy of the function provided in [Lecture → Code 8.7.2.2]:

C++ code 8.15.8: A templated function implementing a generic Gauss-Newton iteration

```
2  template <typename VECTOR, typename FUNCTION, typename JACOBIAN,
3            typename RECORDER =
4                std::function<void(const VECTOR &, const VECTOR &, double)>>
5  VECTOR gn(
6      const VECTOR &init, FUNCTION &&F, JACOBIAN &&J, double rtol = 1.0E-6,
7      double atol = 1.0E-8,
8      RECORDER &&rec = [](const VECTOR &, const VECTOR &, double) -> void {}) {
9    VECTOR x = init;  // Vector for iterates x^(k)
10   // Vector for Gauss-Newton correction s
11   auto f{F(x)};  // Auxiliary vector in the image space of F
12   VECTOR s = J(x).householderQr().solve(f);  //
13   rec(x, s, f.norm());
14   x = x - s;  // Update of iterate
15   // A posteriori termination based on absolute and relative tolerances
16   while ((s.norm() > rtol * x.norm()) &&
17          (s.norm() > atol)) {  //
18     f = F(x);
19     s = J(x).householderQr().solve(f);  //
20     rec(x, s, f.norm());                // Possibly record iteration
         information
21     x = x - s;                          // Update of iterate
22   }
23   return x;
24 }
```

Get it on GitLab (locationestimation.hpp).

Note that this codes uses EIGEN's built-in solver for linear least squares problems in Line 12 and Line 19, *cf.* [Lecture → Code 3.3.4.2]. An alternative would be to use the normal equations

$$\mathrm{D}F(\mathbf{x}^{(k)})^\top \mathrm{D}F(\mathbf{x}^{(k)})\mathbf{s} = \mathrm{D}F(\mathbf{x}^{(k)})^\top F(\mathbf{x}^{(k)}) ,$$

but there is no benefit in doing so.

This version of `gn()` also takes an optional recorder object as an argument to enable inspection of the history of the iteration. In each step that recorder object is given the current iterate $\mathbf{x}^{(k)}$, the Gauss-Newton update $\mathbf{s}$, and $\left\|F(\mathbf{x}^{(k)})\right\|$.

The next code calls `gn()` with suitable lambda functions `F` and `J` providing functors for F and $\mathrm{D}F$, respectively. The initial guesses are set according to the specifications of the sub-problem.

C++ code 8.15.9: Main driver function for source estimation

```
2  template <typename RECORDER = std::function<
3                void(const Eigen::Vector4d &, const Eigen::Vector4d &, double)>>
4  std::pair<Eigen::Vector3d, double> source_estimation(
5      const Eigen::Matrix<double, 3, Eigen::Dynamic> &Q,
6      const Eigen::VectorXd &ta,
7      RECORDER &&rec = [](const Eigen::Vector4d &, const Eigen::Vector4d &,
8                          double) -> void {}) {
```

```
  const unsigned int n = ta.size();
  assert(Q.cols() == n);
  // return variables
  Eigen::Vector3d p = Eigen::VectorXd::Zero(3);
  double t = 0.;
  // TODO: (8-15.c) Compute the source locations and emission times
     using
  // Gauss-Newton.
  // START

  // Lambda function realizing the function F from (8.15.2)
  auto F = [&Q, &ta](const Eigen::Vector4d &x) -> Eigen::VectorXd {
    const unsigned int n = ta.size();
    Eigen::VectorXd result(n);
    for (unsigned int j = 0; j < n; ++j) {
      result[j] = (Q.col(j) - x.head(3)).norm() - ta[j] + x[3];
    }
    return result;
  };
  // Lambda function for Jacobian DF according to (8.15.6)
  auto J = [&Q, &ta](const Eigen::Vector4d &x) -> Eigen::MatrixXd {
    const unsigned int n = ta.size();
    Eigen::MatrixXd Jac(n, 4);
    for (unsigned int j = 0; j < n; ++j) {
      const Eigen::Vector3d qp{Q.col(j) - x.head(3)};
      Jac.block(j, 0, 1, 3) = -qp.transpose() / qp.norm();
      Jac(j, 3) = 1.0;
    }
    return Jac;
  };
  // Compute initial guess for source position
  // as center of gravity of receiver locations
  Eigen::Vector3d cnt = Q.rowwise().sum() / n;
  // Initial guess t_0 for starting time of source signal:
  Eigen::VectorXd::Index minpos;
  const double min_ta = ta.minCoeff(&minpos);
  const double t_0 = min_ta - (Q.col(minpos) - cnt).norm();

  // Build vector for intial guess
  Eigen::Vector4d init = (Eigen::Vector4d() << cnt, t_0).finished();

  // Gauss-Newton method with correction-based termination
  constexpr double rtol = 1.0E-6;
  constexpr double atol = 1.0E-8;
  Eigen::Vector4d xs = gn(init, F, J, rtol, atol, rec);
  p = xs.head(3);
  t = xs[3];
  // END
  return {p, t};
}
```

Get it on GitLab (locationestimation.hpp).

Of course the function `gn()` need not be used and the Gauss-Newton loop can directly be integrated into `source_estimation()`.

Note that implementations with recorder objects have not been requested. The provided templates just supply the simpler interfaces.

SOLUTION of (8-15.d):

The key quantity of examine is the morm of the iteration error $\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|_2$. We roughly observe a doubling of the number of correct digits in iteration steps #1 through #5. This is evidence of quadratic convergence. In the last step the gain in accuracy is not what quadratic convergence would predict, because we have already reached an error of magnitude $\approx$ EPS, EPS $\approx 10^{-16}$ the machine precision for **double**.

Following [Lecture → Rem. 8.2.2.12] we can also approach the question more quantitatively. We write $\epsilon_k := \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|_2$ and augment the table

k	$\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|_2$	$\|\mathbf{s}\|_2$	$\left\|F(\mathbf{x}^{(k)})\right\|$	$(\log\epsilon_k - \log\epsilon_{k-1}) : (\log\epsilon_{k-1} - \log\epsilon_{k-2})$
1	1.021791e+00	1.135326e+00	3.442935e+00	
2	2.588111e-01	2.527947e-01	6.041095e-01	
3	7.550346e-03	7.504712e-03	1.559289e-02	2.5739
4	6.192137e-05	6.192055e-05	9.969019e-05	1.3590
5	9.310988e-10	9.310988e-10	1.862087e-09	2.3199
6	1.240973e-16	7.295652e-17	3.159274e-16	1.4256

The bottom number in the fourth column should be discarded due to the impact of round-off errors. The estimates for the order of convergence fluctuate wildly (around 2). Something similar we have see in [Lecture → Ex. 8.2.2.13].

HINT 1 for (8-16.a): Rewrite (8.16.1) using that $\cos(\xi) = \operatorname{Re}\exp(-\imath\xi)$ for all $\xi \in \mathbb{R}$. ⌟

SOLUTION of (8-16.a):

Since $x_j \in \mathbb{R}$ as indicated in the hint the formula (8.16.1) can be converted into

$$u_N(t) = \mathrm{Re}\left\{ \sum_{j=0}^{N} x_j e^{2\pi \imath j t} \right\}, \quad t \in \mathbb{R},$$

which means

$$u_N(\frac{k}{M}) = \mathrm{Re}\left\{ \sum_{j=0}^{N} x_j e^{-2\pi \imath j \frac{kj}{M}} \right\}, \quad k = 0, \ldots, M-1. \tag{8.16.3}$$

Next recall the discrete Fourier transform from [Lecture → Def. 4.2.1.18]:

Definition [Lecture → Def. 4.2.1.18]. Discrete Fourier transform

The linear map $\mathrm{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathrm{DFT}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called **discrete Fourier transform** (DFT), i.e. for $[c_0, \ldots, c_{n-1}] := \mathrm{DFT}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} = \sum_{j=0}^{n-1} y_j \exp\left(-2\pi \imath \frac{kj}{n}\right), \quad k = 0, \ldots, n-1. \qquad [\text{Lecture} \rightarrow \text{Eq. (4.2.1.19)}]$$

Inspired by (8.16.3) we aim to apply it for $n = M$. Since $M > N$ this entails zero-padding of the vectors of coefficients. We define

$$\widetilde{x}_j := \begin{cases} x_j & \text{for } j \in \{0, \ldots, N\}, \\ 0 & \text{for } j \in \{N+1, \ldots, M-1\}. \end{cases}, \qquad j = 0, \ldots, M-1,$$

and collect these numbers into the vector $\widetilde{\mathbf{x}} \in \mathbb{R}^M$. Then (8.16.3) can be recast as

$$\left[u_N(\frac{k}{M}) \right]_{k=0}^{M-1} = \mathrm{Re}\{\mathrm{DFT}_M(\widetilde{\mathbf{x}})\}.$$

The implementation based on EIGEN's built-in FFT-based DFT facilities [Lecture → § 4.2.1.21] is straightforward.

C++ code 8.16.4: Evaluation of trigonometric sum (8.16.1) at M equidistant points

```
2  Eigen::VectorXd eval_uN(const Eigen::VectorXd &x, unsigned int M) {
3    const unsigned int N = x.size() - 1;
4    assert((M > N) && "Number of evaluation points must be > N");
5    Eigen::VectorXd u = Eigen::VectorXd::Zero(M);
6    // TODO: (8-16.a) Evaluate uN efficiently, i.e. with complexity O(M
       log M)
7    // START
8    // Zero-pad coefficient vector and copy it into a complex-valued
       vector
9    Eigen::VectorXcd xt = Eigen::VectorXcd::Zero(M);
10   xt.head(N + 1) = x;
11   // Compute DFT of the vector xt
12   Eigen::FFT<double> fft;
```

```
13    Eigen::VectorXcd y = fft.fwd(xt);
14    u = y.real();
15    // END
16    return u;
17  }
```

Get it on GitLab (periodiccollocation.hpp).

Note that we have to introduce vectors of type **Eigen::VectorXcd**, because EIGEN's FFT only operates on those.

HINT 1 for (8-16.b): Remember the derivatives $\cos' = -\sin$ and $\sin' = \cos$. ⌟

SOLUTION of (8-16.b):

By the chain rule we find for any $\alpha \in \mathbb{R}$

$$\frac{d^2}{dt^2}\{t \mapsto \cos(2\pi j t)\}(t) = -(2\pi j)^2 \cos(2\pi j t)\ .$$

Using this, it is straightforward from (8.16.2) that

$$n = N+1\,, \quad \mathbf{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_N \end{bmatrix}\,,$$

$$F(\mathbf{x}) = \left[\sum_{j=0}^{N} x_j \cdot 4\pi^2 j^2 \cos\left(2\pi \tfrac{jk}{N+1}\right) + \left(\sum_{j=0}^{N} x_j \cdot \cos\left(2\pi \tfrac{jk}{N+1}\right)\right)^3 - \sin\left(\pi \tfrac{k}{N+1}\right)\right]_{k=0}^{N}\ . \qquad (8.16.5)$$

SOLUTION of (8-16.c):

Given $\mathbf{x} = [x_0, \dots, x_N]^\top \in \mathbb{R}^{N+1}$ we define

$$\begin{aligned} \mathbf{d} &:= \left[4\pi^2 j^2 x_j\right]_{j=0}^{N} \in \mathbb{R}^{N+1}\,, \\ \mathbf{y} &:= \texttt{eval\_uN}(\mathbf{d}, N+1)\,, \\ \mathbf{u} &:= \texttt{eval\_uN}(\mathbf{x}, N+1)\,. \end{aligned}$$

Then, from (8.16.5) it is immediate that

$$F(\mathbf{x}) = \mathbf{y} + \left[(\mathbf{u})_k^3\right]_{k=0}^{N} - \left[\sin(\pi \tfrac{k}{N+1})\right]_{k=0}^{N}\,.$$

This can be translated into C++ directly.

C++ code 8.16.6: Evaluation of F, with $F(\mathbf{x}) = \mathbf{0} \mathrel{\hat{=}}$ collocation equations (8.16.2)

```
Eigen::VectorXd eval_F(const Eigen::VectorXd &x) {
  const unsigned int N = x.size() - 1;
  Eigen::VectorXd Fx = Eigen::VectorXd::Zero(N + 1);
  // TODO: (8-16.c) Evaluate F at x.
  // START
  Eigen::ArrayXd s = Eigen::ArrayXd::LinSpaced(N + 1, 0, N);
  // Contribution [sin(π k/(N+1))]_k
  const Eigen::VectorXd rhs = ((M_PI / (N + 1)) * s).sin().matrix();
  // Vector [x_j · 4π²j²]_j
  s = 4 * M_PI * M_PI * s * s;
  const Eigen::VectorXd d{(x.array() * s).matrix()};
  Fx = eval_uN(d, N + 1) + (eval_uN(x, N + 1).array().pow(3)).matrix() - rhs;
  // END
  return Fx;
}
```

Get it on GitLab (periodiccollocation.hpp).

Of course, this is only one of many ways how this function can be implemented.

HINT 1 for (8-16.d): The function `eval_uN()` from Sub-problem (8-16.a) may be used. ⌟

SOLUTION of (8-16.d):

For the function $F(\mathbf{x}) = [F_0(\mathbf{x}), \ldots, F_N(\mathbf{x})]^\top$ encoding (8.16.2) and found in Sub-problem (8-16.b) the Jacobian $\mathrm{D}\,F(\mathbf{x}) \in \mathbb{R}^{N+1,N+1}$ for $\mathbf{x} \in \mathbb{R}^{N+1}$ is the matrix $\left[\frac{\partial F_k}{\partial x_j}(\mathbf{x})\right]_{k,j=0}^{N}$.

From (8.16.5) we directly see, $\mathbf{x} = [x_0, \ldots, x_N]$,

$$F_k(\mathbf{x}) = \sum_{\ell=0}^{N} x_\ell \cdot 4\pi^2\ell^2 \cos(2\pi\ell\tfrac{k}{N+1}) + \left(\sum_{\ell=0}^{N} x_\ell \cdot \cos(2\pi\ell\tfrac{k}{N+1})\right)^3 - \sin(\pi\tfrac{k}{N+1}) \,.$$

Forming the partial derivative with respect to x_j is straightforward using the chain rule on the second summand:

$$\frac{\partial F_k}{\partial x_j}(\mathbf{x}) = 4\pi^2 j^2 \cos(2\pi j\tfrac{k}{N+1}) + 3\Bigl(\underbrace{\sum_{\ell=0}^{N} x_\ell \cdot \cos(2\pi\ell\tfrac{k}{N+1})}_{=u_N(\frac{k}{N+1})}\Bigr)^2 \cos(2\pi j\tfrac{k}{N+1}) \,.$$

C++ code 8.16.7: Evaluation of the Jacobian $\mathrm{D}\,F(\mathbf{x})$

```
Eigen::MatrixXd eval_DF(const Eigen::VectorXd &x) {
  const unsigned int N = x.size() - 1;
  const Eigen::VectorXd u{eval_uN(x, N + 1)};
  Eigen::MatrixXd J = Eigen::MatrixXd::Zero(N + 1, N + 1);
  constexpr double fourpisq = 4 * M_PI * M_PI;
  // TODO: (8-16.d) Evaluate the Jacobian of F at x.
  // START
  for (unsigned int k = 0; k <= N; ++k) {
    const double fac = (2 * M_PI * k) / static_cast<double>(N + 1);
    const double uNksq = 3.0 * u[k] * u[k];
    for (unsigned int j = 0; j <= N; ++j) {
      J(k, j) = (fourpisq * j * j + uNksq) * std::cos(fac * j);
    }
  }
  // END
  return J;
}
/* SAM_LISTING_BEGIN_4 */

#endif
```

Get it on GitLab (periodiccollocation.hpp).

SOLUTION of (8-17.a):

Definition [Lecture → Def. 8.2.2.10]. Order of convergence

A *convergent* sequence $\mathbf{x}^{(k)}$, $k = 0, 1, 2, \ldots$, in $\mathbb{R}^n$ with limit $\mathbf{x}^* \in \mathbb{R}^n$ converges with **order** p, $p \geq 1$, if

$$\exists C > 0: \quad \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| \leq C \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|^p \quad \forall k \in \mathbb{N}_0 \,,$$

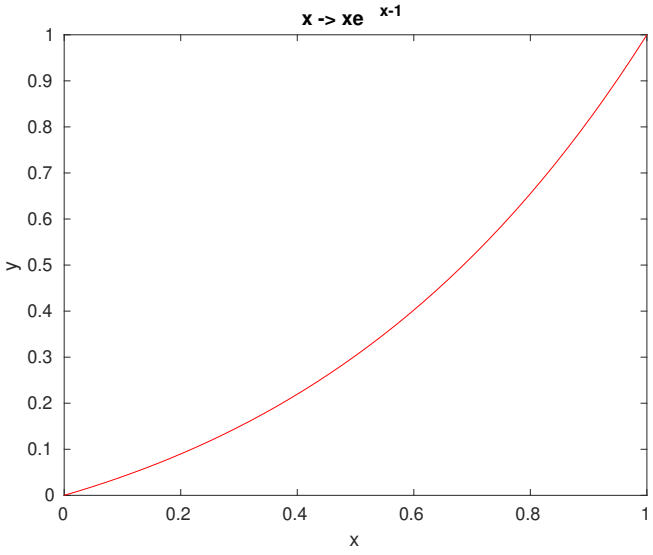
and, in addition, $C < 1$ in the case $p = 1$.

SOLUTION of (8-17.b):

See [Lecture → Code 8.5.3.6] for the solution.

```
template <typename FuncType, typename JacType, typename VecType>
void newton(const FuncType &F, const JacType &DF, VecType &x,
            double rtol, double atol) {
  double sn;
  do {
    auto jacfac = DF(x).lu(); // LU-factorize Jacobian ]
    x -= jacfac.solve(F(x));  // Compute next iterate
    // Compute norm of simplified Newton correction
    sn = jacfac.solve(F(x)).norm();
  }
  // Termination based on simplified Newton correction
  while ((sn > rtol * x.norm()) && (sn > atol));
}
```

SOLUTION of (8-17.c):



The domain of definition of g agrees with the domain of values of f, which is $[0,1]$. Since $f(g(y)) = y$ we know that $x := g(y) \in \mathbb{R}$ solves

$$F(x) = 0 \quad \text{with} \quad F(\xi) := f(\xi) - y \,.$$

With the derivative (y is just a constant!)

$$F'(\xi) = \xi e^{\xi-1} + e^{\xi-1} = (\xi+1)e^{\xi-1} > 0\,, \quad \xi \in [0,1]\,,$$

we arrive at the Newton iteration

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} = x^{(k)} - \frac{x^{(k)}e^{x^{(k)}-1} - y}{(x^{(k)}+1)e^{x^{(k)}-1}}\,, \quad k \in \mathbb{N}_0\,.$$

HINT 1 for (8-2.a): First try to understand what is accomplished in Lines 5–12 of this code. Recall the properties of logarithms. ⌟

HINT 2 for (8-2.a): Line 15–18 realize the Newton iteration for finding a zero of a real valued function. ⌟

SOLUTION of (8-2.a):

The C++ code computes $y = \log(x)$ for a given x. The loop in lines 15–18 realizes the Newton iterations for finding the zero of the function

$$f(z) = e^z - x\,, \quad z \in \mathbb{R}\,, \tag{8.2.2}$$

where z is the independent variable and x is given. We compute $f'(z) = e^z$ and obtain the Newton iteration

$$z^{(k+1)} = z^{(k)} - \frac{\exp(z^{(k)}) - x}{\exp(z^{(k)})} = z^{(k)} - s\,, \quad s = 1 - \frac{x}{\exp(z^{(k)})}\,. \tag{8.2.3}$$

C++ code 8.2.4: The documented function

```
2  double myfunction(double x) {
3    // We require that x > 0.
4    assert(x > 0);
5    // First we factorize x as x = 2^{n+r} = 2^n * x_new
6    // with integer n and r ∈ [0.5, 1.5].
7    // Set y = n log(2) and z = log(x_new),
8    // then log(x) = y + z.
9    constexpr double dy = 0.693147180559945;  // = std::log(2)
10   double y = 0.;
11   // If x x = 2^{n+r} with positive n:
12   while (x > 2. * std::sqrt(2.)) {
13     x /= 2.;
14     y += dy;
15   }  //
16   // If x x = 2^{n+r} with negative n:
17   while (x < std::sqrt(2.)) {
18     x *= 2.;
19     y -= dy;
20   }  //
21   // Now, we have found y, and we use Newton iteration for the
22   // function f(z) = exp(z) - x_new to find z = log(x_new).
23   // Initial guess:
24   double z = x - 1.;  //
25   // The update is dz = -f(z)/f'(z)
26   double dz = x * std::exp(-z) - 1.;
27   while (std::abs(dz / z) > std::numeric_limits<double>::epsilon()) {
28     z += dz;
29     dz = x * std::exp(-z) - 1.;
30   }
31   return y + z + dz;  //
32 }
```

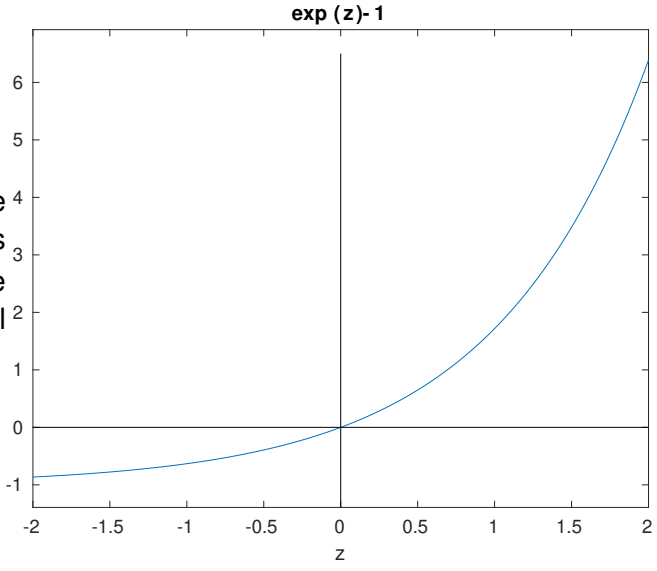
Get it on GitLab (codequiz.hpp).

SOLUTION of (8-2.b):

The **while** loops rely 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 $x \in [\sqrt{2}, 2\sqrt{2}]$.



The rationale for this can be seen by looking at the graph of f as defined in (8.2.2). 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.

SOLUTION of (8-2.c):

The `while`-loop computes the zero of the function $z \mapsto f(z)$ from (8.2.2) using Newton iterations, see also (8.2.3):

$$\begin{cases} z^{(0)} & := & x - 1\,, \\ z^{(n+1)} & := & z^{(n)} - \frac{e^{z^{(n)}} - x}{e^{z^{(n)}}}\,, \quad n \geq 1 \end{cases} .$$

SOLUTION of (8-2.d):

This is a correction based termination criterium, see [Lecture → § 8.2.3.4]. It is meant to ensure that the relative error of the final iterate is of the size of the machine precision EPS, because in C++ EPS for a floating point data type **T** is available as **std**`::numeric_limits<T>::epsilon()`.

HINT 1 for (8-2.e): Using Taylor expansion and the fixed-point form of the Newton iteration as in [Lecture → § 8.4.2.5], establish a recursion for the iteration errors $e^{(n)} := z^{(n)} - z^*$. 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 (8-2.e):

Denote the zero of $f(z) := e^z - x$ with z^*, write for the derivative $f'(z) = e^z$, and set $e^{(n)} := z^{(n)} - z^*$ for the iteration errors.



As in [Lecture → § 8.4.2.5] and the proof of [Lecture → Lemma 8.3.2.15] we

- rewrite the Newton iteration as a fixed-point iteration,
- use a Taylor expansion of its iteration function around z^*,
- and drop a powers $\left(e^{(n)}\right)^p$ for $p > 2$ (asymptotic perspective).

As in [Lecture → Eq. (8.4.2.6)] the Newton iteration can be rewritten as

$$z^{(k+1)} = \Phi(z^{(k)}) \quad \text{with} \quad \Phi(z) = z - \frac{f(z)}{f'(z)} \,. \tag{8.2.5}$$

As in [Lecture → Eq. (8.4.2.7)] we find by the quotient rule

$$\frac{d\Phi}{dz}(z) = 1 - \frac{\left(f'(z)\right)^2 - f(z)f''(z)}{\left(f'(z)\right)^2} = \frac{f(z)f''(z)}{\left(f'(z)\right)^2} \,.$$

Plugging in the concrete function f with $f(z) = e^z - x$, $f'(z) = e^z$, $f''(z) = e^z$, we get

$$\Phi'(z) := \frac{d\Phi}{dz}(z) = \frac{(e^z - x)e^z}{\left(e^z\right)^2} = 1 - xe^{-z} \,,$$

and by straightforward differentiation

$$\Phi''(z) := \frac{d^2\Phi}{dz^2}(z) = xe^{-z} \,.$$

For the iteration error of a fixed-point iteration like (8.2.5) we have

$$e^{(k+1)} = z^{(k+1)} - z^* = \Phi(z^{(k)}) - \Phi(z^*) = \Phi(z^* + e^{(k)}) - \Phi(z^*) \,. \tag{8.2.6}$$

Recall the Taylor expansion [Lecture → Eq. (1.5.4.28)] of a function $g \in C^3([a,b])$ around $t^* \in]a,b[$:

$$g(t^* + h) = g(t^*) + g'(t^*)h + \tfrac{1}{2}g''(t^*)h^2 + O(h^3) \quad \text{for} \quad h \to 0 \,. \tag{8.2.7}$$

Apply this formula to the right side of (8.2.6) with $g := \Phi$, $t^* := z^*$, $h := e^{(k)}$:

$$e^{(k+1)} = \Phi'(z^*)e^{(k)} + \tfrac{1}{2}\Phi''(z^*)\left(e^{(k)}\right)^2 + O\Big(\left(e^{(k)}\right)^3\Big) \quad \text{for} \quad |e^{(k)}| \approx 0 \,.$$

Since $e^{z^*} = x$, we immediately compute

$$\Phi'(z^*) = 0 \quad , \quad \Phi''(z^*) = 1 \,,$$

which implies

$$e^{(k+1)} = \tfrac{1}{2}\left(e^{(k)}\right)^2 + O\Big(\left(e^{(k)}\right)^3\Big) \,.$$

If we ignore higher powers of the iteration error, which is justified if the initial guess is already close to the solution,

$$|e^{(n+1)}| \doteq \frac{1}{2}|e^{(n)}|^2 \doteq \cdots \doteq (\frac{1}{2})^{1+2+\cdots+2^n}(e^{(0)})^{2^{n+1}} \doteq 2\cdot(\frac{1}{2})^{2^{n+1}}(e^{(0)})^{2^{n+1}} \doteq 2\cdot\left(\frac{1}{2}|e^{(0)}|\right)^{2^{n+1}},$$

where $\doteq$ expresses equality "up to terms of size $O\left((e^{(k)})^3\right)$". Further, the initial error is

$$e^{(0)} = z^{(0)} - z^* = x - 1 - \log(x), \quad x \in \left[\sqrt{2}, 2\sqrt{2}\right].$$

By discussing the simple function $x \mapsto x - 1 - \log(x)$ we find that the initial error $e^{(0)}$ attains its maximum value for $x = 2\sqrt{2}$:

$$|e^{(0)}| < \epsilon_{\max} := 2\sqrt{2} - 1 - \log(2\sqrt{2}) \approx 0.7887.$$

We can determine the minimal required number n of iteration steps by finding the smallest integer satisfying

$$\left|\frac{e^{(n)}}{\log x}\right| \leq \text{EPS}.$$

We are on the safe side, when replacing $|\log(x)|$ with its smallest possible value $\frac{1}{2}\log 2$, as $x \in \left[\sqrt{2}, 2\sqrt{2}\right]$. This leads to the requirement

$$|e^{(n)}| \leq \tfrac{1}{2}\log 2 \ \text{EPS}.$$

Then we can plug in the expression for $e^{(n)}$ found above:

$$2\left(\tfrac{1}{2}e^{(0)}\right)^{2^n} \leq \tfrac{1}{2}\log 2 \ \text{EPS}.$$

Next, we replace $e^{(0)}$ with its upper bound and get

$$\left(\tfrac{1}{2}\epsilon_{\max}\right)^{2^n} \leq \tfrac{1}{4}\log 2 \ \text{EPS},$$

from which we conclude that we need at least

$$n_{\min} := \left\lceil \log_2 \left(\frac{-\log\left(\frac{1}{4}\log 2 \ \text{EPS}\right)}{-\log(\frac{1}{2}\epsilon_{\max})} \right) \right\rceil$$

steps, where $\lceil \xi \rceil$ means the smallest integer $\geq \xi$.

The code in Code 8.2.8 demonstrates the use of this prediction for the number of required Newton iterations.

C++ code 8.2.8: Modified `myfunction`.

```
2 double myfunction_modified(double x) {
3   constexpr double dy = 0.693147180559945;  // = std::log(2)
```

```
  double y = 0.;
  while (x > 2. * std::sqrt(2.)) {
    x /= 2.;
    y += dy;
  }
  while (x < std::sqrt(2.)) {
    x *= 2.;
    y -= dy;
  }
  double z = x - 1.;
  double dz = x * std::exp(-z) - 1.;
  // TODO: (8-2.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
  const double e_max = 2. * std::sqrt(2.) - 1. - std::log(2. * std::sqrt(2.));
  constexpr double eps = std::numeric_limits<double>::epsilon();

  const int k_min = std::ceil(log2((-std::log(0.25 * std::log(2.) * eps)) /
                                   (-std::log(0.5 * std::abs(e_max)))));

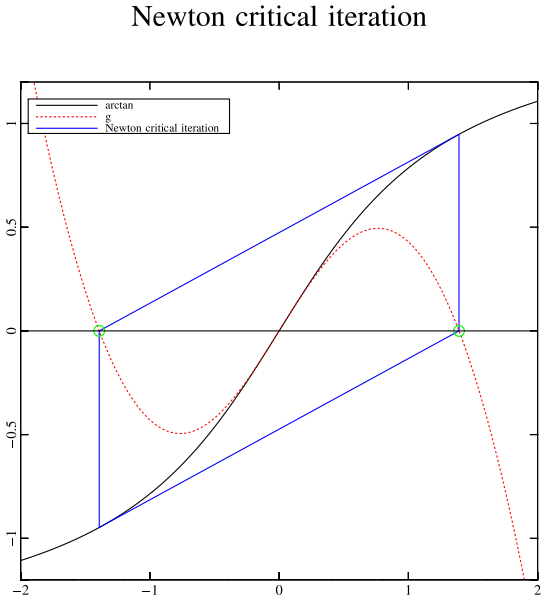
  for (int i = 1; i < k_min; ++i) {
    z = z + dz;
    dz = x * std::exp(-z) - 1.;
  }
  // END
  return y + z + dz;
}
```

Get it on GitLab (codequiz.hpp).

HINT 1 for (8-3.a): Refer to [Lecture → Fig. 305] and find out for which initial guesses the Newton method oscillates between two values. ⌟

HINT 2 for (8-3.a):

Clearly the solution is $x^* = 0$.
Graphical considerations may help you to find the solution, see Fig. 90. 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 $x^{(0)}$.



SOLUTION of (8-3.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

$$g(x) := 2x - (1 + x^2) \arctan x \quad \text{with} \quad g'(x) = 1 - 2x \arctan x \ .$$

SOLUTION of (8-3.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 8.3.1: Solution of Sub-problem (8-3.b)

```
2  double newton_arctan(double x0_ = 2) {
3    // TODO: (9-3.c) define a suitable Newton iteration to find the
4    // critical value x0 for F(x) = arctan(x).
5    double x0 = x0_;
6    // START
7    double upd = 1;
8    constexpr double eps = std::numeric_limits<double>::epsilon();
9
10   while (upd > eps) {
11     // Newton iteration for g(x)
12     double x1 =
13         (-x0 + (1 - x0 * x0) * std::atan(x0)) / (1 - 2 * x0 * std::atan(x0));
14     // Relative size of the Newton update
15     upd = std::abs((x1 - x0) / x1);
16     x0 = x1;
17   }
18   // END
19   return x0;
20 }
```

Get it on GitLab (newtarctan.hpp).

SOLUTION of (8-4.a):

The implementation of (8.4.1) is straightforward. The finite-precision-based stopping rule can be implemented relying on the actual machine precision or by comparing two successive iterates.

C++-code 8.4.2: Solution of Sub-problem (8-4.a)

```
2  template <class Function>
3  double steffensen(Function &&f, double x0) {
4    double x_old = x0;
5    double x = x0;
6
7    // TODO: (8-4.a) Implement the Steffensen's method for a function f.
8    // START
9
10   double upd = 1;
11
12   // Get machine precision
13   const double eps = std::numeric_limits<double>::epsilon();
14
15   // Iterate until machine precision is reached
16   while (std::abs(upd) > eps * std::abs(x_old)) {
17     // Storing the old iterate
18     x_old = x;
19
20     // Only 2 evaluations of f at each step
21     const double fx = f(x);
22
23     if (fx != 0) {
24       // Compute update
25       upd = fx * fx / (f(x + fx) - fx);
26
27       // New iterate
28       x -= upd;
29     } else {
30       upd = 0;
31     }
32   }
33
34   // END
35
36   return x;
37 }
```

Get it on GitLab (quadraticconvergence.hpp).

SOLUTION of (8-4.b):

C++ code 8.4.3: Testing function for Steffensen's method

```
2  void testSteffensen() {
3    // TODO: (8-4.b) write a test of your implementation, that prints
4    // an estimate of the zero of f(x) = xe^x - 1
5    // START
6
7    auto f = [](double x) { return x * std::exp(x) - 1; };
8    double x0 = 1.0;
9
10   const double x = steffensen(f, x0);
11
12   std::cout << "The iterative method converges to " << x << std::endl;
13
14   // END
15 }
```

Get it on GitLab (quadraticconvergence.hpp).

HINT 1 for (8-4.c): [Lecture → Def. 8.2.2.10] defines the order of convergence of an iteration for solving $f(x) = 0$, while [Lecture → Rem. 8.2.2.12] explains how to extract an estimate for the order from measured errors. ⌟

SOLUTION of (8-4.c):

C++ code 8.4.4: Testing the order of Steffensen's method

```
void orderSteffensen() {
  auto f = [](double x) { return x * std::exp(x) - 1; };

  constexpr double x_star = 0.567143290409784; // use as exact value

  // TODO: (8-4.c) Tabulate values from which you can read the
  // order of Steffensen's method.
  // 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

  std::vector<double> store;

  auto log = [&store](double x) -> void { store.push_back(x); };

  steffensen_log(f, 1.0, log);

  const unsigned n = store.size();

  Eigen::VectorXd errs(n), log_errs(n);
  for (unsigned i = 0; i < n; ++i) {
    errs(i) = std::abs(store[i] - x_star);
    log_errs(i) = std::log(errs(i));
  }

  Eigen::VectorXd ratios = Eigen::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) << store[i] << std::setw(20) << errs(i)
              << std::setw(20) << ratios(i) << std::endl;
  }

  // END
}
```

Get it on GitLab (quadraticconvergence.hpp).

C++ code 8.4.5: Steffensen's method with logging of iterates

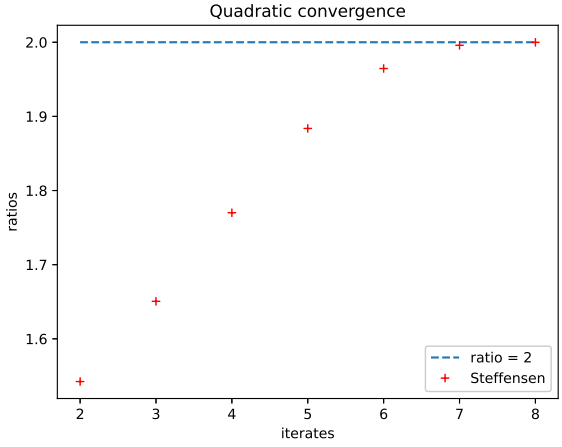
```
template <class Function, typename LOGGER>
double steffensen_log(Function &&f, double x0,
    LOGGER &&log = [](double) -> void {}) {
```

```
5
6     double x = x0;
7
8     // TODO: (8-4.c) Modify the function steffensen:  use the logger to
9     // save all the iterations x of the Steffensen's method.
10    // START
11
12    // Add x to the log.  Note:  This is an idle lambda function if none
      is
13    // provided.  See the default value in the function signature.
14    log(x);
15
16    double upd = 1;
17    const double eps = std::numeric_limits<double>::epsilon();
18
19    // Iterate until machine precision is reached
20    while (std::abs(upd) > eps * x) {
21      // Only 2 evaluations of f at each step
22      const double fx = f(x);
23
24      if (fx != 0) {
25        // Compute update
26        upd = fx * fx / (f(x + fx) - fx);
27
28        // New iterate
29        x -= upd;
30
31        log(x);
32      } else {
33        upd = 0;
34      }
35    }
36
37    // END
38
39    return x;
40  }
```

Get it on GitLab (quadraticconvergence.hpp).

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



◁ 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. 91. As in the experiments shown in class, roundoff affects the estimated order, once the iteration error approaches the machine precision.

HINT 1 for (8-4.d): If f is a function and $h : [a,b] \to \mathbb{R}$ with $h(x) \neq 0,\ \forall x \in [a,b]$, then $(fh)(x) = 0 \iff f(x) = 0$. ⌟

SOLUTION of (8-4.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 a larger set of positive initial points. Of course, $\tilde{f}(x) = 0$ exactly when $f(x) = 0$.

HINT 1 for (8-5.a): Study the estimates of [Lecture → § 8.4.3.4]. ⌟

SOLUTION of (8-5.a):

We have to ensure that the norm of the iteration error tends to zero:

$$\lim_{k\to\infty} x^{(k)} = x^* \iff \lim_{k\to\infty} \|x^{(k)} - x^*\| = 0 \,.$$

Thus, we seek an upper bound $B(k)$ for $\|x^{(k)} - x^*\|$:

$$\begin{aligned}
\|x^{(k)} - x^*\| &\le C\|x^{(k-1)} - x^*\|^p \\
&\le C \cdot C^p \|x^{(k-2)} - x^*\|^{p^2} \\
&\le C \cdot C^p \cdot C^{p^2} \|x^{(k-3)} - x^*\|^{p^3} \\
&\vdots \\
&\le C \cdots C^{p^{k-1}} \|x^{(0)} - x^*\|^{p^k} \\
&= C^{\sum_{i=0}^{k-1} p^i} \|x^{(0)} - x^*\|^{p^k} \\
&\overset{\text{geom. series}}{=} C^{\frac{p^k-1}{p-1}} \|x^{(0)} - x^*\|^{p^k} \\
&\le C^{\frac{p^k-1}{p-1}} \epsilon_0^{p^k} = \underbrace{C^{\frac{1}{1-p}}}_{\text{const.}} \cdot \left(C^{\frac{1}{p-1}} \epsilon_0\right)^{p^k} = B(k) \,.
\end{aligned}$$

For what values of ϵ_0 will we have $\lim_{k\to\infty} B(k) = 0$? We can read this off the expression we found for $B(k)$:

$$\lim_{k\to\infty} B(k) = 0 \quad \Longleftrightarrow \quad C^{\frac{1}{p-1}} \epsilon_0 < 1 \quad \Longrightarrow \quad \boxed{0 < \epsilon_0 < C^{\frac{1}{1-p}}} \,. \tag{8.5.3}$$

See also [Lecture → Eq. (8.4.3.9)].

SOLUTION of (8-5.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:

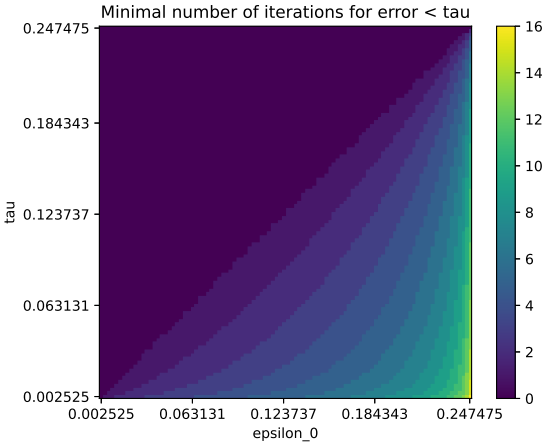
$$\ln\left(C^{\frac{1}{1-p}}\right) + p^k \cdot \underbrace{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}_{<0} < \ln\tau \,,$$

$$\blacktriangleright \quad k > \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}\right) \cdot \frac{1}{\ln(p)} \,,$$

$$\blacktriangleright \quad k_{min} = \left\lceil \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}\right) \cdot \frac{1}{\ln(p)} \right\rceil \in \mathbb{N} \,, \tag{8.5.4}$$

where $\lceil\cdot\rceil$ stands for the next larger integer. A similar, slightly simplified formula is [Lecture → Eq. (8.4.3.10)

SOLUTION of (8-5.c):



◁ Image of the minimum number $k_{\min}$ of iteration according to (8.5.4) as a function of ϵ_0 and the error reduction factor τ.

We see that the number of iterations increases only mildly as τ is reduced, but rather strongly depends on ϵ_0 as this values approaches the threshold $C^{\frac{1}{1-p}}$ found in Sub-problem (8-5.a).

A similar plot is given in [Lecture → Fig. 301].

C++-code 8.5.5: Solution of Sub-problem (8-5.c)

```
2  void kminplot() {
3    constexpr double p = 1.5;
4    constexpr double C = 2.;
5    const double eps_max = std::pow(C, 1. / (1. - p));
6    constexpr unsigned int ngp = 100;  // number of grid points
7
8    plt::figure();
9
10   // TODO: (8-5.c) Plot kmin using matplotlibcpp.  To that end, start by
       creating
11   // a mesh using the provided meshgrid function in meshgrid.hpp.
12   // START
13   Eigen::VectorXd eps_lin = Eigen::VectorXd::LinSpaced(ngp, 0, eps_max);
14   Eigen::VectorXd tau_lin = eps_lin;
15   // we need an open interval, i.e.  remove first and last element
16   Eigen::VectorXd eps_lin_seg = eps_lin.segment(1, eps_lin.size() - 2);
17   Eigen::VectorXd tau_lin_seg = tau_lin.segment(1, tau_lin.size() - 2);
18   Eigen::MatrixXd eps_msh, tau_msh;
19   meshgrid(eps_lin_seg, tau_lin_seg, eps_msh, tau_msh);
20
21   // apply formula to mesh
22   Eigen::MatrixXd kmin;
23   Eigen::MatrixXd num = tau_msh.array().log();
24   num += (1. / (p - 1.)) * std::log(C) *
25          Eigen::MatrixXd::Ones(tau_msh.rows(), tau_msh.cols());
26   Eigen::MatrixXd den = (eps_msh * std::pow(C, 1. / (p - 1.))).array().log();
27   kmin = (num.cwiseQuotient(den)).array().log() / std::log(p);
28
29   // Consider only gridpoints where eps is larger than tau
30   for (unsigned int i = 0; i < kmin.rows(); ++i) {
31     for (unsigned int j = 0; j < kmin.cols(); ++j) {
32       kmin(i, j) = std::ceil(kmin(i, j));
33       if (i > j) {
34         kmin(i, j) = 0;
35       }
36     }
37   }
```

```
38
39    std::vector<double> ticks(5);
40    std::vector<std::string> xlabels(5), ylabels(5);
41    for (unsigned int i = 0; i < 5; ++i) {
42      ticks[i] = i * (ngp - 3) / 4;
43      Eigen::Index id = (Eigen::Index)ticks[i];
44      xlabels[i] = std::to_string(eps_lin_seg(id));
45      ylabels[i] = std::to_string(tau_lin_seg(id));
46    }
47
48    plt::imshow(kmin, {{"cmap", "viridis"}, {"origin", "lower"}});
49    plt::colorbar();
50    plt::title("Minimal number of iterations for error < tau");
51    plt::xlabel("epsilon_0");
52    plt::ylabel("tau");
53    plt::xticks(ticks, xlabels);
54    plt::yticks(ticks, ylabels);
55
56    // END
57    plt::savefig("./cx_out/k_min.png");
58  }
```

Get it on GitLab (ordconviter.hpp).

HINT 1 for (8-6.a): Consult [Lecture → Rem. 8.2.2.12] to learn how to estimate orders of convergence from sequences of error norms. ⌟

SOLUTION of (8-6.a):

To estimate the order of convergence according to [Lecture → Def. 8.2.2.10] we have to track the quotients

$$\frac{\log e^{(k+1)} - \log e^{(k)}}{\log e^{(k)} - \log e^{(k-1)}} .$$

C++11-code 8.6.2: Measuring order of convergence

```
2  double testOrder(const unsigned int n = 20) {
3    double order = 0;
4    // TODO: (8-6.a) Compute the order of the method using recursion
         (8.6.1)
5    // Use as initial error guess e_1 = 1 and e_2 = 0.8
6    // START
7    std::vector<double> e(n), loge(n);
8    e[0] = 1; // initial error
9    loge[0] = std::log(e[0]);
10   e[1] = 0.8;
11   loge[1] = std::log(e[1]);
12   for (unsigned int k = 1; k < n - 1; ++k) {
13     e[k + 1] = e[k] * std::sqrt(e[k - 1]); // error recursion
14     loge[k + 1] = std::log(e[k + 1]);
15   }
16   order = (loge[n - 1] - loge[n - 2]) / (loge[n - 2] - loge[n - 3]);
17   // END
18   return order;
19 }
```

Get it on GitLab (recursionorder.hpp).

The order appears to be approximately 1.36603.

HINT 1 for (8-6.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 (8-6.b): Study how the order p of convergence was derived from the error recurrence [Lecture → Eq. (8.4.2.37)] in [Lecture → Rem. 8.4.2.33]. ⌟

HINT 3 for (8-6.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 (8-6.b):

$\|e^{(n)}\| = C\|e^{(n-1)}\|^p$ (p is the largest convergence order and C is some constant). Then

$$\|e^{(n+1)}\| = C\|e^{(n)}\|^p = C(C\|e^{(n-1)}\|^p)^p = C^{p+1}\|e^{(n-1)}\|^{p^2} , \tag{8.6.3}$$

We assume equality in the error recusion, because this is the worst case. Thus,

$$\|e^{(n+1)}\| = \|e^{(n)}\| \cdot \|e^{(n-1)}\|^{\frac{1}{2}} = C\|e^{(n-1)}\|^{p+\frac{1}{2}} . \tag{8.6.4}$$

Combining the previous equations:

$$C^{p+1}\|e^{(n-1)}\|^{p^2} = C\|e^{(n-1)}\|^{p+\frac{1}{2}} ,$$

i.e.

$$C^p\|e^{(n-1)}\|^{p^2-p-\frac{1}{2}} = 1 . \tag{8.6.5}$$

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.

SOLUTION of (8-7.b):

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 a circuit element. In particular we use the constitutive relation for the Ohmic resistor, $I = U/R$, here with $R = 1$.

① $3U_1 - 0 - U_2 - U_3 = 0$,

② $3U_2 - U_1 - U_3 - U_{\text{in}} = 0$,

③ $3U_3 - 0 - U_1 - U_2 + \alpha\left(e^{\beta\left(\frac{U_3 - U_{\text{in}}}{U_T}\right)} - 1\right) = 0.$

Thus the nonlinear system of equations reads

$$\mathbf{F}(\mathbf{u}) = \begin{bmatrix} 3U_1 - U_2 - U_3 \\ 3U_2 - U_1 - U_3 - U_{\text{in}} \\ 3U_3 - U_1 - U_2 + \alpha e^{-\frac{\beta}{U_T}U_{\text{in}}} \cdot e^{\frac{\beta}{U_T}U_3} - \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{for } \mathbf{u} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix}.$$

SOLUTION of (8-7.c):

According to [Lecture → Section 8.5.1] the iteration of the multi-dimensional Newton method is

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} - \left(\mathrm{D}\,\mathbf{F}(\mathbf{u}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{u}^{(k)})\,.$$

The components of the Jacobian $\mathrm{D}\,\mathbf{F}$ 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

$$\mathrm{D}\,\mathbf{F}(\mathbf{u}) = \begin{bmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 + \frac{\alpha\beta}{U_T} e^{\beta\frac{U_3 - U_{\mathrm{in}}}{U_T}} \end{bmatrix}.$$

We use EIGEN's built-in Gaussian elimination solver, see [Lecture → § 2.5.0.4].

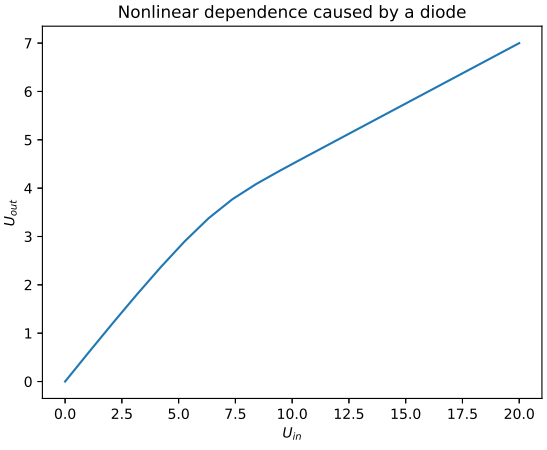
C++-code 8.7.1: Solution

```
2  void circuit(const double& alpha, const double& beta,
3               const Eigen::VectorXd& Uin, Eigen::VectorXd& Uout) {
4    constexpr double Ut = 0.5;
5    constexpr double tau = 1e-6;
6    const unsigned int n = Uin.size();
7
8    // TODO: (8-7.b) Compute the output voltages for the given circuit
9    // START
10   double Uin_;  // Uin of current node
11   // lambda function for evaluation of F
12   auto F = [alpha, beta](const Eigen::VectorXd& U, const double Uin_) {
13     Eigen::VectorXd f(3);
14     f << 3. * U(0) - U(1) - U(2), 3. * U(1) - U(0) - U(2) - Uin_,
15         3. * U(2) - U(0) - U(1) +
16             alpha * (std::exp(beta * (U(2) - Uin_) / Ut) - 1);
17     return f;
18   };
19
20   Eigen::MatrixXd J(3, 3);                 // the Jacobian
21   J << 3, -1, -1, -1, 3, -1, -1, -1, 0;    // dummy in J(2,2)
22   Eigen::VectorXd f(3);                    // the function
23
24   for (unsigned int i = 0; i < n; ++i) {
25     Uin_ = Uin(i);
26     Eigen::VectorXd U = Eigen::VectorXd::Random(3);  // random initial guess
27     Eigen::VectorXd h = Eigen::VectorXd::Ones(3);
28     while (h.cwiseAbs().maxCoeff() > tau * U.norm()) {
29       J(2, 2) = 3. + (alpha * beta) / Ut * std::exp(beta * (U(2) - Uin_) / Ut);
30       f = F(U, Uin_);
31       h = J.partialPivLu().solve(f);
32       U -= h;
33     }
34     Uout(i) = U(0);
35   }
36   // END
37 }
```

Get it on GitLab (nonlinear_circuit.hpp).

SOLUTION of (8-7.d):

The nonlinear effects can be observed by calculating the differences between the solutions U_{out}: see file `nonlinear_circuit.hpp`.
The relationship $U_{\text{out}} = U_{\text{out}}(U_{\text{in}})$ is obviously not affine-linear.



C++-code 8.7.2: Solution

```
2  void plotU() {
3    constexpr unsigned int n = 20;  // number of grid points
4    constexpr double alpha = 8;
5    constexpr double beta = 1;
6
7    plt::figure();
8
9    // TODO: (8-7.c) Plot Uout as a function of Uin with matplotlibcpp
10   // START
11   Eigen::VectorXd Uin = Eigen::VectorXd::LinSpaced(n, 0, 20);
12   Eigen::VectorXd Uout(n);
13   circuit(alpha, beta, Uin, Uout);
14
15   plt::plot(Uin, Uout);
16   plt::title("Nonlinear dependence caused by a diode");
17   plt::xlabel("$U_{in}$");
18   plt::ylabel("$U_{out}$");
19   // END
20   plt::savefig("./cx_out/Udiode.png");
21 }
```

Get it on GitLab (nonlinear_circuit.hpp).

SOLUTION of (8-8.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. (8.8.1) is equivalent to the 2×2 non-linear system of equations

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} x^3 - 3xy^2 - 1 \\ 3x^2 y - y^3 \end{bmatrix} = \mathbf{0} \, , \quad \mathbf{x} := \begin{bmatrix} x \\ y \end{bmatrix} \, . \tag{8.8.3}$$

SOLUTION of (8-8.b):

The Newton iteration in $\mathbb{R}^n$ reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(D\mathbf{F}(\mathbf{x}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{x}^{(k)}) \,. \tag{8.8.4}$$

see [Lecture → Eq. (8.5.1.6)], where $\mathrm{D}\,\mathbf{F}$ is the Jacobian

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ 6xy & 3x^2 - 3y^2 \end{bmatrix} . \tag{8.8.5}$$

SOLUTION of (8-8.c):

We use the initialization of matrices via the << operator offered by EIGEN, see [Lecture → § 1.2.1.3].

C++ code 8.8.6: Evaluation of F

```
Eigen::Vector2d juliaF(const Eigen::Vector2d &z) {
  Eigen::Vector2d Fz = Eigen::Vector2d::Zero();
  // TODO: (8-8.c) 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;
}
```

Get it on GitLab (julia.hpp).

C++ code 8.8.7: Evaluation of Jacobian DF

```
Eigen::Matrix2d juliaDF(const Eigen::Vector2d &z) {
  Eigen::Matrix2d DFz = Eigen::Matrix2d::Zero();
  // TODO: (8-8.c) Implement the Jacobian of F at z.
  // START
  const double a = 3.0 * (z(0) * z(0) - z(1) * z(1));
  const double b = 6.0 * z(0) * z(1);
  DFz << a, -b, b, a;
  // END
  return DFz;
}
```

Get it on GitLab (julia.hpp).

HINT 1 for (8-8.d): You may stop the iteration once you are closer in distance to one of the three roots of unity than 10^{-3}.

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 (8-8.d):

C++ code 8.8.8: Plotting of attractors of complex roots

```
void julia() {
  // Exact solutions of z^3 = 1 for complex z.
  // Use 2-dimensional real vectors to represent complex numbers.
  Eigen::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.
  constexpr double tol = 1e-4;
  constexpr unsigned int N_it = 20;

  // The image will have a resolution of res*res.
  constexpr unsigned 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.
  Eigen::MatrixXd C(res, res);

  // TODO: (8-8.d) 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 (unsigned int i = 0; i < res; i++) {
    for (unsigned int j = 0; j < res; j++) {
      // Newton iteration starting at Z(i,j)=z=(x,y).
      const double x = -2 + 4.0 * i / (res - 1);
      const double y = -2 + 4.0 * j / (res - 1);
      Eigen::Vector2d z(x, y);
      for (unsigned int k = 0; k < N_it; k++) {
        z -= juliaDF(z).lu().solve(juliaF(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 (unsigned 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");
}
```

Get it on GitLab (julia.hpp).

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 $|x_1^{(k)} + ix_2^{(k)} - z_\ell| < 10^{-3}$, $\ell \in \{1,2,3\}$, 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.

A GPU-based implemenation by Alexander Pietak, CSE BSc student in 2022, can be accessed on ShaderToy.

SOLUTION of (8-9.a):

❶ 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.

❷ Conversely, if $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} = \ldots = \mathbf{x}^{(0)}$, then, by the recursion of the Newton method,

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})\,, \\ \mathbf{x}^{(k)} &= \mathbf{y}^{(k)} - \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{y}^{(k)})\,. \end{aligned}$$

Then we plug the first equation into the second and infer

$$\begin{aligned} &\mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)}) - \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})\mathbf{F}(\mathbf{x}^{(k)})) = \mathbf{0}\,, \\ [\mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)}) \text{ regular}] \quad &\Longrightarrow \quad \mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{F}(\mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})\mathbf{F}(\mathbf{x}^{(k)})) = \mathbf{0}\,. \end{aligned}$$

Finally, we exploit the injectivity (8.9.2) of $\mathbf{F}$ and again the regularity of $\mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})$:

$$\mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{F}(\mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})) \quad \Leftrightarrow \quad D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)}) = 0 \quad \Leftrightarrow \quad \mathbf{F}(\mathbf{x}^{(k)}) = 0\,.$$

SOLUTION of (8-9.b):

C++ code 8.9.3: Implementation of `mod_newt_step_scalar`.

```
2  template <typename Scalar, class Function, class Jacobian>
3  Scalar mod_newt_step_scalar(const Scalar& x, Function&& f, Jacobian&& df) {
4    double return_value = 0.;
5    // TODO: (8-9.b) Compute a step of the modified Newton method for a
       scalar
6    // function
7    // START
8    Scalar y = x + f(x) / df(x);
9    return_value = y - f(y) / df(x);
10   // END
11   return return_value;
12 }
```

Get it on GitLab (modnewt.hpp).

HINT 1 for (8-9.c): The exact solution is $x = \tan(a) = 0.123624065869274\ldots$.

HINT 2 for (8-9.c): Remember that $\arctan'(x) = \frac{1}{1+x^2}$. ⌟

SOLUTION of (8-9.c):

We implement a generic function `sample_nonlinear_solver()` 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++ code 8.9.4: 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,
                             unsigned int max_itr = 100) {
  // TODO: (8-9.c,f) Implement this generic function that iteratively solves a
  // nonlinear equation.  The function should return whether the iteration
  // converged in the maximum number of iterations
  // START.
  // Temporary where to store new step
  Vector x_new = x;
  double r = 1;

  for (unsigned 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;
  // END
  return false;
}
```

Get it on GitLab (modnewt.hpp).

We employ the method discussed in [Lecture → Rem. 8.2.2.12] to estimate the order of convergence of the iteration. The estimator is

$$\texttt{estord} := \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}}, \quad \epsilon_k := \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|_2 . \tag{8.9.5}$$

C++ code 8.9.6: Implementation of `mod_newt_ord`.

```cpp
void mod_newt_ord() {
  // Setting up values, functions and jacobian
  constexpr double a = 0.123;
  auto f = [](double x) {
    return atan(x) - a;
  };                    // Function which sees a as it is constexpr
  double x_scalar = 5;  // initial guess

  // TODO: (8-9.c) Generate suitable terminal output to determine the order of
  // convergence of the modified Newton method applied to the scalar equation
  // that is given by $f = 0$.  You may want to implement the templated function
  // sample_nonlinear_solver for the iterations.
  // START
  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 step
  std::vector<double> sol;
  std::vector<double> err;

  // Compute error and push back to err, used in
  // sample_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
  sol.push_back(x_scalar);
  errf(x_scalar);

  // Lambda performing the next step, used to define a proper
  // function handle to be passed to sample_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
```

```
56    auto space = std::setw(15);
57    std::cout << space << "sol." << space << "err." << space << "order"
58              << std::endl;
59    for (unsigned i = 0; i < sol.size(); ++i) {
60      std::cout << space << sol.at(i) << space << err.at(i);
61      if (i >= 3) {
62        std::cout << space
63                  << (log(err.at(i)) - log(err.at(i - 1))) /
64                         (log(err.at(i - 1)) - log(err.at(i - 2)));
65      }
66      std::cout << std::endl;
67    }
68    // END
69  }
```

Get it on GitLab (modnewt.hpp).

solution	error	estord
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

The method converges very fast so that the iteration error quickly drops to the level of machine precision, where it is polluted by roundoff errors. So it is difficult to estimate the rate of convergence from the data.

We can just infer that the rate seems to be between 2 and 3.

HINT 1 for (8-9.d): Do not forget to perform a single LU-decomposition of the Jacobian and reuse it. See [Lecture → Code 8.5.3.6] for an example. ⌟

SOLUTION of (8-9.d):

C++ code 8.9.7: Implementation of `mod_newt_step_system`.

```
template <typename Vector, class Function, class Jacobian>
Vector mod_newt_step_system(const Vector& x, Function&& f, Jacobian& df) {
  Vector return_value = x;  // this is just a dummy, you should overwrite it
  // TODO: (8-9.d) Efficiently perform one step of the modified Newton method
  // for a system of equations.
  // START
  auto lu = df(x).lu();
  // Reusing LU decomposition
  Vector y = x + lu.solve(f(x));
  return_value = y - lu.solve(f(y));
  // END
  return return_value;
}
```

Get it on GitLab (modnewt.hpp).

SOLUTION of (8-9.e):

We can simply rely on component-wise partial derivatives to derive

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \mathbf{A} + \begin{bmatrix} c_1 e^{x_1} & & & \\ & c_2 e^{x_2} & & \\ & & \ddots & \\ & & & c_n e^{x_n} \end{bmatrix} \in \mathbb{R}^{n,n} \,. \tag{8.9.9}$$

SOLUTION of (8-9.f):

C++ code 8.9.10: Implementation of `mod_newt_sys`.

```
Eigen::VectorXd mod_newt_sys(const Eigen::MatrixXd& A, const Eigen::VectorXd& c,
                             double tol = 1.0e-6, int maxit = 100) {
  Eigen::VectorXd x_system = Eigen::VectorXd::Zero(c.size());  // Initial guess

  // TODO: (8-9.f) Solve the non-linear system of equations (9.9.6) using the
  // modified Newton method.  You may want to use your implementation of
  // sample_nonlinear_solver.
  // START

  // Handler for function and jacobian in standard format.  Must see matrix A
  // and vector c.
  auto F = [&A, &c](const Eigen::VectorXd& x) {
    Eigen::VectorXd tmp =
        A * x + c.cwiseProduct(x.array().exp().matrix()).eval();
    return tmp;
  };
  auto dF = [&A, &c](const Eigen::VectorXd& x) {
    Eigen::MatrixXd C = A;
    Eigen::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](Eigen::VectorXd& x) {
    if (err.size() > 0)
      return err.back();
    else
      return 1.;
  };

  // Refactor (i.e. wrap) the step such that it is compatible with
  // sample_nonlinear_solver.  Must see F, dF and 'err'.  We also store the
  // error.
  auto newt_system_step = [&F, &dF,
                           &err](const Eigen::VectorXd& x) -> Eigen::VectorXd {
    Eigen::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, tol, maxit);
  // END
  return x_system;
}
```

Get it on GitLab (modnewt.hpp).

HINT 1 for (8-9.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 (8-9.g):

We implement a “stepping” function and reuse the previous codes.

We iterate the Modified Newton 5 times and print out the error for each step with corresponding order of convergence.

The order of convergence is estimate as recommended in [Lecture → Rem. 8.2.2.12], see (8.9.5).

C++ code 8.9.11: Implementation of `mod_newt_sys_test()`.

```
2  void mod_newt_sys_test() {
3    Eigen::MatrixXd A(4, 4);
4    A << 2., -1., 0., 0., -1., 2., -1., 0., 0., -1., 2., -1., 0., 0., -1., 2.;
5    Eigen::VectorXd c(4);
6    c << 1., 2., 3., 4.;
7
8    // TODO: (8-9.g) Determine the order of convergence from suitable
       tabulated
9    // values.
10   // determine a reference solution with very low tolerance
11   // (machine precision)
12   // START
13   constexpr double reference_tol = std::numeric_limits<double>::epsilon();
14   Eigen::VectorXd reference_sol = mod_newt_sys(A, c, reference_tol);
15
16   std::vector<double> err;
17   std::vector<Eigen::VectorXd> sol;
18
19   // seperate calculation and printing to not pollute the table from
       output of
20   // iteration function increase the number of maximal iterations
21   sol.push_back(Eigen::VectorXd::Zero(4));
22   err.push_back(reference_sol.norm());
23   for (unsigned k = 1; k < 5; ++k) {
24     sol.push_back(mod_newt_sys(A, c, 0., k));
25     err.push_back((reference_sol - sol.at(k)).norm());
26   }
27
28   // Perform convergence study with Modified newton for system of
       equations
29   std::cout << std::endl
30             << "*** Modified Newton method (system) ***" << std::endl
31             << std::endl;
32   std::cout << "Exact: " << reference_sol.transpose() << std::endl;
33
34   auto space = std::setw(15);
35   std::cout << space << "k" << space << "err." << space << "order" << std::endl;
36
37   // print iteration number, error and order
38   for (unsigned k = 0; k < 5; ++k) {
39     std::cout << space << k << space << err.at(k);
40     if (k >= 1 && k < 4) {
41       std::cout << space
42                 << (log(err.at(k + 1)) - log(err.at(k))) /
43                        (log(err.at(k)) - log(err.at(k - 1)));
44     }
45     std::cout << std::endl;
```

```
46    }
47    // END
48  }
```

Get it on GitLab (modnewt.hpp).

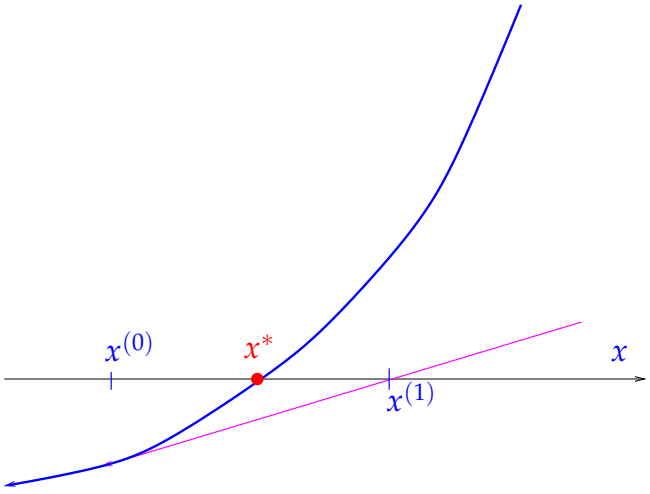
k	error	estord
0	2.87469	
1	0.359054	3.39385
2	0.000308353	2.74505
3	1.18145e-12	0.442692
4	2.22045e-16	

◁ Error table as printed by `mod_newt_sys()`.

As in Sub-problem (8-9.c) the method converges very fast and it is difficult to tease out a convergence order, because only few meaningful data points are generated before the iteration error becomes polluted by roundoff error.

The order of convergence seems to be > 2 consistent with what we observed in Sub-problem (8-9.c).

SOLUTION of (9-1.a):

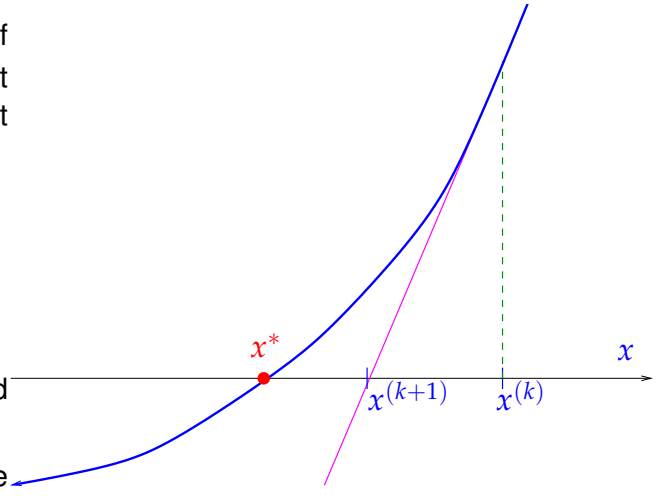


If $x^{(0)} < x^*$, where $F(x^*) = 0$, then $x^{(1)} > x^*$ and $F(x^{(1)}) > 0$.
Therefore, we may assume $x^{(1)} > x^*$ from now.

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

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} < x^{(k)} .$$



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:

$$x^+ = x^+ - \frac{F(x^+)}{F'(x^+)} \quad \blacktriangleright \quad F(x^+) = 0 .$$

SOLUTION of (9-10.a):

The fixed-point iteration $\mathbf{x}^{(k)} \to \mathbf{x}^{(k+1)}$, $k \in \mathbb{N}$, is defined by

$$\mathbf{x}^{(k+1)} \in \mathbb{R}^n: \quad \mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k+11)} = \boldsymbol{v} \quad \Leftrightarrow \quad \mathbf{x}^{(k+1)} = \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b} \; . \tag{9.10.3}$$

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.

```
2  Eigen::VectorXd fixed_point_step(const Eigen::VectorXd &xk, const Eigen::VectorXd
       &b) {
3    Eigen::VectorXd x_new;
4    // TO DO (9-10.b): implement one step of the fixed point iteration
5    // START
6    // Define matrix by fixed point x^k
7    const int n = xk.size();
8    double nrm = xk.norm();
9    Eigen::SparseMatrix<double> A(n, n);
10   A.reserve(3 * n);
11   for (int i = 0; i < n; ++i) {
12     if (i > 0)
13       A.insert(i, i - 1) = 1;
14     A.insert(i, i) = 3 + nrm;
15     if (i < n - 1)
16       A.insert(i, i + 1) = 1;
17   }
18   // solve linear system
19   Eigen::SparseLU<Eigen::SparseMatrix<double>> Ax_lu(A);
20   x_new = Ax_lu.solve(b);
21   // END
22   return x_new;
23 }
```

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;

     // Check if Fixed point method is successful
    if (err < atol || err < rtol * x_new.norm()) {
      break;
    }
  }
  // END
  return x;
}
```

SOLUTION of (9-10.d):

The Newton iteration, as provided in the lecture document in [Lecture → Ex. 8.5.1.30], reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2} \right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \; . \tag{9.10.7}$$

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.

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;
}
```

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;

     // Check if Newton method is successful
    if (err < atol || err < rtol * x_new.norm()) {
      break;
    }
  }
  // END
  return x;
}
```

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

$$\begin{aligned}
\Phi_B(t) &= e^{-\lambda_2 t} B_0 + \int_0^t e^{-\lambda_2 (t-\tau)} \lambda_1 e^{-\lambda_1 \tau} A_0 \, d\tau && = e^{-\lambda_2 t} B_0 + \lambda_1 e^{-\lambda_2 t} A_0 \int_0^t e^{(\lambda_2 - \lambda_1)\tau} \, d\tau \\
&= e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} (e^{(\lambda_2 - \lambda_1) t} - 1) A_0 && = e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \Big(e^{-\lambda_1 t} - e^{-\lambda_2 t} \Big) A_0.
\end{aligned}$$

It satisfies the initial condition $\Phi_B(0) = B_0$, we test if it is a solution of the ODE (9.11.1):

$$\begin{aligned}
\Phi_B'(t) &= -\lambda_2 e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \Big(-\lambda_1 e^{-\lambda_1 t} + \lambda_2 e^{-\lambda_2 t} \Big) A_0 \\
&= -\lambda_2 e^{-\lambda_2 t} B_0 + \lambda_2 \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} A_0 - \lambda_1 \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_1 t} A_0 \\
&= -\lambda_2 \Phi_B(t) + \lambda_2 \frac{\lambda_1}{\lambda_2 - \lambda_1} \Phi_A(t) - \lambda_1 \frac{\lambda_1}{\lambda_2 - \lambda_1} \Phi_A(t) \\
&= -\lambda_2 \Phi_B(t) + \lambda_1 \Phi_A(t).
\end{aligned}$$

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

$$F_i(\mathbf{x}) \quad = \quad \Phi_B(t_i) - m_i \quad = \quad e^{-\lambda_2 t_i} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i} \right) A_0 - m_i, \qquad i = 1, \ldots, N.$$

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:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{F}(\mathbf{x})\|_2^2 = \underset{\mathbf{x}}{\operatorname{argmin}} \sum_{i=1}^{N} |F_i(\mathbf{x})|^2.$$

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, $i = 1, \ldots, N$, 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

$$\mathbf{s} = \operatorname*{argmin}_{\mathbf{a} \in \mathbb{R}^4} \|\mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{DF}(\mathbf{x}^{(k)})\, \mathbf{a}\|_2.$$

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)

```
2       std::vector<double> gn_update;
3       Eigen::Vector4d x(1., 1., 1., 0.1), s;
4       do {
5           Eigen::Vector4d s = DF(x, t, m).colPivHouseholderQr().solve(F(x, t, m));
6           x -= s;
7           gn_update.push_back(s.lpNorm<Eigen::Infinity>());
8       } while (gn_update.back() > 1e-14);
```

An examination of the changes in `gn_update` shows linear convergence.

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

eq. (9.12.1) $\Leftrightarrow \quad x^2 - 2m_1x + m_1^2 + y^2 - 2m_2y + m_2^2 - r^2 = 0 \quad \Leftrightarrow \quad -2xm_1 - 2ym_2 - c = -x^2 - y^2$

Thus, for the data (x_i, y_i), $i = 1, \ldots, N$, we have the $N \times 3$ linear system

$$\begin{pmatrix} -2x_1 & -2y_1 & -1 \\ \vdots & \vdots & \vdots \\ -2x_N & -2y_N & -1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ c \end{pmatrix} \overset{L.S.}{=} \begin{pmatrix} -x_1^2 - y_1^2 \\ \vdots \\ -x_N^2 - y_N^2 \end{pmatrix}.$$

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.

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).

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 $i = 1, \ldots, N$, 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 $\|\mathbf{F}(\mathbf{z})\|^2 = \sum_{i=1}^{n} d_i^2$.

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 8.7.1], 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

$$\begin{aligned} \mathbf{H}\Phi(\mathbf{z}) &= \underbrace{\mathbf{DF}(\mathbf{z})^T}_{\in \mathbb{R}^{3,N}} \cdot \underbrace{\mathbf{DF}(\mathbf{z})}_{\in \mathbb{R}^{N,3}} + \sum_{i=1}^{N} \underbrace{F_i(\mathbf{z})}_{\in \mathbb{R}} \underbrace{\mathbf{H}F_i(\mathbf{x})}_{\in \mathbb{R}^{3,3}} \\ &= \sum_{i=1}^{N} \begin{bmatrix} \frac{x_i - m_1}{R_i} \\ \frac{y_i - m_2}{R_i} \\ 1 \end{bmatrix} \begin{bmatrix} \frac{x_i - m_1}{R_i}, & \frac{y_i - m_2}{R_i}, & 1 \end{bmatrix} + \sum_{i=1}^{N} (R_i - r) \begin{bmatrix} \frac{1}{R_i} - \frac{(x_i - m_1)^2}{R_i^3} & \frac{-(x_i - m_1)(x_i - m_2)}{R_i^3} & 0 \\ \frac{-(x_i - m_1)(x_i - m_2)}{R_i^3} & \frac{1}{R_i} - \frac{(y_i - m_2)^2}{R_i^3} & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} N + r \sum_{i=1}^{N} \left(\frac{(x_i - m_1)^2}{R_i^3} - \frac{1}{R_i} \right) & r \sum_{i=1}^{N} \frac{(x_i - m_1)(y_i - m_2)}{R_i^3} & \sum_{i=1}^{N} \frac{(x_i - m_1)}{R_i} \\ r \sum_{i=1}^{N} \frac{(x_i - m_1)(y_i - m_2)}{R_i^3} & N + r \sum_{i=1}^{N} \left(\frac{(y_i - m_2)^2}{R_i^3} - \frac{1}{R_i} \right) & \sum_{i=1}^{N} \frac{(y_i - m_2)}{R_i} \\ \sum_{i=1}^{N} \frac{(x_i - m_1)}{R_i} & \sum_{i=1}^{N} \frac{(y_i - m_2)}{R_i} & N \end{bmatrix}. \end{aligned}$$

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.

```
2 Vector3d circl_geo_fit(const VectorXd &x, const VectorXd & y) {
3
4 }
```

Get it on GitLab (circle_approx.cpp).

SOLUTION of (9-12.f):

The iteration of the Newton method [Lecture → Eq. (8.7.1.6)] 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).

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 **??**.

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 $c/a < (b_1^2 + b_2^2)/(4a^2)$).

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 $\rightarrow$ 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.

SOLUTION of (9-13.a):

We start with the obvious implication

$$\boldsymbol{x} \in R_{\mathbf{d}} \cap \partial \mathcal{L}_c \quad \Leftrightarrow \quad \exists \xi > 0\colon \quad \boldsymbol{x} = \xi \mathbf{d} \quad \text{and} \quad f(\xi \mathbf{d}) = c\,.$$

Hence, we have to solve the zero-finding problem for a real-valued function $F : \mathbb{R}_0^+ \to \mathbb{R}$

$$\text{find} \quad \xi > 0: \quad F(\xi) := f(\xi \mathbf{d}) - c = 0\,. \tag{9.13.4}$$

By the chain rule

$$F'(\xi) = \mathbf{grad}\, f(\xi \mathbf{d})^\top \mathbf{d}\,, \quad \xi > 0$$

The Newton iteration [Lecture → Eq. (8.4.2.1)] for (9.13.4) is the recursion

$$\xi^{(k+1)} = \xi^{(k)} - \frac{F(\xi^{(k)})}{F'(\xi^{(k)})} = \xi^{(k)} - \frac{f(\xi^{(k)}\mathbf{d}) - c}{\mathbf{grad}\, f(\xi^{(k)}\mathbf{d})^\top \mathbf{d}}\,, \quad k \geq 0\,. \tag{9.13.5}$$

As initial guess we choose some $\xi^{(0)} > 0$.

Note that the denominator in (9.13.5) cannot vanish, because $\mathbf{d} \neq \mathbf{0}$ and f is strictly convex, which means that the derivative of $\xi \mapsto f(\xi \mathbf{d})$ is strictly increasing on $[0, \infty[$.

SOLUTION of (9-13.b):

According to [Lecture → Eq. (8.4.2.30)] the two-point iteration for the secant method applied to (9.13.4) reads:

$$\begin{aligned} \xi^{(k+1)} &= \xi^{(k)} - \frac{\xi^{(k)} - \xi^{(k-1)}}{F(\xi^{(k)}) - F(\xi^{(k-1)})} F(\xi^{(k)}) \\ &= \frac{\xi^{(k)} - \xi^{(k-1)}}{f(\xi^{(k)}\mathbf{d}) - f(\xi^{(k-1)}\mathbf{d})} \big(f(\xi^{(k)}\mathbf{d}) - c\big) \ . \end{aligned} \tag{9.13.6}$$

SOLUTION of (9-13.c):

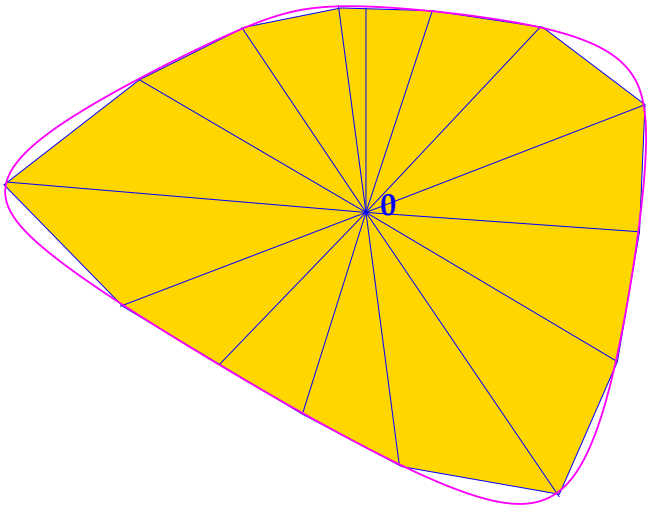
The implementation of (9.13.6) is straightforward. The problem statements asks us to use $\xi^{(0)} = \frac{\|\mathbf{x}_0\|_2}{\|\mathbf{d}\|_2}$ and $\xi^{(1)} = \frac{\|\mathbf{x}_1\|_2}{\|\mathbf{d}\|_2}$. An efficient implementation uses only a single f-evaluation per step.

Concerning the correction-based termination criterion see [Lecture → Code 8.4.2.31].

C++ code 9.13.7: Sub-problem (9-13.c): The function `pointLevelSet()`

```
2  template <typename Functor>
3  Vector2d pointLevelSet(Functor &&f, const Vector2d &d, double c,
4                         const Vector2d &x0, const Vector2d &x1,
5                         double rtol = 1e-10, double atol = 1e-16) {
6    Vector2d intersect;
7    // START
8    // The function whose zero has to found
9    auto F = [&f, &d, &c](double t) -> double { return f(t * d) - c; };
10
11   double upd = 1.;
12   // initial guesses for secand method
13   double t0 = x0.norm()/d.norm();
14   double t1 = x1.norm()/d.norm();
15
16   double fo = F(t0);
17   // check termination criterion
18   while (std::abs(upd) > std::max(atol, rtol * std::min(t0, t1))) {
19     double fn = F(t1);
20     // secant update
21     upd = fn * (t1 - t0) / (fn - fo);
22     // new iteration points
23     t0 = t1;
24     t1 = t1 - upd;
25     fo = fn;
26   }
27   // compute intersection of the boundary of the level set with the ray
28   intersect = t1 * d;
29   // END
30   return intersect;
31 }
```

HINT 1 for (9-13.d): A convex polygonal domain P with $\mathbf{0} \in P$ can be partitioned into (non-overlapping) triangles with one vertex in $\mathbf{0}$ and the other two vertices coinciding with corners of P.



◁ Approximating the area enclosed by a convex level set boundary by the sum of areas of triangles having the same angle $\frac{2\pi}{n}$ at vertex $\mathbf{0}$.

The area of a triangle $\triangle ABC$ with angle α at vertex A is

$$|\triangle ABC| = \tfrac{1}{2}\overline{AB} \cdot \overline{AC} \sin\alpha \ .$$

⌟

HINT 2 for (9-13.d): It is convenient to use the function `pointLevelSet()` to find points on $\partial\mathcal{L}_c$. ⌟

SOLUTION of (9-13.d):

C++ code 9.13.8: Sub-problem (9-13.d): The function `areaLevelSet()`

```
template <typename Functor>
double areaLevelSet(Functor &&f, unsigned int n, double c) {
  assert(n > 2 && "n too small");
  double area;

  // define initial guess
  Vector2d x0 = {1, 0};
  Vector2d x1 = {2, 0};
  // START
  // constant angle
  double sin_ct = std::sin(2 * PI / n) * 0.5;
  VectorXd side(n + 1);
  for (unsigned int j = 0; j < n; ++j) {
    Vector2d d;
    d << std::cos(2 * PI * j / n), std::sin(2 * PI * j / n);
    // compute lengths of archimedean radii
    side(j) = pointLevelSet(f, d, c, x0, x1).norm();
  }
  // periodic extension for vectorization
  side(n) = side(0);
  // multiply pairs of adjacent sides
  area = (side.tail(n).cwiseProduct(side.head(n))).sum() * sin_ct;
  // END
  return area;
}
```

SOLUTION of (9-13.e):

n	error in area
5	0.7322814883
10	0.2020820848
20	0.0539834484
40	0.0136526713
80	0.0034265044

We observe a decrease of the error by roughly a factor of four, when doubling number n of evaluation points. Thus, we conclude *algebraic convergence* with rate 2:

$$\text{error} = O((1/n)^2) \quad \text{for} \quad n \to \infty \,.$$

HINT 1 for (9-14.a): From [Lecture → Section 3.4.4.2] recall the use of the singular value decomposition (SVD) to solve low-rank best approximation problems. ⌟

SOLUTION of (9-14.a):

If $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ is the singular-value decomposition of $\mathbf{M}$, we know from [Lecture → Thm. 3.4.4.19] that $(\mathbf{U})_{:,1}(\mathbf{\Sigma})_{1,1}(\mathbf{V})_{:,1}^\top$ is a rank-1 matrix closest to $\mathbf{M}$ in Frobenius norm.

If $\mathbf{M} = \mathbf{M}^\top$, then $\mathbf{U} = \mathbf{V}$ in its SVD, which is an immediate consequence of [Lecture → Lemma 3.4.1.6]. As a consequence, the closest rank-1 matrix is also symmetric! This means that

$$\mathbf{M} = \mathbf{M}^\top \quad \blacktriangleright \quad (\mathbf{U})_{:,1}\sqrt{(\mathbf{\Sigma})_{1,1}} \in \operatorname*{argmin}_{\mathbf{z}\in\mathbb{R}^n} \left\| \mathbf{M} - \mathbf{z}\mathbf{z}^\top \right\|_F .$$

Implementation in EIGEN is straightforward, see [Lecture → Code 3.4.2.1].

C++ code 9.14.3: Sub-problem (9-14.a): The function `symRankOneBestApproxSym()`

```
2  VectorXd symRankOneBestApproxSym(const MatrixXd &M) {
3    // ensure that M is symmetric
4    assert(M.cols() == M.rows() && M == M.transpose() && "M is not symmetric");
5    // Vector to be used to return result
6    VectorXd z(M.cols());
7    // TODO: (9-14.a)
8    //  solve equ. (9.14.1) for a symmetric matrix with the aid of singular value
9    //  decomposition (SVD)
10   // START
11   // Compute economical SVD of M
12   JacobiSVD<MatrixXd> svd(M, ComputeThinU | ComputeThinV);
13   // Extract first column of u and first singular value
14   VectorXd first_U = svd.matrixU().col(0);
15   VectorXd sv = svd.singularValues();
16   // Scaling
17   z = first_U * std::sqrt(sv[0]);
18   // END
19   return z;
20 }
```

Get it on GitLab (symrank1.hpp).

There is no difference between economical and full SVD for a square matrix.

HINT 1 for (9-14.b): You can exploit that the Kronecker product is a bilinear operation and use the general product rule. ⌟

SOLUTION of (9-14.b):

① We can find the Jacobian of $\mathbf{\Phi}$ by a perturbation approach. Using that the Kronecker product is a bi-linear operation we get

$$\mathbf{\Phi}(\mathbf{x}+\mathbf{h}) - \mathbf{\Phi}(\mathbf{x}) = -\mathbf{x}\otimes\mathbf{h} - \mathbf{h}\otimes\mathbf{x} - \mathbf{h}\otimes\mathbf{h} \overset{!}{=} \mathsf{D}\,\mathbf{\Phi}(\mathbf{x})\mathbf{h} + O(\|\mathbf{h}\|^2) \,,$$

$$\blacktriangleright \qquad \mathsf{D}\,\mathbf{\Phi}(\mathbf{x})\mathbf{h} = -\mathbf{x}\otimes\mathbf{h} - \mathbf{h}\otimes\mathbf{x} \,. \tag{9.14.5}$$

② The same result follows from the general product rule [Lecture → Eq. (8.5.1.17)].

- Product rule: $F : D \subset V \mapsto W$, $G : D \subset V \mapsto U$ sufficiently smooth, $b : W \times U \mapsto Z$ *bilinear*, *ie.*, linear in each argument:

$$\begin{aligned} & T(\mathbf{x}) = b(F(\mathbf{x}), G(\mathbf{x})) \\ \Rightarrow \quad & \mathsf{D}\,T(\mathbf{x})\mathbf{h} = b(\mathsf{D}\,F(\mathbf{x})\mathbf{h}, G(\mathbf{x})) + b(F(\mathbf{x}), \mathsf{D}\,G(\mathbf{x})\mathbf{h}) \,, \\ & \mathbf{h} \in V, \mathbf{x} \in D \,. \end{aligned} \tag{9.14.6}$$

We apply it with

- $D = V = W = U = \mathbb{R}^n$,
- $F \equiv G = \mathsf{Id}$ (identity operators, for which $\mathsf{D}\,F = \mathsf{D}\,G = \mathsf{Id}$),
- $b(\mathbf{x},\mathbf{y}) = -\mathbf{x}\otimes\mathbf{y}$ and $Z = \mathbb{R}^{n^2}$,

such that (9.14.6) gives us

$$\mathsf{D}\,T(\mathbf{x})\mathbf{h} = b(\mathbf{h},\mathbf{x}) + b(\mathbf{x},\mathbf{h}) = -\mathbf{h}\otimes\mathbf{x} - \mathbf{x}\otimes\mathbf{h} \,.$$

HINT 1 for (9-14.c): You may use the following properties of the Kronecker product:

$$(\mathbf{A}\otimes\mathbf{B})(\mathbf{C}\otimes\mathbf{D}) = (\mathbf{AC})\otimes(\mathbf{BD})\,, \tag{9.14.7}$$

$$(\mathbf{A}\otimes\mathbf{B})^\top = \mathbf{A}^\top\otimes\mathbf{B}^\top\,, \tag{9.14.8}$$

$$\mathbf{x}\otimes\mathbf{y} = (\mathbf{x}\otimes\mathbf{I}_n)\mathbf{y} = (\mathbf{I}_n\otimes\mathbf{y})\mathbf{x}\,, \tag{9.14.9}$$

for all matrices $\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}$ of suitable sizes and vectors $\mathbf{x},\mathbf{y}\in\mathbb{R}^n$. ⌟

SOLUTION of (9-14.c):

① The most concise derivation starts from (9.14.5) and uses (9.14.9):

$$\begin{aligned} \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})\mathbf{h} &= -\mathbf{x}\otimes\mathbf{h} - \mathbf{h}\otimes\mathbf{x} \\ &= -(\mathbf{x}\otimes\mathbf{I}_n)([1]\otimes\mathbf{h}) - (\mathbf{I}_n\otimes\mathbf{x})(\mathbf{h}\otimes[1]) \\ &= -(\mathbf{x}\otimes\mathbf{I}_n + \mathbf{I}_n\otimes\mathbf{x})\mathbf{h}\ . \end{aligned}$$

The Jacobian is just the matrix representation of the linear mapping $\mathbf{h}\mapsto \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})\mathbf{h}$ in standard bases, so that

$$\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}) = -\mathbf{x}\otimes\mathbf{I}_n - \mathbf{I}_n\otimes\mathbf{x} \in \mathbb{R}^{n^2,n}\ . \tag{9.14.10}$$

Recalling the definition of the Kronecker product we can explicitly write

$$-\mathbf{x}\otimes\mathbf{I}_n - \mathbf{I}_n\otimes\mathbf{x} = -\begin{bmatrix} x_1 & & & & & \\ & \ddots & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & x_1 \\ \vdots & & & & \vdots & \\ & & & & & \\ \vdots & & & & \vdots & \\ x_n & & & & & \\ & \ddots & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & x_n \end{bmatrix} - \begin{bmatrix} x_1 & & & & & \\ \vdots & & & & & \\ \vdots & & & & & \\ x_n & & & & & \\ & x_1 & & & & \\ & \vdots & & & & \\ & \vdots & & & & \\ & x_n & & & & \\ & & \ddots & & & \\ & & \vdots & & & \\ & & \vdots & & & \\ & & & \ddots & & \\ & & & \vdots & & \\ & & & \vdots & & \\ & & & & x_1 & \\ & & & & \vdots & \\ & & & & \vdots & \\ & & & & x_n & \end{bmatrix},$$

$$\blacktriangleright \quad \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)}) = -\begin{bmatrix} x_1 + x_1 & & & & & & \\ x_2 & \ddots & & & & & \\ \vdots & & \ddots & & & & \\ \vdots & & & \ddots & & & \\ x_{n-1} & & & & \ddots & & \\ x_n & & & & & x_1 & \\ \vdots & & & & & \vdots & \\ \vdots & & & & & \vdots & \\ x_n & & & & & x_1 & \\ & \ddots & & & & x_2 & \\ & & \ddots & & & \vdots & \\ & & & \ddots & & \vdots & \\ & & & & \ddots & x_{n-1} & \\ & & & & & x_n + x_n & \end{bmatrix} \in \mathbb{R}^{n^2,n} .$$

② We can do direct componentwise differentiation of $(x_i := (\mathbf{x})_i)$

$$\boldsymbol{\Phi}(\mathbf{x}) = \mathrm{vec}(\mathbf{M}) + \Big[x_1^2, x_1x_2, \ldots, x_1x_n, x_2x_1, x_2^2, x_2x_3, \ldots, x_2x_n, x_3x_1, \ldots, \ldots, x_nx_1, \ldots, x_n^2\Big]^\top ,$$

which yields, e.g.,

$$\frac{\partial \boldsymbol{\Phi}}{\partial x_1}(\mathbf{x}) = [2x_1, x_2, x_3, \ldots, x_n, x_2, 0, \ldots, 0, x_3, 0, \ldots, \ldots, x_n, 0, \ldots, 0] ,$$
$$\frac{\partial \boldsymbol{\Phi}}{\partial x_2}(\mathbf{x}) = [0, x_1, 0, \ldots, 0, x_1, 2x_2, x_3, \ldots, x_n, 0, x_3, 0, \ldots, \ldots, 0, x_n, 0, \ldots, 0] ,$$
$$\vdots$$

In the case $n = 3$ we obtain

$$\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)}) = -\begin{bmatrix} 2x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & 0 & x_1 \\ x_2 & x_1 & 0 \\ 0 & 2x_2 & 0 \\ 0 & x_3 & x_2 \\ x_3 & 0 & x_1 \\ 0 & x_3 & x_2 \\ 0 & 0 & 2x_3 \end{bmatrix} .$$

HINT 1 for (9-14.d): The system matrix $\mathbf{T}$ of the normal equations is a symmetric rank-1 matrix added to a scaled identity:

$$\mathbf{T} = \alpha\mathbf{I} + \mathbf{v}\mathbf{v}^\top , \quad \alpha \in \mathbb{R} , \quad \mathbf{v} \in \mathbb{R}^n , \tag{9.14.12}$$

α and $\mathbf{v}$ to be determined. ⌟

SOLUTION of (9-14.d):

The normal equations for (9.14.11) read

$$\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b} \quad \blacktriangleright \quad \mathbf{T} = \mathbf{A}^\top \mathbf{A} \,.$$

Here, we have $\mathbf{A} = \mathsf{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})$ and, thus, we have to compute $\mathsf{D}\,\boldsymbol{\Phi}(\mathbf{x})^\top \mathsf{D}\,\boldsymbol{\Phi}(\mathbf{x})$ for $\mathbf{x} = \mathbf{x}^{(k)}$. For the sake of simplicity we write $\mathbf{x}$ in place of $\mathbf{x}^{(k)}$.

Approach ①: Use (9.14.10) together with (9.14.8) and, then, (9.14.9),

$$\begin{aligned}
\mathsf{D}\,\boldsymbol{\Phi}(\mathbf{x})^\top \mathsf{D}\,\boldsymbol{\Phi}(\mathbf{x}) &= (\mathbf{x} \otimes \mathbf{I}_n + \mathbf{I}_n \otimes \mathbf{x})^\top (\mathbf{x} \otimes \mathbf{I}_n + \mathbf{I}_n \otimes \mathbf{x}) \\
&= (\mathbf{x} \otimes \mathbf{I}_n)^\top (\mathbf{x} \otimes \mathbf{I}_n) + (\mathbf{x} \otimes \mathbf{I}_n)^\top (\mathbf{I}_n \otimes \mathbf{x}) + \\
&\quad (\mathbf{I}_n \otimes \mathbf{x})^\top (\mathbf{x} \otimes \mathbf{I}_n) + (\mathbf{I}_n \otimes \mathbf{x})^\top (\mathbf{I}_n \otimes \mathbf{x}) \\
&= (\mathbf{x}^\top \otimes \mathbf{I}_n)(\mathbf{x} \otimes \mathbf{I}_n) + (\mathbf{x}^\top \otimes \mathbf{I}_n)(\mathbf{I}_n \otimes \mathbf{x}) + \\
&\quad (\mathbf{I}_n \otimes \mathbf{x}^\top)(\mathbf{x} \otimes \mathbf{I}_n) + (\mathbf{I}_n \otimes \mathbf{x}^\top)(\mathbf{I}_n \otimes \mathbf{x}) \\
&= (\mathbf{x}^\top \mathbf{x}) \otimes \mathbf{I}_n + \mathbf{x}^\top \otimes \mathbf{x} + \mathbf{x}^\top \otimes \mathbf{x} + \mathbf{I}_n \otimes (\mathbf{x}^\top \mathbf{x}) \\
&= \|\mathbf{x}\|^2 \mathbf{I}_n + \mathbf{x}\mathbf{x}^\top + \mathbf{x}\mathbf{x}^\top + \|\mathbf{x}\|^2 \mathbf{I}_n \,.
\end{aligned}$$

$$\blacktriangleright \quad \mathbf{T} = \boxed{2\|\mathbf{x}\|^2 \mathbf{I}_n + 2\mathbf{x}\mathbf{x}^\top} \,.$$

HINT 1 for (9-14.e): Exploit the Sherman-Morrison-Woodbury formula or the inverse of a matrix under low-rank modification:

Lemma [Lecture → Lemma 2.6.0.21]. Sherman-Morrison-Woodbury formula

For regular $\mathbf{A} \in \mathbb{K}^{n,n}$, *and* $\mathbf{U}, \mathbf{V} \in \mathbb{K}^{n,k}$, $n, k \in \mathbb{N}$, $k \leq n$, *holds*

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I}_k + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^H\mathbf{A}^{-1} \,,$$

if $\mathbf{I}_k + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U}$ *is regular.*

HINT 2 for (9-14.e): If you failed to compute $\mathbf{T}$, you may use the general formula from (9.14.12). ⌟

SOLUTION of (9-14.e):

We apply the Sherman-Morrison-Woodbury formula [Lecture → Lemma 2.6.0.21] for $k = 1$ with $\mathbf{A} = 2\|\mathbf{x}\|^2$, $\mathbf{U} = 2\mathbf{x} \in \mathbb{R}^{n,1}$, $\mathbf{V} = \mathbf{x} \in \mathbb{R}^{n,1}$, and get

$$\mathbf{T}^{-1} = \frac{1}{2\|\mathbf{x}\|^2}\left(\mathbf{I} - \frac{1}{2}\frac{\mathbf{x}\mathbf{x}^\top}{\|\mathbf{x}\|^2}\right) . \tag{9.14.13}$$

Hence, solving a linear system of equations with system matrix $\mathbf{T}$ can be accomplished by carrying out a few simple vector operations.

HINT 1 for (9-14.f): The reshaping of vectors into matrices via EIGEN's **Eigen::Map** wrapper class may be useful, see [Lecture → Code 1.2.3.7]. ⌟

SOLUTION of (9-14.f):

The considerations follow [Lecture → Ex. 1.4.3.8].

Writing $v_j := (\mathbf{v})_j$, by the formulas for the Kronecker product we have

$$\mathbf{v}^\top \otimes \mathbf{I}_n = [v_1\mathbf{I}_n\ ,\ v_2\mathbf{I}_n\ ,\ \ldots\ ,\ v_n\mathbf{I}_n] \in \mathbb{R}^{n,n^2}\ ,$$
$$\mathbf{I}_n \otimes \mathbf{v}^\top = \begin{bmatrix} \mathbf{v}^\top & & & \\ & \mathbf{v}^\top & & \\ & & \mathbf{v}^\top & \\ & & & \mathbf{v}^\top \end{bmatrix} \in \mathbb{R}^{n,n^2}\ .$$

Next, chop up $\mathbf{b} \in \mathbb{R}^{n^2}$ into n sections of length n:

$$\mathbf{b} = \begin{bmatrix} \mathbf{b}^1 \\ \vdots \\ \mathbf{b}^n \end{bmatrix}\ ,\quad \mathbf{b}^\ell \in \mathbb{R}^n\ ,\quad \ell \in \{1,\ldots,n\}\ ,$$

and introduce the matrix with those parts as columns:

$$\mathbf{B} = \left[\mathbf{b}^1, \mathbf{b}^2, \ldots, \mathbf{b}^n\right] \in \mathbb{R}^{n,n}\ .$$

In the code this matrix can be formed by means of **Eigen::Map**.

By the rules of matrix×vector multiplication:

$$(\mathbf{v}^\top \otimes \mathbf{I}_n)\mathbf{b} = [v_1\mathbf{I}_n, v_2\mathbf{I}_n, \ldots, v_n\mathbf{I}_n] \begin{bmatrix} \mathbf{b}^1 \\ \vdots \\ \mathbf{b}^n \end{bmatrix} = v_1\mathbf{b}^1 + \cdots + v_n\mathbf{b}^n = \mathbf{B}\mathbf{v}\ ,$$
$$(\mathbf{I}_n \otimes \mathbf{v}^\top)\mathbf{b} = \begin{bmatrix} \mathbf{v}^\top & & & \\ & \mathbf{v}^\top & & \\ & & \mathbf{v}^\top & \\ & & & \mathbf{v}^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}^1 \\ \vdots \\ \mathbf{b}^n \end{bmatrix} = \begin{bmatrix} \mathbf{v}^\top\mathbf{b}^1 \\ \mathbf{v}^\top\mathbf{b}^2 \\ \vdots \\ \mathbf{v}^\top\mathbf{b}^n \end{bmatrix} = \mathbf{B}^\top\mathbf{v}\ .$$

In the implementation you should not create a temporary matrix to hold $\mathbf{B}$; rather use **Eigen::Map** twice.

C++ code 9.14.15: Sub-problem (9-14.f): The function `computeKronProdVecMult()`

```
2  VectorXd computeKronProdVecMult(const VectorXd &v, const VectorXd &b) {
3    const unsigned int n = v.size();
4    assert(b.size() == n * n && "Size mismatch");
5    // Vector to be used to return result
6    VectorXd res;
7    // TODO: (9-14.f)
8    //  evaluate the expression (9.14.14) efficiently with the aid of Eigen::map
9    // START
10   // Raw data array for argument vector b
11   const double *data = b.data();
12   // Twice reinterpretation of data, no copy
13   res =
```

```
14          MatrixXd::Map(data, n, n) * v + MatrixXd::Map(data, n, n).transpose() * v;
15      // END
16      return res;
17  }
```

Get it on GitLab (symrank1.hpp).

HINT 1 for (9-14.g): Take the cue from [Lecture → Code 8.7.2.2]. ⌟

HINT 2 for (9-14.g): It may be convenient to call `computeKronProdVecMult()` at one point. ⌟

HINT 3 for (9-14.g): **If and only if** you failed to find $\mathbf{T}^{-1}$ in Sub-problem (9-14.e) and the vector $\mathbf{b}$ of (9.14.11), then use

- $\mathbf{T}^{-1} = \operatorname{diag}(1, 2, \ldots, n)$,
- $\mathbf{b} = \operatorname{vec}(\mathbf{M})$,
- $\mathsf{D}\,\boldsymbol{\Phi}(\mathbf{x}) = [x_1\mathbf{I}_n, x_2\mathbf{I}_n, \ldots, x_n\mathbf{I}_n]^\top$.

SOLUTION of (9-14.g):

Using (9.14.13) in the k-th step of the iteration, the solution of the normal equations is

$$\mathbf{s} = \frac{1}{2\|\mathbf{x}^{(k)}\|^2}\left(\mathbf{I} - \frac{1}{2}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|^2}\right) \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})^\top(\mathrm{vec}(\mathbf{M}) - \mathbf{x}^{(k)} \otimes \mathbf{x}^{(k)}) \,. \tag{9.14.16}$$

From the result of (9.14.10) and (9.14.7) we infer

$$\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})^\top = -\mathbf{x}^\top \otimes \mathbf{I}_n - \mathbf{I}_n \otimes \mathbf{x}^\top \quad \text{with} \quad \mathbf{x} := \mathbf{x}^{(k)} \,.$$

Thus, after you have computed $\mathbf{b} := \mathrm{vec}(\mathbf{M}) - \mathbf{x}^{(k)} \otimes \mathbf{x}^{(k)}$ you can call `computeKronProdVecMult()` to obtain $\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})^\top\mathbf{b}$. Everything else in (9.14.16) is straightforward.

The vector $\mathbf{s}$ then supplies the correction in the current Gauss-Newton step.

C++ code 9.14.17: Sub-problem (9-14.g): The function `symmRankOneApprox()`

```
VectorXd symmRankOneApprox(const MatrixXd &M, const double rtol = 1e-6,
                           const double atol = 1e-8) {
  // ensure M is square
  assert(M.cols() == M.rows() && "Matrix must be square");

  const unsigned int n = M.cols();
  // Initial guess: Column of symmetric part of M with largest norm
  auto Msym = 0.5 * (M + M.transpose());
  Eigen::Index j, dummy;
  Msym.colwise().norm().maxCoeff(&dummy, &j);
  // Set initial guess. This vector is also used to return the result
  VectorXd z{Msym.col(j)};
  // TODO: (9-14.g)
  //  solve equ. (9.14.1) using Gauss-Newton iteration with tolerances rtol and
  //  atol
  // START
  // Gauss-Newton correction vector
  VectorXd s(n);
  // Direct access to the array storing the entries of M
  const double *dataM = M.data();

  std::cout << " Correction s to current newton step:\n";
  // Main Gauss-Newton loop with correction-based termination
  do {
    // Compute b = vec(M) - kron(z,z)
    const MatrixXd Z = z * z.transpose();
    const double *dataz = Z.data();
    const VectorXd b =
        MatrixXd::Map(dataM, n * n, 1) - MatrixXd::Map(dataz, n * n, 1);
    // Solve normal equations
    const double normalize = 2 * z.squaredNorm();
    VectorXd kronMult = computeKronProdVecMult(z, b);
    s = (kronMult - z.dot(kronMult) / normalize * z) / normalize;
    // Apply Gauss-Newton update
    z = z + s;
    std::cout << "|s| = " << s.norm() << std::endl;
  } while (s.norm() > std::max(atol, rtol * z.norm()));
  std::cout << "\n";
```

```
40    // END
41    return z;
42  }
```

Get it on GitLab (symrank1.hpp).

Remark. Write tr for the **trace** of a square matrix, that is, the sum of its diagonal entries. Then

$$\mathbb{R}^{n,n} \times \mathbb{R}^{n,n} \to \mathbb{R} \quad , \quad (\mathbf{X}, \mathbf{Y}) \mapsto (\mathbf{X}, \mathbf{Y})_F := \operatorname{tr}(\mathbf{X}^\top \mathbf{Y}) \, ,$$

defines an *inner product* on the matrix space $\mathbb{R}^{n,n}$. The Frobenius norm is the associated norm! By straightforward computations we find

$$\operatorname{tr}(\mathbf{AB}) = \operatorname{tr}(\mathbf{BA}) \quad \forall \mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n} \, .$$

As a consequence, symmetric and skew-symmetric matrices are $(\cdot,\cdot)_F$-orthogonal

$$\mathbf{X} = \mathbf{X}^\top \, , \quad \mathbf{Y} = -\mathbf{Y}^\top \quad \Rightarrow \quad (\mathbf{X}, \mathbf{Y})_F = 0 \, ,$$

because

$$(\mathbf{X}, \mathbf{Y})_F = \operatorname{tr}(\mathbf{X}^\top \mathbf{Y}) = -\operatorname{tr}(\mathbf{X}\mathbf{Y}^\top) = -\operatorname{tr}(\mathbf{Y}^\top \mathbf{X}) = -(\mathbf{Y}, \mathbf{X})_F \, .$$

Thus, by Pythagoras' theorem we have

$$\begin{aligned} \left\| \mathbf{M} - \mathbf{x}\mathbf{x}^\top \right\|_F^2 &= \left\| \tfrac{1}{2}(\mathbf{M} + \mathbf{M}^\top) + \tfrac{1}{2}(\mathbf{M} - \mathbf{M}^\top) - \mathbf{x}\mathbf{x}^\top \right\|_F^2 \\ &= \left\| \tfrac{1}{2}(\mathbf{M} + \mathbf{M}^\top) - \mathbf{x}\mathbf{x}^\top \right\|_F^2 + \left\| \tfrac{1}{2}(\mathbf{M} - \mathbf{M}^\top) \right\|_F^2 \, . \end{aligned}$$

Hence the symmetric rank-1 best approximation of the symmetric part $\frac{1}{2}(\mathbf{M} + \mathbf{M}^\top)$ is the symmetric rank-1 best approximation also for the general matrix $\mathbf{M}$!

As a consequence the initial guess `symRankOneBestApproxSym`$(1/2(\mathbf{M} + \mathbf{M}^\top))$ is already the final solution in `symRankOneApprox()` and the Gauss-Newton iteration does have any effect! What a waste of time that we programmed it before plunging into deep mathematical considerations. Yet, this is how it usually works in computational mathematics: With hindsight one could have avoided a lot of coding by clever mathematical analysis, but only the computational results suggested that analysis.

HINT 1 for (9-2.a): First try to understand what is accomplished in Lines 5–12 of this code. Recall the properties of logarithms. ⌟

HINT 2 for (9-2.a): Line 15–18 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 loop in lines 15–18 realizes the Newton iterations for finding the zero of the function

$$f(z) = e^z - x\,, \quad z \in \mathbb{R}\,, \tag{9.2.2}$$

where z is the independent variable and x is given. We compute $f'(x) = e^z$ and obtain the Newton iteration

$$z^{(k+1)} = z^{(k)} - \frac{\exp(z^{(k)}) - x}{\exp(z^{(k)})} = z^{(k)} - s\,, \quad s = 1 - \frac{x}{\exp(z^{(k)})}\,. \tag{9.2.3}$$

C++11-code 9.2.4: The documented function

```
2  double myfunction(double x) {
3    // We require that x > 0.
4    // First we factorize x as x = 2^{n+r} = 2^n * x_new
5    // with integer n and r ∈ [0.5,1.5].
6    // Set y = n log(2) and z = log(x_new),
7    // then log(x) = y + z.
8    double dy = 0.693147180559945; // dy = std::log(2)
9    double y = 0.;
10   // If x x = 2^{n+r} with positive n:
11   while (x > 2. * std::sqrt(2.)) {
12     x /= 2.;
13     y += dy;
14   } //
15   // If x x = 2^{n+r} with negative n:
16   while (x < std::sqrt(2.)) {
17     x *= 2.;
18     y -= dy;
19   } //
20   // Now, we have found y, and we use Newton iteration for the
21   // function f(z) = exp(z) - x_new to find z = log(x_new).
22   // Initial guess:
23   double z = x - 1.; //
24   // The update is dz = -f(z)/f'(z)
25   double dz = x * std::exp(-z) - 1.;
26   while (std::abs(dz / z) > std::numeric_limits<double>::epsilon()) {
27     z += dz;
28     dz = x * std::exp(-z) - 1.;
29   }
30   return y + z + dz; //
31 }
```

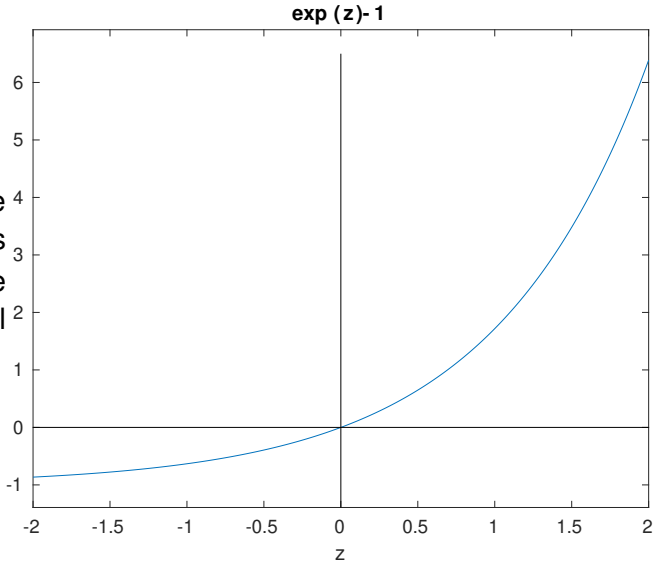
Get it on GitLab (codequiz.hpp).

SOLUTION of (9-2.b):

The **while** loops rely 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 $x \in [\sqrt{2}, 2\sqrt{2}]$.



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.

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, see also (9.2.3):

$$\begin{cases} z^{(0)} & := & x-1\,, \\ z^{(n+1)} & := & z^{(n)} - \frac{e^{z^{(n)}}-x}{e^{z^{(n)}}}\,, \quad n \geq 1 \end{cases} .$$

SOLUTION of (9-2.d):

This is a correction based termination criterium, see [Lecture → § 8.2.3.4]. It is meant to ensure that the relative error of the final iterate is of the size of the machine precision EPS, because in C++ EPS for a floating point data type **T** is available as **std**`::numeric_limits<T>::epsilon()`.

HINT 1 for (9-2.e): Using Taylor expansion and the fixed-point form of the Newton iteration as in [Lecture → § 8.4.2.5], establish a recursion for the iteration errors $e^{(n)} := z^{(n)} - z^*$. 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 $f(z) := e^z - x$ with z^*, write for the derivative $f'(z) = e^z$, and set $e^{(n)} := z^{(n)} - z^*$ for the iteration errors.



As in [Lecture → § 8.4.2.5] and the proof of [Lecture → Lemma 8.3.2.15] we

- rewrite the Newton iteration as a fixed-point iteration,
- use a Taylor expansion of its iteration function around z^*,
- and drop a powers $\left(e^{(n)}\right)^p$ for $p > 2$ (asymptotic perspective).

As in [Lecture → Eq. (8.4.2.6)] the Newton iteration can be rewritten as

$$z^{(k+1)} = \Phi(z^{(k)}) \quad \text{with} \quad \Phi(z) = z - \frac{f(z)}{f'(z)} \,. \tag{9.2.5}$$

As in [Lecture → Eq. (8.4.2.7)] we find by the quotient rule

$$\frac{d\Phi}{dz}(z) = 1 - \frac{\left(f'(z)\right)^2 - f(z)f''(z)}{\left(f'(z)\right)^2} = \frac{f(z)f''(z)}{\left(f'(z)\right)^2} \,.$$

Plugging in the concrete function f with $f(z) = e^z - x$, $f'(z) = e^z$, $f''(z) = e^z$, we get

$$\Phi'(z) := \frac{d\Phi}{dz}(z) = \frac{(e^z - x)e^z}{\left(e^z\right)^2} = 1 - xe^{-z} \,,$$

and by straightforward differentiation

$$\Phi''(z) := \frac{d^2\Phi}{dz^2}(z) = xe^{-z} \,.$$

For the iteration error of a fixed-point iteration like (9.2.5) we have

$$e^{(k+1)} = z^{(k+1)} - z^* = \Phi(z^{(k)}) - \Phi(z^*) = \Phi(z^* + e^{(k)}) - \Phi(z^*) \,. \tag{9.2.6}$$

Recall the Taylor expansion [Lecture → Eq. (1.5.4.28)] of a function $g \in C^3([a,b])$ around $t^* \in]a,b[$:

$$g(t^* + h) = g(t^*) + g'(t^*)h + \tfrac{1}{2}g''(t^*)h^2 + O(h^3) \quad \text{for} \quad h \to 0 \,. \tag{9.2.7}$$

Apply this formula to the right side of (9.2.6) with $g := \Phi$, $t^* := z^*$, $h := e^{(k)}$:

$$e^{(k+1)} = \Phi'(z^*)e^{(k)} + \tfrac{1}{2}\Phi''(z^*)\left(e^{(k)}\right)^2 + O\Big(\left(e^{(k)}\right)^3\Big) \quad \text{for} \quad |e^{(k)}| \approx 0 \,.$$

Since $e^{z^*} = x$, we immediately compute

$$\Phi'(z^*) = 0 \quad , \quad \Phi''(z^*) = 1 \,,$$

which implies

$$e^{(k+1)} = \tfrac{1}{2}\left(e^{(k)}\right)^2 + O\Big(\left(e^{(k)}\right)^3\Big) \,.$$

If we ignore higher powers of the iteration error, which is justified if the initial guess is already close to the solution,

$$|e^{(n+1)}| \doteq \frac{1}{2}|e^{(n)}|^2 \doteq \cdots \doteq (\frac{1}{2})^{1+2+\cdots+2^n}(e^{(0)})^{2^{n+1}} \doteq 2 \cdot (\frac{1}{2})^{2^{n+1}}(e^{(0)})^{2^{n+1}} \doteq 2 \cdot \left(\frac{1}{2}|e^{(0)}|\right)^{2^{n+1}} ,$$

where $\doteq$ expresses equality "up to terms of size $O\left((e^{(k)})^3\right)$". Further, the initial error is

$$e^{(0)} = z^{(0)} - z^* = x - 1 - \log(x) , \quad x \in \left[\sqrt{2}, 2\sqrt{2}\right] .$$

By discussing the simple function $x \mapsto x - 1 - \log(x)$ we find that the initial error $e^{(0)}$ attains its maximum value for $x = 2\sqrt{2}$:

$$|e^{(0)}| < \epsilon_{\max} := 2\sqrt{2} - 1 - \log(2\sqrt{2}) \approx 0.7887 .$$

We can determine the minimal required number n of iteration steps by finding the smallest integer satisfying

$$\left|\frac{e^{(n)}}{\log x}\right| \leq \text{EPS} .$$

We are on the safe side, when replacing $|\log(x)|$ with its smallest possible value $\frac{1}{2}\log 2$, as $x \in \left[\sqrt{2}, 2\sqrt{2}\right]$. This leads to the requirement

$$|e^{(n)}| \leq \tfrac{1}{2}\log 2 \ \text{EPS} .$$

Then we can plug in the expression for $e^{(n)}$ found above:

$$2\left(\tfrac{1}{2}e^{(0)}\right)^{2^n} \leq \tfrac{1}{2}\log 2 \ \text{EPS} .$$

Next, we replace $e^{(0)}$ with its upper bound and get

$$\left(\tfrac{1}{2}\epsilon_{\max}\right)^{2^n} \leq \tfrac{1}{4}\log 2 \ \text{EPS} ,$$

from which we conclude that we need at least

$$n_{\min} := \lceil \log_2 \left(\frac{-\log\left(\frac{1}{4}\log 2 \ \text{EPS}\right)}{-\log(\frac{1}{2}\epsilon_{\max})} \right) \rceil$$

steps, where $\lceil \xi \rceil$ means the smallest integer $\geq \xi$.

The code in Code 9.2.8 is for your reference.

C++11-code 9.2.8: Modified `myfunction`.

```
2 double myfunction_modified(double x) {
3   double dy = 0.693147180559945; // dy = std::log(2)
4   double y = 0.;
5   while (x > 2. * std::sqrt(2.)) {
```

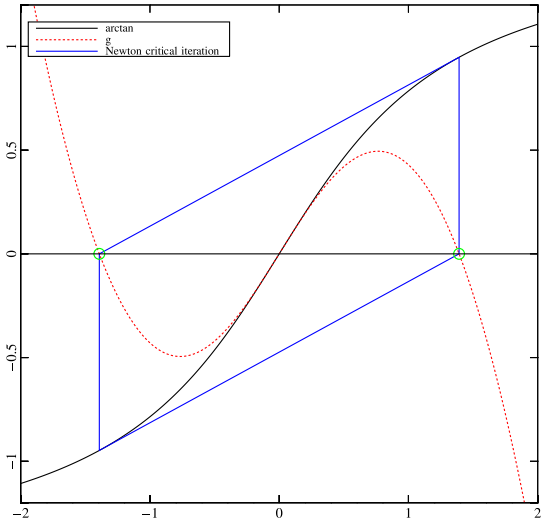
```
6       x /= 2.;
7       y += dy;
8     }
9     while (x < std::sqrt(2.)) {
10      x *= 2.;
11      y -= dy;
12    }
13    double z = x - 1.;
14    double dz = x * std::exp(-z) - 1.;
15    // TO DO: (9-2.e) Write a for-loop that achieves the same accuracy
16    // as the third while-loop of myfunction().  I.e.  fix the number of
17    // iterations before looping.
18    // START
19    double e_max = 2. * std::sqrt(2.) - 1. - std::log(2. * std::sqrt(2.));
20    double eps = std::numeric_limits<double>::epsilon();
21
22    int k_min = std::ceil(log2((-std::log(0.25 * std::log(2.) * eps)) /
23                               (-std::log(0.5 * std::abs(e_max)))));
24
25    for (int i = 1; i < k_min; ++i) {
26      z = z + dz;
27      dz = x * std::exp(-z) - 1.;
28    }
29    // END
30    return y + z + dz;
31  }
```

Get it on GitLab (codequiz.hpp).

HINT 1 for (9-3.a): Refer to [Lecture → Fig. 303] and find out for which initial guesses the Newton method oscillates between two values. ⌟

HINT 2 for (9-3.a):

Newton critical iteration



Clearly the solution is $x^* = 0$.
Graphical considerations may help you to find the solution, see Fig. 77. 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 $x^{(0)}$.

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

$$g(x) := 2x - (1 + x^2) \arctan x \quad \text{with} \quad g'(x) = 1 - 2x \arctan x \ .$$

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;
}
```

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 or by comparing two successive iterates.

C++-code 9.4.2: Solution of Sub-problem (9-4.a)

```
2  template <class Function> double steffensen(Function &&f, double x0) {
3    double x_old = x0;
4    double x = x0;
5    // TO DO (9-4.a): implement the Steffensen's method for a function f
6    // START
7    double upd = 1;
8    constexpr double eps = std::numeric_limits<double>::epsilon();
9    // Iterate until machine precision is reached
10   while (std::abs(upd) > eps * std::abs(x_old)) {
11     x_old = x;              // Storing the old iterate
12     const double fx = f(x); // Only 2 evaluations of f at each step
13     if (fx != 0) {
14       upd = fx * fx / (f(x + fx) - fx);
15       x -= upd; // New iterate
16     } else {
17       upd = 0;
18     }
19   }
20   // END
21   return x;
22 }
```

Get it on GitLab (quadraticconvergence.hpp).

SOLUTION of (9-4.b):

C++ code 9.4.3: Testing function for Steffensen's method

```
2  void testSteffensen(void) {
3    // TO DO (9-4.b):  write a test of your implementation, that prints
4    // an estimate of the zero of f(x) = xe^x - 1
5    // START
6    const double x =
7        steffensen([](double x) { return x * std::exp(x) - 1; }, 1.0);
8    std::cout << "The iterative method converges to " << x << std::endl;
9    // END
10 }
```

Get it on GitLab (quadraticconvergence.hpp).

HINT 1 for (9-4.c): [Lecture → Def. 8.2.2.10] defines the order of convergence of an iteration for solving $f(x) = 0$, while [Lecture → Rem. 8.2.2.12] 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

```
2  template <class T> class Logger {
3    std::vector<T> info;
4
5  public:
6    // Hint:  there is no need to implement a constructor
7    // since info is initialized as empty vector by default.
8
9    // TO DO: overload the operator ( ) so that it adds its argument to
10   // the member "info".
11   void operator()(T val) {
12     // START
13     info.push_back(val);
14     // END
15   }
16   // TO DO: Define a member function that fetches the data contained in
       "info".
17   // START
18   std::vector<T> getInfo(void) { return info; }
19   // END
20
21   void print_log() {
22     for (auto v : info) {
23       std::cout << v << std::endl;
24     }
25   }
26 };
```

Get it on GitLab (quadraticconvergence.hpp).

C++ code 9.4.5: Steffensen's method with logging of iterates

```
2  template <class Function>
3  double steffensen_log(Function &&f, double x0,
4                        Logger<double> *logger_p = nullptr) {
5
6    double x = x0;
7    // TO DO (9-4.c): Modify the function steffensen: use the class
       Logger to
8    // save all the iterations x of the Steffensen's method.
9    // START
10   bool log_enabled = false;
11   if (logger_p != nullptr) {
12     log_enabled = true;
13     (*logger_p)(x);
14   }
15   double upd = 1;
16   constexpr double eps = std::numeric_limits<double>::epsilon();
17   // Iterate until machine precision is reached
18   while (std::abs(upd) > eps * x) {
19     const double fx = f(x); // Only 2 evaluations of f at each step
20     if (fx != 0) {
21       upd = fx * fx / (f(x + fx) - fx);
```

```
      x -= upd;
      if (log_enabled) {
        (*logger_p)(x);
      }
    } else {
      upd = 0;
    }
  }
  // END
  return x;
}
```

Get it on GitLab (quadraticconvergence.hpp).

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; };
  constexpr double x_star = 0.567143290409784; // use as exact value
  // TO DO (9-4.c): tabulate values from which you can read the
  // order of Steffensen's method.
  // 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();
  const unsigned n = myData.size();

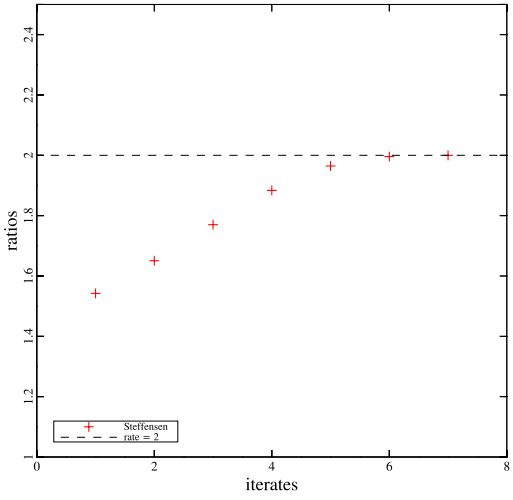
  VectorXd errs(n), log_errs(n);
  for (unsigned i = 0; i < n; ++i) {
    errs(i) = std::abs(myData[i] - x_star);
    log_errs(i) = std::log(errs(i));
  }
  VectorXd ratios = VectorXd::Zero(n);
  for (unsigned i = 2; i < n; ++i) {
    ratios(i) =
        (log_errs(i) - log_errs(i - 1)) / (log_errs(i - 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
}
```

Get it on GitLab (quadraticconvergence.hpp).

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

Quadratic convergence



◁ 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. 78. As in the experiments shown in class, roundoff affects the estimated order, once the iteration error approaches the machine precision.

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 $(fh)(x) = 0 \iff f(x) = 0$. ⌟

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 a larger set of positive initial points. Of course, $\tilde{f}(x) = 0$ exactly when $f(x) = 0$.

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 $\lim_{k\to\infty} B(k) = 0$.

$$\begin{aligned}
\|x^{(k)} - x^*\| &\le C\|x^{(k-1)} - x^*\|^p \\
&\le C \cdot C^p \|x^{(k-2)} - x^*\|^{p^2} \\
&\le C \cdot C^p \cdot C^{p^2} \|x^{(k-3)} - x^*\|^{p^3} \\
&\vdots \\
&\le C \cdots C^{p^{k-1}} \|x^{(0)} - x^*\|^{p^k} \\
&= C^{\sum_{i=0}^{k-1} p^i} \|x^{(0)} - x^*\|^{p^k} \\
&\overset{\text{geom. series}}{=} C^{\frac{p^k-1}{p-1}} \|x^{(0)} - x^*\|^{p^k} \\
&\le C^{\frac{p^k-1}{p-1}} \epsilon_0^{p^k} = \underbrace{C^{\frac{1}{1-p}}}_{\text{const.}} \cdot \left(C^{\frac{1}{p-1}}\epsilon_0\right)^{p^k} = B(k)\,.
\end{aligned}$$

Finally, $\lim_{k\to\infty} B(k) = 0 \quad \iff \quad C^{\frac{1}{p-1}}\epsilon_0 < 1 \quad \implies \quad 0 < \epsilon_0 < C^{\frac{1}{1-p}}$

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:

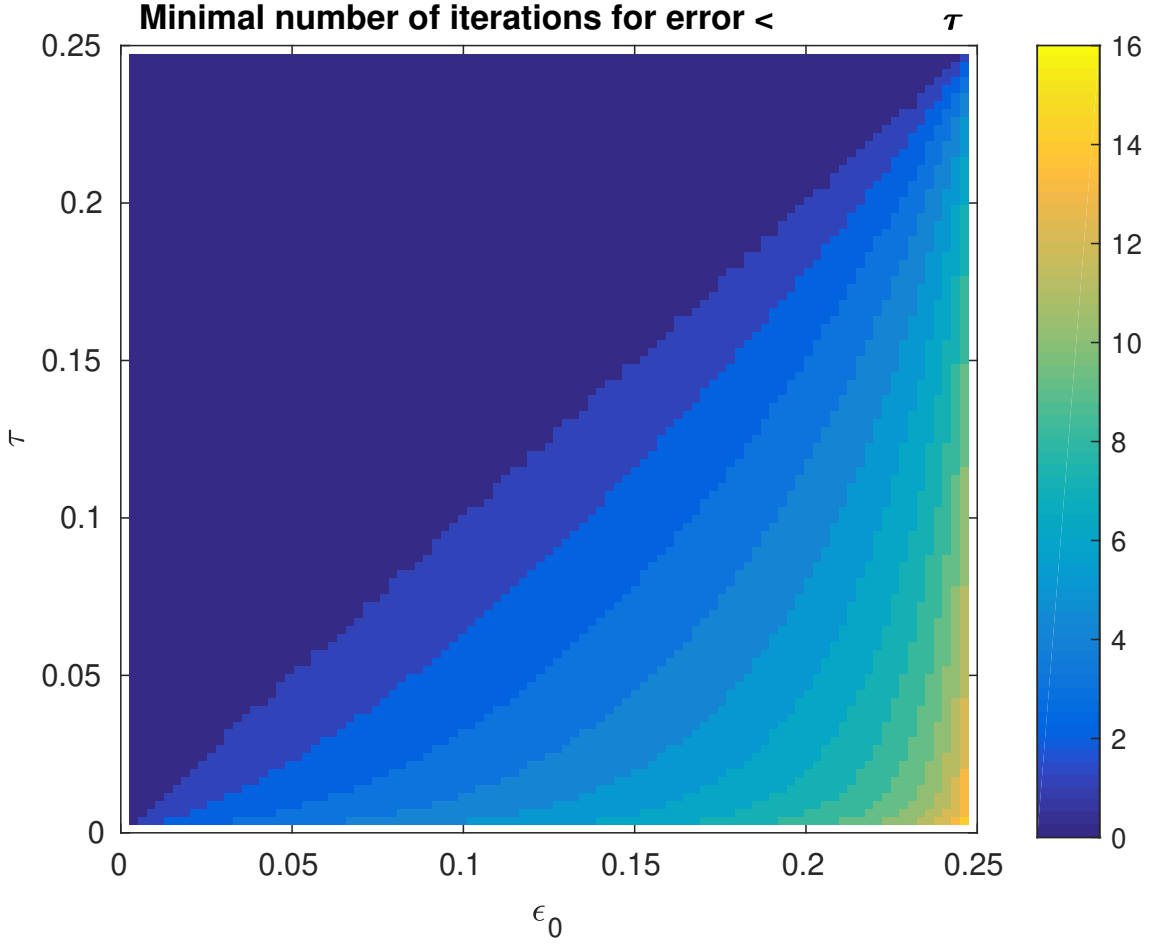
$$\ln\left(C^{\frac{1}{1-p}}\right) + p^k \cdot \underbrace{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}_{<0} < \ln\tau$$

$$k > \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}\right) \cdot \frac{1}{\ln(p)}, \qquad k_{\min} \in \mathbb{N}$$

$$k_{min} = \left\lceil \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}\right) \cdot \frac{1}{\ln(p)} \right\rceil$$

SOLUTION of (9-5.c):

The minimum number of iterations given ϵ_0 and τ is shown in Fig. 79, which was produced with the MATLAB code `k_min_plot.m`.



HINT 1 for (9-6.a): Consult [Lecture → Rem. 8.2.2.12] 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. 8.2.2.10] 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

```
2   int n = 10; // compute ten error terms
3   std::vector<double> e(n), loge(n);
4   e[0] = 1; // initial error
5   loge[0] = std::log(e[0]);
6   e[1] = 0.8;
7   loge[1] = std::log(e[1]);
8   for (int k = 1; k < n; ++k) {
9     e[k + 1] = e[k] * std::sqrt(e[k - 1]); // the error recursion
10    loge[k + 1] = std::log(e[k + 1]); // logarith of the error
11    std::cout << (loge[k + 1] - loge[k]) / (loge[k] - loge[k - 1]) << std::endl;
12  }
```

The order appear to be arrproximatively 1.36585.

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. (8.4.2.37)] in [Lecture → Rem. 8.4.2.33].

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.

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

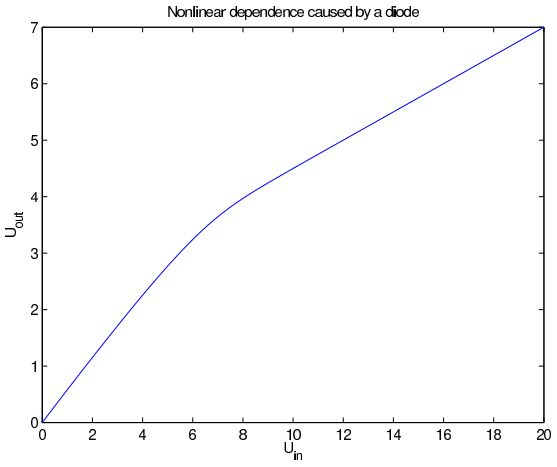
$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 3U_1 - U_2 - U_3 \\ 3U_2 - U_1 - U_3 - U \\ 3U_3 - U_1 - U_2 + \alpha e^{-\frac{\beta}{U_T}U} \cdot e^{\frac{\beta}{U_T}U_3} - \alpha \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad \text{for } \mathbf{u} = \begin{pmatrix} U_1 \\ U_2 \\ U_3 \end{pmatrix}.$$

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`.

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.



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

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} x^3 - 3xy^2 - 1 \\ 3x^2 y - y^3 \end{bmatrix} = \mathbf{0} \, , \quad \mathbf{x} := \begin{bmatrix} x \\ y \end{bmatrix} \, . \tag{9.8.3}$$

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 $\rightarrow$ Eq. (8.5.1.6)], where $\mathrm{D}\,\mathbf{F}$ is the Jacobian

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ 6xy & 3x^2 - 3y^2 \end{bmatrix} . \tag{9.8.5}$$

SOLUTION of (9-8.c):

We use the initialization of matrices via the << operator offered by EIGEN, see [Lecture → § 1.2.1.3].

C++ code 9.8.6: Evaluation of F

```
Vector2d juliaF(const Vector2d &z) {
  Vector2d Fz = Vector2d::Zero();
  // TODO: (9-8.c) 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;
}
```

Get it on GitLab (julia.hpp).

C++ code 9.8.7: Evaluation of Jacobian D F

```
Matrix2d juliaDF(const Vector2d &z) {
  Matrix2d DFz = Matrix2d::Zero();
  // TODO: (9-8.c) Implement the Jacobian of F at z.
  // START
  const double a = 3.0 * (z(0) * z(0) - z(1) * z(1));
  const double b = 6.0 * z(0) * z(1);
  DFz << a, -b, b, a;
  // END
  return DFz;
}
```

Get it on GitLab (julia.hpp).

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^{-3}.

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.
  constexpr double tol = 1e-4;
  constexpr unsigned int N_it = 20;

  // The image will have a resolution of res*res.
  constexpr unsigned 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);

  // TODO: (9-8.d) 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 (unsigned int i = 0; i < res; i++) {
    for (unsigned int j = 0; j < res; j++) {
      // Newton iteration starting at Z(i,j)=z=(x,y).
      const double x = -2 + 4.0 * i / (res - 1);
      const double y = -2 + 4.0 * j / (res - 1);
      Vector2d z(x, y);
      for (unsigned int k = 0; k < N_it; k++) {
        z -= juliaDF(z).lu().solve(juliaF(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 (unsigned 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");
}
```

Get it on GitLab (julia.hpp).

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 $|x_1^{(k)} + ix_2^{(k)} - z_\ell| < 10^{-3}$, $\ell \in \{1,2,3\}$, 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.

SOLUTION of (9-9.a):

❶ 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.

❷ Conversely, if $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$, then, by the recursion of the Newton method,

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})\,, \\ \mathbf{x}^{(k)} &= \mathbf{y}^{(k)} - \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{y}^{(k)})\,. \end{aligned}$$

Then we plug the first equation into the second and infer

$$\begin{aligned} &\mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)}) - \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})\mathbf{F}(\mathbf{x}^{(k)})) = \mathbf{0}\,, \\ \blacktriangleright\quad &\mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{F}(\mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})\mathbf{F}(\mathbf{x}^{(k)})) = \mathbf{0}\,. \end{aligned}$$

Finally, we exploit the injectivity (9.9.2) of $\mathbf{F}$:

$$\mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{F}(\mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})) \quad\Leftrightarrow\quad D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)}) = 0 \quad\Leftrightarrow\quad \mathbf{F}(\mathbf{x}^{(k)}) = 0\,.$$

SOLUTION of (9-9.b):

C++11-code 9.9.3: Implementation of `mod_newt_step_scalar`.

```
2  template <typename Scalar, class Function, class Jacobian>
3  Scalar mod_newt_step_scalar(const Scalar& x, Function&& f, Jacobian&& df) {
4    double return_value = 0.;
5    // TODO: (9-9.b) Compute a step of the modified Newton method for a
       scalar
6    // function
7    // START
8    Scalar y = x + f(x) / df(x);
9    return_value = y - f(y) / df(x);
10   // END
11   return return_value;
12 }
```

Get it on GitLab (modnewt.hpp).

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 `sample_nonlinear_solver()` 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 9.9.4: 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,
                             unsigned int max_itr = 100) {
  // TODO: (9-9.c,f) Implement this generic function that iteratively solves a
  // nonlinear equation.  The function should return whether the iteration
  // converged in the maximum number of iterations
  // START.
  // Temporary where to store new step
  Vector x_new = x;
  double r = 1;

  for (unsigned 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;
  // END
  return false;
}
```

Get it on GitLab (modnewt.hpp).

We employ the method discussed in [Lecture → Rem. 8.2.2.12] to estimate the order of convergence of the iteration. The estimator is

$$\texttt{estord} := \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}}, \quad \epsilon_k := \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|_2 . \tag{9.9.5}$$

C++11-code 9.9.6: Implementation of `mod_newt_ord`.

```
void mod_newt_ord(void) {
  // Setting up values, functions and jacobian
  constexpr double a = 0.123;
  auto f = [](double x) {
    return atan(x) - a;
  };                    // Function which sees a as it is constexpr
  double x_scalar = 5;  // initial guess

  // TODO: (9-9.c) Generate suitable terminal output to determine the order of
  // convergence of the modified Newton method applied to the scalar equation
  // that is given by f = 0.  You may want to implement the templated function
  // sample_nonlinear_solver for the iterations.
  // START
  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 step
  std::vector<double> sol;
  std::vector<double> err;

  // Compute error and push back to err, used in
  // sample_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
  sol.push_back(x_scalar);
  errf(x_scalar);

  // Lambda performing the next step, used to define a proper
  // function handle to be passed to sample_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
```

```
56    auto space = std::setw(15);
57    std::cout << space << "sol." << space << "err." << space << "order"
58              << std::endl;
59    for (unsigned i = 0; i < sol.size(); ++i) {
60      std::cout << space << sol.at(i) << space << err.at(i);
61      if (i >= 3) {
62        std::cout << space
63                  << (log(err.at(i)) - log(err.at(i - 1))) /
64                         (log(err.at(i - 1)) - log(err.at(i - 2)));
65      }
66      std::cout << std::endl;
67    }
68    // END
69  }
```

Get it on GitLab (modnewt.hpp).

solution	error	estord
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

The method converges very fast so that the iteration error quickly drops to the level of machine precision, where it is polluted by roundoff errors. So it is difficult to estimate the rate of convergence from the data.

We can just infer that the rate seems to be between 2 and 3.

HINT 1 for (9-9.d): Do not forget to perform a single LU-decomposition of the Jacobian and reuse it. See [Lecture → Code 8.5.3.6] for an example. ⌟

SOLUTION of (9-9.d):

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, Function&& f, Jacobian& df) {
  Vector return_value = x; // this is just a dummy, you should overwrite it
  // TODO: (9-9.d) Efficiently perform one step of the modified Newton method
  // for a system of equations.
  // START
  auto lu = df(x).lu();
  // Reusing LU decomposition
  Vector y = x + lu.solve(f(x));
  return_value = y - lu.solve(f(y));
  // END
  return return_value;
}
```

Get it on GitLab (modnewt.hpp).

SOLUTION of (9-9.e):

We can simply rely on component-wise partial derivatives to derive

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \mathbf{A} + \begin{bmatrix} c_1 e^{x_1} & & & \\ & c_2 e^{x_2} & & \\ & & \ddots & \\ & & & c_n e^{x_n} \end{bmatrix} \in \mathbb{R}^{n,n} \,. \tag{9.9.9}$$

SOLUTION of (9-9.f):

C++11-code 9.9.10: Implementation of `mod_newt_sys`.

```
Eigen::VectorXd mod_newt_sys(const Eigen::MatrixXd& A, const Eigen::VectorXd& c,
                             double tol = 1.0e-6, int maxit = 100) {
  Eigen::VectorXd x_system = Eigen::VectorXd::Zero(c.size());  // Initial guess

  // TODO: (9-9.f) Solve the non-linear system of equations (9.9.6) using the
  // modified Newton method.  You may want to use your implementation of
  // sample_nonlinear_solver.
  // START

  // Handler for function and jacobian in standard format.  Must see matrix A
  // and vector c.
  auto F = [&A, &c](const Eigen::VectorXd& x) {
    Eigen::VectorXd tmp =
        A * x + c.cwiseProduct(x.array().exp().matrix()).eval();
    return tmp;
  };
  auto dF = [&A, &c](const Eigen::VectorXd& x) {
    Eigen::MatrixXd C = A;
    Eigen::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](Eigen::VectorXd& x) {
    if (err.size() > 0)
      return err.back();
    else
      return 1.;
  };

  // Refactor (i.e. wrap) the step such that it is compatible with
  // sample_nonlinear_solver.  Must see F, dF and 'err'.  We also store the
  // error.
  auto newt_system_step = [&F, &dF,
                           &err](const Eigen::VectorXd& x) -> Eigen::VectorXd {
    Eigen::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, tol, maxit);
  // END
  return x_system;
}
```

Get it on GitLab (modnewt.hpp).

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 “stepping” function and reuse the previous codes.

We iterate the Modified Newton 5 times and print out the error for each step with corresponding order of convergence.

The order of convergence is estimate as recommended in [Lecture → Rem. 8.2.2.12], see (9.9.5).

C++11-code 9.9.11: Implementation of `mod_newt_sys`.

```
2  void mod_newt_sys_test(void) {
3    Eigen::MatrixXd A(4, 4);
4    A << 2., -1., 0., 0., -1., 2., -1., 0., 0., -1., 2., -1., 0., 0., -1., 2.;
5    Eigen::VectorXd c(4);
6    c << 1., 2., 3., 4.;
7
8    // TODO: (9-9.g) Determine the order of convergence from suitable
       tabulated
9    // values.
10   // determine a reference solution with very low tolerance
11   // (machine precision)
12   // START
13   constexpr double reference_tol = std::numeric_limits<double>::epsilon();
14   Eigen::VectorXd reference_sol = mod_newt_sys(A, c, reference_tol);
15
16   std::vector<double> err;
17   std::vector<Eigen::VectorXd> sol;
18
19   // seperate calculation and printing to not pollute the table from
       output of
20   // iteration function increase the number of maximal iterations
21   sol.push_back(Eigen::VectorXd::Zero(4));
22   err.push_back(reference_sol.norm());
23   for (unsigned k = 1; k < 5; ++k) {
24     sol.push_back(mod_newt_sys(A, c, 0., k));
25     err.push_back((reference_sol - sol.at(k)).norm());
26   }
27
28   // Perform convergence study with Modified newton for system of
       equations
29   std::cout << std::endl
30             << "*** Modified Newton method (system) ***" << std::endl
31             << std::endl;
32   std::cout << "Exact: " << reference_sol.transpose() << std::endl;
33
34   auto space = std::setw(15);
35   std::cout << space << "k" << space << "err." << space << "order" << std::endl;
36
37   // print iteration number, error and order
38   for (unsigned k = 0; k < 5; ++k) {
39     std::cout << space << k << space << err.at(k);
40     if (k >= 1 && k < 4) {
41       std::cout << space
42                 << (log(err.at(k + 1)) - log(err.at(k))) /
43                        (log(err.at(k)) - log(err.at(k - 1)));
44     }
45     std::cout << std::endl;
```

```
46    }
47    // END
48  }
```

Get it on GitLab (modnewt.hpp).

k	error	estord
0	2.87469	
1	0.359054	3.39385
2	0.000308353	2.74505
3	1.18145e-12	0.442692
4	2.22045e-16	

◁ Error table as printed by `mod_newt_sys()`.

As in Sub-problem (9-9.c) the method converges very fast and it is difficult to tease out a convergence order, because only few meaningful data points are generated before the iteration error becomes polluted by roundoff error.

The order of convergence seems to be > 2 consistent with what we observed in Sub-problem (9-9.c).