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

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2021, Version of October 2, 2022
(C) Seminar für Angewandte Mathematik, ETH Zürich

Link to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

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

0.1.2 Importance of Homework

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

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

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

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

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

Problem 2-16: Grid functions ⚀

This exercise deals with construction of sparse matrices from triplet format, see [Lecture → Section 2.7.2]. This task is relevant for applications like image processing and the numerical solution of partial differential equations.

Connected with [Lecture → Section 2.7]

Consider the following matrix $\mathbf{S} \in \mathbb{R}^{3,3}$:

$$\mathbf{S} := \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Denote by $s_{i,j}$ the entries of $\mathbf{S}$. Consider the following *linear* operator from $\mathbb{R}^{n,m}$ to $\mathbb{R}^{n,m}$:

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

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

(2-16.a) ⚀ (10 min.) Show that L is a linear operator.

SOLUTION 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.

▲

(2-16.b) ⚀ The space $\mathbb{R}^{n,m}$, as a vector-space, is isomorphic to $\mathbb{R}^{n\cdot m}$, via the mapping $\mathbf{X} \mapsto \mathrm{vec}(\mathbf{X})$, see [Lecture → Eq. (1.2.3.5)].

From linear algebra, we know that any linear operator over a finite dimensional (real) vector space can be represented by a (real) matrix multiplication. We define the matrix $\mathbf{A} \in \mathbb{R}^{nm,nm}$ s.t.

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

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

SOLUTION of (2-16.b):

The matrix $\mathbf{A}$ is given by:

$$\mathbf{A} = \begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix}$$

▲

(2-16.c) ⊡ (15 min.) Write a C++ function

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

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

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

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

SOLUTION 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
```

```
  // START
  auto f = [n, m](index_t i, index_t j) -> double {
    if (i > n / 4 && i < n * 3 / 4 && j > m / 4 && j < m * 3 / 4)
      return 1.;
    else
      return 0.;
  };
  return f;
  // END
}
```

Get it on GitLab (gridfun.hpp).

▲

(2-16.d) ⚂ (30 min.) [depends on Sub-problem (2-16.c)]

Implement a function

```
Eigen::SparseMatrix<double> build_matrix(const Eigen::Matrix3d & S,
    Eigen::Index Xn, Eigen::Index Xm);
```

which, given the stencil matrix $\mathbf{S} \in \mathbb{R}^{3,3}$ and the size of the matrix $\mathbf{X}$ (passed through `Xn,Xm`), builds and returns the sparse matrix $\mathbf{A}$ in CRS format. The matrix $\mathbf{A}$ can be built using an intermediate triplet/COO format.

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()`

```
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 $\mathbf{A}$

```
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 × 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) {
        for (index_t l = 0; l < 3; ++l) {
          if (i + k - 1 >= 0 && i + k - 1 <= size[0] - 1 && j + l - 1 >= 0 &&
              j + l - 1 <= size[1] - 1) {
            // The index of vec(X) w.r.t
            // the indices (i,j) of X
            const index_t I = to_vector_index(i, j, size);
            // Index on vec(X) relative to index
            // (i+k-2,j+l-2)
            // NOTE: C++ indexing starts from 0, so we change the
            // offset.
            const index_t J = to_vector_index(i + k - 1, j + l - 1, size);
            // Add triplet of corresponding stencil matrix entry
            triplets.push_back(Eigen::Triplet<double>(J, I, S(k, l)));
          }
        }
      }
    }
  }

  // Build from triplets and compress matrix (no need to compress through
  // makeCompressed(); already included in setFromTriplets()).
  A.setFromTriplets(triplets.begin(), triplets.end());
  // END

  return A;
}
```

Get it on GitLab (gridfun.hpp).

▲

(2-16.e) ⚂ (20 min.) Implement a function

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

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

SOLUTION of (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).

▲

(2-16.f) ⚄ (20 min.)

Implement a function

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

which, given a matrix $\mathbf{Y}$ and the sparse matrix $\mathbf{A}$, returns the matrix $\mathbf{X}$ s.t. $\mathrm{vec}(\mathbf{Y}) := \mathbf{A}\,\mathrm{vec}(\mathbf{X})$. You can use objects of type `Eigen::Map` to "reshape" a matrix into a column vector. Objects of this type are read and write compatible.

SOLUTION of (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).

▲

End Problem 2-16, 95 min.

Problem 2-17: Efficient sparse matrix-matrix multiplication in COO format

The *triplet (or COOrdinate) list format* works very well for defining a matrix or adding/modifying its elements. This is however not so true for matrix operations such as matrix-matrix multiplications, unlike the CSC/CSR storage.
This problem is about (sparse) matrix-matrix multiplication in COO format. We will consider the following items:

1. The worst case of sparse matrix-matrix multiplication, when one wants to use sparse matrix storage formats.
2. The most efficient way to tackle this problem with the COO format.
3. The asymptotic complexity of such algorithm (which is, by definition, the complexity under the worst-case scenario).

Related to [Lecture → Section 2.7.1]

For simplicity, in the following we will only deal with *binary* matrices. Binary matrices are only made of 0 or 1. At the same time, we will allow for duplicates in our implementations of the triplet format (see [Lecture → § 2.7.1.1]). The matrix entry associated to the pair (i,j) is defined to be the sum of all triplets corresponding to it.

For the subproblems involving coding, we define types `Trip = Triplet<double>` and `TripVec = std::vector<trip>`.

(2-17.a) ⚀ Consider multiplications between sparse matrices: $\mathbf{AB} = \mathbf{C}$. Is the product $\mathbf{C}$ guaranteed to be sparse? Make an example of two sparse matrices which, when multiplied, return a dense matrix.

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.

▲

(2-17.b) ⚀ Implement a C++ function which returns the input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ in COO format:

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

You can use EIGEN classes.

SOLUTION 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).

▲

(2-17.c) ⚀ Implement a C++ function which computes the product between two matrices in COO format:

```
TripVec COOprod_naive(const TripVec &A, const TripVec &B);
```

You can use EIGEN classes.

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)

```
2  TripVec COOprod_naive(const TripVec& A, const TripVec& B) {
3    TripVec C;
4
```

```
  // 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).

▲

(2-17.d) ⊡ What is the asymptotic complexity of your naive implementation in (2-17.c)?

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.

▲

(2-17.e) ⚄ Implement a C++ function which computes the product between two matrices in COO format in an efficient way:

```
TripVec COOprod_effic(TripVec &A, TripVec &B);
```

You can use EIGEN classes.

Can you do better than (2-17.c)?

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)

```
2  TripVec COOprod_effic(TripVec& A, TripVec& B) {
3    TripVec C;
4
5    // TODO: (2-17.e) Compute the matrix-matrix product efficiently
6    // START
7
8    // sort A along columns
9    std::sort(A.begin(), A.end(),
10             [](const Trip& a1, const Trip& a2) { return a1.col() < a2.col(); });
11   // and B along rows
12   std::sort(B.begin(), B.end(),
13             [](const Trip& b1, const Trip& b2) { return b1.row() < b2.row(); });
14
15   std::size_t i_A = 0, i_B = 0;
16   std::set<unsigned int> intersect;
17   while (i_A != A.size() && i_B != B.size()) {
18     if (A[i_A].col() < B[i_B].row()) {
19       ++i_A;
20     } else if (B[i_B].row() < A[i_A].col()) {
21       ++i_B;
22     } else {
23       intersect.insert(A[i_A].col());
24       ++i_A;
25       ++i_B;
26     }
27   }
28
29   // iteration along intersection of nonempty columns in A and rows in
       B
30   TripVec::iterator A_idx = A.begin();
31   TripVec::iterator B_idx = B.begin();
32   for (auto i : intersect) {
33     A_idx = std::find_if(A_idx, A.end(),
34                          [i](const Trip& a) { return a.col() == i; });
35     B_idx = std::find_if(B_idx, B.end(),
36                          [i](const Trip& b) { return b.row() == i; });
37
38     TripVec::iterator A_it;
39     TripVec::iterator B_it;
40     for (A_it = A_idx; A_it != A.end(); ++A_it) {
41       if (A_it->col() != i) {
42         break;
43       } else {
44         for (B_it = B_idx; B_it != B.end(); ++B_it) {
45           if (B_it->row() != i) {
46             break;
47           } else {
48             // actual multiplication
49             Trip triplet(A_it->row(), B_it->col(),
50                          A_it->value() * B_it->value());
51             C.push_back(triplet);
52           }
53         }
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).

▲

(2-17.f) What is the asymptotic complexity of your efficient implementation in (2-17.e)?

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.

▲

(2-17.g) Compare the timing of `COOprod_naive` (2-17.c) and `COOprod_effic` (2-17.e) for random matrices with different dimensions n. Perform the comparison twice: first only for products between sparse matrices, then for any kind of matrix.

HINT 1 for (2-17.g): The `main()` code to run your simulations is already provided in the template.

SOLUTION of (2-17.g):

Comparison of timings – sparse matrices

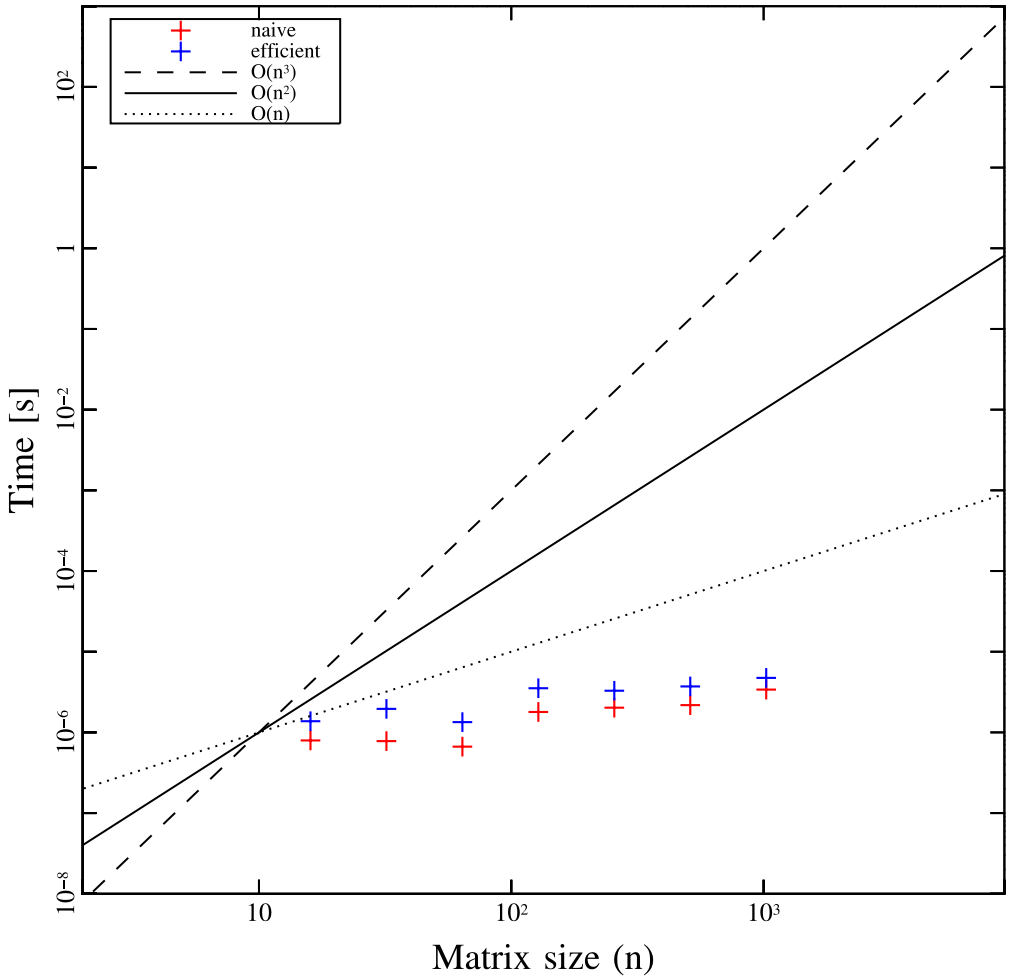


Fig. 9

Comparison of timings − sparse/dense matrices

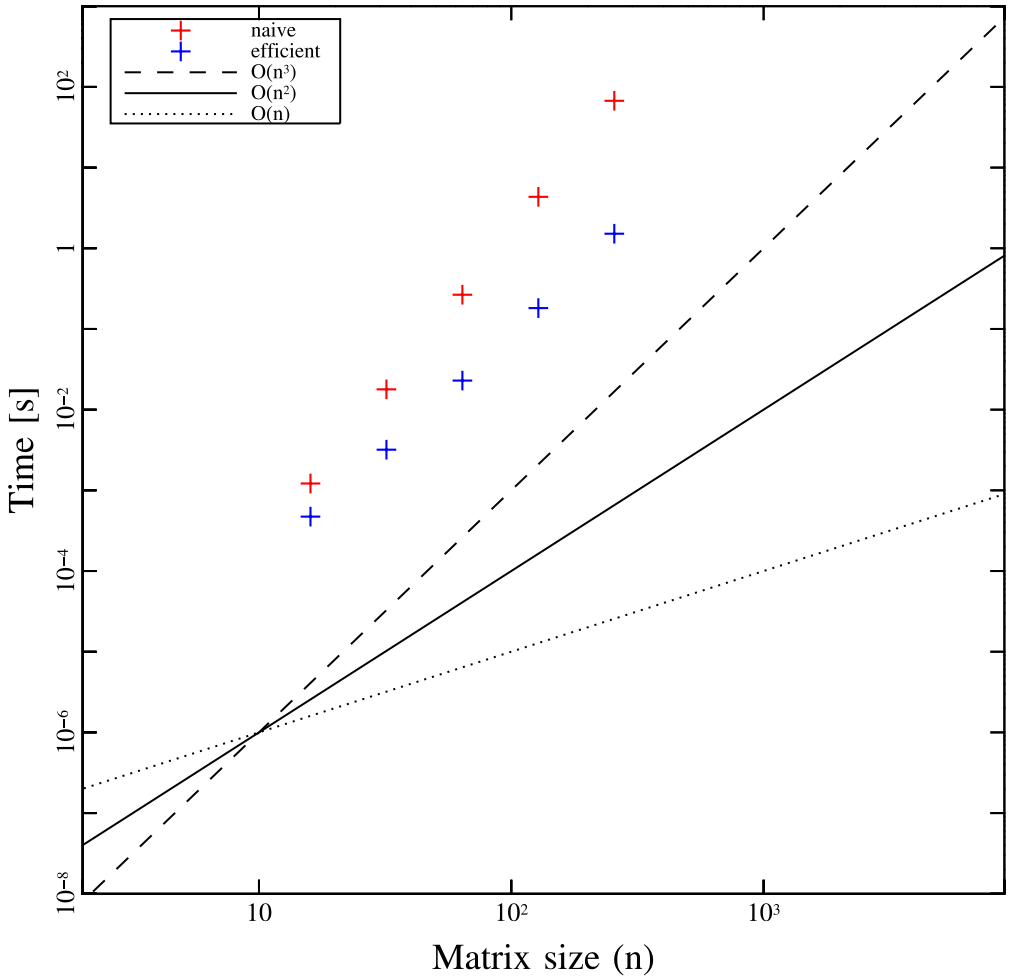


Fig. 10

Both the coded algorithms are faster than the standard matrix-matrix multiplications, as we do not compute the sum $\sum_{k=1}^{n} a_{ik}b_{kj}$ for any cell c_{ij} of the product matrix. The naive algorithm outperforms the efficient algorithm in the special case of sparse matrices: the product of sparse matrices can be dense (and therefore the number of common indices k_{common} can be high), as seen in (2-17.a). However, the efficient algorithm is generally better when considering any kind of matrix.

▲

End Problem 2-17

Problem 2-18: Gauss-Seidel Iteation for a Sparse Linear System of Equations

This problem studies are particular iterative method defined by a matrix and a vector, the so-called **Gauss-Seidel iteration**. We consider its algorithmic aspects when the matrix is given in a raw CRS format.

Familiarity with [Lecture → Section 2.7.1] is esssential for this problem. Some concepts from [Lecture → Section 8.2.2] will also be used. Implementation in C++ based on EIGEN will be requested.

Given the *regular* square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ and the vector $\mathbf{b} \in \mathbb{R}^n$, $n \in \mathbb{N}$, the **Gauss-Seidel iteration** $\mathbf{x}^{(k+1)} := \mathbf{\Phi}(\mathbf{x}^{(k)})$, $\mathbf{\Phi} : \mathbb{R}^n \to \mathbb{R}^n$, is defined as

$$\left(\mathbf{x}^{(k+1)}\right)_i = (\mathbf{A})_{i,i}^{-1}\left((\mathbf{b})_i - \sum_{j=1}^{i-1}(\mathbf{A})_{i,j}\left(\mathbf{x}^{(k+1)}\right)_j - \sum_{j=i+1}^{n}(\mathbf{A})_{i,j}\left(\mathbf{x}^{(k)}\right)_j\right), \quad i = 1,2,\ldots,n \text{ (in this order!)}. \tag{2.18.1}$$

Here sums for which the lower index is larger than the upper are meant to evaluate to zero.

The matrix $\mathbf{A}$ is available in a raw compressed row-storage (CRS) format, encapsulated in the following data type:

```
struct CRSMatrix {
  unsigned int m;                      // number of rows
  unsigned int n;                      // number of columns
  std::vector<double> val;             // value array
  std::vector<unsigned int> col_ind;   // same length as value array
  std::vector<unsigned int> row_ptr;   // length m+1, row_ptr[m] ==
     val.size()
};
```

Refer to [Lecture → § 2.7.1.4] for further explanations on the meaning of the fields of **CRSMatrix**.

(2-18.a) ⚀ (10 min.) For what square matrices $\mathbf{A}$ and vectors $\mathbf{b}$ is the Gauss-Seidel iteration (2.18.1) well-defined?

Iteration (2.18.1) well-defined $\forall \mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$: .

SOLUTION 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 $\forall \mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$: $(\mathbf{A})_{i,i} \neq 0 \quad \forall i \in \{1,\ldots,n\}$.

▲

(2-18.b) ⚂ (45 min.)

In the file `gaussseidelcrs.hpp` implement the C++ function

```
bool GaussSeidelstep_crs(
  const CRSMatrix &A, const Eigen::VectorXd &b,
  Eigen::VectorXd &x);
```

that carries out a single step of the Gauss-Seidel iteration according to (2.18.1). The vector $\mathbf{x}^{(k)}$ is passed through `x` and it should return $\mathbf{x}^{(k+1)}$ also in `x`. **false** is returned in case the iteration is not well-defined, **true** otherwise.

SOLUTION 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 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 (int i = 0; i < n; ++i) {
    if (A(i, i) == 0.0) return false;
    double s = b[i];
    for (int j = 0; j < i; ++j) s -= A(i, j) * x[j];
    for (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 (gaussseidelcrs.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");

```

```
  // START student code
  // 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.
    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)
      x[i] = s / Aii;
    else
      return false;
  }
  // END student code
  return true;
}
```

Get it on GitLab (gaussseidelcrs.hpp).

▲

(2-18.c) ⚀ (20 min.) For given regular $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{b} \in \mathbb{R}^n$ assume that the Gauss-Seidel iteration (2.18.1) is well-defined. Characterize the set of all **fixed points** of the Gauss-Seidel iteration (2.18.1) by a compact formula:

$$\{\mathbf{x}^* \in \mathbb{R}^n : \mathbf{x}^* \text{ is fixed point of (2.18.1)}\} = \quad .$$

A fixed point of an iteration $\mathbf{x}^{(k+1)} = F(\mathbf{x}^{(k)})$ is a vector $\mathbf{x}^*$ such that $\mathbf{x}^* = F(\mathbf{x}^*)$.

SOLUTION 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

$$(\mathbf{x}^*)_i = (\mathbf{A})_{i,i}^{-1}\left((\mathbf{b})_i - \sum_{j=1}^{i-1}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j - \sum_{j=i+1}^{n}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j\right), \quad i = 1,2,\ldots,n$$

$$\Leftrightarrow \quad (\mathbf{A})_{i,i}(\mathbf{x}^*)_i = (\mathbf{b})_i - \sum_{j=1}^{i-1}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j - \sum_{j=i+1}^{n}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j, \quad i = 1,2,\ldots,n$$

$$\Leftrightarrow \quad \sum_{j=1}^{n}(\mathbf{A})_{i,j}(\mathbf{x}^*)_j = (\mathbf{b})_i, \quad i = 1,2,\ldots,n$$

$$\Leftrightarrow \quad \mathbf{A}\mathbf{x}^* = \mathbf{b} \quad \text{!}.$$

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

▲

(2-18.d) ⚂ (30 min.) [depends on Sub-problem (2-18.b)]

Develop a C++ function

```
bool GaussSeidel_iteration(
  const CRSMatrix &A, const Eigen::VectorXd &b, Eigen::VectorXd &x,
  double atol = 1.0E-8, double rtol = 1.0E-6,
  unsigned int maxit = 100);
```

which carries out Gauss-Seidel iterations for the data $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$ (arguments `A` and `b`), and with initial guess $\mathbf{x}^{(0)}$ passed in `x`. The same argument is used to return the final iterate. The iteration is to terminate after `maxit` steps. If this happens, **false** should be returned, **true** otherwise.

The function should rely on the correction-based termination criterion [Lecture → Eq. (8.5.3.2)] with relative and absolute tolerance passed in `rtol` and `atol`, respectively and based on the Euclidean vector norm.

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     // START student code
15     if (!GaussSeidelstep_crs(A, b, x)) return false;
16     const double delta_x_norm = (x - x_old).norm();
17     const double x_norm = x.norm();
18     rec(x_norm, delta_x_norm);  //
19     if ((delta_x_norm < atol) || (delta_x_norm < rtol * x_norm)) {  //
```

```
20        return true;
21      }
22      x_old = x;
23      // END student code
24    }
25    // No termination
26    return false;
27  }
```

Get it on GitLab (gaussseidelcrs.hpp).

The stopping rule (2.18.5) is implemented in Line 19.

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");
6    assert(A.n == b.size() && "Vector b length mismatch");
7    assert(A.n == x.size() && "Vector x length mismatch");
8
9    // Main loop for Gauss-Seidel iteration
10   for (unsigned int k = 0; k < maxit; ++k) {
11     // START student code
12     double x_normsq = 0.0;
13     double delta_x_normsq = 0.0;
14     for (unsigned int i = 0; i < A.n; ++i) {
15       double Aii = 0.0;
16       double s = b[i];
17       for (unsigned int l = A.row_ptr[i]; l < A.row_ptr[i + 1]; ++l) {
18         const unsigned int j = A.col_ind[l];
19         if (j != i) {
20           s -= A.val[l] * x[j];
21         } else {
22     // Fetch diagonal entry of A
23           Aii = A.val[l];
24         }
25       }
26       if (Aii != 0.0) {
27         // Update iterate and compute Eucidean norm of current iterate
28         // and of correction
29         const double tmp = x[i];
30         x[i] = s / Aii;
31         delta_x_normsq += (x[i] - tmp) * (x[i] - tmp);
32         x_normsq += x[i] * x[i];
33       } else
34         return false;
35     }
36     // Correction based termination cirterion
37     if ((delta_x_normsq < atol * atol) ||
38         (delta_x_normsq < rtol * rtol * x_normsq)) {
39       return true;
40     }
41     // END student code
```

```
    }
    // No termination
    return false;
}
```

Get it on GitLab (gaussseidelcrs.hpp).

▲

(2-18.e) ⚄ (5 min.) [depends on Sub-problem (2-18.c)]

We apply the Gauss-Seidel iteration for

$$\mathbf{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 2 & 8 & 2 & 0 & 0 \\ 3 & 0 & 1 & 10 & 5 & 0 \\ 0 & 2 & 0 & 3 & 13 & 2 \\ 0 & 4 & 0 & 0 & 2 & 11 \end{bmatrix} \quad , \quad \mathbf{b} = \mathbf{0} \, .$$

and initial guess $\mathbf{x}^{(0)} = [1,2,3,4,5,6]^\top$. The following table displays the progress of the iteration:

k	$\left\|\mathbf{x}^{(k)}\right\|_2$	$\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\|$	$\left\|\mathbf{x}^{(k)}\right\|_2 : \left\|\mathbf{x}^{(k-1)}\right\|_2$
1	3.863865e+00	1.127304e+01	
2	8.297690e-01	3.751824e+00	2.147510e-01
3	6.432366e-02	7.857529e-01	7.751996e-02
4	1.476504e-02	5.984439e-02	2.295429e-01
5	4.068577e-03	1.098126e-02	2.755548e-01
6	9.204751e-04	3.155445e-03	2.262401e-01
7	1.880926e-04	7.327086e-04	2.043429e-01
8	3.636497e-05	1.517466e-04	1.933355e-01
9	6.803531e-06	2.956273e-05	1.870902e-01
10	1.246867e-06	5.556761e-06	1.832676e-01
11	2.254491e-07	1.021425e-06	1.808125e-01
12	4.039721e-08	1.850525e-07	1.791855e-01
13	7.194116e-09	3.320315e-08	1.780844e-01
14	1.275722e-09	5.918398e-09	1.773286e-01

Describe qualitatively and quantitatively the observed *asymptotic* convergence of the iteration for this example.

SOLUTION 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.

▲

End Problem 2-18 , 110 min.

Problem 2-19: A special Sylvester Equation

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

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

which yields a linear system of equations for the entries of the unknown matrix $\mathbf{X} \in \mathbb{R}^{n,n}$. In this exercise we consider a very special specimen of Sylvester equation and recast it for the sake of efficient direct solution with EIGEN.

Related to various topics from class and also Problem 2-8.

(2-19.a) ⚀ (5 min.) Refresh your knowledge about s.p.d. matrices, see [Lecture → Def. 1.1.2.6] and [Lecture → Lemma 1.1.2.7]. ▲

(2-19.b) ⚁ (5 min.) Recall the "vectorization" $\mathrm{vec}(\mathbf{X}) \in \mathbb{R}^{n^2}$ of a matrix $\mathbf{X} \in \mathbb{R}^{n,n}$ from [Lecture → Rem. 1.2.3.4] and jot down $\mathrm{vec}(\mathbf{X})$ for

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

SOLUTION of (2-19.b):

The vectorized matrix is obtained by stacking the columns of the matrix on top of each other:

$$\mathrm{vec}(\mathbf{X}) = [x_{11}, x_{21}, x_{12}, x_{22}]^\top \in \mathbb{R}^4 \, .$$

▲

(2-19.c) ⚂ (15 min.) For $\mathbf{S}, \mathbf{T} \in \mathbb{R}^{2,2}$ find the matrix $\mathbf{C} \in \mathbb{R}^{4,4}$ such that

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

that is, express the entries of $\mathbf{C}$ in terms of the entries $(\mathbf{S})_{i,j}$ and $(\mathbf{T})_{i,j}$ of $\mathbf{S}$ and $\mathbf{T}$.

SOLUTION 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

$$\mathbf{C}_{:,1} = \mathrm{vec}\left(\begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \right) = \mathrm{vec}\left(\begin{bmatrix} s_{11} & 0 \\ s_{21} & 0 \end{bmatrix} \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \right)$$

$$= \mathrm{vec}\left(\begin{bmatrix} s_{11}t_{11} & s_{11}t_{12} \\ s_{21}t_{11} & s_{21}t_{12} \end{bmatrix} \right) = \begin{bmatrix} s_{11}t_{11} \\ s_{21}t_{11} \\ s_{11}t_{12} \\ s_{21}t_{12} \end{bmatrix} \, .$$

Thus we find

$$\mathbf{C} = \begin{bmatrix} s_{11}t_{11} & s_{12}t_{11} & s_{11}t_{21} & s_{12}t_{21} \\ s_{21}t_{11} & s_{22}t_{11} & s_{21}t_{21} & s_{22}t_{21} \\ s_{11}t_{12} & s_{12}t_{12} & s_{11}t_{22} & s_{12}t_{22} \\ s_{21}t_{12} & s_{22}t_{12} & s_{21}t_{22} & s_{22}t_{22} \end{bmatrix} = \mathbf{T}^\top \otimes \mathbf{S} \,.$$

▲

(2-19.d) ⚄ (20 min.) Based on the **Kronecker product** $\otimes$ from [Lecture → Def. 1.4.3.7] write down a matrix $\mathbf{C} \in \mathbb{R}^{n^2,n^2}$ depending on the given square matrices $\mathbf{S},\mathbf{T} \in \mathbb{R}^{n,n}$ such that

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

HINT 1 for (2-19.d): Read **[LAN70]**. ⌟

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

$$(\mathbf{SE}_{k,m}\mathbf{T})_{i,j} = s_{i,k}t_{m,j} \,. \tag{2.19.2}$$

Write $P(i,j) \in \{1,\ldots,n^2\}$ for the **p**osition of the matrix entry $(\mathbf{Z})_{i,j}$ in the vector $\operatorname{vec}(Z)$, $\mathbf{Z} \in \mathbb{R}^{n,n}$, $i,j \in \{1,\ldots,n\}$. Hence, from (2.19.2) we conclude

$$(\mathbf{C})_{P(i,j),P(k,m)} = s_{i,k}t_{m,j} = \left(\mathbf{T}^\top \otimes \mathbf{S}\right)_{P(i,j),P(k,m)} \quad \Leftrightarrow \quad \boxed{\mathbf{C} = \mathbf{T}^\top \otimes \mathbf{S}} \,. \tag{2.19.3}$$

▲

Throughout the remainder of this problem let $\mathbf{A} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, stand for a *symmetric, positive definite* (s.p.d.) matrix, whose rows and columns contain at most five non-zero entries. Hence, for large n the matrix $\mathbf{A}$ can be considered *sparse* in the sense of [Lecture → Notion 2.7.0.1].

Then consider the special Sylvester equation:

$$\text{seek} \quad \mathbf{X} \in \mathbb{R}^{n,n}: \quad \mathbf{XA}^{-1} + \mathbf{AX} = \mathbf{I} \,, \tag{2.19.4}$$

and the equivalent matrix equation:

$$\text{seek} \quad \mathbf{X} \in \mathbb{R}^{n,n}: \quad \mathbf{X} + \mathbf{AXA} = \mathbf{A} \,. \tag{2.19.5}$$

(2-19.e) ⚀ (5 min.) Both (2.19.4) and (2.19.5) can be recast as an $n^2 \times n^2$ linear system of equations for the unknown vector $\mathrm{vec}(\mathbf{X})$. Based on the result of Sub-problem (2-19.d) express their coefficient matrices by means of the Kronecker product [Lecture → Def. 1.4.3.7].

SOLUTION 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 $\mathbf{A}$.

▲

(2-19.f) ⚄ (10 min.) Based on the result of Sub-problem (2-19.e) and in light of the observation made in [Lecture → Ex. 2.7.4.4] give a sharp upper bound for the number of non-zero entries of the coefficient matrices of those linear systems for (2.19.4) and (2.19.5).

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

$$\mathrm{nnz}(\mathbf{A}) \leq 5n \,.$$

However, as explained in the hint, $\mathrm{nnz}(\mathbf{A}^{-1}) = n^2$ is possible. Moreover, for the Kronecker product we have the simple estimate

$$\mathrm{nnz}(\mathbf{S} \otimes \mathbf{T}) \leq \mathrm{nnz}(\mathbf{S}) \cdot \mathrm{nnz}(\mathbf{T}) \,. \tag{2.19.8}$$

Combining all these estimates gives us

$$\mathrm{nnz}(\mathbf{A}^{-1} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{A}) \leq n^3 + 5n^2 \,,$$
$$\mathrm{nnz}(\mathbf{I} \otimes \mathbf{I} + \mathbf{A} \otimes \mathbf{A}) \leq 26n^2 \,.$$

Hence, (2.19.5) features a much sparser coefficient matrix than (2.19.4).

▲

(2-19.g) ⚀ (15 min.) Show that both (2.19.4) and (2.19.5) give rise to linear systems of equations for the entries of $\mathbf{X}$ with *s.p.d.* coefficient matrices.

For your argument you can appeal to the following result:

Theorem 2.19.9. Eigenvalues of Kronecker-product matrices

Assume that both $\mathbf{S},\mathbf{T} \in \mathbb{R}^{n,n}$ can be diagonalized and write $\sigma(\mathbf{S}),\sigma(\mathbf{T}) \subset \mathbb{C}$ for the sets of their eigenvalues. Then the set of eigenvalues of the Kronecker product is given by

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

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 → Lemma 1.1.2.7], these matrices are s.p.d.

▲

(2-19.h) ⚁ (15 min.) Temporarily assume that $\mathbf{A}$ is a *diagonal* matrix. Implement a C++ function

```
template <typename Vector>
  Eigen::SparseMatrix<double> solveDiagSylvesterEq(const Vector
     &diagA);
```

that solves (2.19.4) with the diagonal of $\mathbf{A}$ passed through the vector argument `diagA`. The result $\mathbf{X}$ is returned as a sparse matrix. The type **Vector** must supply a `size()` method and component access through **operator** `[]`.

SOLUTION 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).

▲

(2-19.i) ⚀ (45 min.) Without using EIGEN's (beta-status) implementation of the Kronecker product, realize a C++ function

```
Eigen::SparseMatrix<double> sparseKron(const
   Eigen::SparseMatrix<double> &M);
```

that computes the Kronecker product $\mathbf{M} \otimes \mathbf{M}$ for the square sparse matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ and returns the result in sparse-matrix format.

For the sake of efficient initialization you have to use the `reserve()` function and determine the maximum number of non-zero entries per column for $\mathbf{A}$ in advance. To do this use the member functions of **SparseMatrix** that allow direct access to the data vectors of the CCS format, *cf.* [Lecture → § 2.7.1.4]:

- **const double** `*valuePtr()` **const**: returns a pointer to the array of values of non-zero matrix entries,
- **const** `StorageIndex* innerIndexPtr()` **const**: returns a pointer to the array of row indices for every non-zero matrix entry,
- **const** `StorageIndex* outerIndexPtr()` **const**: returns a pointer to the vector of starting positions of columns.

These access methods also permit you to run through all non-zero entries of $\mathbf{A}$ and retrieve their locations.

SOLUTION 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 $\mathbf{M}$

```
2  Eigen::SparseMatrix<double> sparseKron(const Eigen::SparseMatrix<double> &M) {
3    int n = M.rows();
4    assert(n == M.cols() && "Matrix M must be square.");
5
6    // JM is an array of size n+1, with
7    // JM[j] = number of non-zero entries in all columns 0,..., (j-1) of M.
8    const int *JM = M.outerIndexPtr();
9
10   // Define nnzColwiseM(j) = nnz( M.col(j) )
11   Eigen::VectorXi nnzColwiseM(n);
12   for (int j = 0; j < n; j++) {
13     nnzColwiseM(j) = JM[j + 1] - JM[j];
14   }
15
16   // B will be the Kronecker product of M with itself.
17   Eigen::SparseMatrix<double> B(n * n, n * n);
18
19   // TODO: (2-19.i) Fill in the entries of B.
20   // Hint: You can use nnzColwiseM when reserving space for B.
21   // Use M.valuePtr() and M.innerIndexPtr() to define arrays similar
22   // to JM. Use those arrays to access the non-zero entries of M.
23
24   // START
25
26   Eigen::VectorXi nnzColwiseB;
27   // We have that nnz(B.col(j*n+l)) = nnz(M.col(j)) * nnz(M.col(l)).
28   Eigen::MatrixXi tmp = nnzColwiseM * nnzColwiseM.transpose();  // O(n^2)
29   nnzColwiseB = Eigen::MatrixXi::Map(tmp.data(), n * n, 1);
30
31   // Reserve sufficient space.
32   B.reserve(nnzColwiseB);
33
34   // We have that B.block(i*n,j*n,n,n) = M(i,j) * M,
35   // i.e. B( i*n + k, j*n + l ) = M(i,j)*M(k,l).
36   // Hence, we loop over i,j,k,l.
37
38   // Row indices of non-zero entries (size = nnz):
39   const int *IM = M.innerIndexPtr();
40   // Values of non-zero entries (size = nnz):
41   const double *valM = M.valuePtr();
42
43   // Assuming nnz in each column is bounded by a constant, the below is O(n^2).
44   // Loop over the column index j:
45   for (int j = 0; j < n; j++) {
46     // Loop over the non-zero entries in column j:
47     for (int s = JM[j]; s < JM[j + 1]; s++) {
48       int i = IM[s];          // Row index of non-zero entry #s.
49       double Mij = valM[s];   // M(i,j) = Mij
50
51       // 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).

▲

(2-19.j) ⚂ (20 min.) [depends on Sub-problem (2-19.i), Sub-problem (2-19.e)]

Write a C++ function

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

that solves (2.19.4) and returns the solution $\mathbf{X}$.

SOLUTION of (2-19.j):

C++ code 2.19.12: Solving Sylvester equation (2.19.4).

```
2  Eigen::MatrixXd solveSpecialSylvesterEq(const Eigen::SparseMatrix<double> &A) {
3    const int n = A.rows();
4    assert(n == A.cols() && "Matrix A must be square.");
5    Eigen::MatrixXd X(n, n);
6
7    // TODO: (2-19.j) Solve the equation X*A^{-1} + A*X = I .
8
9    // START
10   // The equation is equivalent to X + A*X*A = A,
11   // which has the system matrix C = kron(I,I) + kron(A,A).
12   // Define left hand side of C*Vec(X) = b.
13   Eigen::SparseMatrix<double> C(n * n, n * n);
14   C.setIdentity();
15   // May involve expensive copying of data?
16   C += sparseKron(A);
17
18   // Define right hand side.
19   // Do not use a "sparse vector"!
20   Eigen::VectorXd b(n * n);
21   b.setZero();
22   const int *JA = A.outerIndexPtr();
23   const int *IA = A.innerIndexPtr();
24   const double *valA = A.valuePtr();
25   // Form b by stacking the columns of A.
26   // Iterate over columns of A:
27   for (int j = 0; j < n; j++) {
28     // Loop over the non-zero entries in column j:
```

```
29      for (int s = JA[j]; s < JA[j + 1]; s++) {
30        const int i = IA[s];      // Row index of non-zero entry #s.
31        b[j * n + i] = valA[s];   // b(j*n+i) = Aij
32      }
33    }
34    // Call sparse direct solver.
35    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
36    // Since C is s.p.d., we could also use
37    // SimplicialLLT<Eigen::SparseMatrix<double>> solver;
38    solver.compute(C);
39    // x = Vec(X)
40    Eigen::VectorXd x;
41    x = solver.solve(b);
42    X = Eigen::MatrixXd::Map(x.data(), n, n);
43    // END
44
45    return X;
46  }
```

Get it on GitLab (Sylvester.hpp).

▲

End Problem 2-19 , 155 min.

Problem 2-20: Matrix-Vector Product in CRS format

Sparse matrices have to be stored in special formats in order to save memory and inform algorithms about the position of non-zero entries. This problem will examine the CRS format

This problems is related to [Lecture → § 2.7.1.4] and assumes familiarity with C++.

The following data structure is used to store an $m \times n$-matrix in compressed row-storage (CRS) format:

```
struct CRSMatrix {
  unsigned int m;                       // number of rows
  unsigned int n;                       // number of columns
  std::vector<double> val;              // value array
  std::vector<unsigned int> col_ind;    // same length as value array
  std::vector<unsigned int> row_ptr;    // length m+1, row_ptr[m] ==
     val.size()
};
```

This format is defined by the relationship ("C++ indexing")

$$\texttt{val}[k] = a_{ij} \quad \Leftrightarrow \quad \begin{cases} \texttt{col\_ind}[k] = j\,, \\ \texttt{row\_ptr}[i] \leq \texttt{k} < \texttt{row\_ptr}[i+1]\,, \end{cases} \qquad 0 \leq k \leq \texttt{val.size()}\,,$$

for $i \in \{0,\ldots,m-1\}$, $j \in \{0,\ldots,n-1\}$.

(2-20.a) ⚀ (40 min.) The function `crsmv` is supposed to return the product of a matrix in CRS format passed through `M` and of a vector given as argument `x`. Supplement the missing parts of the following listing code by writing valid C++ code into the boxes.

```
template <typename VECTORTYPE_I, typename VECTORTYPE_II>
VECTORTYPE_I crsmv(const CRSMatrix &M,
                   const VECTORTYPE_II &x) {
  assert((x.size() == M.          )
         && "Size mismatch between x and M");
  VECTORTYPE_I y(M.m);
  for (int k = 0; k <           ; ++k) {
    y[k] = 0;
    for (int j = M.           ; j <           ; ++j) {
      y[           ] += M.                    * x[                    ];
    }
  }
  return y;
}
```

SOLUTION 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()`

```
template <typename VECTORTYPE_I, typename VECTORTYPE_II>
VECTORTYPE_I crsmv(const CRSMatrix &M, const VECTORTYPE_II &x) {
  assert((x.size() == M.n) && "Size mismatch between x and M");
  VECTORTYPE_I y(M.m);
  for (int k = 0; k < M.m; ++k) {
    y[k] = 0;
    for (int j = M.row_ptr[k]; j < M.row_ptr[k + 1]; ++j) {
      y[k] += M.val[j] * x[M.col_ind[j]];
    }
  }
  return y;
}
```

▲

End Problem 2-20 , 40 min.

Chapter 3

Direct Methods for Linear Least Squares Problems

Problem 3-1: Hidden linear regression

This problem is about a hidden linear regression: in fact, at first glance it will seem that we are dealing with a nonlinear system of equations. However, we will be able to reduce the system to a linear form.

Linear regression was introduced in [Lecture → Ex. 3.0.1.1] and [Lecture → Ex. 3.0.1.4]. You have to be familiar with normal equations, see [Lecture → Section 3.1.2] and, in particular, [Lecture → Ex. 3.1.2.4]. This problems also addresses the solution of linear least-squares problems in EIGEN, see [Lecture → Code 3.2.0.1] and [Lecture → Code 3.3.4.2].

We consider the function $f(t) = \alpha e^{\beta t}$ with unknown parameters $\alpha > 0$, $\beta \in \mathbb{R}$. Given are measurements (t_i, y_i), $i = 1, \ldots, n$, $2 < n \in \mathbb{N}$. We want to determine $\alpha, \beta \in \mathbb{R}$ such that $f(t)$ fulfills the following condition "to the best of its ability" (in the least square sense):

$$f(t_i) = y_i, \quad \text{with} \quad y_i > 0,\ i = 1, \ldots, n\ . \tag{3.1.1}$$

(3-1.a) ⊡ (10 min.) Which overdetermined system of *nonlinear* equations directly stems from (3.1.1)?

SOLUTION 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.

▲

(3-1.b) ⚂ (15 min.) In which overdetermined system of *linear* equations can you transform the nonlinear system derived in (3-1.a)?

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:

$$g(t) = \log f(t) = \log \alpha + \log e^{\beta t} = a + \beta t \; .$$

Moreover, we impose $b_i := \log y_i$.

The new problem is therefore expressed by the following overdetermined linear system:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \text{with } \mathbf{A} = \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \vdots \\ 1 & t_n \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} a \\ \beta \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \log y_1 \\ \log y_2 \\ \vdots \\ \log y_n \end{bmatrix} .$$

▲

(3-1.c) ⚀ (15 min.) Determine the **normal equations** for the overdetermined linear system of (3-1.b).

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 $\alpha = e^a$ and β in function f.

▲

(3-1.d) ⚄ (20 min.) Consider the data $(t_i, y_i) \in \mathbb{R}^2$, $i = 1, \ldots, m$, stored in two vectors `t` and `y` of equal length. In the file `adaptedlinreg.hpp` implement a C++ function

```
Eigen::VectorXd linReg(const Eigen::VectorXd &t, const
   Eigen::VectorXd &y);
```

that uses the *method of normal equations* discussed in [Lecture → Section 3.2] to solve the 1D linear regression problem as described in [Lecture → Ex. 3.0.1.1] in least-squares sense. This function should return the estimated parameters α and β in a vector of length 2.

SOLUTION 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) {
```

```
  // TODO: (3-1.d) Use the method of normal equations to solve the 1D linear
  // regression problem in least-square sense.
  // Note:  The tests anticipate the outputs in the order (alpha, beta), and not
  // (beta, alpha).

  // START
  assert(t.size() == y.size() && "t and y must have same size");

  // Coefficient matrix of overdetermined linear system.
  Eigen::MatrixXd A(t.size(), 2);
  A.col(0).setOnes();
  A.col(1) = t;

  // Normal equation
  Eigen::MatrixXd lhs = A.transpose() * A;  // Left-hand side
  Eigen::VectorXd rhs = A.transpose() * y;  // Right-hand side

  // Intercept and slope.
  Eigen::VectorXd x = lhs.lu().solve(rhs);

  // END

  return x;
}
```

Get it on GitLab (adaptedlinreg.hpp).

▲

(3-1.e) ⚀ (15 min.) [depends on Sub-problem (3-1.d), Sub-problem (3-1.a)]

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

```
Eigen::VectorXd expFit(const Eigen::VectorXd &t, const Eigen::VectorXd &y);
```

that returns a least squares estimate of α and β, given the nonlinear relationship (3.1.1) between $\mathbf{t}$ and $\mathbf{f}$:

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)

```
Eigen::VectorXd expFit(const Eigen::VectorXd &t, const Eigen::VectorXd &y) {
  // TODO: (3-1.e) Implement least square estimate of alpha and beta using
  // the previously implemented function linReg().
  // Note:  You don't need to have implemented linReg() to solve this subproblem.
  // The tests will compile and use the master solution for linReg().
  // Note:  The tests anticipate the outputs in the order (alpha, beta), and not
  // (beta, alpha).
```

```
9
10    // START
11    assert(t.size() == y.size() && "t and y must have same size");
12
13    // Transform to a linear equation
14    // b := log(y) = x(0) + x(1)*t
15    // We use array() for coefficient-wise log() and matrix() to
          transform it
16    // back to a vector.
17    Eigen::VectorXd b = y.array().log().matrix();
18
19    // Solve and transform back to the nonlinear equation
20    // y = exp(x(0)) * exp(x(1)*t) = alpha*exp(beta*t)
21    Eigen::VectorXd x = linReg(t, b);
22    x(0) = exp(x(0));  // Eigen's coefficient-wise exp()
23
24    // END
25
26    return x;
27  }
```

Get it on GitLab (adaptedlinreg.hpp).

▲

End Problem 3-1 , 75 min.

Problem 3-2: Estimating a Tridiagonal Matrix

In [Lecture → Section 3.1] we learned that to determine the least squares solution of an overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ we minimize the residual norm $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$ w.r.t. $\mathbf{x}$. However, we also face a linear least squares problem when minimizing the residual norm w.r.t. the entries of $\mathbf{A}$. This is the unusual situation considered in this problem.

This is an exercise in converting a problem statement into a proper linear least squares problem in the form of an overdetermined linear system of equations as in [Lecture → Section 3.0.1] and then solving it through the normal equations, see [Lecture → Section 3.2].

The following least-squares problem arises in the estimation of material parameters in one-dimensional diffusion problems: Let two vectors $\mathbf{z}, \mathbf{c} \in \mathbb{R}^n$, $n > 2 \in \mathbb{N}$ of measured values be given. The two real numbers α^* and β^* are defined as:

$$(\alpha^*, \beta^*) = \operatorname*{argmin}_{\alpha,\beta\in\mathbb{R}} \left\| \mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} \right\|_2 , \tag{3.2.1}$$

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

$$\mathbf{T}_{\alpha,\beta} = \begin{bmatrix} \alpha & \beta & 0 & \dots & 0 \\ \beta & \alpha & \beta & \ddots & \vdots \\ 0 & \beta & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \alpha & \beta \\ 0 & \dots & 0 & \beta & \alpha \end{bmatrix} . \tag{3.2.2}$$

(3-2.a) ⚄ (20 min.) Reformulate (3.2.1) as a linear least squares problem in standard form according to [Lecture → Def. 3.1.1.1]

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^k} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \tag{3.2.3}$$

For appropriate choice of $m, k \in \mathbb{N}$ define suitable $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{x} \in \mathbb{R}^k$ and $\mathbf{b} \in \mathbb{R}^m$, with $m, k \in \mathbb{N}$.

HINT 1 for (3-2.a): Compute the matrix×vector product $\mathbf{T}_{\alpha,\beta}\mathbf{z}$ for using $\mathbf{z} = [z_1, z_2, \dots, z_n]^\top$. Express the resulting vector as the product of a matrix and the vector $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$. ⌟

SOLUTION of (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 **??** can be written as:

$$\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} = \begin{bmatrix} \alpha z_1 + \beta z_2 \\ \alpha z_2 + \beta(z_1 + z_3) \\ \dots \\ \alpha z_{n-1} + \beta(z_{n-2} + z_n) \\ \alpha z_n + \beta z_{n-1} \end{bmatrix} - \mathbf{c} = \begin{bmatrix} z_1 & z_2 \\ z_2 & z_1 + z_3 \\ \vdots & \vdots \\ z_{n-1} & z_{n-2} + z_n \\ z_n & z_{n-1} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} - \mathbf{c} =: \mathbf{A}\mathbf{x} - \mathbf{b} \tag{3.2.4}$$

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.

▲

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

Explicitly state the normal equations [Lecture → Thm. 3.1.2.1] associated with the linear least-squares problem (3.2.3).

SOLUTION 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

$$\mathbf{A}^\top \mathbf{A} = \begin{bmatrix} \sum_{\ell=1}^{n} z_\ell^2 & 2\sum_{\ell=1}^{n-1} z_\ell z_{\ell+1} \\ 2\sum_{\ell=1}^{n-1} z_\ell z_{\ell+1} & z_1^2 + 2\sum_{\ell=2}^{n-1}\left(z_\ell^2 + z_{\ell-1}z_{\ell+1}\right) + z_n^2 \end{bmatrix} \in \mathbb{R}^{2,2}\,.$$

▲

(3-2.c) ⚂ (30 min.) [depends on Sub-problem (3-2.b)]

Write a C++ function

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

that computes the optimal (in least-squares sense) parameters α^* and β^* according to **??** from data vectors `z` and `c` (i.e. $\mathbf{z}$ and $\mathbf{c}$ from **??**). Use the *method of normal equations*.

HINT 1 for (3-2.c): Similarly to Problem 3-1, $\mathbf{A}\mathbf{x} = \mathbf{b}$ 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 $\mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b}$ 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).

▲

End Problem 3-2 , 65 min.

Problem 3-3: Linear Data Fitting

Abstract linear data fitting and its connection to the linear least squares problem [Lecture → Eq. (3.1.3.7)] is discussed in [Lecture → Ex. 3.0.1.1]. In this problem we practise these techniques for a concrete example.

This problem relies on the methods and EIGEN facilities introduced in [Lecture → Section 3.2] and [Lecture → Section 3.3.4].

For certain points in time t_i we have measured data f_i for a physical quantity f whose time-dependence $t \mapsto f(t)$ is only known qualitatively, that is, up to a few unknown parameters, see [Lecture → Ex. 3.0.1.1]:

t_i	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
f_i	100	34	17	12	9	6	5	4	4	2

$f(t)$ is assumed to be a linear combination

$$f(t) = \sum_{j=1}^{4} \gamma_j \phi_j(t) \tag{3.3.1}$$

of the functions

$$\phi_1(t) = \frac{1}{t}, \quad \phi_2(t) = \frac{1}{t^2}, \quad \phi_3(t) = \mathrm{e}^{-(t-1)}, \quad \phi_4(t) = \mathrm{e}^{-2(t-1)} . \tag{3.3.2}$$

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

$$\sum_{i=1}^{10} (f(t_i) - f_i)^2 \longrightarrow \min . \tag{3.3.3}$$

(3-3.a) ⚀ (30 min.) In the file `linfit.hpp` write a C++ function

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

that computes a least squares estimate for the coefficients γ_i by related to the data (3.3.2) and (3.3.3) by means of solving the *normal equations*.

SOLUTION of (3-3.a):

We have to recast (3.3.3) as a least squares problem arising from an overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$. We find

$$\sum_{i=1}^{10} (f(t_i) - f_i)^2 = \sum_{i=1}^{10} \left(\sum_{j=1}^{4} \gamma_j \phi_j(t_i) - f_i \right)^2 ,$$

which means

$$\mathbf{A} = \begin{bmatrix} \phi_1(t_1) & \phi_2(t_1) & \phi_3(t_1) & \phi_4(t_1) \\ \vdots & & & \vdots \\ \phi_1(t_{10}) & \phi_2(t_{10}) & \phi_3(t_{10}) & \phi_4(t_{10}) \end{bmatrix} \in \mathbb{R}^{10,4} .$$

First we look at the helper function `make_A`. This only creates the matrix $\mathbf{A}$ by applying the four model functions to the vector t and building up the matrix column wise. 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    size_t n = b.size();
4    Eigen::MatrixXd A(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 (size_t i = 0; i < n; i++) {
10     double t_vec = t[i];
11     // Initialize row i of the matrix
12     A(i, 0) = 1.0 / t_vec;
13     A(i, 1) = 1.0 / (t_vec * t_vec);
14     A(i, 2) = std::exp(-(t_vec - 1.0));
15     A(i, 3) = std::exp(-2.0 * (t_vec - 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**

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}\,,$$

we calculate $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{b}$ and then solve the system 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_vec,
3                                  const Eigen::VectorXd &b) {
4    // TODO (3-3.a) Solve normal equations to find the coefficients of the
5    // linear fitting
6    Eigen::VectorXd gamma(4);
7
8    // START
9    // Create matrix A
10   Eigen::MatrixXd A = make_A(t_vec);
11
12   // Transpose matrix A
13   Eigen::MatrixXd At = A.transpose();
14
15   // Compute LHS (A^T*A) of the normal equation
16   Eigen::MatrixXd AtA = At * A;
17
18   // Compute RHS (A^T*b) of the normal equation
19   Eigen::VectorXd Atb = At * b;
20
21   // Solve normal equation to get coefficients
22   gamma = AtA.ldlt().solve(Atb);
23   // END
24
25   return gamma;
26 }
```

Get it on GitLab (linfit.hpp).

▲

(3-3.b) ⚀ (30 min.) Write a C++ function

```
Eigen::VectorXd data_fit_qr(const Eigen::VectorXd &t, const
   Eigen::VectorXd &f);
```

that uses *orthogonal transformation techniques* as introduced in [Lecture → Section 3.3.4] for the same task as in the previous sub-problem.

SOLUTION 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_vec,
3                              const Eigen::VectorXd &b) {
4    // TODO (3-3.b) Find the coefficients for the linear
5    // fitting by means of the QR decomposition of A
6    Eigen::VectorXd gamma(4);
7
8    // START
9    // Make matrix A
10   Eigen::MatrixXd A = make_A(t_vec);
11
12   // Compute QR decomposition
13   gamma = A.colPivHouseholderQr().solve(b);
14   // END
15
16   return gamma;
17 }
```

Get it on GitLab (linfit.hpp).

▲

(3-3.c) ⚀ (15 min.) Write a C++ function

```
Eigen::VectorXd fitted_function(const Eigen::VectorXd &gamma,
   const Eigen::VectorXd &t_vec)
```

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

SOLUTION 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) {
  // TODO (3-3.c): Define the data for the first plot by
  // evaluating the function f at the grid defined by t_vec
  unsigned int n = t_vec.size();
  Eigen::VectorXd y(n);

  // START
  Eigen::MatrixXd A = make_A(t_vec);
  y = A * gamma;
  // END

  return y;
}
```

Get it on GitLab (linfit.hpp).

▲

(3-3.d) ⚂ (15 min.) Using EIGEN write a C++ function

```
Eigen::VectorXd fitting_error(const Eigen::VectorXd &gamma);
```

that returns the vector

$$\left[(f(t_i) - f_i)^2\right]_{i=1}^{10} \in \mathbb{R}^{10}$$

where f is defined according to (3.3.1) using the values for the γ_j-coefficients passed in the `gamma` argument.

SOLUTION 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) {
  // TODO (3-3.c): Compute the vector of squared errors of your
  // fit at the provided sample points
  Eigen::VectorXd err(10);

  // START
  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.;

  // Compute error
  err = fitted_function(gamma, t_vec) - f;
  err = err.cwiseProduct(err);
  // END

  return err;
}
```

Get it on GitLab (linfit.hpp).

▲

End Problem 3-3 , 90 min.

Problem 3-4: QR-Factorization of Tridiagonal Matrices

In [Lecture → Rem. 3.3.3.26] we saw that a QR-factorization of a tri-diagonal matrix features an upper triangular factor that is tridiagonal again. In this exercise we exploit this fact for the design of efficient algorithms, in particular for the stable solution of tridiagonal linear systems of equations.

This exercise assume familiarity with the concept of QR-decomposition/factorization [Lecture → Section 3.3.3] and, in particular, Givens rotations [Lecture → § 3.3.3.15] and their compressed storage [Lecture → Rem. 3.3.3.21].

Throughout this problem, in order to be compatible with some software, for generic real-valued square tridiagonal matrices we are supposed to use the data structure

```
#include <Eigen/Dense>
struct TriDiagonalMatrix {
  Eigen::Index n; // Matrix size:  n × n
  Eigen::VectorXd d; // n-vector of diagonal entries
  Eigen::VectorXd l; // n−1-vector, entries of first lower diagonal
  Eigen::VectorXd u; // n−1-vector, entries of first upper diagonal
  // Simple constructor
  TriDiagonalMatrix(const Eigen::VectorXd &d,
  const Eigen::VectorXd &l,
  const Eigen::VectorXd &u)
  : n(d.size()), d(d), l(l), u(u) {
    assert((n-1) == l.size() && (n-1) == u.size());
  }
};
```

Note that in full-fledged C++ implementations, this would be a C++ class with four different constructors and a destructor explicitly defined, private data members, and access member functions. The rudimentary definition above is proposed solely for the sake of simplicity.

Let us write $\mathbf{d} \in \mathbb{R}^n$, $\mathbf{l} \in \mathbb{R}^{n-1}$, and $\mathbf{u} \in \mathbb{R}^{n-1}$ for the vectors stored in `d`, `l`, `u`. Then the matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in an **TriDiagonalMatrix** object is defined by

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

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

(3-4.a) ⚀ (30 min.) Prove that for a *tridiagonal* invertible matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ a QR-factorization $\mathbf{A} = \mathbf{QR}$, $\mathbf{Q} \in \mathbb{R}^{n,n}$ orthogonal, $\mathbf{R} \in \mathbb{R}^{n,n}$ upper triangular with positive diagonal entries, satisfies

$$(\mathbf{R})_{i,j} = 0 \quad \text{for} \quad j > i+2 \quad \text{and} \quad (\mathbf{Q})_{i,j} = 0 \quad \text{for} \quad i > j+1\,. \tag{3.4.1}$$

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

$$\mathbf{A},\mathbf{B}\ \begin{cases}\text{diagonal}\\ \text{upper triangular}\\ \text{lower triangular}\end{cases}\ \Rightarrow\ \mathbf{AB}\ \text{ and }\ \mathbf{A}^{-1}\ \begin{cases}\text{diagonal}\\ \text{upper triangular}\\ \text{lower triangular}\end{cases}.$$

(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

$$\mathrm{Span}\{(\mathbf{A})_{:,1},\ldots,(\mathbf{A})_{:,k}\} \subset \mathrm{Span}\{\mathbf{e}_1,\ldots,\mathbf{e}_k\} = \left\{ \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \xi_j \in \mathbb{R} \right\}. \tag{3.4.5}$$

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 $\mathbf{Q}^\top$.

▲

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

```
class TriDiagonalQR {
  public:
  explicit TriDiagonalQR(const TriDiagonalMatrix &A);

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

  private:
  Eigen::Index n; // size of the square matrix
  Eigen::Matrix B; // Three non-zero upper diagonals of R
  Eigen::VectorXd rho; // Encoded Givens rotations

};
```

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

$$(\mathbf{R})_{i,j} := \begin{cases} (\mathbf{B})_{i,1} & \text{, if } i = j\,, \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,2} & \text{, if } j = i+1\,, \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,3} & \text{, if } j = i+2\,, \\ 0 & \text{else,} \end{cases} \qquad i,j \in \{1,\ldots,n\}\,, \tag{3.4.6}$$

that is, $\mathbf{R} = \begin{bmatrix} x_1 & y_1 & z_1 & & & & \\ & x_2 & y_2 & z_2 & & & \\ & & x_3 & y_3 & z_3 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & x_{n-2} & y_{n-2} & z_{n-2} \\ & & & & & x_{n-1} & y_{n-1} \\ & & & & & & x_n \end{bmatrix} \Rightarrow \mathbf{B} = \begin{bmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_{n-2} & y_{n-2} & z_{n-2} \\ x_{n-1} & y_{n-1} & 0 \\ x_n & 0 & 0 \end{bmatrix}$.

Note that three entries of $\mathbf{B}$ will never be used.

The $\mathbf{Q}$-factor is stored in compressed format through the underlying $n-1$ Givens rotations whose target rows are clear a priori, see [Lecture → Rem. 3.3.3.26]. Every Givens rotation is stored through a single number ρ following the convention described in [Lecture → Rem. 3.3.3.21]. These $n-1$ numbers are collected in the data member `rho`. Its j-th entry corresponds to the Givens rotation acting on rows j & $j+1$.

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

```
std::tuple<double,double,double>
  compGivensRotation(Eigen::Vector2d a);
```

that computes a Givens rotation mapping the vector $\mathbf{a} \in \mathbb{R}^2$ onto a multiple of the first Cartesian coordinate vector $\mathbf{e}_1 \in \mathbb{R}^2$. Use the algorithm implemented in [Lecture → Code 3.3.3.17]. The function is to return the triplet (ρ, γ, σ), where $\mathbf{G} = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix}$ is the transformation matrix such that $\mathbf{G}^\top \mathbf{a} = (\pm\|\mathbf{a}\|, 0)^\top$, and $\rho \in \mathbb{R}$ encodes it according to the rule set

$$\rho := \begin{cases} \operatorname{sign}(\sigma) & \text{, if } \gamma = 0\,, \\ \sqrt{2}\operatorname{sign}(\gamma)\sigma & \text{, if } |\sigma| < |\gamma|\,, \\ \frac{1}{2}\sqrt{2}\operatorname{sign}(\sigma)/\gamma & \text{, if } |\sigma| \geq |\gamma|\,. \end{cases} \qquad \text{[Lecture → Eq. (3.3.3.22)]}$$

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));
}
```

▲

(3-4.c) ⚄ (45 min.) [depends on [Lecture → Rem. 3.3.3.26]]

Write an *efficient* code for the constructor of **TriDiagonalQR**, which initializes the data members, that is, computes the QR-factorization of the tridiagonal matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ passed to it in `A`.

HINT 1 for (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:

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{12}^\top} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{n-1,n}^\top \cdots \mathbf{G}_{23}^\top} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

$*$ $\hat{=}$ entry set to zero by Givens rotation, $*$ $\hat{=}$ new non-zero entry

C++ code 3.4.8: Constructor of TriDiagonalQR

```
2  TriDiagonalQR::TriDiagonalQR(const TriDiagonalMatrix &A) : n(A.n) {
3    rho = Eigen::VectorXd::Zero(n - 1);
4    B = Eigen::MatrixXd::Zero(n, 3);
5    B.col(0) = A.d;
6    B.col(1).tail(n - 1) = A.l;
7    B.col(2).head(n - 1) = A.u;
8    // Currently we have for k=0,...,n-1:
9    // B(k,0) = A(k,k), B(k,1) = A(k,k-1), B(k,2) = A(k,k+1).
10
11   // A bit clumsy, but less error prone than juggling indices
12   Eigen::MatrixXd Ablock(2, 3);
13   double gamma, sigma;
14   for (unsigned int k = 0; k < n - 1; ++k) {
15     // The block of the current A that will be affected by the next
         rotation.
16     Ablock << B(k, 0), B(k, 2), 0, B(k + 1, 1), B(k + 1, 0), B(k + 1, 2);
17     // TODO (3-4.c): Find the k-th Givens rotation and update rho and
         B.
18     // START
19     std::tuple<double, double, double> params =
20         compGivensRotation(Ablock.col(0));
21     rho[k] = std::get<0>(params); // Store rotation parameter
22     gamma = std::get<1>(params);
23     sigma = std::get<2>(params);
24
25     // Update B
26     Eigen::Matrix2d G;
27     G << gamma, -sigma, sigma, gamma;
28     Ablock = G * Ablock; // Rotate to eliminate Ablock(1,0)
29     B.row(k) = Ablock.row(0);
30     B(k + 1, 0) = Ablock(1, 1);
31     B(k + 1, 2) = Ablock(1, 2);
32     // END
33   }
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].

▲

(3-4.d) ⚀ (30 min.) Provide an *efficient* implementation of the method `applyQT()` of **TriDiagonalQR** that computes $\mathbf{Q}^\top\mathbf{x}$ for a supplied vector $\mathbf{x} \in \mathbb{R}^n$ (passed in `x`).

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

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

```
2  inline Eigen::Matrix2d Givens(double rho) {
3    double gamma, sigma;
4    if (std::abs(rho) < 1.0) {
5      sigma = 0.5 * std::sqrt(2.0) * rho;
6      gamma = std::sqrt(1.0 - sigma * sigma);
7    } else if (std::abs(rho) > 1.0) {
8      gamma = 0.5 * std::sqrt(2.0) / rho;
9      sigma = std::sqrt(1 - gamma * gamma);
10   } else {
11     gamma = 0.0;
12     sigma = sign(rho);
13   }
14   Eigen::Matrix2d G;
15   G << gamma, sigma, -sigma, gamma;
16   return G;
17 }
```

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

```
2  template <typename VecType>
3  Eigen::VectorXd TriDiagonalQR::applyQT(const VecType &x) const {
4    assert(n == x.size() &&
5           "Input must have size equal to the number of rows in Q.");
6    Eigen::VectorXd y(n);
7    // TODO (3-4.d): Fill in the entries of y.
8    // START
9    for (unsigned int k = 0; k < n; ++k) {
10     y[k] = x[k];
11   }
12   Eigen::Matrix2d G;
13   for (unsigned int k = 0; k < n - 1; ++k) {
14     G = Givens(rho[k]).transpose();
15     y.segment(k, 2) = G * y.segment(k, 2);
16   }
17   // END
18   return y;
19 }
```

Get it on GitLab (tridiagqr.hpp).

▲

(3-4.e) ⚄ (30 min.) [depends on Sub-problem (3-4.d) and [Lecture → Rem. 3.3.4.3]]

Devise an efficient implementation of the method `solve()` of **TriDiagonalQR** that accepts a vector $\mathbf{b} \in \mathbb{R}^n$ as arguments and solves the linear systems of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$ the tridiagonal matrix stored in the current **TriDiagonalQR** object, returning the solution $\mathbf{x} \in \mathbb{R}^n$. A **std::runtime_error** exception should be thrown in case the matrix $\mathbf{A}$ is not invertible.

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):

- First step: $\mathbf{c} = \mathbf{Q}^\top \mathbf{b}$ by `applyQT()`
- Second step: back-substitution for a tridiagonal matrix

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).

▲

(3-4.f) ⚂ (10 min.) [depends on Sub-problem (3-4.b), Sub-problem (3-4.d), Sub-problem (3-4.e)]

What is the asymptotic computational effort for your implementations of the constructor of **TriDiagonalQR** and of the methods `applyQT()` and `solve()` in terms of $n \to \infty$?

SOLUTION of (3-4.f):

All functions/methods enjoy the asymptotic complexity $O(n)$ for $n \to \infty$ as they all involve a single loop.

▲

(3-4.g) ⚂ (30 min.) The following templated C++ function implements (in a non-optimal way) a so-called **inverse iteration** [Lecture → Section 9.3.2].

```
template <typename MatrixType>
unsigned int invit(const MatrixType &A,Eigen::VectorXd &x,
double TOL = 1E-6, unsigned int maxit = 100) {
  x.normalize();
  Eigen::VectorXd x_old(x.size());
  int it = 0;
  do {
    x_old = x;
    x = A.lu().solve(x_old);
    x.normalize();
  }
  while (((x-x_old).norm() > TOL) && (it++ < maxit));
  return it;
}
```

Implement a **specialization** (C++ reference) of this function for **MatrixType = TriDiagonalMatrix**.

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.

▲

End Problem 3-4 , 195 min.

Problem 3-5: Conditional minimum with SVD

A *Singular Value Decomposition* (SVD, see [Lecture → Def. 3.4.1.3]) is the generalized version of the diagonalization process for any $m \times n$ matrix. We will see that the diagonal and orthogonal matrices arising from an SVD have certain properties such that it is immediate to identify the minimum of a least squares functional for that matrix. In other words, computing the SVD of a matrix is equivalent to solving its corresponding least square problem.

The solution of this problem is discussed in [Lecture → Section 3.4.4.1]; the algorithm is implemented in [Lecture → Code 3.4.4.2].

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

$$\mathbf{x}^* = \operatorname*{argmin}_{\|\mathbf{x}\|_2=1} \{\|\mathbf{A}\mathbf{x}\|_2\} \tag{3.5.1}$$

(3-5.a) ⚄ (45 min.) Show that the solution of (3.5.1) is:

$$\mathbf{x}^* = \mathbf{v}_n := \mathbf{V}_{:,n} \tag{3.5.2}$$

Show that the following holds:

$$\|\mathbf{A}\mathbf{v}_n\|_2 = \sigma_n := \mathbf{\Sigma}_{n,n}. \tag{3.5.3}$$

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:

$$\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \min_{\|\mathbf{y}\|=1} \|\mathbf{\Sigma}\mathbf{y}\| \tag{3.5.7}$$

$$= \min_{\|\mathbf{y}\|=1} \sqrt{\sigma_1^2 y_1^2 + \ldots + \sigma_n^2 y_n^2} \tag{3.5.8}$$

$$\geq \min_{\|\mathbf{y}\|=1} \sqrt{\sigma_n^2 y_1^2 + \ldots + \sigma_n^2 y_n^2} \tag{3.5.9}$$

$$= \sigma_n \min_{\|\mathbf{y}\|=1} \sqrt{y_1^2 + \ldots + y_n^2} \tag{3.5.10}$$

$$= \sigma_n \min_{\|\mathbf{y}\|=1} \|\mathbf{y}\| = \sigma_n. \tag{3.5.11}$$

Here we first use the previously obtained equation and then compute $\boldsymbol{\Sigma}\mathbf{y}$ using the fact that $\boldsymbol{\Sigma}$ is a diagonal matrix (when computing the norm).

Secondly, we prove "$\leq$". We can bound the minimum by:

$$\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \min_{\|\mathbf{y}\|=1} \|\boldsymbol{\Sigma}\mathbf{y}\| \leq \|\boldsymbol{\Sigma}\mathbf{e}_n\| = \sigma_n \,.$$

We have therefore proven the second equality (3.5.6), i.e. that $\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \sigma_n$, and $\mathbf{e}_n = \operatorname{argmin}_{\|\mathbf{y}\|=1} \|\boldsymbol{\Sigma}\mathbf{y}\|$.

Let us reconstruct $\mathbf{x}$ from $\mathbf{y} = \mathbf{V}^\top \mathbf{x}$. Since $\mathbf{V}$ is orthogonal, we have that:

$$\mathbf{e}_n = \mathbf{y} = \mathbf{V}^\top \mathbf{x} \quad \Longrightarrow \quad \mathbf{x} = \mathbf{V}\mathbf{y} = \mathbf{V}\mathbf{e}_n = \mathbf{V}_{:,n} =: \mathbf{v}_n. \tag{3.5.12}$$

▲

End Problem 3-5 , 45 min.

Problem 3-6: Sparse Approximate Inverse (SPAI [GRH97])

This problem studies the least squares aspects of the SPAI method, a technique used in the numerical solution of partial differential equations. We encounter an "exotic" sparse matrix technique and rather non-standard least squares problems. Please note that the matrices to which SPAI techniques are applied will usually be huge and extremely sparse, say, of dimension $10^7 \times 10^7$ with only 10^8 non-zero entries. Therefore sparse matrix techniques must be applied.

Requires only familiarity with [Lecture → Section 3.1] and [Lecture → Section 3.2].



Sub-problem (3-6.d) is connected with [Lecture → Section 10.3]. In case this section had not (yet) been covered in the course that sub-problem should be skipped.

Let $\mathbf{A} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$, be a regular sparse matrix with at most $n \ll N$ non-zero entries per row and column. We define the space of matrices with the same pattern (on non-zero entries) as $\mathbf{A}$:

$$\mathcal{P}(\mathbf{A}) := \{\mathbf{X} \in \mathbb{R}^{N,N} : \ (\mathbf{A})_{ij} = 0 \ \Rightarrow \ (\mathbf{X})_{ij} = 0\} \ . \tag{3.6.1}$$

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

$$\mathbf{B} := \operatorname*{argmin}_{\mathbf{X} \in \mathcal{P}(\mathbf{A})} \|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F \ , \tag{3.6.2}$$

where $\|\cdot\|_F$ stands for the Frobenius norm [Lecture → Def. 3.4.4.17]. The solution of (3.6.2) can be used as a so-called preconditioner for the acceleration of iterative methods for the solution of linear systems of equations, see [Lecture → Chapter 10]. An extended "self-learning" variant of the SPAI method is presented in [**GRH97**].

(3-6.a) ⚂ (20 min.) Show that the columns of $\mathbf{B}$ can be computed independently of each other by solving linear least squares problems. In the statement of these linear least squares problems write $\mathbf{b}_i$ for the columns of $\mathbf{B}$.

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 $\mathbf{I} - \mathbf{A}\mathbf{X}$ depend only on the values in the vector $\mathbf{x}_i$. Hence, in order to minimize the Frobenius norm $\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2$ one needs to (independently) minimize each of the terms $\|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2$:

$$\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{I}\mathbf{e}_i - \mathbf{A}\mathbf{X}\mathbf{e}_i\|_2^2 = \sum_{i=1}^{n} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2 .$$

The corresponding minimization problems read

$$\mathbf{b}_i := \operatorname*{argmin}_{\mathbf{x}_i \in \mathcal{P}((\mathbf{A})_{:,i})} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2 , \quad \text{for} \quad i = 1, \ldots, N, \tag{3.6.3}$$

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

$$\mathbf{y} \in \mathbb{R}^m: \quad \|\mathbf{e}_i - \mathbf{A}_i\mathbf{y}\|_2 \to \min \, .$$

▲

(3-6.b) ⚄ (60 min.) [depends on Sub-problem (3-6.a)]

Implement an efficient C++ function

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

for the computation of $\mathbf{B}$ according to (3.6.2). You may rely on the normal equations associated with the linear least squares problems for the columns of $\mathbf{B}$ or you may simply invoke the least squares solver of EIGEN, see [Lecture → Code 3.4.3.14].

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 $\mathbf{B}$ in EIGEN using the intermediate triplet format (`std::vector<Triplet<double>>`), 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),

$$\mathbf{b}_i := \operatorname*{argmin}_{\mathbf{x}_i \in \mathbb{R}^{m_i}} \|\mathbf{e}_i - \mathbf{C}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \ldots, N.$$

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).

▲

(3-6.c) ⚀ (15 min.) [depends on Sub-problem (3-6.b)]

What is the total asymptotic computational effort of your implementation of `spai()` in terms of the problem size parameters N and n.

SOLUTION 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 $O(Nn^3)$.

▲

(3-6.d) ⚁ (45 min.) [depends on Sub-problem (3-6.b)]

For $n \in \mathbb{N}$, consider a sparse s.p.d. [Lecture → Def. 1.1.2.6] matrix $\mathbf{A}$ given by the following C++-code

C++11-code 3.6.5: Initialization of matrix $\mathbf{A}$

```
2 | Eigen::SparseMatrix<double> init_A(unsigned int n) {
3 |   Eigen::SparseMatrix<double> L(n, n);
4 |   Eigen::SparseMatrix<double> R(n, n);
5 |   L.reserve(n);
6 |   R.reserve(n + 2 * (n - 1));
7 |   L.insert(0, 0) = std::exp(1. / n);
```

```
  R.insert(0, 0) = 2.;
  for (unsigned int i = 1; i < n; ++i) {
    L.insert(i, i) = std::exp((i + 1.) / n);
    R.insert(i, i) = 2.;
    R.insert(i - 1, i) = -1.;
    R.insert(i, i - 1) = -1.;
  }
  Eigen::SparseMatrix<double> A = kroneckerProduct(L, R);
  A.makeCompressed();
  return A;
}
```

Get it on GitLab (spai.hpp).

Solve the linear system of equations $\mathbf{Ax} = \mathbf{b}$ using EIGEN's **ConjugateGradient** (EIGEN documentation) iterative solver with initial guess $\mathbf{0}$ and default tolerance. Do this without a preconditioner and using the preconditioner $\frac{1}{2}(\mathbf{B} + \mathbf{B}^T)$, where $\mathbf{B}$ solves (3.6.2). Use $\mathbf{b} = [1,1,\ldots,1]^\top$. Write a C++ function

```
std::vector<std:pair<unsigned int, unsigned int>>
testSPAIPrecCG(unsigned int L);
```

that returns the number of CG iterations (with and without the SPAI preconditioner) versus the system size $N = n^2$ for $n = 2^{1,\ldots,L}$.

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).

▲

End Problem 3-6 , 140 min.

Problem 3-7: Shape identification

This problem deals with pattern recognition, a central problem in image processing (e.g. in aerial photography, self-driving cars, and recognition of pictures of cats). We will deal with a very simplistic problem.

This problem practises the solution of a linear least squares problem based on the normal-equation method, see [Lecture → Section 3.2].

A routine within the program of a self driving-car is tasked with the job of identifying road signs. The task is the following: given a collection of points $\mathbf{p}_i \in \mathbb{R}^2, i = 1, \ldots, n$, we have to decide whether those point represent a stop sign, or a "priority road sign". In this exercise we consider $n = 8$.

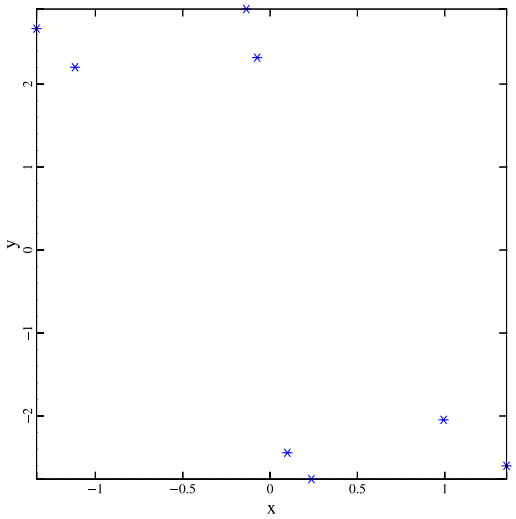


Fig. 11

A set of possible input points $\mathbf{p}^i$: corners detected in the photo.

Which traffic sign had been in the focus of the camera?



Fig. 12

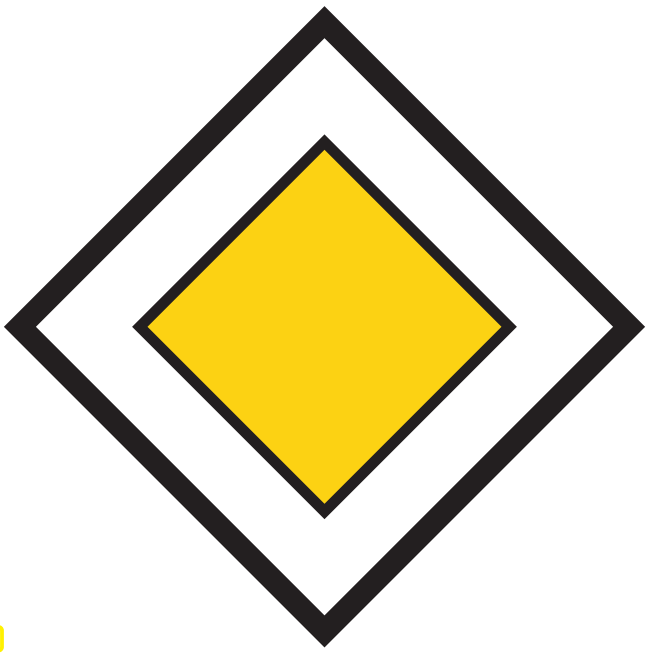


Fig. 13

The shape of the sign can be represented by a 2×8 matrix with the 8 coordinates in $\mathbb{R}^2$ defining the shape of the sign. We assume that the stop sign resp. the priority road sign are defined by the "model" points (known a priori) $\mathbf{x}^i_{stop} \in \mathbb{R}^2$ resp. $\mathbf{x}^i_{priority} \in \mathbb{R}^2$ for $i = 1, \ldots, 8$.

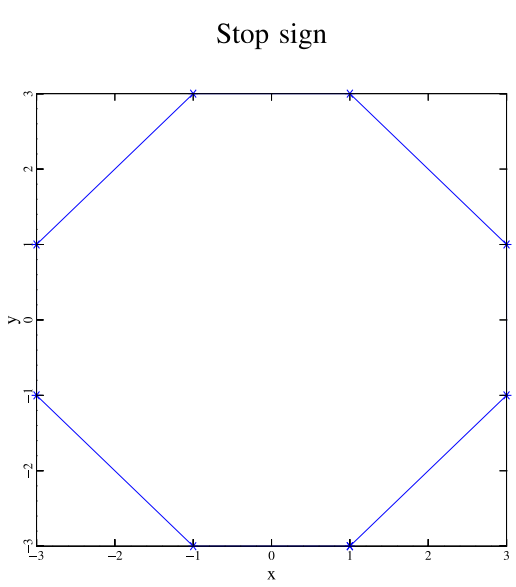


Fig. 14

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

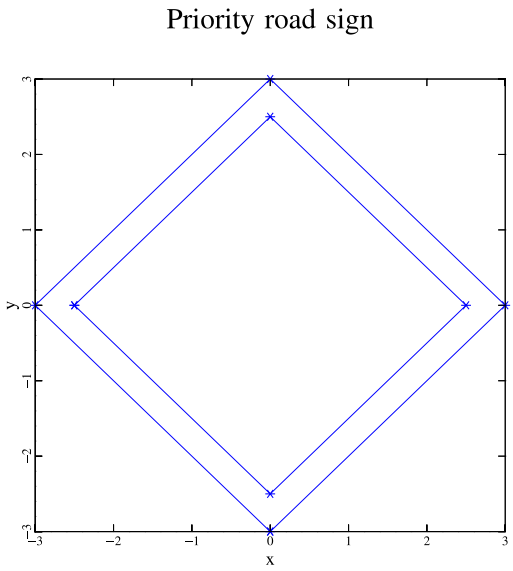


Fig. 15

The 8-points $\mathbf{x}^i_{priority}$ defining the model of a priority road sign.

However, in a real case scenario, one can imagine that the photographed shape is not exactly congruent to the one specified by the "model" points. Instead, one can assume that the points on the photo ($\mathbf{p}^i \in \mathbb{R}^2$) are the result of a *linear transformation* of the original points $\mathbf{x}^i \in \mathbb{R}^2$ for $i = 1, \ldots, 8$; i.e. we can assume that there exists a matrix $\mathbf{A} \in \mathbb{R}^{2,2}$, s.t.

$$\mathbf{p}^i = \mathbf{A}\mathbf{x}^i, i = 1, \ldots, n \,. \tag{3.7.1}$$

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

Moreover, we have some error in the data, i.e. our points do not represent exactly one of the linearly transformed shapes (it is plausible to imagine that there is some measurement error, and that the photographed shape is not exactly the same as our "model" shape), i.e. (3.7.1) is satisfied only "approximately".

With this problem, we will try to use the least square method to find the matrix $\mathbf{A}$ and to *classify* our points, i.e. to decide whether they represent a stop or a priority road sign.

(3-7.a) ⚄ (20 min.) For the moment, assume we know the points $\mathbf{x}^i$ (i.e. we know the shape of our sign). Explicitly write (3.7.1) as an overdetermined linear system:

$$\mathbf{B}\mathbf{v} = \mathbf{w} \,,$$

whose least-squares solution $\mathbf{v}^*$ will allow to determine the "best" linear transformation $\mathbf{A}$ (in the least-squares sense). How is the vector of unknowns related to $\mathbf{A}$?

SOLUTION of (3-7.a):

Let $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$. Then, for every $i = 1, \ldots, n$:

$$\begin{bmatrix} p^i_1 \\ p^i_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x^i_1 \\ x^i_2 \end{bmatrix} = \begin{bmatrix} a_{11}x^i_1 + a_{12}x^i_2 \\ a_{21}x^i_1 + a_{22}x^i_2 \end{bmatrix} \tag{3.7.2}$$

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.

▲

(3-7.b) ⚀ (10 min.) When does the matrix $\mathbf{B}$ have full rank? Give a geometric interpretation of this condition.

SOLUTION of (3-7.b):

The matrix has $\operatorname{rank}(\mathbf{B}) \leq 4$. The matrix has not full rank, if and only if its columns are linearly dependent, which is the case if and only if there exists a $c \in \mathbb{R}$ s.t. $x_2^i = cx_1^i$. This is equivalent to $\mathbf{x}^i = x_1^i[1, c]^\top$, i.e. all points $\mathbf{x}^i$ lie on the same line.

▲

(3-7.c) ⚀ (15 min.) [depends on Sub-problem (3-7.a)]

In the file `shape_ident.hpp` implement a C++ function

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

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

SOLUTION 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).

▲

(3-7.d) ⚂ (30 min.) [depends on Sub-problem (3-7.c)]

In the file `shape_ident.hpp` implement a C++ function

```
double solve_lsq(const Eigen::MatrixXd & X, const Eigen::MatrixXd
    & P,
                 Eigen::MatrixXd & A);
```

that computes the matrix $\mathbf{A}$ by solving the least squares problem based on the normal equations, see [Lecture → Section 3.2]. It should return the Euclidean norm of the residual of the least square solution. Pass the vectors $\mathbf{x}^i$ and the vectors $\mathbf{p}^i$ as a $2 \times n$ EIGEN matrix `X` resp. `P`.

SOLUTION 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.

```
2  double solve_lsq(const Eigen::MatrixXd &X, const Eigen::MatrixXd &P,
3                   Eigen::MatrixXd &A) {
4    assert(X.rows() == 2 && "X must have 2 rows!");
```

```
5   assert(P.rows() == 2 && "P must have 2 rows!");
6   assert(X.cols() == P.cols() && "P and X must have same size!");
7
8   unsigned n = X.cols();
9   double norm_of_residual = 0.;
10
11  // TODO: (3-7.d) solve LSQ problem, return best linear approximation
        and
12  // residual
13
14  // START
15  // Build system matrix
16  Eigen::MatrixXd B = shape_ident_matrix(X);
17
18  // Solve LSQ system using normal equation
19  // We need to do some reshaping in order to properly set up the
        system
20  A = Eigen::Map<Eigen::MatrixXd>(
21          Eigen::MatrixXd(
22              // Solve LSQ problem using normal equation
23              (B.transpose() * B)
24                  .ldlt()
25                  .solve(B.transpose() *
26                         // Need to vectorize matrix
27                         Eigen::Map<const Eigen::MatrixXd>(P.data(), 2 * n, 1))
28              // Pass an array to "EIgen::Map"
29              )
30              .data(),
31          // Need to reshape vector to 2x2 matrix
32          2, 2)
33          .transpose();
34
35  // Residual:  must reshape P
36  norm_of_residual =
37      (Eigen::Map<const Eigen::MatrixXd>(P.data(), 2, n) - A * X).norm();
38  // END
39
40  return norm_of_residual;
41 }
```

Get it on GitLab (shape_ident.hpp).

▲

(3-7.e) ⚁ (30 min.) [depends on Sub-problem (3-7.d)]

Explain how the norm of the residual of the least square solution can be used to identify the shape defined by the points $\mathbf{p}^i$. Implement a function

```
enum Shape { Stop, Priority };
Shape identify(const Eigen::MatrixXd Xstop,
               const Eigen::MatrixXd Xpriority,
               const Eigen::MatrixXd & P,
               Eigen::MatrixXd & A);
```

that identifies the shape (either Stop or Priority sign) of the input points $\mathbf{p}^i$. The function returns an `enum` that classifies the shape of points specified by `P`. Return the linear transformation in the matrix `A`. The "model points" $\mathbf{x}^i_{stop}$ resp. $\mathbf{x}^i_{priority}$ are passed through `Xstop` resp. `Xpriority`.

SOLUTION 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).

▲

End Problem 3-7, 105 min.

Problem 3-8: Low rank approximation of matrices

As explained in the course, large $m \times n$ matrices of low rank $k \ll \min\{m,n\}$ can be stored using only $k(m+n)$ real numbers when using their SVD factorization/representation as sum of tensor products [Lecture → Eq. (3.4.1.9)]. Thus, low rank matrices are of considerable interest for *matrix compression* (see [Lecture → Ex. 3.4.4.24]). Unfortunately, adding two low rank matrices usually leads to an increase of the rank and entails "recompression" by computing a low-rank best approximation of the sum. This problems demonstrates an efficient approach to recompression.

Study [Lecture → Section 3.4.4.2] to prepare for this exercise. The algorithms have to be implemented based on EIGEN and they rely on the SVD [Lecture → Section 3.4.2] and QR factorization [Lecture → Section 3.3.3.4].

(3-8.a) ⚀ (10 min.) Show that for a matrix $\mathbf{X} \in \mathbb{R}^{m,n}$ the following statements are equivalent:

(i) $\operatorname{rank}(\mathbf{X}) = k$

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

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 $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, with $\mathbf{\Sigma} = \operatorname{diag}(\sigma_1,\ldots,\sigma_k,0,\ldots,0)$, see [Lecture → Eq. (3.4.1.14)].

Thus, we can omit the last rows/columns of the matrices in the decomposition: $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{U}_{:,1:k}\mathbf{\Sigma}_{1:k,1:k}(\mathbf{V}_{:,1:k})^\top$.

By defining $\left\{ \begin{array}{l} \mathbf{A} := \mathbf{U}_{:,1:k}\mathbf{\Sigma}_{1:k,1:k} \\ \mathbf{B} := \mathbf{V}_{:,1:k} \end{array} \right\}$, the claim follows.

(ii)⇒(i):The matrix $\mathbf{B}^\top$ corresponds to a linear mapping from $\mathbb{R}^n$ to $\mathbb{R}^k$, with $k \le n$. 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 $k \le m$. 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:

$$\exists \mathbf{b}_\ell \in \mathbb{R}^n: \quad f_\ell(\mathbf{z}) = \mathbf{b}_\ell^\top \mathbf{z} \quad \forall \mathbf{z} \in \mathbb{R}^n \,.$$

$$\blacktriangleright \qquad \mathbf{X}\mathbf{z} = \sum_{\ell=1}^{k} (\mathbf{A})_{:,\ell} \boldsymbol{v}_\ell^\top \mathbf{z} = \mathbf{A}\mathbf{B}^\top \mathbf{z} \quad \forall \mathbf{z} \in \mathbb{R}^n \quad \text{with} \quad \mathbf{B} := \left[\mathbf{b}_1, \cdots, \mathbf{b}_k\right] \in \mathbb{R}^{n,k} \,.$$

Of course, there are many more ways to prove the asserted equivalence.

▲

(3-8.b) ⚀ (30 min.) [depends on Sub-problem (3-8.a)]

In the file `lowrankrep.hpp` write an EIGEN-based C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
factorize_X_AB(const Eigen::MatrixXd & X, unsigned int k);
```

that factorizes the matrix $\mathbf{X} \in \mathbb{R}^{m,n}$ with $\mathrm{rank}(\mathbf{X}) = k$ into $\mathbf{A}\mathbf{B}^\top$, as in Sub-problem (3-8.a), and returns the two matrices $\mathbf{A} \in \mathbb{R}^{m,k}$ and $\mathbf{B} \in \mathbb{R}^{n,k}$ in this order.

The function should issue a warning in case $\mathrm{rank}(\mathbf{X}) = k$ is in doubt.

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 $\mathrm{rank}(X) = k \le \min\{m,n\}$, 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 $\mathrm{rank}(\mathbf{X}) = k$ 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 × 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 $\operatorname{rank}(\mathbf{X}) > k$ please revisit [Lecture → Ex. 3.4.2.3], which explains how to check, if a singular value is "relatively close to zero".

▲

(3-8.c) ⚀ (30 min.) Let $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{B} \in \mathbb{R}^{n,k}$, with $k \leq \min\{m,n\}$. Based on the economical QR-factorization [Lecture → Thm. 3.3.3.4] of both $\mathbf{A}$ and $\mathbf{B}$

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

devise a way, how the economical singular value decomposition $\mathbf{A}\mathbf{B}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}$ [Lecture → Eq. (3.4.1.5)] of $\mathbf{A}\mathbf{B}^\top$ can be computed using only the singular value decomposition of a $k \times k$-matrix.

SOLUTION 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 $\mathbf{S}$:

$$\mathbf{S} = \tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^\top \quad , \quad \tilde{\mathbf{U}}, \tilde{\mathbf{\Sigma}}, \tilde{\mathbf{V}} \in \mathbb{R}^{k,k} \, , \quad \tilde{\mathbf{U}}, \tilde{\mathbf{V}} \text{ orthogonal} \, , \quad \tilde{\mathbf{\Sigma}} \text{ diagonal} \, . \tag{3.8.4}$$

Then, insert (3.8.4) into (3.8.3):

$$\mathbf{A}\mathbf{B}^\top = \mathbf{Q}_A \underbrace{\mathbf{R}_A\mathbf{R}_B^\top}_{=\tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^\top} \mathbf{Q}_B^\top = \underbrace{(\mathbf{Q}_A\tilde{\mathbf{U}})}_{=:\mathbf{U}}\mathbf{\Sigma}\underbrace{(\mathbf{Q}_B\tilde{\mathbf{V}})}_{=:\mathbf{V}}^\top \, . \tag{3.8.5}$$

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 $\mathbf{A}\mathbf{B}^\top$.

▲

(3-8.d) ⚄ (30 min.) [depends on Sub-problem (3-8.c)]

Given $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{B} \in \mathbb{R}^{n,k}$, with $k \ll m,n$, write an efficient C++ function

```
std::tuple<Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd>
svd_AB(const MatrixXd & A, const MatrixXd & B);
```

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

HINT 1 for (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)

```
2  std::tuple<Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd> svd_AB(
3      const Eigen::MatrixXd& A, const Eigen::MatrixXd& B) {
4    assert(A.cols() == B.cols() &&
5           "Matrices A and B should have the same column number");
6    assert(A.cols() < A.rows() && A.cols() < B.rows() &&
7           "Matrices A and B should have less columns than rows");
8    std::size_t m = A.rows();
9    std::size_t n = B.rows();
10   std::size_t k = A.cols();
11   Eigen::MatrixXd U, S, V;
12   // TODO: (3-8.d) Define the tuple of
13   // matrices U,S,V of the svd decomposition of A*B^T
14   // Hint: You can define a tuple of objects using std::make_tuple
15   // START
16   // QA: m x k; QB: n x k; RA,RB k x k
17   Eigen::HouseholderQR<Eigen::MatrixXd> QRA = A.householderQr();
18   Eigen::MatrixXd QA =
19       QRA.householderQ() * Eigen::MatrixXd::Identity(m, std::min(m, k));
20   Eigen::MatrixXd RA = Eigen::MatrixXd::Identity(std::min(m, k), m) *
21                        QRA.matrixQR().triangularView<Eigen::Upper>();
22   Eigen::HouseholderQR<Eigen::MatrixXd> QRB = B.householderQr();
23   Eigen::MatrixXd QB =
```

```
24        QRB.householderQ() * Eigen::MatrixXd::Identity(n, std::min(n, k));
25    Eigen::MatrixXd RB = Eigen::MatrixXd::Identity(std::min(n, k), n) *
26                         QRB.matrixQR().triangularView<Eigen::Upper>();
27
28    // U,V: k x k
29    Eigen::JacobiSVD<Eigen::MatrixXd> svd(
30        RA * RB.transpose(), Eigen::ComputeFullU | Eigen::ComputeFullV);
31    // Thin matrices are unnecessary here as RA*RB' is
32    // a square k×k matrix!
33    // Moreover, if you had computed the thin matrices U and V
34    // of the direct SVD of AB',
35    // you would still have dealt with the minimum between m and n,
36    // without exploiting k<<m,n!
37
38    S = svd.singularValues().asDiagonal();
39    U = svd.matrixU();
40    V = svd.matrixV();
41    // U: m x k; V: n x k
42    U = QA * U;
43    V = QB * V;
44    // END
45    return std::make_tuple(U, S, V);
46  }
```

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 subproblems, their columns would be orthogonal (i.e. $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{B}^\top\mathbf{B}$ would be diagonal). Therefore, the triangular part of their QR decompositions would be diagonal, which can be used to speed up Code 3.8.6. However, our code works for every choice of matrices $\mathbf{A},\mathbf{B}$ with the same number of columns k.

▲

(3-8.e) ⚀ (10 min.) [depends on Sub-problem (3-8.d)]

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

SOLUTION 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.

▲

(3-8.f) ⚀ (10 min.) If $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m,n}$ satisfy $\operatorname{rank}(\mathbf{Y}) = \operatorname{rank}(\mathbf{X}) = k$, show that $\operatorname{rank}(\mathbf{Y}+\mathbf{X}) \leq 2k$.

SOLUTION of (3-8.f):

From $\operatorname{rank}(\mathbf{X}) = k$ it follows that there is a basis for the image of $\mathbf{X}$ with k basis vectors, i.e. each column of $\mathbf{X}$ is a linear combination of these k vectors. The same holds for $\mathbf{Y}$ with other k basis vectors, where some can be obtainable from linearcombinations of the basis vectors of $\mathbf{X}$. Hence, each column of $\mathbf{X}+\mathbf{Y}$ is a linear combination of *at most* $2k$ vectors. Thus, $\operatorname{rank}(\mathbf{X}+\mathbf{Y}) \leq 2k$.

▲

(3-8.g) ⚄ (10 min.) Consider $\mathbf{A}_X, \mathbf{A}_Y \in \mathbb{R}^{m,k}$, $\mathbf{B}_X, \mathbf{B}_Y \in \mathbb{R}^{n,k}$, $\mathbf{X} = \mathbf{A}_X\mathbf{B}_X^\top$ and $\mathbf{Y} = \mathbf{A}_Y\mathbf{B}_Y^\top$. Find a factorization of the sum $\mathbf{X}+\mathbf{Y}$ as $\mathbf{X}+\mathbf{Y} = \mathbf{A}\mathbf{B}^\top$, with $\mathbf{A} \in \mathbb{R}^{m,2k}$ and $\mathbf{B} \in \mathbb{R}^{n,2k}$.

SOLUTION 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

$$\mathbf{A}\mathbf{B}^\top = [\mathbf{A}_X\ \mathbf{A}_Y]\begin{bmatrix}\mathbf{B}_X^\top \\ \mathbf{B}_Y^\top\end{bmatrix} = \mathbf{A}_X\mathbf{B}_X^\top + \mathbf{A}_Y\mathbf{B}_Y^\top = \mathbf{X}+\mathbf{Y}\,.$$

▲

(3-8.h) ⚅ (30 min.) [depends on Sub-problem (3-8.d), Sub-problem (3-8.g)]

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

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

that returns two matrix factors $\mathbf{A}_Z \in \mathbb{R}^{m,k}$ and $\mathbf{B}_Z \in \mathbb{R}^{n,k}$ whose product $\mathbf{Z} = \mathbf{A}_Z\mathbf{B}_Z^\top$ yields the rank-k best approximation of the matrix sum $\mathbf{X}+\mathbf{Y} = \mathbf{A}_X\mathbf{B}_X^\top + \mathbf{A}_Y\mathbf{B}_Y^\top$:

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

Here the matrices $\mathbf{A}_X, \mathbf{A}_Y \in \mathbb{R}^{m,k}$ and $\mathbf{B}_X, \mathbf{B}_Y \in \mathbb{R}^{n,k}$ are given and passed to the function through the arguments `Ax`, `Ay`, `Bx`, and `By`. You may only consider the case where $2k < m, n$.

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   Eigen::MatrixXd U, S, V;
26   std::tuple<Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd> svd_tuple =
27       svd_AB(A, B);
28   U = std::get<0>(svd_tuple);
29   S = std::get<1>(svd_tuple);
30   V = std::get<2>(svd_tuple);
31   std::size_t k = Ax.cols();
32   Az = U.leftCols(k) * S.topLeftCorner(k, k);
33   Bz = V.leftCols(k);
34   // END
35   return std::make_pair(Az, Bz);
36 }
```

Get it on GitLab (lowrankrep.hpp).

▲

(3-8.i) ⚀ (10 min.) [depends on Sub-problem (3-8.h)]

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

SOLUTION 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 $m,n \to \infty$.

▲

End Problem 3-8 , 170 min.

Problem 3-9: Matrix Fitting by Constrained Least Squares

In this problem we look at a particular *constrained* linear least squares problem, as they have been discussed in [Lecture → Section 3.6]. In particular, we will study the augmented normal equation approach to solve such type of least squares problem, see [Lecture → Section 3.6.1]. In this context we refresh our understanding of the Lagrange multiplier technique.

Related to [Lecture → Section 3.6]; involves implementation with EIGEN.

Consider the following problem of finding the minimal-norm matrix fitting a linear system of equations with given solution and right-hand side vector:

$$\text{Given } n \in \mathbb{N},\ \mathbf{z} \in \mathbb{R}^n,\ \mathbf{g} \in \mathbb{R}^n, \quad \text{find} \quad \mathbf{M}^* = \operatorname*{argmin}_{\mathbf{M}\in\mathbb{R}^{n,n},\ \mathbf{Mz}=\mathbf{g}} \|\mathbf{M}\|_F \ , \tag{3.9.1}$$

where $\|\cdot\|_F$ denotes the Frobenius norm of a matrix as introduced in [Lecture → Def. 3.4.4.17].

Definition [Lecture → Def. 3.4.4.17]. Frobenius norm

The Frobenius norm of $\mathbf{A} \in \mathbb{K}^{m,n}$ is defined as

$$\|\mathbf{A}\|_F^2 := \sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2 \ .$$

(3-9.a) ⚀ (30 min.) Reformulate the problem as an equivalent standard linearly constrained least squares problem [Lecture → Eq. (3.6.0.1)]

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^N,\ \mathbf{Cx}=\mathbf{d}} \|\mathbf{Ax}-\mathbf{b}\|_2 \ , \tag{3.9.2}$$

for suitable matrices $\mathbf{A}$, $\mathbf{C}$ and vectors $\mathbf{b}$ and $\mathbf{d}$, see [Lecture → Eq. (3.6.0.1)]. These matrices and vectors have to be specified based on $\mathbf{z}$ and $\mathbf{g}$.

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 $\mathbf{Cm}^* = \mathbf{d}$, 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,

$$\mathbf{C} = \begin{bmatrix} \mathbf{z}^\top & \mathbf{0}^\top & \cdots & \mathbf{0}^\top \\ \mathbf{0}^\top & \mathbf{z}^\top & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{0} \\ \mathbf{0}^\top & \cdots & \mathbf{0}^\top & \mathbf{z}^\top \end{bmatrix} \in \mathbb{R}^{n,n^2} \ . \tag{3.9.3}$$

▲

(3-9.b) ⚁ (15 min.) [depends on Sub-problem (3-9.a)]

State a necessary and sufficient condition for the matrix $\mathbf{C}$ found in Sub-problem (3-9.a) to possess full rank.

SOLUTION 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 $\mathbf{z} \neq \mathbf{0}$.

▲

(3-9.c) ⚄ (15 min.) [depends on Sub-problem (3-9.a)]

State the **augmented normal equations** [Lecture → Eq. (3.6.1.6)] corresponding to the constrained linear least squares problem found in Sub-problem (3-9.a) and give necessary and sufficient conditions on $\mathbf{g}$ and $\mathbf{z}$ that ensure existence and uniqueness of solutions.

HINT 1 for (3-9.c): Generic **augmented normal equations** associated with (3.9.2) read

$$\mathbf{S}\begin{bmatrix} \mathbf{m}^* \\ \mathbf{p} \end{bmatrix} := \begin{bmatrix} \mathrm{Id}_{n^2} & \mathbf{C}^\top \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{m}^* \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{g} \end{bmatrix} =: \mathbf{h}. \tag{3.9.4}$$

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 → 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).

▲

(3-9.d) ⚄ (30 min.) [depends on Sub-problem (3-9.c)]

Write a C++ function

```
Eigen::MatrixXd min_frob(const Eigen::VectorXd &z, const
   Eigen::VectorXd &g);
```

that computes the solution $\mathbf{M}^*$ of the minimization problem (3.9.1) given the vectors $\mathbf{z}, \mathbf{g} \in \mathbb{R}^n$. Use the augmented normal equations [Lecture → Eq. (3.6.1.6)] derived in ((3-9.c)).

It is convenient to use EIGEN's built-in Kronecker product class **Eigen::KroneckerProduct** (**#include** `<unsupported/Eigen/KroneckerProduct>` required).

SOLUTION 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

```
2  Eigen::MatrixXd min_frob(const Eigen::VectorXd &z, const Eigen::VectorXd &g) {
3    assert(z.size() == g.size() && "Size mismatch!");
4    const unsigned int n = g.size();
5    Eigen::MatrixXd M_mat = Eigen::MatrixXd::Zero(n, n);
6    // TODO: (3-9.d) solve the augmented normal equations
7    // Hint:  Use Map to reshape a vector into a matrix
8    // START
9    // Build temporary matrix C.
10   Eigen::MatrixXd C =
11       kroneckerProduct(Eigen::MatrixXd::Identity(n, n), z.transpose());
12   Eigen::VectorXd minus_p = (C * C.transpose()).ldlt().solve(g);
13   Eigen::VectorXd M = C.transpose() * minus_p;
14   M_mat = Eigen::Map<Eigen::MatrixXd>(M.data(), n, n).transpose();
15   // END
16   return M_mat;
17 }
```

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.

▲

(3-9.e) ⚂ (15 min.) [depends on Sub-problem (3-9.d)]

Using the test vectors $\mathbf{z} = [1, 2, \ldots, n]^\top \in \mathbb{R}^n$ and $\mathbf{g} = [1, -1, 1, \ldots, 1, -1]^\top \in \mathbb{R}^n$, implement a C++ function

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

that checks *in a numerical experiment* whether

$$\mathbf{M} = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2} \tag{3.9.7}$$

is a solution to (3.9.1). To that end call the function `min_frob` implemented in Sub-problem (3-9.d), conduct the test and return **true** if it succeeds.

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

```
bool testMformula(unsigned int n) {
  constexpr double tol = 1e-10;
  // TODO: (3-9.e) compare the formula:  return true if the relative
  // error in Frobenius norm is less than tol.
  // Hint:  Define z using LinSpaced and g using replicate()
  // START
  Eigen::VectorXd z = Eigen::VectorXd::LinSpaced(n, 1, n);
  Eigen::Vector2d v = {1, -1};
  Eigen::VectorXd g = v.replicate(static_cast<int>(n) / 2, 1);

  Eigen::MatrixXd Mstar = min_frob(z, g);

  Eigen::MatrixXd M = g * z.transpose() / z.squaredNorm();  // M
  // Safe test for realtive smallness of error
  return ((Mstar - M).norm() < tol * M.norm());
  // END
  return false;
}
```

Get it on GitLab (matrixfit.hpp).

▲

(3-9.f) ⚂ (30 min.) [depends on Sub-problem (3-9.c)]

Now, give a rigorous proof that (3.9.7) gives a solution of (3.9.1), provided that $\mathbf{z} \neq \mathbf{0}$.

SOLUTION 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" $\mathbf{m}^*$!

▲

So far, we have simply applied the formulas from [Lecture → Section 3.6.1] to (3.9.1) and its reformulation as a constrained linear least squares problem. Now we take a closer look at the method of Lagrangian multipliers, which was used to derive these equations in class.

(3-9.g) ⚂ (20 min.) As in [Lecture → Eq. (3.6.1.2)] and [Lecture → Eq. (3.6.1.3)] from the lecture notes, find a **Lagrange functional** $L : \mathbb{R}^{n,n} \times \mathbb{R}^n \longrightarrow \mathbb{R}$ such that:

$$\mathbf{M}^* = \operatorname*{argmin}_{\mathbf{M}\in\mathbb{R}^{n,n}} \left\{ \sup_{\mathbf{m}\in\mathbb{R}^n} [L(\mathbf{M},\mathbf{m})] \right\}. \tag{3.9.10}$$

The matrix $\mathbf{M}^*$ should provide the solution of (3.9.1).

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:

$$L(\mathbf{M},\mathbf{m}) = \|\mathbf{M}\|_F^2 + \mathbf{m}^\top(\mathbf{Mz} - \mathbf{g}) \ .$$

▲

(3-9.h) ⚂ (15 min.) Find the derivative $\mathbf{grad}\,\Phi \in \mathbb{R}^{n,n}$ of the function Φ defined as:

$$\Phi : \mathbb{R}^{n,n} \to \mathbb{R}, \quad \Phi(\mathbf{X}) = \|\mathbf{X}\|_F^2 \tag{3.9.11}$$

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:

$$\frac{\partial}{\partial x_{ij}}\Phi(\mathbf{X}) = 2(\mathbf{X})_{ij} \quad \Rightarrow \quad \mathbf{grad}\,\Phi(\mathbf{X}) = 2\mathbf{X} \ . \tag{3.9.12}$$

▲

(3-9.i) ⚂ (20 min.) [depends on Sub-problem (3-9.h)]

Use the result of Sub-problem (3-9.h) to derive **saddle point equations** for $\mathbf{grad}\,L(\mathbf{M},\mathbf{m}) = 0$, for the L obtained in Sub-problem (3-9.g).

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):

Using $\mathbf{grad}\,\Phi(\mathbf{X}) = 2\mathbf{X}$, the derivative of L w.r.t. $\mathbf{M}$ is:

$$\frac{\partial L}{\partial \mathbf{M}} = 2\mathbf{M} + \mathbf{m}\mathbf{z}^\top = 0\,. \tag{3.9.13}$$

The derivative of L w.r.t. $\mathbf{m}$ is:

$$\frac{\partial L}{\partial \mathbf{m}} = \mathbf{M}\mathbf{z} - \mathbf{g} = 0\,. \tag{3.9.14}$$

▲

(3-9.j) ⚀ (15 min.) [depends on Sub-problem (3-9.i)]

Solve the saddle point equations obtained in Sub-problem (3-9.i) in symbolic form.

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 (3.9.13) into (3.9.14) we obtain:

$$-\frac{1}{2}\mathbf{m}\|\mathbf{z}\|_2^2 - \mathbf{g} = 0 \quad\Longrightarrow\quad \mathbf{m} = -2\frac{\mathbf{g}}{\|\mathbf{z}\|_2^2} \quad\overset{((3.9.13))}{\Longrightarrow}\quad \mathbf{M}^* = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2}\,. \tag{3.9.15}$$

The matrix $\mathbf{M}^*$ is the tensor product of two vectors, see [Lecture → Section 1.3.1].

▲

End Problem 3-9 , 205 min.

Problem 3-10: Fitting of (relative) Point Locations from Distances

Given *all* pairwise distance of points on a line, this redundant information can be used to determine their positions through least-squares fitting. Pertinent algorithms are discussed in this problem.

This problem addresses the algorithmic details of the least-squares solution of the sparse overdetermined linear system of equations described in [Lecture → Ex. 3.0.1.9]. The focus is on normal equation methods as introduced in [Lecture → Section 3.2].

The numbers $x_1, \ldots, x_n \in \mathbb{R}$ describe the location of n points lying on a line that is identified with the real axis $\mathbb{R}$. We know measured values for the $m := \binom{n}{2} = \frac{1}{2}n(n-1)$ signed distances $d_{i,j} := x_j - x_i \in \mathbb{R}$, $1 \le i < j \le n$, from which we have to determine the x_i up to a shift. To fix the shift, we set $x_n := 0$.

This amounts to solving the overdetermined linear system of equations

$$\begin{cases} x_j - x_i = d_{i,j}\,, & 1 \le i < j \le n-1\,, \\ -x_i = d_{i,n}\,, & i \in \{1,\ldots,n-1\}\,. \end{cases} \tag{3.10.1}$$

Collecting the unknown positions in the vector $\mathbf{x} := [x_1, \ldots, x_{n-1}]^\top \in \mathbb{R}^{n-1}$, the equations (3.10.1) can be recast as an $m \times (n-1)$ linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{b} \in \mathbb{R}^m$ contains each of the distances $d_{i,j}$, $1 \le i < j \le n$, exactly once.

(3-10.a) ⚄ (30 min.) How many real numbers and integers have to be stored at least in order to represent the matrix $\mathbf{A}$, regarded as a matrix with real entries, in

- COO/triplet format and
- CRS format?

SOLUTION 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,

$$\operatorname{nnz}(\mathbf{A}) = 2\binom{n}{2} - (n-1) = n(n-1) - (n-1) = (n-1)^2 \tag{3.10.3}$$

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 $\operatorname{nnz}(A) = (n-1)^2$ 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).

▲

(3-10.b) ⚀ (25 min.) Based on EIGEN implement (in the file `disfitting.hpp`) an *efficient* C++ function for the initialization of a sparse-matrix data structure for the matrix $\mathbf{A}$:

```
Eigen::SparseMatrix<double> initA(unsigned int n);
```

It goes without saying that the parameter `n` passes the number n of points.

SOLUTION 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).

▲

(3-10.c) ⚀ (45 min.) [depends on Sub-problem (3-10.b)]

According to [Lecture → § 3.2.0.7] the **extended normal equations** for the generic overdetermined linear system $\mathbf{M}\mathbf{x} = \mathbf{c}$, $\mathbf{M} \in \mathbb{R}^{m,\ell}$, $\mathbf{c} \in \mathbb{R}^m$, $m \geq \ell$, boil down to the linear system of equations

$$\begin{bmatrix} -\mathbf{I} & \mathbf{M} \\ \mathbf{M}^\top & \mathbf{O} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix} . \tag{3.10.5}$$

Implement an efficient C++ function

```
Eigen::VectorXd solveExtendedNormalEquations(
  const Eigen::MatrixXd &D);
```

that uses a sparse direct solver provided by EIGEN to solve the extended normal equations for the position fitting problem (3.10.1) and returns its least-squares solution $\mathbf{x} \in \mathbb{R}^{n-1}$. The argument `D` passes an $n \times n$-matrix $\mathbf{D}$ whose strict upper triangular part contains the distances

$$[\mathbf{D}]_{i,j} := d_{i,j} , \quad 1 \leq i < j \leq n .$$

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

$$\mathbf{A}_{\mathrm{ext}} := \begin{bmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^\top & \mathbf{O} \end{bmatrix} \in \mathbb{R}^{n-1+m,n-1+m} \tag{3.10.6}$$

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 $\mathbf{A}_{\mathrm{ext}}$:

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
```

```
  std::vector<Eigen::Triplet<double>> tripletsAext;
  for (unsigned int k = 0; k < m; ++k) {
    // - I block
    tripletsAext.push_back(Eigen::Triplet<double>(k, k, -1.0));
  }
  for (auto t : tripletsA) {
    // A block
    tripletsAext.push_back(
        Eigen::Triplet<double>(t.row(), m + t.col(), t.value()));
    // A.transpose() block
    tripletsAext.push_back(
        Eigen::Triplet<double>(m + t.col(), t.row(), t.value()));
  }

  Eigen::SparseMatrix<double> A_ext(n - 1 + m, n - 1 + m);
  A_ext.setFromTriplets(tripletsAext.begin(), tripletsAext.end());
  // solve extended system and extract the last n-1 entries
  // of the solution vector
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver(A_ext);
  x = solver.solve(b_ext).tail(n - 1);
  // END
  return x;
}
```

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.

▲

(3-10.d) ⚄ (20 min.) [depends on Sub-problem (3-10.b)]

The coefficient matrix $\mathbf{M} \in \mathbb{R}^{n-1,n-1}$ of the **normal equations** [Lecture → Eq. (3.1.2.2)] associated with the overdetermined linear system of equations (3.10.1) is of the form

$$\mathbf{M} = \mathbf{D} - \mathbf{z}\mathbf{z}^\top \quad \text{with} \quad \begin{array}{l} \text{a diagonal matrix} \quad \mathbf{D} \in \mathbb{R}^{n-1,n-1}\,, \\ \text{and a column vector} \quad \mathbf{z} \in \mathbb{R}^{n-1}\,. \end{array} \tag{3.10.8}$$

Compute $\mathbf{D}$ and $\mathbf{z}$, and find a simple expression for the inverse $\mathbf{M}^{-1}$.

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

$$[\mathbf{M}]_{i,j} = \begin{cases} -1 & \text{, if } i \neq j\,, \\ n-1 & \text{, if } i = j \end{cases} \quad \Longrightarrow \quad \mathbf{M} = n\mathbf{I}_{n-1} - \mathbf{1}\mathbf{1}^\top\,. \tag{3.10.9}$$

We conclude that

$$\mathbf{D} = n\mathbf{I}_{n-1} \quad , \quad \mathbf{z} = [1,\ldots,1]^\top \in \mathbb{R}^{n-1} \; .$$

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 $\mathbf{U},\mathbf{V} \in \mathbb{K}^{n,k}$, $n,k \in \mathbb{N}$, $k \le n$, 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.*

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 $\mathbf{V} \leftarrow \mathbf{1}$.

$$\blacktriangleright \quad \mathbf{M}^{-1} = \tfrac{1}{n}I_{n-1} + \tfrac{1}{n}\mathbf{1}\frac{1}{1-\frac{1}{n}\mathbf{1}^\top\mathbf{1}}\tfrac{1}{n}\mathbf{1}^\top = \tfrac{1}{n}\left(\mathbf{I}_{n-1} + \mathbf{1}\mathbf{1}^\top\right) \; , \tag{3.10.10}$$

since $\mathbf{1}^\top\mathbf{1} = n-1$. This formula can be verified directly

$$(n\mathbf{I}_{n-1} - \mathbf{1}\mathbf{1}^\top)\frac{1}{n}\left(\mathbf{I}_{n-1} + \mathbf{1}\mathbf{1}^\top\right) = \mathbf{I}_{n-1} + (-\frac{1}{n} + 1 - \frac{n-1}{n})\mathbf{1}\mathbf{1}^\top = \mathbf{I}_{n-1} \; .$$

▲

(3-10.e) ⚂ (45 min.) [depends on Sub-problem (3-10.b) and Sub-problem (3-10.d)]

Write an *efficient* C++ function

```
Eigen::VectorXd solveNormalEquations(
  const Eigen::MatrixXd &D);
```

that returns the least-squares solution of (3.10.1) by solving the corresponding normal equations. The signature is the same as for `solveExtendedNormalEquations()` from Sub-problem (3-10.c).

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()`

```
Eigen::VectorXd solveNormalEquations(const Eigen::MatrixXd& D) {
  Eigen::VectorXd x;
  const unsigned int n = D.cols();
  const unsigned int m = n * (n - 1) / 2;
  // TODO: (3-10.e) Solve the normal equations by exploiting the
  // Sherman-Morrison-Woodbury formula, i.e. by using your results from the
  // previous subproblem
  // START
  // initialize right hand side
  Eigen::VectorXd b = Eigen::VectorXd::Zero(m);
  unsigned int curr = 0;
  for (unsigned int i = 0; i < n - 1; ++i) {
    for (unsigned int k = i + 1; k < n; ++k) {
      b(curr) = D(i, k);
      ++curr;
    }
  }
  // Direct computation of A^T*b
  Eigen::SparseMatrix<double> AT = initA(n).transpose();
  Eigen::VectorXd ATb = AT * b;

  // Alternative version

  // Initialize the extended D
  Eigen::MatrixXd D_full = Eigen::MatrixXd::Zero(n, n);
  D_full.triangularView<Eigen::Upper>() =
      D.triangularView<Eigen::Upper>();  // in case D is not just a upper
                                         // triangular matrix
  Eigen::MatrixXd D_fullT = D_full.transpose();
  D_full = D_full - D_fullT;
  // Calculate ATb in a more efficient way
  Eigen::VectorXd ATb_alt =
      D_full.transpose().block(0, 0, n - 1, n) * Eigen::VectorXd::Ones(n);

  // We use ATb on forward but ATb == ATb_alt

  // Use explicit expression of the inverse of A^T*A
  // to compute the solution
  x = (Eigen::MatrixXd::Identity(n - 1, n - 1) +
       Eigen::MatrixXd::Ones(n - 1, n - 1)) *
      ATb / n;
  // END
  return x;
}
```

Get it on GitLab (distfitting.hpp).

Of course, the structure of $\mathbf{A}$ could be exploited. First, one completely fills the matrix $\mathbf{D}$:

$$[\mathbf{D}]_{i,j} = \begin{cases} d_{i,j} & \text{, if } i < j\,, \\ 0 & \text{, if } i = j\,, \\ -d_{i,j} & \text{, if } i > j\,. \end{cases}$$

Then we have

$$\left[\mathbf{A}^\top \mathbf{b}\right]_\ell = \sum_{k=1}^{n} [\mathbf{D}]_{k,\ell}\,, \quad \ell = 1,\ldots,n-1 \quad \Rightarrow \quad \mathbf{A}^\top \mathbf{b} = \left[\mathbf{D}^\top\right]_{1:n-1,:} \mathbf{1}\,.$$

▲

(3-10.f) ⚂ (15 min.) [depends on Sub-problem (3-10.c) and Sub-problem (3-10.e)]

Write a C++ function

```
bool testNormalEquations(const MatrixXd &D);
```

that calls both `solveExtendedNormalEquations()` and `solveNormalEquations()` and returns **true**, if the results agree.

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).

▲

(3-10.g) ⚁ (10 min.) [depends on Sub-problem (3-10.e)]

What is the asymptotic complexity of your implementation of `solveNormalEquations()` in Sub-problem (3-10.e) for $n \to \infty$?

SOLUTION of (3-10.g):

The straightforward matrix×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×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.

▲

End Problem 3-10 , 190 min.

Problem 3-11: A Class Representing Low-Rank Matrices

Low-rank matrices play a central role in modern algorithms for data compression and the efficient handling of non-local operations. In this problem we devise a dedicated data type for efficiently storing and manipulating low-rank matrices.

This problem is connected with [Lecture → Section 1.4.3] and [Lecture → Section 3.4.4]. It is closely related to Problem 3-8.

Recall the following result from elementary linear algebra:

Theorem 3.11.1. Representation of low-rank matrices

Every matrix $\mathbf{M} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, *with rank* $r := \operatorname{rank}(\mathbf{M}) > 0$ *can be factored as*

$$\mathbf{M} = \mathbf{A}\mathbf{B}^\top \quad \text{with} \quad \begin{array}{l} \mathbf{A} \in \mathbb{R}^{m,r} \,, \\ \mathbf{B} \in \mathbb{R}^{n,r} \,. \end{array}$$

This factorization is used in the design of the following EIGEN-based C++ class (File `matrixlowrank.hpp`) meant to store real matrices $\mathbf{M} \in \mathbb{R}^{m,n}$ with maximal rank $r < \min\{m,n\}$ and to provide special operations on them:

```
class MatrixLowRank {
public:
  MatrixLowRank(unsigned int m, unsigned int n, unsigned int r);
  MatrixLowRank(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B);

  Eigen::Index rows() const { return _m; };
  Eigen::Index cols() const { return _n; };
  Eigen::Index rank() const { return _r; };

  Eigen::MatrixXd operator*(const Eigen::MatrixXd &)const;
  MatrixLowRank &operator*=(const Eigen::MatrixXd &);
  MatrixLowRank &addTo(const MatrixLowRank &, double tol = 1E-6);
private:
  unsigned int _m;     // num.  of rows
  unsigned int _n;     // num.  of columns
  unsigned int _r;     // maximal rank, =1 for zero matrix
  Eigen::MatrixXd _A; // factor matrix A
  Eigen::MatrixXd _B; // factor matrix B
};
```

The convention for the case $\mathbf{M} = \mathbf{O}$ is that we set `_r = 1` and the low-rank factors to zero.

(3-11.a) ⚀ (15 min.)

Give a proof of Thm. 3.11.1.

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:

$$\begin{aligned} &\exists \mathbf{b}_j \in \mathbb{R}^n : \quad \eta_j(\mathbf{x}) = \mathbf{b}_j^\top \mathbf{x} , \quad j = 1, \ldots, r , \\ \blacktriangleright \quad &\mathbf{M}\mathbf{x} = \sum_{j=1}^{r} \mathbf{a}_j \mathbf{b}_j^\top \mathbf{x} = \mathbf{A}\mathbf{B}^\top \mathbf{x} , \quad \begin{aligned} \mathbf{A} &:= [\mathbf{a}_1, \ldots, \mathbf{a}_r] , \\ \mathbf{B} &:= [\mathbf{b}_1, \ldots, \mathbf{b}_r] . \end{aligned} \end{aligned}$$

▲

(3-11.b) ⚂ (15 min.) In the file `matrixlowrank.hpp` implement the method

```
Eigen::MatrixXd MatrixLowRank::operator *
  (const Eigen::MatrixXd &X) const;
```

which is supposed to return the product of the $m \times n$ low-rank matrix stored in the **MatrixLowRank** object (left factor) with a generic dense matrix $\mathbf{X} \in \mathbb{R}^{n,k}$ (right factor). Take care, that your code is efficient.

SOLUTION 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]:

$$\mathbf{M}\mathbf{X} = \mathbf{A}(\mathbf{B}^\top \mathbf{X}) , \quad \mathbf{X} \in \mathbb{R}^{n,k} .$$

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).

▲

(3-11.c) ⚀ (5 min.) [depends on Sub-problem (3-11.b)]

What is the asymptotic complexity of your implementation of `MatrixLowRank::operator *` from Sub-problem (3-11.b) in terms of $m, n, k \to \infty$, $r := \operatorname{rank}(\mathbf{M})$ fixed. Here, the size of the argument matrix is $n \times k$, $k \in \mathbb{N}$, and the **MatrixLowRank** object stores $\mathbf{M} \in \mathbb{R}^{m,n}$.

SOLUTION 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.

▲

(3-11.d) ⚄ (10 min.) Provide an efficient implementation of the method (File `matrixlowrank.hpp`)

```
MatrixLowRank &operator *= (const Eigen::MatrixXd &X);
```

for the *in-situ (right) multiplication* of a low-rank $m \times n$ matrix stored in a **MatrixLowRank** object with a generic matrix $\mathbf{X} \in \mathbb{R}^{n,k}$. "In-situ" means that after the operation the **MatrixLowRank** object stores the result of the matrix multiplication. The "internal rank" `_r` of the **MatrixLowRank** object should not change.

SOLUTION 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** $*=$ 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`!

▲

(3-11.e) ⚄ (40 min.) With an eye on efficiency, in the file `matrixlowrank.hpp` complete the implementation of the method

```
MatrixLowRank &MatrixLowRank::addTo(
  const MatrixLowRank &X, double atol, double rtol);
```

that replaces the matrix $\mathbf{M} \in \mathbb{R}^{m,n}$ stored in the **MatrixLowRank** object with $\operatorname{trunc}_{\mathtt{tol}}(\mathbf{M}+\mathbf{X})$, where

- $\mathbf{X} \in \mathbb{R}^{m,n}$ is the matrix passed through the argument `X`,
- and, for $\mathbf{Y} \in \mathbb{R}^{m,n}$,

$$\operatorname{trunc}_{\mathtt{tol}}(\mathbf{Y}) := \operatorname{argmin}\left\{ \operatorname{rank}(\mathbf{R}) : \mathbf{R} \in \mathbb{R}^{m,n}, \begin{array}{c} \|\mathbf{Y}-\mathbf{R}\|_2 \leq \mathrm{rtol}\|\mathbf{Y}\|_2\,, \\ \textbf{or} \\ \|\mathbf{Y}-\mathbf{R}\|_2 \leq \mathrm{atol} \end{array} \right\},$$

where $\|\cdot\|_2$ designates the Euclidean matrix norm. The "internal rank" `_r` must be updated.

Your implementation may assume that

$$\operatorname{rank}(\mathbf{M}) + \operatorname{rank}(\mathbf{X}) \leq \min\{m,n\}\,.$$

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

$$\mathbf{M}+\mathbf{X} = \mathbf{A}\mathbf{B}^\top + \mathbf{A}_X\mathbf{B}_X^\top = \left[\mathbf{A},\mathbf{A}_X\right]\left[\mathbf{B},\mathbf{B}_X\right]^\top =: \widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top\,.$$

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

$$\mathbf{R}_A\mathbf{R}_B^\top = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top\,, \quad \mathbf{U},\mathbf{V} \in \mathbb{R}^{r+s,r+s} \text{ orthogonal},\ \boldsymbol{\Sigma} \in \mathbb{R}^{r+s,r+s} \text{ diagonal}\,,$$

for the singular-value decomposition of $\mathbf{R}_A\mathbf{R}_B^\top$, we obtain an economical singular value decomposition of $\widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top$:

$$\widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top = \widetilde{\mathbf{U}}\widetilde{\boldsymbol{\Sigma}}\widetilde{\mathbf{V}}^\top \quad , \quad \begin{aligned} \widetilde{\mathbf{U}} &:= \mathbf{Q}_A\mathbf{U} \in \mathbb{R}^{m,r+s}\,, \\ \widetilde{\boldsymbol{\Sigma}} &:= \boldsymbol{\Sigma} \in \mathbb{R}^{r+s,r+s} \\ \widetilde{\mathbf{V}} &:= \mathbf{Q}_B\mathbf{V} \in \mathbb{R}^{n,r+s}\,. \end{aligned}$$

Denote by $\sigma_1 \ge \sigma_2 \ge \cdots \ge \sigma_{r+s} \ge 0$ the sorted diagonal entries of $\widetilde{\boldsymbol{\Sigma}}$, that is, the singular values of $\widetilde{\mathbf{A}}\widetilde{\mathbf{B}}^\top$. Define (for $\mathtt{tol} < 1$)

$$\widetilde{r} := \max\left\{ r' \in \{1,\ldots,r+s\} : \frac{\sigma_{r'}}{\sigma_1} > \mathtt{rtol},\ \sigma_{r'} > \mathrm{atol} \right\}\,. \tag{3.11.4}$$

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 $\le \mathtt{rtol}$ and an absolute error $\le \mathrm{atol}$ in the Euclidean matrix norm. That matrix, realizing the best approximation is

$$\mathrm{trunc}_{\mathtt{tol}} = \underbrace{\left(\mathbf{Q}_A[\mathbf{U}]_{:,1:\widetilde{r}}[\boldsymbol{\Sigma}]_{1:\widetilde{r},1:\widetilde{r}}\right)}_{=:\mathbf{A}_{\mathrm{new}}} \underbrace{\left(\mathbf{Q}_B[\mathbf{V}]_{:,1:\widetilde{r}}\right)^\top}_{=:\mathbf{B}_{\mathrm{new}}}\,,$$

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 $\sigma_1 \leq$ `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++;
  }

```

```
52    _r = rtilde;
53    _A = qrA.householderQ() * svdRR.matrixU().leftCols(rtilde) *
54         Sigma.head(rtilde).asDiagonal();
55    _B = qrB.householderQ() * svdRR.matrixV().leftCols(rtilde);
56    // END
57
58    return *this;
59  }
```

Get it on GitLab (matrixlowrank.hpp).

▲

End Problem 3-11 , 85 min.

Problem 3-12: Polar Decomposition of Matrices

In class we have seen various product decompositions/factorizations of matrices like the LU-decomposition [Lecture → Section 2.3.2], the QR-decomposition [Lecture → Section 3.3.3], and the singular-value decomposition (SVD) [Lecture → Section 3.4]. In this problems we study another factorization, which is sometimes used in numerical methods, though it is not as important as the decompositions listed before.

This problem is related to [Lecture → Section 3.3.3.2] and [Lecture → Section 3.4.2] and requires familiarity with the EIGEN-based C++ implementation discussed in those sections.

Theorem 3.12.1. Polar decomposition

For every matrix $\mathbf{X} \in \mathbb{R}^{m,n}$, $m \geq n$, there is a matrix $\mathbf{Q} \in \mathbb{R}^{m,n}$ with orthonormal columns, $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}_n$, and a symmetric positive semi-definite [Lecture → Def. 1.1.2.6] matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ such that $\mathbf{X} = \mathbf{Q}\mathbf{M}$.

The matrix factorization postulated in Thm. 3.12.1 is called **polar decomposition** of $\mathbf{X}$.

(3-12.a) ⚄ (30 min.) Give a proof of Thm. 3.12.1.

HINT 1 for (3-12.a): Thm. 3.12.1 can be shown based on the singular value decomposition (SVD) of $\mathbf{X}$ [Lecture → Thm. 3.4.1.1]. ⌟

SOLUTION of (3-12.a):

Recall the economical SVD of $\mathbf{X} \in \mathbb{R}^{m,n}$, $m \geq n$:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top, \quad \mathbf{U} \in \mathbb{R}^{m,n}, \ \mathbf{U}^\top\mathbf{U} = \mathbf{I}_n, \quad \mathbf{V} \in \mathbb{R}^{n,n} \text{ orthogonal},$$

where $\mathbf{\Sigma} \in \mathbb{R}^{n,n}$ is a diagonal matrix with non-negative diagonal entries. Setting

$$\mathbf{Q} := \mathbf{U}\mathbf{V}^\top \quad \text{and} \quad \mathbf{M} := \mathbf{V}\mathbf{\Sigma}\mathbf{V}^\top \tag{3.12.2}$$

we have already found the polar decomposition of $\mathbf{X}$, because

$$\begin{aligned}
&\mathbf{M}^\top = \left(\mathbf{V}\mathbf{\Sigma}\mathbf{V}^\top\right)^\top = \left(\mathbf{V}^\top\right)^\top \mathbf{\Sigma}^\top \mathbf{V}^\top = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{M}\,,\\
&\mathbf{Q}\mathbf{M} = \mathbf{U}\mathbf{V}^\top\mathbf{V}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{X} \quad \mathbf{V} \text{ orthogonal!}\,,\\
&\mathbf{Q}^\top\mathbf{Q} = \mathbf{V}\mathbf{U}^\top\mathbf{U}\mathbf{V}^\top = \mathbf{V}\mathbf{V}^\top = \mathbf{I}_n\,,\\
&\mathbf{z}^\top\mathbf{M}\mathbf{z} = (\mathbf{V}\mathbf{z})^\top\mathbf{\Sigma}(\mathbf{V}\mathbf{z}) \geq 0 \quad \forall \mathbf{z} \in \mathbb{R}^n\,.
\end{aligned}$$

▲

Based on the data types of EIGEN, the polar decomposition of a "tall/slim" real matrix is to be implemented as the following C++ class.

```
class PolarDecomposition {
 public:
  explicit PolarDecomposition(const Eigen::MatrixXd &X) { initialize(X); }
  PolarDecomposition(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B);
  PolarDecomposition(const PolarDecomposition &) = default;
  ~PolarDecomposition() = default;
```

```
  // Left multiplication of M with the Q-factor of the polar decomposition
  void applyQ(Eigen::MatrixXd &Y) const { Y.applyOnTheLeft(Q_); }
  // Left multiplication of M with the M-factor of the polar decomposition
  void applyM(Eigen::MatrixXd &Y) const { Y.applyOnTheLeft(M_); }

 private:
  void initialize(const Eigen::MatrixXd &X);
  Eigen::MatrixXd Q_;  // factor Q
  Eigen::MatrixXd M_;  // factor M
};
```

The following specification of the class if given:

- The constructor

  ```
  PolarDecomposition(const Eigen::MatrixXd &X);
  ```

  should compute the polar decomposition factors $\mathbf{Q}$ and $\mathbf{M}$ according to Thm. 3.12.1 of the matrix passed in `X` and store them in the data members `Q_` and `M_`.

- The constructor

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

  is supposed the initialize the data members `Q_` and `M_` with the polar decomposition factors $\mathbf{Q} \in \mathbb{R}^{m,n}$ and $\mathbf{M} \in \mathbb{R}^{n,n}$ of the matrix $\mathbf{AB}^\top \in \mathbb{R}^{m,n}$, where the matrices $\mathbf{A} \in \mathbb{R}^{m,k}$ and $\mathbf{B} \in \mathbb{R}^{n,k}$ are passed through the arguments `A` and `B`.

- The methods `applyQ()` and `applyM()` realize the operations

  $$\mathbf{Y} \leftarrow \mathbf{QY} \quad , \quad \mathbf{Y} \leftarrow \mathbf{MY} \, ,$$

  where $\mathbf{Q}$ and $\mathbf{M}$ are the factors of the polar decomposition stored in the **PolarDecomposition** object.

(3-12.b) ⚀ (15 min.) Regardless of the implementation, what is the *minimal* asymptotic computational cost, that is, a *sharp lower bound* of the asymptotic computational effort, for a call of the second constructor

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

of a **PolarDecomposition** object for a $m \times n$-matrix, $m \geq n$, for $m, n \to \infty$, and small fixed k?

Minimal asymptotic cost $= O($ ☐ $)$ for $m, n \to \infty$.

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.

▲

(3-12.c) ⚂ (30 min.) [depends on Sub-problem (3-12.a)]

In the file `polardecomposition.hpp` implement the method

```
void PolarDecomposition::intialize(const Eigen::MatrixXd &X);
```

that sets the data members `_Q` and `_M` of the class **PolarDecomposition**. These data members store the factors $\mathbf{Q}$ and $\mathbf{M}$ of the polar decomposition of the argument matrix `X` according to Thm. 3.12.1.

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    // TO DO: 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(X, Eigen::ComputeThinU |
9                                             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).

▲

(3-12.d) ⚄ (120 min.) [depends on Sub-problem (3-12.b)]

Write a code for the constructor

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

which is supposed to initialize the data members `Q_` and `M_` with the polar decomposition factors $\mathbf{Q} \in \mathbb{R}^{m,n}$ and $\mathbf{M} \in \mathbb{R}^{n,n}$ of the matrix $\mathbf{X} := \mathbf{A}\mathbf{B}^\top \in \mathbb{R}^{m,n}$, where the matrices $\mathbf{A} \in \mathbb{R}^{m,k}$ and $\mathbf{B} \in \mathbb{R}^{n,k}$ are passed through the arguments `A` and `B`. We assume that k is small and fixed and $k \leq n < m$. Your code should have *optimal complexity* with respect to $m, n \to \infty$.

HINT 1 for (3-12.d): One idea is to harness economical/thin QR-decompositions. ⌟

SOLUTION of (3-12.d):

You cannot simply call `intialize(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 $\mathbf{R}_A\mathbf{R}_B^\top$

$$\mathbf{R}_A\mathbf{R}_B^\top \in \mathbb{R}^{k,k}\text{:}\quad \mathbf{R}_A\mathbf{R}_B^\top = \widehat{\mathbf{U}}\widehat{\mathbf{\Sigma}}\widehat{\mathbf{V}}^\top\,. \tag{3.12.4}$$

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.

```
2  PolarDecomposition::PolarDecomposition(const Eigen::MatrixXd &A,
3                                         const Eigen::MatrixXd &B) {
4    const long m = A.rows();  // No. of rows of X
5    const long n = B.rows();  // No. of columns of X
6    const long k = A.cols();  // Maximal rank of X
7    // We assume k <= n <= m
8    assert(m >= n);
9    assert(k < n);
10   assert(B.cols() == k);
11   // TO DO: Implement a method to initialize the data members Q_ and M_
12   // for X := AB^T = QM, with optimal complexity
13   // START
14   // Compute QR-decompositions in an encoded way, see \lref{par:ecovsfull}
15   Eigen::HouseholderQR<Eigen::MatrixXd> QRA(A);  // cost = O(mk^2)
16   Eigen::HouseholderQR<Eigen::MatrixXd> QRB(B);  // cost = O(nk^2)
17   const Eigen::MatrixXd RA{
18       QRA.matrixQR().block(0, 0, k, k).template triangularView<Eigen::Upper>()};
19   const Eigen::MatrixXd RB{
20       QRB.matrixQR().block(0, 0, k, k).template triangularView<Eigen::Upper>()};
21   // SVD of small k x k-matrix R_A R_B^T
22   // cost = O(k^3)
23   Eigen::JacobiSVD<Eigen::MatrixXd> svdh(
24       RA * RB.transpose(), Eigen::ComputeFullU | Eigen::ComputeFullV);
25   // Extract kxk orthogonal factor matrices
26   const Eigen::MatrixXd &Vh{svdh.matrixV()};
27   const Eigen::MatrixXd &Uh{svdh.matrixU()};
28   // Build auxiliary mxn block matrix
29   Eigen::MatrixXd W{Eigen::MatrixXd::Zero(m, n)};  // cost = O(mn)
30   W.block(0, 0, k, k) = Uh * Vh.transpose();       // cost = O(k^3)
31   W.block(k, k, n - k, n - k) = Eigen::MatrixXd::Identity(n - k, n - k);
32   // Compute the Q-factor, cost = O(kmn + kn^2)
33   Q_ =
34       (QRB.householderQ() * ((QRA.householderQ() * W).transpose())).transpose();
35   // Small kxk matrix containing singular values on diagonal
36   const auto Sigma{svdh.singularValues().asDiagonal()};
37   // Form M-factor of polar decomposition
38   // 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).

▲

End Problem 3-12 , 195 min.

Problem 3-13: QR-Iteration

We have learned about the QR-decomposition/factorization as a tool for solving linear systems of equations [Lecture → Rem. 3.3.4.3] and linear least-squares problems [Lecture → Section 3.3.4]. Yet, this matrix factorization has many more important applications. For instance, it is a crucial ingredient for the so-called QR-algorithm, an iteration for solving algebraic eigenvalue problems [**GOL13**]. This problem will examine some algorithmic aspects of that algorithm.

You are supposed to know the contents of [Lecture → Section 3.3.3] and implementation in EIGEN.

We recall the QR-decomposition of matrices:

Theorem [Lecture → Thm. 3.3.3.4]. QR-decomposition

For any matrix $\mathbf{A} \in \mathbb{K}^{n,k}$ *with* $\operatorname{rank}(\mathbf{A}) = k$ *there exists*

(i) a unique *Matrix* $\mathbf{Q}_0 \in \mathbb{R}^{n,k}$ *that satisfies* $\mathbf{Q}_0^{\mathrm{H}}\mathbf{Q}_0 = \mathbf{I}_k$, *and a* unique *upper triangular Matrix* $\mathbf{R}_0 \in \mathbb{K}^{k,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\ldots,k\}$, *such that*

$$\mathbf{A} = \mathbf{Q}_0 \cdot \mathbf{R}_0 \qquad (\textit{"economical" QR-decomposition}) ,$$

(ii) a unitary Matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$ *and a* unique *upper triangular* $\mathbf{R} \in \mathbb{K}^{n,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\ldots,n\}$, *such that*

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \qquad (\textit{full QR-decomposition}) .$$

If $\mathbb{K} = \mathbb{R}$ *all matrices will be real and* $\mathbf{Q}$ *is then orthogonal.*

(3-13.a) ⚄ (30 min.) Prove the following fact:

Lemma 3.13.1. Lower bandwidth of Q-factor

Let $\mathbf{T} = \mathbf{QR}$ *be a QR-decomposition of a regular tridiagonal matrix* $\mathbf{T} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$. *Then* $\mathbf{Q}$ *has lower bandwidth 1, that is,*

$$i,j \in \{1,\ldots,n\} , \quad i > j+1 \quad \Rightarrow \quad (\mathbf{Q})_{i,j} = 0 .$$

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

$$\operatorname{Span}\{(\mathbf{Q})_{:,1}, \ldots, (\mathbf{Q})_{:,m}\} = \operatorname{Span}\{(\mathbf{T})_{:,1}, \ldots, (\mathbf{T})_{:,m}\} \subset \operatorname{Span}\{\mathbf{e}_1, \ldots, \mathbf{e}_{m+1}\} ,$$

$m \in \{1, \ldots, n\}$, where $\mathbf{e}_\ell \in \mathbb{R}^n$ stands for the ℓ-th coordinate vector. Hence,

$$\begin{array}{c} (\mathbf{Q})_{:,m} \in \operatorname{Span}\{\mathbf{e}_1, \ldots, \mathbf{e}_{m+1}\} \\ \Updownarrow \\ (\mathbf{Q})_{:,m} = \xi_1 \mathbf{e}_1 + \xi_2 \mathbf{e}_2 + \cdots + \xi_m \mathbf{e}_m + \xi_{m+1} \mathbf{e}_{m+1} , \quad \xi_1, \ldots, \xi_{m+1} \in \mathbb{R} , \end{array}$$

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.

▲

(3-13.b) ⚂ (30 min.) Give a proof of the following result:

Lemma 3.13.3. QR-based similarity transform

If $\mathbf{T} = \mathbf{Q}\mathbf{R}$ is the QR-decomposition of a regular, symmetric, and tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, then $\mathbf{T}' := \mathbf{R}\mathbf{Q}$ is also symmetric and tridiagonal.

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

$$i,j \in \{1,\ldots,n\}\text{:} \quad \begin{matrix} i > j+1 \quad \Rightarrow \quad (\mathbf{M})_{i,j} = 0\,, \\ (\mathbf{M})_{i,j} = (\mathbf{M})_{j,i} \end{matrix} \quad \blacktriangleright \quad (\mathbf{M})_{i,j} = 0 \quad \text{for} \quad |i-j| > 1\,.$$

▲

(3-13.c) ⚄ (120 min.) A *symmetric* and *tridiagonal* matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ can be encoded by two vectors $\mathbf{d} = \mathbf{d}(\mathbf{T}) \in \mathbb{R}^n$ and $\mathbf{u} = \mathbf{u}(\mathbf{T}) \in \mathbb{R}^{n-1}$ when setting

$$(\mathbf{T})_{i,j} = \begin{cases} (\mathbf{d})_i & \text{, if } \quad i = j\,, \\ (\mathbf{u})_i & \text{, if } \quad j = i+1\,, \\ (\mathbf{u})_j & \text{, if } \quad i = j+1\,, \\ 0 & \text{else.} \end{cases} \qquad i,j \in \{1,\ldots,n\}:$$

$$\blacktriangleright \quad \mathbf{T} = \begin{bmatrix} (\mathbf{d})_1 & (\mathbf{u})_1 & 0 & \ldots & & & \ldots & 0 \\ (\mathbf{u})_1 & (\mathbf{d})_2 & (\mathbf{u})_2 & 0 & & & & 0 \\ 0 & (\mathbf{u})_2 & (\mathbf{d})_3 & (\mathbf{u})_3 & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & 0 & (\mathbf{u})_{n-2} & (\mathbf{d})_{n-1} & (\mathbf{u})_{n-1} \\ 0 & \ldots & & & & 0 & (\mathbf{u})_{n-1} & (\mathbf{d})_n \end{bmatrix}.$$

In the file `qriteration.hpp` implement a C++ function

```
std::pair<Eigen::VectorXd, Eigen::VectorXd>
  qrStep(const Eigen::VectorXd &dT,
         const Eigen::VectorXd &uT);
```

that takes the vectors $\mathbf{d}(\mathbf{T}) \in \mathbb{R}^n$ and $\mathbf{u}(\mathbf{T}) \in \mathbb{R}^{n-1}$ describing a symmetric, tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ as arguments, and

- returns the tuple $(\mathbf{d}(\mathbf{T}'), \mathbf{u}(\mathbf{T}'))$ of the defining vectors $\mathbf{d}(\mathbf{T}')$ and $\mathbf{u}(\mathbf{T}')$ of the symmetric, tridiagonal matrix $\mathbf{T}' := \mathbf{Q}^\top \mathbf{T}\mathbf{Q}$, where $\mathbf{Q}$ is the Q-factor of the QR-decomposition $\mathbf{T} = \mathbf{Q}\mathbf{R}$ of $\mathbf{T}$, *cf.* Lemma 3.13.1,
- and features an asymptotic complexity of $O(n)$ for $n \to \infty$.

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 $\mathbf{U}$:

$$(\mathbf{U})_{i-j+3,j} := \left(\mathbf{T}^{(k)}\right)_{i,j} \quad \text{for} \quad i-j \in \{-2,-1,0,1\}\,, \quad j=1,\ldots,n:$$

$$\blacktriangleright \quad \mathbf{U} = \begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \ldots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \ldots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \ldots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \ldots & t_{n,n-1} & 0 \end{bmatrix} \quad , \quad t_{i,j} := \left(\mathbf{T}^{(k)}\right)_{i,j}\,.$$

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:

$$\begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \ldots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \ldots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \ldots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \ldots & t_{n,n-1} & 0 \end{bmatrix}.$$

We write `givens` for a function computing the relevant parameters $c := \cos\varphi$ and $s := \sin\varphi$ of the Givens rotation. Then

$$\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)}) = \begin{bmatrix} c & -s & 0 & \ldots & 0 \\ s & c & & & \vdots \\ 0 & & 1 & & \\ \vdots & & & \ddots & \vdots \\ 0 & \ldots & & \ldots & 1 \end{bmatrix}, \quad \begin{bmatrix} c \\ s \end{bmatrix} = \texttt{givens}\left(\begin{bmatrix} t_{11} \\ t_{21} \end{bmatrix}\right),$$

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 $\mathbf{G}_{2,3}^2(\mathbf{T}^{(2)})$.

$$\mathbf{G}_{2,3}^1(\mathbf{T}^{(2)}) \cdot \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} .$$

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, $\mathbf{T}^{(k)} \to \mathbf{T}^{(k+1)}$, $k = 1, \dots, n-1$, we have to perform the following transformation of $\mathbf{U}$: For $k = 1, 2, \dots, n-1$

$$\begin{bmatrix} c \\ s \end{bmatrix} := \texttt{givens}\left(\begin{bmatrix} (\mathbf{U})_{3,k} \\ (\mathbf{U})_{4,k} \end{bmatrix} \right) ,$$

$$\begin{bmatrix} (\mathbf{U})_{3,k} \\ (\mathbf{U})_{4,k} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{3,k} \\ (\mathbf{U})_{4,k} \end{bmatrix} , \quad \begin{bmatrix} (\mathbf{U})_{2,k+1} \\ (\mathbf{U})_{3,k+1} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{2,k+1} \\ (\mathbf{U})_{3,k+1} \end{bmatrix} ,$$

$$\begin{bmatrix} (\mathbf{U})_{1,k+2} \\ (\mathbf{U})_{2,k+2} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{1,k+2} \\ (\mathbf{U})_{2,k+2} \end{bmatrix} \quad \text{, if} \quad k < n-1 .$$

Eventually, $\mathbf{U}$ will encode the $\mathbf{R}$-factor of the QR-decomposition of $\mathbf{T}$:

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & r_{1,3} & r_{2,4} & r_{3,5} & \dots & r_{n-3,n-1} & r_{n-2,n} \\ 0 & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,5} & \dots & r_{n-2,n-2} & r_{n-1,n} \\ r_{1,1} & r_{2,2} & r_{3,3} & r_{4,4} & r_{5,5} & \dots & r_{n-1,n-1} & r_{n,n} \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix} \quad , \quad r_{i,j} := (\mathbf{R})_{i,j} .$$

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

$$\left(\mathbf{G}_{i,j}^{\ell}\right)^{\top} = \begin{bmatrix} 1 & 0 & \dots & & & & & & \dots & 0 \\ 0 & \ddots & \ddots & & & & & & & \vdots \\ \vdots & \ddots & 1 & 0 & & & & & & \\ & & 0 & c & 0 & \dots & 0 & s & & \\ & & & 0 & 1 & & & 0 & & \\ & & & \vdots & & \ddots & & \vdots & & \\ & & & 0 & & & 1 & 0 & & \\ & & & -s & 0 & \dots & 0 & c & 0 & \\ & & & & & & & 0 & 1 & \ddots & \vdots \\ \vdots & & & & & & & & \ddots & \ddots & 0 \\ 0 & \dots & & & & & & & \dots & 0 & 1 \end{bmatrix}, \quad i,j \in \{1,\dots,n\},\ i \neq j\,,$$

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:

$$\begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}_{2,3}^{2}\right)^{\top} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix},$$

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:

$$\begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}_{2,3}^2\right)^\top = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & \star & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & * & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

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(n);
8    Eigen::VectorXd uT_p(n - 1);
9
10   // TO DO : Compute the defining vectors d(T') and u(T') of T':= RQ
         with O(n)
11   // asymptotic complexity, given the defining vectors d(T) and U(T)
         of T = QR
12   // START
13   // Auxiliary 4xn matrix U ∈ R^{4,n}
14   Eigen::MatrixXd U{Eigen::MatrixXd::Zero(4, n)};
15   U.block(1, 1, 1, n - 1) = uT.transpose();
```

```
16    U.block(2, 0, 1, n) = dT.transpose();
17    U.block(3, 0, 1, n - 1) = uT.transpose();
18    // Stage I: QR decomposition by n-1 Givens rotations
19    std::vector<Eigen::Matrix2d> givensrots;  // Store rotations!
20    for (long i = 0; i < n - 1; ++i) {
21      const Eigen::Vector2d g = givens(Eigen::Vector2d(U(2, i), U(3, i)));
22      const Eigen::Matrix2d G{
23          (Eigen::Matrix2d(2, 2) << g[0], -g[1], g[1], g[0]).finished()};
24      U.block(2, i, 2, 1).applyOnTheLeft(G);
25      U.block(1, i + 1, 2, 1).applyOnTheLeft(G);
26      if (i < n - 2) {
27        U.block(0, i + 2, 2, 1).applyOnTheLeft(G);
28      }
29      givensrots.push_back(G);
30    }
31    // At this point U encodes the R-factor of the QR-decomposition of
32    // T.  In particular, the bottom row of U should contain
33    // only entries that are "numerically zero".
34
35    // Stage II: Apply transposed rotations from right to R according to
36    // formulas (3.13.6)
37    for (long i = 1; i < n; ++i) {
38      const Eigen::Matrix2d &G{givensrots[i - 1]};  // An alias
39      Eigen::Vector2d tmp;
40      tmp = G * Eigen::Vector2d(U(3, i - 1), U(2, i));
41      U(3, i - 1) = tmp[0];
42      U(2, i) = tmp[1];
43      tmp = G * Eigen::Vector2d(U(2, i - 1), U(1, i));
44      U(2, i - 1) = tmp[0];
45      U(1, i) = tmp[1];
46      tmp = G * Eigen::Vector2d(U(1, i - 1), U(0, i));
47      U(1, i - 1) = tmp[0];
48      U(0, i) = tmp[1];
49    }
50    // Extract diagonal and sub-/super-diagonal of T'.'
51    dT_p = U.block(2, 0, 1, n).transpose();
52    uT_p = U.block(1, 1, 1, n - 1).transpose();
53    // END
54
55    return {dT_p, uT_p};
56  }
```

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 $n \to \infty$.

▲

End Problem 3-13 , 180 min.

Problem 3-14: Economical Singular-Value Decomposition

For most applications requiring the singular-value decomposition (SVD) of matrix, its economical (thin) variant is sufficient. This problems examines some of its features.

This problems is based on the contents of [Lecture → Section 3.4.1].

Given a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, $\mathbf{A} \neq \mathbf{O}$, we write $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ for its **economical (thin) SVD**.

(3-14.a) ⚀ (12 min.) Fill in the correct matrix sizes

$\mathbf{U} \in \mathbb{R}^{\square,\square}$,

$\mathbf{\Sigma} \in \mathbb{R}^{\square,\square}$,

$\mathbf{V} \in \mathbb{R}^{\square,\square}$.

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):

$$\mathbf{U} \in \mathbb{R}^{m,\min\{m,n\}} \quad , \quad \mathbf{\Sigma} \in \mathbb{R}^{\min\{m,n\},\min\{m,n\}} \quad , \quad \mathbf{V} \in \mathbb{R}^{n,\min\{m,n\}} \, .$$

▲

(3-14.b) ⚀ (24 min.) Decide, whether the following statements are true for *every* $\mathbf{A} \in \mathbb{R}^{m,n} \setminus \{\mathbf{O}\}$, $m,n \in \mathbb{N}$.

1. The matrix $\mathbf{U}$ is an orthogonal matrix.

   ○ TRUE ○ FALSE

2. The set of all columns of $\mathbf{U}$ is an orthonormal basis of $\mathcal{R}(\mathbf{A})$.

   ○ TRUE ○ FALSE

3. $\mathbf{V}$ has orthogonal rows.

   ○ TRUE ○ FALSE

4. $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$.

○ TRUE ○ FALSE

5. $\mathrm{nnz}(\mathbf{\Sigma}) := \sharp\{(i,j) : (\mathbf{\Sigma})_{i,j} \neq 0\} \leq n$.

○ TRUE ○ FALSE

6. $\mathbf{U}\mathbf{V}^\top = \mathbf{I}$.

○ TRUE ○ FALSE

SOLUTION 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 $m < n$, $\mathbf{V}\mathbf{V}^\top = \mathbf{I}$.
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].

▲

(3-14.c) ⚀ (12 min.) What is the asymptotic computational effort for computing the economical SVD of a dense matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$?

$$\mathrm{cost}(\text{economical SVD of } \mathbf{A} \in \mathbb{R}^{m,n}) = O(\quad\quad)$$

for $m,n \to \infty$.

SOLUTION of (3-14.c):

According to [Lecture → § 3.4.2.2] we have

$$\mathrm{cost}(\text{economical SVD of } \mathbf{A} \in \mathbb{R}^{m,n}) = O(\min\{m,n\}^2 \max\{m,n\}) \quad \text{for} \quad m,n \to \infty\,.$$

▲

End Problem 3-14, 48 min.

Problem 3-15: Low-rank Approximation of Concatenated Matrices

Low-rank approximation is a widely used classical technique for the compression of data sets. In this problem we study the recompression of two already compressed data sets after they have been merged. In linaer-algebra terms this boils down to computing the economical singular-value decomposition of a matrix arising from concatenating two low-rank factorized matrices.

This problem requires the concepts, techniques and results presented in [Lecture → Section 3.4.2] and [Lecture → Section 3.4.4.2].

Given are two **low-rank matrices** in factorized form

$$\begin{aligned} \mathbf{X}_1 &:= \mathbf{A}_1\mathbf{B}_1^\top & \mathbf{A}_1, \mathbf{A}_2 &\in \mathbb{R}^{m,k} , \\ \mathbf{X}_2 &:= \mathbf{A}_2\mathbf{B}_2^\top \quad , & \mathbf{B}_1, \mathbf{B}_2 &\in \mathbb{R}^{n,k} , \end{aligned} \tag{3.15.1}$$

for $m, n \in \mathbb{N}$, $k \leq \min\{m, n\}$. We "concatenate" the two matrices $\mathbf{X}_1$ and $\mathbf{X}_2$ into

$$\mathbf{X} := [\mathbf{X}_1 \ \mathbf{X}_2] \in \mathbb{R}^{m,2n} . \tag{3.15.2}$$

Our ultimate goal is the efficient low-rank compression of $\mathbf{X}$, given $\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2$.

(3-15.a) ⚀ (15 min.) Give a *sharp bound* for the maximal rank of the matrix $\mathbf{X}$ defined in (3.15.2)?

$$\operatorname{rank}(\mathbf{X}) \leq \boxed{} .$$

SOLUTION of (3-15.a):

For the range space of the matrix $\mathbf{X}$ we find

$$\begin{aligned} & \mathcal{R}(\mathbf{X}) = \mathcal{R}(\mathbf{X}_1) + \mathcal{R}(\mathbf{X}_2) . \\ \blacktriangleright \quad & \dim \mathcal{R}(\mathbf{X}) \leq \dim \mathcal{R}(\mathbf{X}_1) + \dim \mathcal{R}(\mathbf{X}_2) , \\ \blacktriangleright \quad & \operatorname{rank}(\mathbf{X}) \leq 2k . \end{aligned}$$

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 $\in \mathbb{R}^m$.

▲

(3-15.b) ⚄ (90 min.) In the sequel we assume $2k \leq \min\{m, n\}$. In the file `concatmatlrc.hpp` implement a C++ function

```
std::tuple<Eigen::MatrixXd, Eigen::VectorXd, Eigen::MatrixXd>
  eco_svd_concat(
    const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
    const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2);
```

which accepts four matrices $\mathbf{A}_1, \mathbf{B}_1, \mathbf{A}_2, \mathbf{B}_2$ with sizes $\mathbf{A}_1, \mathbf{A}_2 \in \mathbb{R}^{m,k}$ and $\mathbf{B}_1, \mathbf{B}_2 \in \mathbb{R}^{n,k}$, see (3.15.1), as arguments. The function is supposed to compute the factors

$$\text{(i)} \quad \mathbf{U} \in \mathbb{R}^{m,2k} \quad \text{with} \quad \mathbf{U}^\top \mathbf{U} = \mathbf{I}_{2k} \,, \tag{3.15.3a}$$

$$\text{(ii)} \quad \mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_{2k}) \in \mathbb{R}^{2k,2k} \quad \text{with} \quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{2k} \geq 0 \,, \tag{3.15.3b}$$

$$\text{(iii)} \quad \mathbf{V} \in \mathbb{R}^{2n,2k} \quad \text{with} \quad \mathbf{V}^\top \mathbf{V} = \mathbf{I}_{2k} \,, \tag{3.15.3c}$$

of the **economical/thin SVD** $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ of the concatenation $\mathbf{X}$, $\mathbf{X} := \left[\mathbf{A}_1\mathbf{B}_1^\top \, \mathbf{A}_2\mathbf{B}_2^\top\right]$ as defined in (3.15.2). The function should return the tuple $(\mathbf{U}, [\sigma_1, \ldots, \sigma_{2k}]^\top, \mathbf{V})$.

The asymptotic complexity of your implementation for $m, n \to \infty$ and for small, fixed k, $2k \leq \min\{m, n\}$, must be *optimal*.

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 $\mathbf{B}_2$:

$$\begin{aligned} \mathbf{B}_1 &= \mathbf{Q}_1\mathbf{R}_1 \,, \quad \mathbf{Q}_1 \in \mathbb{R}^{n,k} \,, \quad \mathbf{R}_1 \in \mathbb{R}^{k,k} \,, \quad \mathbf{Q}_1^\top\mathbf{Q}_1 = \mathbf{I}_k \,, \\ \mathbf{B}_2 &= \mathbf{Q}_2\mathbf{R}_2 \,, \quad \mathbf{Q}_2 \in \mathbb{R}^{n,k} \,, \quad \mathbf{R}_2 \in \mathbb{R}^{k,k} \,, \quad \mathbf{Q}_2^\top\mathbf{Q}_2 = \mathbf{I}_k \,. \end{aligned}$$

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>
3  eco_svd_concat(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 int m = A1.rows();
7    assert(A2.rows() == m);
8    const int n = B1.rows();
9    assert(B2.rows() == n);
10   const 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(m, 2 * k);
14   Eigen::VectorXd s(2 * k);
15   Eigen::MatrixXd V(2 * n, 2 * k);
16   // START: Student solution
17   // Computation of economical QR-decompositions of B-factors
18   // Incurs a computational effort of O(nk^2)
19   Eigen::HouseholderQR<Eigen::MatrixXd> qrB1(B1); //
20   const Eigen::MatrixXd Q1{qrB1.householderQ() *
21                            Eigen::MatrixXd::Identity(n, k)}; //
22   Eigen::HouseholderQR<Eigen::MatrixXd> qrB2(B2);           //
23   const Eigen::MatrixXd Q2{qrB2.householderQ() *
24                            Eigen::MatrixXd::Identity(n, k)}; //
25   // Computation of the matrix Z as defined in (3.15.5)
26   // Uses EIGEN's matrix concatenation operator
27
28   Eigen::MatrixXd Z(m, 2 * k);
29   Z << (A1 * (qrB1.matrixQR()
30                  .block(0, 0, k, k)
31                  .template triangularView<Eigen::Upper>())
32                 .transpose()),
33       (A2 * (qrB2.matrixQR()
34                  .block(0, 0, k, k)
35                  .template triangularView<Eigen::Upper>())
36                 .transpose());
37
38   // Economical SVD of Z
39   Eigen::JacobiSVD<Eigen::MatrixXd> svdZ(
40       Z, Eigen::ComputeThinU | Eigen::ComputeThinV); //
41   // Extract desired factors of final economical SVD of X
42   U = svdZ.matrixU();       // left factor of eco-SVD of X
43   s = svdZ.singularValues(); // singular values in a vector
44   // Build right factor of final economical SVD
45   Eigen::MatrixXd VZ{svdZ.matrixV()};
46   V.block(0, 0, n, 2 * k) = Q1 * VZ.block(0, 0, k, 2 * k); //
47   V.block(n, 0, n, 2 * k) = Q2 * VZ.block(k, 0, k, 2 * k); //
48   // END: Student solution
49   return {U, s, V};
50 }
```

Get it on GitLab (concatmatlrc.hpp).

The construction of the $n \times k$-matrices `Q1` and `Q2` in Line 21 and Line 24 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 19 and Line 22: $O(k^2 n)$
- Construction of the $n \times k$-matrices `Q1` and `Q2` in Line 21 and Line 24: $O(k^2 n)$, 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 40: $O(nk^2)$, see [Lecture → § 3.4.2.2]
- Matrix products in Line 46 and Line 47: $O(nk^2)$

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.

▲

(3-15.c) ⚂ (30 min.) Complete the C++ function

```
bool test_eco_svd_concat(
    const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
    const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2);
```

that verifies that for the given argument matrices the implemented function `eco_svd_concat()` complies with the specifications, that is, that its return values provide a factorization of $\mathbf{X}$ from (3.15.2) and meet the requirements (3.15.3). Return **true**, if this is the case.

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

$$\left\|\mathbf{U}^\top\mathbf{U} - \mathbf{I}_{2k}\right\|_F \le \tau\ . \tag{3.15.7}$$

- Check the orthonormality of the columns of $\mathbf{V}$ by the test

$$\left\|\mathbf{V}^\top\mathbf{V} - \mathbf{I}_{2k}\right\|_F \le \tau\ . \tag{3.15.8}$$

- 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 int m = A1.rows();
7    const int n = B1.rows();
8    const 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 (int i = 1; i < 2 * k; ++i) {
16     ok = ok && (s[i - 1] >= s[i]) && (s[i] >= 0.0);
17   }
18   // START: Student solution
19   // Check whether the columns of U are orthonormal
20   ok = ok &&
21       ((U.transpose() * U - Eigen::MatrixXd::Identity(2 * k, 2 * k)).norm() <=
22         tau);
23   // Check whether the columns of V are orthonormal
24   ok = ok &&
25       ((V.transpose() * V - Eigen::MatrixXd::Identity(2 * k, 2 * k)).norm() <=
26         tau);
27   // Check correctness of matrix factorization
28   const Eigen::MatrixXd P{U * s.asDiagonal() * V.transpose()};
29   const double ref = tau * s.norm();
30   ok = ok && ((A1 * B1.transpose() - P.block(0, 0, m, n)).norm() <= ref);
31   ok = ok && ((A2 * B2.transpose() - P.block(0, n, m, n)).norm() <= ref);
32   // END Student solution
33   return ok;
34 }
```

Get it on GitLab (concatmatlrc.hpp).

▲

(3-15.d) ⚂ (30 min.)

Based on `eco_svd_concat()` as specified in Sub-problem (3-15.b) in the file `concatmatlrc.hpp` implement a C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
  concat_low_rank_best(
    const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
    const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2,
    double tol);
```

that computes a matrix $\mathbf{M} \in \mathbb{R}^{m,2n}$ of *minimal rank* $r \in \mathbb{N}$ such that

$$\|\mathbf{X} - \mathbf{M}\|_2 \leq \mathrm{tol} \cdot \|\mathbf{X}\|_2 \,, \tag{3.15.11}$$

where $\mathbf{X}$ is defined in (3.15.2), $\|\cdot\|_2$ stands for the Euclidean matrix norm, and the tolerance $\mathrm{tol} \in]0,1[$ is passed as an argument.

The matrix $\mathbf{M}$ should be returned in low-rank factorized form

$$\mathbf{M} = \mathbf{A}_M \mathbf{B}_M^\top \quad , \quad \mathbf{A}_M \in \mathbb{R}^{m,r} \,, \quad \mathbf{B}_M \in \mathbb{R}^{2n,r} \,, \tag{3.15.12}$$

as tuple $(\mathbf{A}_M, \mathbf{B}_M)$.

SOLUTION 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>
3  concat_low_rank_best(const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
4                       const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2,
5                       double tol) {
6    assert((tol > 0) && (tol < 1.0));
7    // Obtain economical SVD of X, Sub-problem (3-15.b)
8    auto [U, s, V] = eco_svd_concat(A1, B1, A2, B2);
9    int L = s.size();
10   // START student solution
```

```
11    // Find minimal rank r, see (3.15.14)
12    int r;
13    for (r = 1; (r < L) && (s[r] / s[0] > tol); ++r)
14      ;
15    // Rescale columns of U
16    for (int j = 0; j < r; ++j)
17      U.col(j) *= s[j];
18    // Extract sub-matrices
19    return {U.leftCols(r), V.leftCols(r)};
20    // END student solution
21  }
```

Get it on GitLab (concatmatlrc.hpp).

▲

End Problem 3-15 , 165 min.

Problem 3-16: Compact Storage Format for QR-Factorization

It is essential for the efficiency of orthogonalization techniques that orthogonal matrices, which commonly occur as factors in matrix products. are stored as products of elementary orthogonal transformations like Householder reflections or Givens rotations, see [Lecture → Rem. 3.3.3.21]. This approach is adopted in all numerical libraries including EIGEN. This problem addresses some related algorithmic issues.

This problem assumes familiarity with orthogonal matrices [Lecture → Section 3.3.2] and Householder reflections [Lecture → § 3.3.3.11].

A legacy FORTRAN-77 numerical library routine returns a raw array (type **double** $*$) of n^2 **double** floating-point numbers, $n \in \mathbb{N}$, which represents an $n \times n$-matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ according to the following specification:

(I) The array encodes an $n \times n$ matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ in column-major format [Lecture → Section 1.2.3].

(II) The column vectors $\mathbf{u}_j \in \mathbb{R}^n$, $j = 1, \ldots, n-1$, defined as[1]

$$\left(\mathbf{u}_j\right)_k := \begin{cases} 0 & \text{for } k < j\,, \\ \sqrt{1 - \sum\limits_{\ell=k+1}^{n} (\mathbf{M})_{\ell,j}^2} & \text{for } k = j\,, \\ (\mathbf{M})_{k,j} & \text{for } k > j\,, \end{cases} \qquad j = 1, \ldots, n-1\,, \tag{3.16.1}$$

are the building blocks for the product matrix

$$\mathbf{Q} = (\mathbf{I}_n - 2\mathbf{u}_1\mathbf{u}_1^\top)(\mathbf{I}_n - 2\mathbf{u}_2\mathbf{u}_2^\top) \cdot \cdots \cdot (\mathbf{I}_n - 2\mathbf{u}_{n-1}\mathbf{u}_{n-1}^\top)\,. \tag{3.16.2}$$

(III) The matrix $\mathbf{A}$ is given as $\mathbf{A} = \mathbf{QR}$, where $\mathbf{R} \in \mathbb{R}^{n,n}$ is the *upper triangular* part of $\mathbf{M}$:

$$(\mathbf{R})_{k,j} := \begin{cases} (\mathbf{M})_{k,j} & \text{for } k \le j\,, \\ 0 & \text{else,} \end{cases} \qquad k, j \in \{1, \ldots, n\}\,. \tag{3.16.3}$$

Your task is to write an EIGEN-based wrapper class for the raw data that offers certain operations with the matrix $\mathbf{A}$. The class definition is a follows:

C++ code 3.16.4: Class definition of CompactStorageQR

```
2  class CompactStorageQR {
3  public:
4   // Constructor taking raw data array, which has to be persistent
5   CompactStorageQR(const double *data, unsigned int n) : n_(n), M_(data, n, n) {
6     for (unsigned int k = 0; k < n_ - 1; ++k) {
7       double sn{0};
8       for (unsigned int j = k + 1; j < n_; ++j) {
9         const double t{M_(j, k)};
10        sn += t * t;
11      }
12      if (sn > 1.0) {
13        throw std::runtime_error(
14            "CompactStorageQR: Illegal subdiagonal column norm!");
15      }
16    }
```

[1] A sum whose lower summation bound is larger than the upper summation bound is understood to evaluate to zero.

```
  }
  // Determinant of the matrix stored in this object
  double det() const;
  // Right multiplication with another matrix
  Eigen::MatrixXd matmult(const Eigen::MatrixXd &X) const;
  // Solution of linear systems of equations
  Eigen::MatrixXd solve(const Eigen::MatrixXd &B) const;
 private:
  unsigned int n_;                        // Matrix dimensions
  Eigen::Map<const Eigen::MatrixXd> M_;   // Raw data wrapped into a matrix
};
```

(3-16.a) ⚄ (45 min.) In the file `compactstorageqr.hpp` implement the method

```
Eigen::MatrixXd CompactStorageQR::matmult(
  const Eigen::MatrixXd &X) const;
```

which returns the matrix product $\mathbf{A}\cdot\mathbf{X}$ of the matrix $\mathbf{A}\in\mathbb{R}^{n,n}$ stored in the **CompactStorageQR** object and of a matrix $\mathbf{X}\in\mathbb{R}^{n,k}$, $k\in\mathbb{N}$, passed in `X`. Your implementation must not incur asymptotic computational cost worse than $O(n^2k)$ for $n,k\to\infty$.

SOLUTION 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

```
Eigen::MatrixXd CompactStorageQR::matmult(const Eigen::MatrixXd &X) const {
  assert((X.rows() == n_) && "Wrong size of matrix factor");
  Eigen::MatrixXd Y(n_, X.rows());
  // START Student code
  // Multiplication with R-factor
  Y = M_.template triangularView<Eigen::Upper>() * X;
  // Multiplication with Q-factors
  Eigen::VectorXd u{Eigen::VectorXd::Zero(n_)};
  for (int k = n_ - 2; k >= 0; --k) {
    u.tail(n_ - k - 1) = M_.block(k + 1, k, n_ - k - 1, 1);
    u[k] = std::sqrt(1.0 - u.squaredNorm());
    Y -= 2 * u * (u.transpose() * Y);
  }
  // END Student code
  return Y;
}
```

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.

▲

(3-16.b) ⚂ (30 min.) In the file `compactstorageqr.hpp` supply the missing parts of the implementation of

```
Eigen::MatrixXd CompactStorageQR::solve(const Eigen::MatrixXd &B)
    const;
```

which is to compute $\mathbf{A}^{-1}\mathbf{B}$ for $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in the current **CompactStorageQR** object and $\mathbf{B} \in \mathbb{R}^{n,k}$ contained in the argument `B`. Again, the asymptotic complexity of your implementation must be $O(n^2k)$ for $n, k \to \infty$.

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 $\mathbf{A}$,

$$\mathbf{A} = \mathbf{Q}\mathbf{R} = \mathbf{Q}_1\mathbf{Q}_2 \cdot \cdots \cdot \mathbf{Q}_{n-1}\mathbf{R}\,, \quad \mathbf{Q}_j := \mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top\,,$$

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 $\mathbf{R}\mathbf{x} = (\mathbf{Y})_\ell$, $\ell = 1, \ldots, k$, 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    // START student code
6    // Check invertibility by inspecting diagonal elements of R-factor
7    const double atol = 1.0E-16;
8    const double rtol = 1.0E-8;
9    double maxrii = std::abs(M_(0, 0));
10   for (unsigned int k = 1; k < n_; ++k) {
11     const double tmp = std::abs(M_(k, k));
12     if (maxrii < tmp) {
13       maxrii = tmp;
14     }
15   }
```

```
  for (unsigned int k = 1; k < n_; ++k) {
    const double rii = std::abs(M_(k, k));
    if ((rii < rtol * maxrii) || (rii < atol)) {
      throw std::runtime_error("CompactStorageQR::solve: matrix not regular");
    }
  }
  // Multiplication with transposed Q-factors
  Eigen::VectorXd u(n_);
  for (unsigned int k = 0; k < n_ - 1; ++k) {
    double sn{0};
    for (unsigned int j = k + 1; j < n_; ++j) {
      u[j] = M_(j, k);
      sn += u[j] * u[j];
    }
    u[k] = std::sqrt(1.0 - sn);
    std::cout<<"output here"<< u << std::endl;
    // Multiplication with current Q-factor
    X -= 2 * u * (u.transpose() * X);
    u[k] = 0.0;
  }
  // Solve upper triangular linear system RX = Q^T B
  X = M_.template triangularView<Eigen::Upper>().solve(X);
  // END student code
  return X;
}
```

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.

▲

(3-16.c) ⚂ (30 min.) In the file `compactstorageqr.hpp` write the code for the method

```
double CompactStorageQR::det() const;
```

which gives the **determinant** $\det \mathbf{A}$ of the matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in the current **CompactStorageQR** object.

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 $\mathbf{u}_j$:

$$\mathbf{U}_j \mathbf{Q}_j \mathbf{U}_j = \mathbf{U}_j^\top (\mathbf{I}_n - 2\mathbf{u}_j \mathbf{u}_j^\top)\mathbf{U}_j = \operatorname{diag}(-1,1,\ldots,1)\ .$$

C++ code 3.16.8: Method `det` of class CompactStorageQR

```
2  double CompactStorageQR::det() const {
3    double d;
4    // START Student code
5    // The determinant is just the product of the diagonal entries of
       the
6    // R-factor, because the determinant of the Q-factor is 1.
7    d = 1.0;
8    for (unsigned int j = 0; j < n_; ++j) {
9      d *= M_(j, j);
10   }
11   // Sign correction, because Household transformations
12   // have determinant -1 and n-1 of them are inviolved.
13   d *= ((n_%2 == 0)?-1.0:1.0);
14   // END Student code
15   return d;
16 }
```

▲

End Problem 3-16, 105 min.

Chapter 4

Filtering Algorithms

Problem 4-1: Autofocus with FFT

In this problem, we will use 2D frequency analysis to find the "best focused" image among a collection of out-of-focus photos. To that end, we will implement an algorithm based on 2D DFT as introduced in [Lecture → Section 4.2.5].

This problem takes for granted that you know about the 2D discrete Fourier transform and its connection with discrete convolutions, see [Lecture → Section 4.2.2] and [Lecture → Section 4.2.5].

Let a grey-scale image consisting of $n \times m$ pixels be given as a matrix $\mathbf{P} \in \mathbb{R}^{n,m}$ as in [Lecture → Ex. 4.2.5.16]; each element of the matrix indicates the gray-value of the pixel as a number between 0 and v_{max}. This image is regarded as the "perfect image".

If a camera is poorly focused due to an inadequate arrangement of the lenses, it will record a blurred image $\mathbf{B} \in \mathbb{R}^{n,m}$. As before [Lecture → Eq. (4.2.5.17)], the blurring operation can be modeled through the 2D (periodic) discrete convolution and can be undone provided that the point spread function (PSF) is known. However, in this problem we must not assume any knowledge of the PSF.

Assume that the only data available to us is the "black-box" C++ function (declared in `autofocus.hpp`):

```
MatrixXd set_focus(double f);
```

which returns the potentially blurred image $\mathbf{B}(f)$, when the focus parameter f is set to a particular value. The actual operation of `set_focus` is utterly obscure. The focus parameter can be read as the distance of the lenses of a camera that can be changed through turning on and off a stepper motor. The following sequence of images illustrates the impact of setting different focus parameters:



Fig. 16



Fig. 17



Fig. 18



Fig. 19

The problem of *autofocusing* is to determine the value of the focus parameter f, which yields an image $\mathbf{B}(f)$ with the least blur. The idea of the autofocusing algorithm is the following: *The less the image is marred by blur, the larger its "high frequency content"*. The latter can found out based on the discrete Fourier transform, more precisely, its 2D version.

To translate the autofocus idea into an algorithm we have to give a quantitative meaning to the notion of "high frequency content" of a (blurred) image $\mathbf{C}$, which is done by looking at the second moments of its 2D discrete Fourier transform:

$$V(\mathbf{C}) = \sum_{k_1=0}^{n-1} \sum_{k_2=0}^{m-1} \left(\left(\frac{n}{2} - \left| k_1 - \frac{n}{2} \right| \right)^2 + \left(\frac{m}{2} - \left| k_2 - \frac{m}{2} \right| \right)^2 \right) |\hat{\mathbf{C}}_{k_1,k_2}|^2 \,.$$

Here, $\hat{\mathbf{C}} \in \mathbb{C}^{n,m}$ stand for the 2D DFT of the image $\mathbf{C}$. Hence, the autofocusing policy can be rephrased as

$$\text{find } f \in \mathbb{R} \text{ such that } V(\mathbf{B}(f)) \text{ becomes maximal.}$$

(4-1.a) ⚀ (10 min.) This sub-problem supplies us with a tool to write an image file from a C++ code. Write a C++ function

```
void save_image(double focus);
```

that saves the image $\mathbf{B}(f)$ returned by `set_focus()` for $f = 0, 1, 2, 3$ in **P**ortable **G**ray**m**ap (PGM) format to the file `./cx_out/image_focus<f>.png`, where `<f>` is a string containing the argument `focus`.

For this task you can use objects of type `PGMObject` (found in "`pgm.hpp`"). You can find example of usages of this class in "`examples/pgm_example.cpp`".

SOLUTION of (4-1.a):

C++11-code 4.1.1: Rendering of images with different focus settings.

```
2  void save_image(double focus) {
3      // Create empty object
4      PGMObject q;
5
6      // TO DO: (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"
```

```
    q.set_data(set_focus(focus));

    // Create and save file
    std::stringstream ss;
    ss << "./cx_out/image_focus="
       << (int) focus
       << ".pgm";
    std::ofstream file(ss.str());
    file << q;
    // END
}
```

Get it on GitLab (autofocus.hpp).

▲

The file `fft.hpp` supplies a few FFT-based convenience functions. Among them is `fft2r`, which performs the 2D discrete Fourier transform [Lecture → Eq. (4.2.5.4)] of a real-values matrix, *cf.* [Lecture → Code 4.2.5.6], which results in a complex matrix however.

C++ code 4.1.2: 2D DFT of a Eigen::MatrixXd

```
Eigen::MatrixXcd fftr(const Eigen::MatrixXd &X) {
  const long m = X.rows(), n = X.cols();
  Eigen::MatrixXcd Y(m, n); // result
  Eigen::FFT<double> fft; // auxiliary FFT object
  for (long j = 0; j < n; ++j) {
    Eigen::VectorXd Xj = X.col(j);
    Y.col(j) = fft.fwd(Xj);
  }
  return Y;
}
```

```
Eigen::MatrixXcd fft2r(const Eigen::MatrixXd &X) {
  return fft(fftr(X).transpose()).transpose();
}
```

(4-1.b) ⚀ (20 min.) The two-dimensional discrete Fourier transform can be performed using the C++ function `fft2r()`. Write a C++ function

```
void plot_freq(double focus);
```

that creates 3D plots of the (modulus of the) 2D DFTs of the images obtained in sub-problem (4-1.a). Clamp the data between 0 and 8000, this means when a value exceeds 8000, then set it to 8000.

HINT 1 for (4-1.b): MATPLOTLIBCPP provides many facilities for rendering 2D data, for instance `plot_surface()`, see Github page. ⌟

SOLUTION of (4-1.b):

C++11-code 4.1.3: Plotting of spectrum of the images.

```
void plot_freq(double focus) {
  int a = 0;
  int b = 8000;
```

```
5   auto clamp = [a, b](double x) { return x < a ? a : x > b ? b : x; };
6
7   // TO DO: (4-1.b) compute D containing the spectrum of
        set_focus(focus).
8   // "clamp" the data between 0 and 8000.
9   // START
10  MatrixXd D = fft2r(set_focus(focus)).cwiseAbs().unaryExpr(clamp) / b;
11  // END
12
13  // Plot values of X.
14  // Axis labels
15  std::vector<double> xticks(5);
16  for (int i = 0; i < 5; ++i) {
17    xticks[i] = i * (D.cols() - 1) / 4;
18  }
19  std::vector<double> yticks(5);
20  for (int i = 0; i < 5; ++i) {
21    yticks[i] = i * (D.rows() - 1) / 4;
22  }
23  std::vector<std::string> labels{"-1", "-0.5", "0", "0.5", "1"};
24
25  plt::figure();
26  plt::imshow(D, {{"cmap", "viridis"}, {"origin", "lower"}});
27  plt::colorbar();
28  plt::xticks(xticks, labels);
29  plt::yticks(yticks, labels);
30  std::stringstream ss;
31  ss << "Spectrum with f = " << focus << ".";
32  plt::title(ss.str().c_str());
33  std::stringstream ss2;
34  ss2 << "./cx_out/spectrum_focus=" << focus << ".eps";
35  plt::savefig(ss2.str().c_str());
36 }
```

Get it on GitLab (autofocus.hpp).

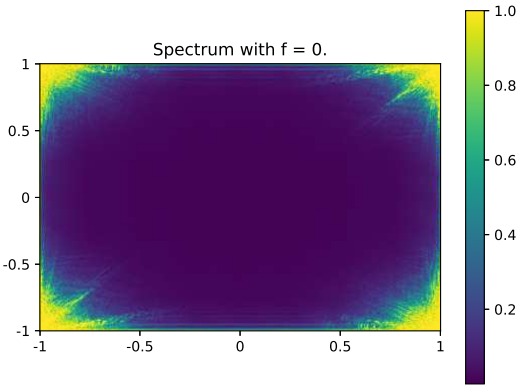


Fig. 20

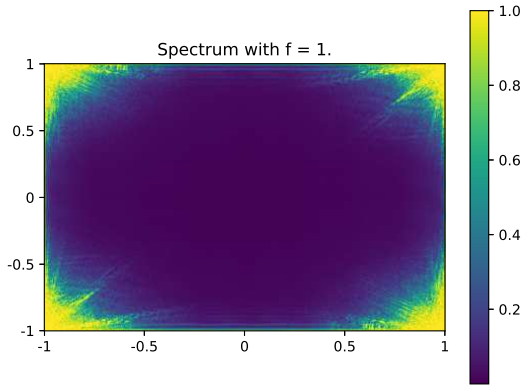


Fig. 21

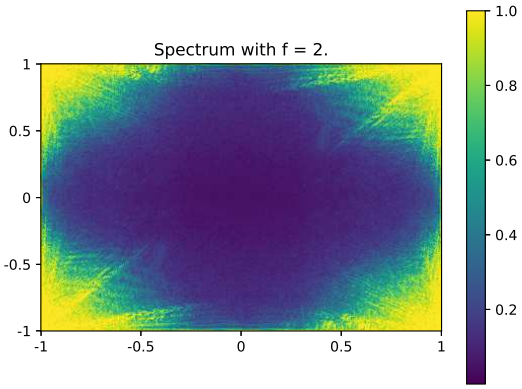


Fig. 22

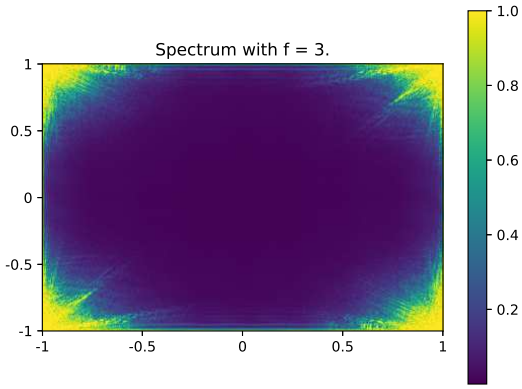


Fig. 23

▲

(4-1.c) ⚄ (15 min.) In the plots generated in Sub-problem (4-1.b), mark (or explain) the regions corresponding to the high and low frequencies.

HINT 1 for (4-1.c): Look at [Lecture → § 4.2.3.6] and [Lecture → Fig. 126] to understand the relationship of Fourier coefficients and low and high frequency content of a signal. ⌟

SOLUTION of (4-1.c):

Centres of the plots correspond to the high frequencies, whereas corners correspond to the low frequencies.

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.

▲

(4-1.d) ⚄ (25 min.) Write a C++ function

```
double high_frequency_content(const MatrixXd & C);
```

that returns the value $V(\mathbf{C})$ for a given matrix $\mathbf{C}$. Write a C++ function

```
void plotV();
```

that plots the function $V(\mathbf{B}(f))$ in terms of the focus parameter f. Use 100 equidistantly spaced sampling points in the interval $[0,5]$.

SOLUTION of (4-1.d):

C++11-code 4.1.4: Computation of high-frequency content.

```
2 double high_frequency_content(const MatrixXd& M) {
3   int n = M.rows(), m = M.cols();
4   double V = 0;
5   // TO DO: (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++11-code 4.1.5: Plot creation for $V(\mathbf{B}(f))$.

```
void plotV() {
  unsigned int N = 20;

  VectorXd x(N), y(N);

  // TO DO: (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.eps");
}
```

Get it on GitLab (autofocus.hpp).

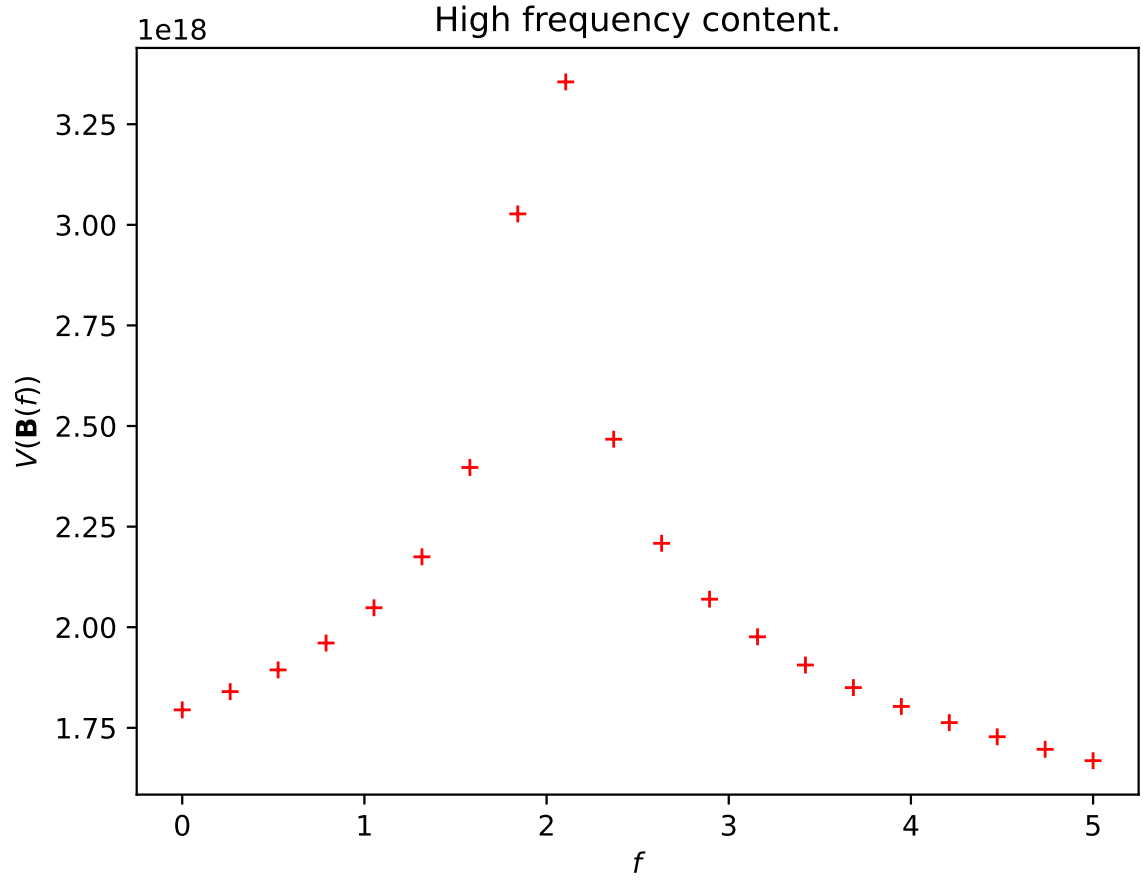


Fig. 24

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.

▲

(4-1.e) ⚄ (40 min.) Write an *efficient* (you want the auto-focus procedure to take not longer than a few seconds!) C++ function

```
double autofocus();
```

that numerically determines optimal focus parameter $f_0 \in [0,5]$ for which the 2nd moment $V(\mathbf{B}(f))$ is maximized. What is the resulting focus parameter f_0?

To locate the change of slope of $V(\mathbf{B}(f))$ in $[0,5]$ you should use the bisection algorithm [Lecture → Section 8.4.1] for finding a zero of the derivative $f \mapsto \frac{dV(\mathbf{B}(f))}{df}$, which will, hopefully, mark the location of the maximum of $V(f)$.

The bisection algorithm for finding a zero of a continuous function $\varphi : [a,b] \to \mathbb{R}$ with $\varphi(a) \cdot \varphi(b) < 0$ is rather simple and lucidly demonstrated in [Lecture → Code 8.4.1.2].

Of course, the derivative is not available, so that we have to approximate it crudely by means of a difference quotient

$$\frac{dV(f)}{df} \approx \frac{V(\mathbf{B}(f+\delta f)) - V(\mathbf{B}(f-\delta f))}{2\delta f} .$$

You can assume that the smallest step of the auto-focus mechanism is 0.05 and so the natural choice

for δf is $\delta f = 0.05$. This maximal resolution also tells you a priori how many bisection steps should be performed.

Use the structure of the function $V(\mathbf{B}(f))$ that you observed in sub-problem (4-1.d). Use as few evaluations of $V(\mathbf{B}(f))$ as possible!

SOLUTION of (4-1.e):

The auto-focus determines that focus parameter should be set to $f \approx 2.0$.

C++11-code 4.1.6: Finding best value of focus parameter.

```
2  double autofocus() {
3    // Minimum focus
4    const double min_focus = 0;
5    // Maximum focus
6    const double max_focus = 5;
7    // Min step
8    const double min_step = 0.05;
9    // Starting guess
10   double f0 = (max_focus - min_focus) / 2.;
11   // Finite differences increment
12   double df = min_step;
13   // Starting step
14   double step = max_focus / 2.;
15   // Max number of iteration
16   unsigned int Niter = std::ceil(std::log2((max_focus - min_focus) / min_step));
17   // TO DO: (4-1.e) Use bisection method to find best focus.
18   // START
19   // Returns V(B(f))
20   auto computeV = [](double focus) {
21     return high_frequency_content(fft2r(set_focus(focus)).cwiseAbs());
22   };
23
24   // Bisection method
25   for (unsigned int i = 0; i < Niter; ++i) {
26     double dV = computeV(f0 + df) - computeV(f0 - df);
27
28     step = step / 2.;
29     f0 = f0 + (dV > 0 ? 1 : -1) * step;
30   }
31   // END
32   return f0;
33 }
```

Get it on GitLab (autofocus.hpp).

▲

End Problem 4-1 , 110 min.

Problem 4-2: FFT and least squares

This problem deals with an application of fast Fourier transformation in the context of a least squares (data fitting) problems.

You should be familiar with both DFT and the idea of least-squares fitting

The orbit of a planet around the sun is a closed, planar curve $\mathcal{C} \subset \mathbb{R}^2$, which can be described by a so-called polar representation, closely related to polar coordinates:

$$\mathcal{C} := \left\{ \begin{bmatrix} d(\varphi)\cos\varphi \\ d(\varphi)\sin\varphi \end{bmatrix} : 0 \le \varphi < 2\pi \right\} ,$$

where $d : [0, 2\pi] \to \mathbb{R}^+$ is a given function of the polar angle φ providing the distance of a point on the curve from the origin in the direction of that angle. ▷

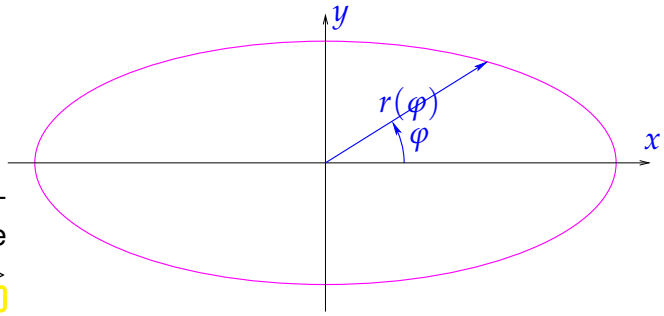


Fig. 25

We put the sun in the origin of the coordinate system. For equispaced polar angles (→ Fig. 25) $\varphi_j = 2\pi\frac{j}{n}$, $j = 0, \ldots, n-1$, $n \in \mathbb{N}$ the distance of the planet from the sun is measured and found to be $d_j \in \mathbb{R}$ $j = 0, \ldots, n-1$. We now discuss a numerical method providing the planet's path in polar representation.

Write $\mathcal{P}_m^C$, $m \in \mathbb{N}$, for the space of real trigonometric polynomials of degree $\le m$. These are 2π-periodic functions $\mathbb{R} \to \mathbb{R}$ of the form

$$p(t) = \sum_{k=0}^{m} c_k \cos(kt) , \quad c_k \in \mathbb{R} . \tag{4.2.1}$$

The idea is to approximate the distance $d(\varphi)$ of the planet from the sun as a function of the angle by a trigonometric polynomial $p^* \in \mathcal{P}_m^C$, which minimized the sum of squares of "distance mismatches":

$$p^* = \operatorname*{argmin}_{p \in \mathcal{P}_m^C} \sum_{j=0}^{n-1} |p(\varphi_j) - d_j|^2 . \tag{4.2.2}$$

Throughout, we assume that the number of distance data is more than twice as big as the degree of p^*: $2m < n$.

(4-2.a) ⚀ (20 min.) Recast this task as a standard linear least squares problem

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^?} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 , \tag{4.2.3}$$

with a suitable matrix $\mathbf{A} \in \mathbb{R}^{?,?}$ and vectors $\mathbf{b}, \mathbf{x} \in \mathbb{R}^?$.

SOLUTION 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,

$$\sum_{j=0}^{n-1} |p(\varphi_j) - d_j|^2 = \left\| \begin{bmatrix} \sum_{k=0}^{m} c_k \cos\left(\frac{2\pi}{n} k \cdot 0\right) - d_0 \\ \sum_{k=0}^{m} c_k \cos\left(\frac{2\pi}{n} k \cdot 1\right) - d_1 \\ \vdots \\ \sum_{k=0}^{m} c_k \cos\left(\frac{2\pi}{n} k(n-1)\right) - d_{n-1} \end{bmatrix} \right\|_2^2 \overset{!}{=} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 .$$

Recalling the definition of the Euclidean norm and the rules of matrix×vector multiplication, we find

$$\mathbf{A} = \begin{bmatrix} \cos(0) & \cos(0) & \cos(0) & \dots & \cos(0) \\ \cos(0) & \cos(\frac{2\pi}{n}) & \cos(\frac{4\pi}{n}) & \dots & \cos(\frac{2\pi m}{n}) \\ \cos(0) & \cos(\frac{4\pi}{n}) & \cos(\frac{8\pi}{n}) & \dots & \cos(\frac{4\pi m}{n}) \\ \dots & \dots & \dots & \dots & \dots \\ \cos(0) & \cos(\frac{2\pi(n-1)}{n}) & \cos(\frac{4\pi(n-1)}{n}) & \dots & \cos(\frac{2\pi(n-1)m}{n}) \end{bmatrix} \in \mathbb{R}^{n,m+1}, \ \mathbf{b} = \begin{bmatrix} d_0 \\ d_1 \\ \dots \\ d_{n-1} \end{bmatrix} \in \mathbb{R}^n .$$

More compactly, letting cos act on a matrix componentwise, we can write

$$\mathbf{A} = \left[\cos\left(\frac{2\pi}{n}\right)\ell k\right]_{\substack{\ell=0,\dots,n-1 \\ k=0,\dots,m}} = \cos\left(\frac{2\pi}{n}\begin{bmatrix} 0 \\ 1 \\ \vdots \\ n-1 \end{bmatrix}\begin{bmatrix} 0 & 1 & \dots & m \end{bmatrix}\right) \in \mathbb{R}^{n,m+1} . \tag{4.2.4}$$

▲

(4-2.b) ⚂ (30 min.) [depends on Sub-problem (4-2.a)]

State the normal equations [Lecture → Thm. 3.1.2.1] for the linear least-squares problem found in Sub-problem (4-2.a) in explicit form.

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*!

$$\mathbf{A}^\top\mathbf{A} = \begin{bmatrix} n & & & \\ & n/2 & & \\ & & \ddots & \\ & & & n/2 \end{bmatrix} \in \mathbb{R}^{m+1,m+1}\,. \tag{4.2.7}$$

▲

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

Write a short EIGEN-based C++ function

```
bool testNormEqMatrix(unsigned int n, unsigned int m);
```

that test, whether the formula for the matrix product $\mathbf{A}^\top\mathbf{A}$ that you have found in Sub-problem (4-2.b) is correct.

To that end, form the dense matrix $\mathbf{A}$ as an object of type **Eigen::MatrixXd**, compute $\mathbf{A}^\top\mathbf{A}$ and compare with the matrix given by your formula. Of course, the comparison has to be done in a numerically sound way following the advice of [Lecture → Rem. 1.5.3.15].

SOLUTION 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 $mn \cdot$ `TOL` with `TOL` $\ll 1$, for instance `TOL` $\approx$ `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);
```

```
  Eigen::VectorXd m_linear = Eigen::VectorXd::LinSpaced(m + 1, 0, m);

  // Building A as described in (4.2.4)
  A = 2.0 * M_PI / n * n_linear * m_linear.transpose();
  A = A.array().cos().matrix();

  // Using the knowledge gained in (4-2.b) to initialize A^T*A directly as a
  // diagonal matrix.
  Eigen::MatrixXd ATA = Eigen::MatrixXd::Zero(m + 1, m + 1);
  ATA.diagonal() = Eigen::VectorXd::Constant(m + 1, n / 2.0);
  ATA(0, 0) = n;

  // Comparing the two approaches to calculate A^T*A.
  // Test for relative smallness of the Frobenius norm of the difference of the
  // matrices
  if ((ATA - A.transpose() * A).norm() < 1e-10 * m * n) {
    return true;
  }
  // END

  return false;
}
```

Get it on GitLab (fftlsq.hpp).

▲

(4-2.d) ⚄ (45 min.) [depends on Sub-problem (4-2.b)]

Write a C++ function

```
VectorXd find_c(const VectorXd & d, unsigned int m);
```

that computes the coefficients c_k, $k = 0, \ldots, m$, of p^* in the form (4.2.1) for arbitrary inputs $d_j \in \mathbb{R}$, $j = 0, \ldots, n-1$. These are passed as vector `d`.

We aim for an efficient implementation: the function must not have asymptotic complexity worse than $O(n \log n)$ in terms of the problem size parameters $m, n \to \infty$.

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;
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).

▲

(4-2.e) ⚄ (30 min.) [depends on Sub-problem (4-2.d)]

Since Kepler we know that the orbits of planets around the sun are ellipses. Choose $n=100$ and sample the distance data form the polar form of an ellipse:

$$d(\varphi)=\frac{1}{\sqrt{1-c^2\cos^2\varphi}}\,,\quad \varphi\in[0,2\pi]\,,\quad c=0.8\,. \tag{4.2.11}$$

1. Using your implementation of `find_c` compute and tabulate the coefficients c_k of p^* for $m=1,2,3$.
2. Using MatPlotLibCpp's `plot()` function create a plot of the ellipse given by (4.2.11).

3. Into the same plot insert the curves described by the trigonometric polynomials p^* of degrees $m = 1, 2, 3$.

This should be accomplished by a C++ function

```
void fitEllipse(void);
```

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()`.

```
2  void fitEllipse() {
3    unsigned int n = 10;
4    unsigned int m = 3;
5
6    // Test points
7    const unsigned int npoints = 10;
8    Eigen::VectorXd d(npoints);
9    d << 0.987214, 1.03579, 0.997689, 0.917471, 1.00474, 0.92209, 1.03517,
10       1.08863, 0.904992, 0.956089;
11
12   plt::figure();
13
14   // TODO: (4-2.e) Tabulate the coefficients of find_c for all m = 1,2,3
15   // plot the ellipse and also the curves of the trigonimetric
        polynomials
16   // START
17   const double c = 0.8;
18   auto d_func =
19       [c](const double phi) { // no need to capture c because it is
            constexpr
20         assert(0. <= phi && phi <= 2. * M_PI);
21         return 1. / std::sqrt(1. - std::pow(c * std::cos(phi), 2));
22       };
23
24   // Sample distance data, we reuse d from above
25   n = 100;
26   d.resize(n);
27   for (std::size_t i = 0; i < n; ++i) {
28     d(i) = d_func((2. * M_PI * i) / n);
29   }
30
31   plt::title("Orbit of planet");
32   plt::xlim(-2, 2);
33   plt::ylim(-2, 2);
34
35   // Plot the ellipse given by (5.2.10) using 10000 points
36   constexpr unsigned int N = 10000;
37   Eigen::VectorXd x_ellipse(N), y_ellipse(N);
38   double temp_d;
39   for (std::size_t i = 0; i < N; ++i) {
40     temp_d = d_func((2.0 * M_PI * i) / N);
```

```
41      x_ellipse(i) = temp_d * std::cos((2.0 * M_PI * i) / N);
42      y_ellipse(i) = temp_d * std::sin((2.0 * M_PI * i) / N);
43    }
44    plt::plot(x_ellipse, y_ellipse, "k", {{"label", "exact orbit"}});
45
46    // Calculate, tabulate and plot the coefficients in one go
47    x_ellipse.resize(n);
48    y_ellipse.resize(n);
49
50    // Create vector of colors for plots
51    std::vector<std::string> colors = {"m", "y--", "r"};
52    std::cout << "degree of polynomial | coefficients" << std::endl;
53    for (m = 1; m <= 3; ++m) {
54      Eigen::VectorXd coefficients = find_c(d, m);
55
56      std::cout << std::setw(20) << m << " | " << std::setw(5)
57                << coefficients.transpose() << std::endl;
58
59      Eigen::VectorXd e = eval_p(coefficients, n);
60      for (std::size_t i = 0; i < n; ++i) {
61        x_ellipse(i) = e(i) * std::cos((2 * M_PI * i) / n);
62        y_ellipse(i) = e(i) * std::sin((2 * M_PI * i) / n);
63      }
64      const std::string label = "fitted trig. poly. for m = " + std::to_string(m);
65      plt::plot(x_ellipse, y_ellipse, colors[m - 1], {{"label", label}});
66    }
67
68    plt::xlabel("x");
69    plt::ylabel("y");
70    plt::legend();
71
72    // END
73    plt::savefig("cx_out/orbit.png");
74  }
```

Get it on GitLab (fftlsq.hpp).

Picture produced by the function ▷

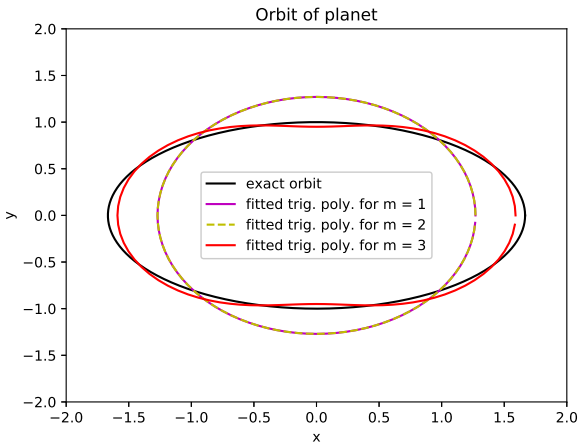


Fig. 26

These are the coefficients of the fitted trigonometric polynomials:

m	$[c_0,\ldots,c_{m-1}]^\top$
1	$[1.27025]^\top$
2	$[1.27025\ 3.60268\cdot 10^{-17}]^\top$
3	$[1.27025\ 3.60268\cdot 10^{-17}\ 0.320124]^\top$

We see, as expected, a better fit for largers degree m. The fact $c_1 = 0$ is due to the symmetry of the ellipse.

▲

End Problem 4-2 , 145 min.

Problem 4-3: Multiplication and division of polynomials based on DFT

In [Lecture → § 4.1.3.6] we saw that the formula of discrete convolution [Lecture → Def. 4.1.3.3] also gives the coefficients of products of polynomials. Thus, in light of the realization of discrete convolutions by means of DFT [Lecture → Section 4.2.2], the Fast Fourier Transform (FFT) becomes relevant for efficiently multiplying polynomials of very high degree.

This problem requires knowledge about DFT and discrete convolutions, see [Lecture → Section 4.1.3] and [Lecture → Section 4.2.2]. For the connection of DFT, FFT, and polynomial multiplication also see the YouTube Video.

Polynomials. We call a polynomial in x of degree $N \in \mathbb{N}$ a function $p : \mathbb{K} \to \mathbb{K}$ of the form

$$p(x) = \sum_{j=0}^{N} \gamma_j x^j \quad \text{with coefficients} \quad \gamma_j \in \mathbb{K}\,, \tag{4.3.1}$$

also called monomial representation. Under pointwise addition and multiplication the polynomials of degree N form a vector space $\mathcal{P}_N$ of dimension $N+1$. Associating the vector $\mathbf{c} := [\gamma_0, \ldots, \gamma_N] \in \mathbb{K}^{N+1}$ to every $p \in \mathcal{P}_N$ as given in (4.3.1) induces an isomorphism of the vector spaces $\mathcal{P}_N$ and $\mathbb{K}^{N+1}$. In other words, we can identify $p \in \mathcal{P}_N$ with its coefficient vector $\mathbf{c}$.

Thus, when we have to represent polynomials $\in \mathcal{P}_N$ in a C++ code we can do this via objects of type **Eigen::VectorXd** that contain their coefficient vectors. This convention is adopted throughout this problem.

Let two large numbers $m, n \gg 1$ and two polynomials of degrees $m-1$ and $n-1$, respectively, in monomial representation

$$u(x) = \sum_{j=0}^{m-1} \alpha_j x^j\,, \quad v(x) = \sum_{j=0}^{n-1} \beta_j x^j\,, \quad \alpha_j, \beta_j \in \mathbb{C}\,, \tag{4.3.2}$$

be given.

The tasks that we are going to tackle in this problem are the following.

1. *Polynomial multiplication*: Efficiently compute the coefficients of the polynomial $p = uv$ (point wise product, a polynomial of degree $\leq m+n-2$).
2. *Polynomial division*: Given the polynomials p and u in terms of their coefficients, the degree of p at least as bis as that of u, find a polynomial v, if it exists, such that $p = uv$.

(4-3.a) ⚀ (25 min.) Write a C++ function

```
Eigen::VectorXd polyMult_naive(const Eigen::VectorXd & u, const
   Eigen::VectorXd & v);
```

that "naively", i.e. using a simple loop-based implementation, computes and returns the vector of coefficients of the polynomial $p = uv$. The coefficient vectors for the polynomials u and v are passed in `u` and `v`.

Also determine the asymptotic complexity of your algorithm for $m, n \to \infty$ (separately).

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)

```
2  Eigen::VectorXd polyMult_naive(const Eigen::VectorXd &u,
3                                 const Eigen::VectorXd &v) {
4    // Fetch degrees of input polynomials
5    int degu = u.size() - 1;
6    int degv = v.size() - 1;
7
8    // Object for product polynomial p = uv
9    int degp = degu + degv;
10
11   Eigen::VectorXd uv(degp + 1);
12
13   // TODO: (4-3.a) Multiply polynomials u and v naively.
14   // START
15   for (int i = 0; i <= degp; ++i) {
16     int fst = std::max(0, i - degv);
17     int lst = std::min(degu, i);
18     for (int j = fst; j <= lst; ++j) {
19       uv(i) += u(j) * v(i - j);
20     }
21   }
22   // END
23
24   return uv;
25 }
```

Get it on GitLab (polydiv.hpp).

Since a double loop is used, the complexity is $\mathcal{O}(nm)$.

▲

(4-3.b) ⚂ (30 min.) Write a C++ function

```
Eigen::VectorXd polyMult_fast(const Eigen::VectorXd & u, const
   Eigen::VectorXd & v);
```

that *efficiently* computes the vector of monomial coefficients of the product polynomial $p = uv$ of the polynomials u and v, passed through their respective coefficient vectors. To that end use the DFT functionality of EIGEN, see [Lecture → Code 4.2.1.22].

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++11-code 4.3.4: Solution of (4-3.b)

```
2  Eigen::VectorXd polyMult_fast(const Eigen::VectorXd &u,
```

```
                                const Eigen::VectorXd &v) {
  // Initialization
  int m = u.size();
  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.

▲

(4-3.c) Determine the complexity of your implementation of algorithm in (4-3.b) making use of the fact that EIGEN's DFT relies on the FFT algorithm [Lecture → Section 4.3] and thus can be carried out with an asymptotic effort $O(n \log n)$ for vector length $n \to \infty$.

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.

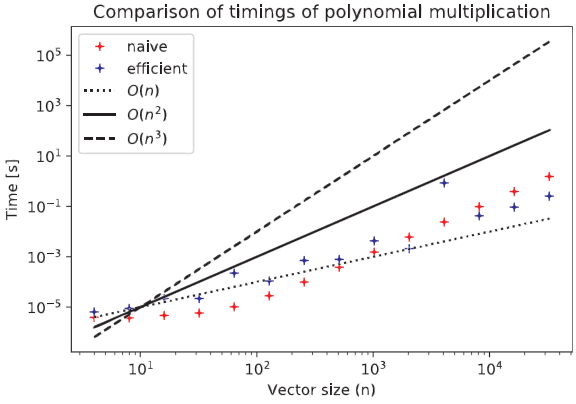


Fig. 27

▲

(4-3.d) ⚄ (20 min.) Give an interpretation of the entries of the vector $\mathrm{DFT}_n\mathbf{u}$ obtained by applying a discrete Fourier transform to the coefficient vector $\mathbf{u}$ of a polynomial u of degree $n-1$? What is an equivalent operation that can be performed for the polynomial u which leads to the same result?

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.

▲

The following sub-problems deal with polynomial division. Polynomial division with remainder can be carried out using Euclid's algorithm and is captured by the following theorem:

Theorem 4.3.6. Polynomial devision with remainder

For every $p \in \mathcal{P}_N$ and $u \in \mathcal{P}_m$, $m \leq N$, there exist unique polynomials $q \in \mathcal{P}_{N-m}$ and $r \in \mathcal{P}_{m-1}$ such that

$$p = uq + r .$$

Using the notations of the theorem, we are interested in determining, whether the polynomial p can be divided by u exactly, that is, whether $r \equiv 0$. In this case we also want to compute the quotient polynomial q.

(4-3.e) ⚂ (45 min.) [depends on Sub-problem (4-3.b)]

Write a C++ function

```
Eigen::VectorXd polyDiv(const Eigen::VectorXd & p, const
   Eigen::VectorXd & u);
```

that takes the coefficient vectors `p` and `u` of two polynomials p and u, respectively, as arguments, where the degree of p must be at least as large as that of u. The function is to return the coefficient vector of a q, such that $p = uv$, if it exists. If not, a vector of length zero should be returned.

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 $(\mathrm{DFT}_{N+1}\widehat{\mathbf{p}})_j = 0$, 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   int mn = uv.size();
5   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);
```

```
  }
  int dim = mn;
  int n = mn - m + 1;

  Eigen::VectorXd v;

  // TODO: (4-3.e) Divide polynomials uv and u efficiently.
  // START

  // zero padding
  Eigen::VectorXd uv_tmp = uv;
  uv_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));
  Eigen::VectorXd u_tmp = u;
  u_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));

  Eigen::FFT<double> fft;
  Eigen::VectorXcd uv_tmp_ = fft.fwd(uv_tmp);
  Eigen::VectorXcd u_tmp_ = fft.fwd(u_tmp);

  // check divisibility:  case(i)
  for (int i = 0; i < dim; ++i) {
    if (abs(uv_tmp_(i)) < 1e-13) {
      if (abs(u_tmp_(i)) < 1e-13) {
        // make cwiseQuotient at i-th position equal 0
        uv_tmp_(i) = 0.0;  // complex assignment (0., 0.)
        u_tmp_(i) = 1.0;   // complex assignment (1., 0.)
      } else {
        std::cerr << "uv can't be divided by u\n";
        return Eigen::VectorXd(0);
      }
    }
  }

  Eigen::VectorXcd tmp = uv_tmp_.cwiseQuotient(u_tmp_);

  v = fft.inv(tmp).real();
  // check divisibility:  case(ii)
  for (int i = n; i < dim; ++i) {
    if (abs(v(i)) > 1e-13) {
      std::cerr << "uv can't be divided by u\n";
      return Eigen::VectorXd(0);
    }
  }

  // reshape v to a suitable size
  v.conservativeResizeLike(Eigen::VectorXd::Zero(n));
  // (mn-1) - (m-1) + 1
  // END

  return v;
}
```

Get it on GitLab (polydiv.hpp).

▲

End Problem 4-3 , 120 min.

Problem 4-4: Solving triangular Toeplitz systems

In [Lecture → Section 4.5] we learned about Toeplitz matrices, the class of matrices with constant diagonals [Lecture → Def. 4.5.1.7]. Obviously, $m \times n$ Toeplitz matrices are *data-sparse* in the sense that it takes only $m+n-1$ steps to encode them completely. Therefore, operations with Toeplitz matrices can often be done with asymptotic computational cost significantly lower than that of the same operations for a generic matrix of the same size. In [Lecture → Section 4.5.2] we have seen this for FFT-based matrix×vector multiplication for Toeplitz matrices.
In this problem we study an efficient FFT-based algorithm for solving triangular linear systems of equations whose coefficient matrix is Toeplitz. Such linear systems are faced, for instance, when inverting a finite, linear, time-invariant, causal channel (LT-FIR) as introduced in [Lecture → Section 4.1].

Template: Get it on GitLab.

In [Lecture → Section 4.1.2] we found that the output operator of a causal finite linear time-invariant filter with impulse response $\mathbf{h} = [h_0, \ldots, h_{n-1}]^\top \in \mathbb{R}^n$ can be obtained by discrete convolution [Lecture → Eq. (4.1.2.4)]; see also [Lecture → Def. 4.1.3.3]. Given an input signal $\mathbf{x} := [x_0, \ldots, x_{n-1}]^\top \in \mathbb{R}^n$, the first n components $y_0, \ldots, y_{n-1}$ of the output are obtained by:

$$\begin{bmatrix} y_0 \\ \vdots \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} h_0 & 0 & \cdots & & \cdots & 0 \\ h_1 & h_0 & 0 & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ h_{n-1} & \cdots & & \cdots & h_1 & h_0 \end{bmatrix}}_{=:\mathbf{H}_n \in \mathbb{R}^{n,n}} \begin{bmatrix} x_0 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix} \tag{4.4.1}$$

Note that the matrix $\mathbf{H}_n \in \mathbb{R}^{n,n}$ is a lower triangular matrix with constant diagonals (linear algebra indexing)

$$(\mathbf{H})_{l,j} = \begin{cases} h_{l-j} & \text{if } l \geq j\,, \\ 0 & \text{else,} \end{cases} \qquad l,j \in \{1,\ldots,n\}$$

This matrix is clearly not circulant: see [Lecture → Def. 4.1.4.12]. At the same time, it still belongs to the class of *Toeplitz matrices*, see [Lecture → Def. 4.5.1.7], which generalizes the class of circulant matrices. (Recall [Lecture → Def. 4.5.1.7]: $\mathbf{T} \in \mathbb{K}^{m,n}$ is a Toeplitz matrix if there is a sequence/array $\mathbf{u} = (u_{-m+1}, \ldots, u_{n-1}) \in \mathbb{K}^{m+n-1}$ such that $t_{ij} = u_{j-i}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$.)

If you want to recover the input signal from the output (*deconvolution*), you will face the following crucial task for digital signal processing: *solve a lower triangular LSE with a Toeplitz system matrix*. Of course, forward elimination, see [Lecture → Section 2.3], can achieve this with asymptotic complexity $\mathcal{O}(n^2)$: see [Lecture → Eq. (2.3.1.6)]. However, since a Toeplitz matrix merely has an information content of $\mathcal{O}(n)$ numbers, there might be a more efficient way – which ought to be explored by you in this problem.

(4-4.a) ⚀ (15 min.) Given a Toeplitz matrix $\mathbf{T} \in \mathbb{R}^{n,n}$, find another matrix $\mathbf{S} \in \mathbb{R}^{n,n}$ such that the following block matrix is a *circulant* matrix:

$$\mathbf{C} = \begin{bmatrix} \mathbf{T} & \mathbf{S} \\ \mathbf{S} & \mathbf{T} \end{bmatrix} \in \mathbb{R}^{2n,2n}\,.$$

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.

▲

(4-4.b) ⚀ (15 min.) [depends on Sub-problem (4-4.a)]

Write the following C++ function:

```
MatrixXd toeplitz(const VectorXd &c, const VectorXd &r)
```

This function should take as input column vectors $\mathbf{c} \in \mathbb{R}^m$ and $\mathbf{r} \in \mathbb{R}^n$ and return a Toeplitz matrix $\mathbf{T} \in \mathbb{R}^{m,n}$ [Lecture → Def. 4.5.1.7] such that (linear-algebra indexing)

$$(\mathbf{T})_{i,j} = \begin{cases} (c)_{i-j+1} & \text{, if } i \geq j \text{ ,} \\ (r)_{j-i+1} & \text{, if } j > i \text{ ,} \end{cases} \quad 1 \leq i \leq m,\ 1 \leq j \leq n \text{ .}$$

SOLUTION of (4-4.b):

C++11-code 4.4.2: Solution of (4-4.b)

```
2 Eigen::MatrixXd toeplitz(const Eigen::VectorXd &c, const Eigen::VectorXd &r) {
3   // Find size of Toeplitz matrix
```

```
  const int m = c.size();
  const int n = r.size();
  Eigen::MatrixXd T(m, n);

  // TODO: build Toeplitz matrix T
  // START
  for (int i = 0; i < n; ++i) {
    T.col(i).tail(m - i) = c.head(m - i);
  }
  for (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).

▲

(4-4.c) ⚂ (20 min.) [depends on Sub-problem (4-4.a)]

Show that the following two C++ functions realize the same linear mapping $\mathbf{x} \mapsto \mathbf{y}$, when supplied with the same arguments.

C++11-code 4.4.3: Function `toepmatmult()`

```
Eigen::VectorXd toepmatmult(const Eigen::VectorXd &c, const Eigen::VectorXd &r,
                            const Eigen::VectorXd &x) {
  assert(c.size() == r.size() && c.size() == x.size() &&
         "c, r, x have different lengths!");
  return toeplitz(c, r) * x;
}
```

Get it on GitLab (toeplitz.hpp).

C++11-code 4.4.4: Function `toepmult()`

```
Eigen::VectorXd toepmult(const Eigen::VectorXd &c, const Eigen::VectorXd &r,
                         const Eigen::VectorXd &x) {
  assert(c.size() == r.size() && c.size() == x.size() &&
         "c, r, x have different lengths!");
  int n = c.size();

  // Complex arithmetic required here
  Eigen::VectorXcd cr_tmp = c.cast<std::complex<double>>();
  Eigen::VectorXcd x_tmp = x.cast<std::complex<double>>();

  // Prepare vector encoding circulant matrix C
  cr_tmp.conservativeResize(2 * n);
  cr_tmp.tail(n) = Eigen::VectorXcd::Zero(n);
  cr_tmp.tail(n - 1).real() = r.tail(n - 1).reverse();

  // Zero padding
  x_tmp.conservativeResize(2 * n);
  x_tmp.tail(n) = Eigen::VectorXcd::Zero(n);

```

```
21   // Periodic discrete convolution from [Lecture → Code 4.2.2.4]
22   Eigen::VectorXd y = pconvfft(cr_tmp, x_tmp).real();
23   y.conservativeResize(n);
24
25   return y;
26 }
```

Get it on GitLab (toeplitz.hpp).

The function `pconvfft` is defined in [Lecture → Section 4.2.2], [Lecture → Code 4.2.2.4], and `toeplitz()` is the simple C++ function from Sub-problem (4-4.b).

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    Eigen::VectorXcd tmp = (fft.fwd(u)).cwiseProduct(fft.fwd(x));
6
7    return fft.inv(tmp);
8  }
```

Get it on GitLab (pconvfft.hpp).

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.

▲

(4-4.d) ⚀ (10 min.) What is the asymptotic complexity of `toepmatmult()` (→ Code 4.4.3) and `toepmult()` (→ Code 4.4.4)?

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 $n \to \infty$.

▲

(4-4.e) ⚁ Explain in detail what the following C++ functions are meant for and why they are algebraically equivalent (i.e. equivalent when ignoring roundoff errors), provided that `h.size()= 2^l`.

C++11-code 4.4.6: Function `ttmatsolve`

```
2  Eigen::VectorXd ttmatsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y) {
3    assert(h.size() == y.size() && "h and y have different lengths!");
4    int n = h.size();
5
6    Eigen::VectorXd h_tmp = Eigen::VectorXd::Zero(n);
7    h_tmp(0) = h(0);
8
9    Eigen::MatrixXd T = toeplitz(h, h_tmp);
10
11   Eigen::VectorXd x = T.fullPivLu().solve(y);
12
13   return x;
14 }
```

Get it on GitLab (toeplitz.hpp).

C++11-code 4.4.7: Function `ttrecsolve`

```
2  Eigen::VectorXd ttrecsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y,
3                             int l) {
4    assert(h.size() == y.size() && "h and y have different lengths!");
5    // Result vector
6    Eigen::VectorXd x;
7    // Trivial case of asn 1x1 LSE
8    if (l == 0) {
9      x.resize(1);
10     x(0) = y(0) / h(0);
11   } else {
12     int n = std::pow(2, l);
13     int m = n / 2;
14
15     // Check matching length of vectors
16     assert(h.size() == n && y.size() == n &&
17            "h and y have length different from 2^l!");
```

```
18
19      Eigen::VectorXd x1 = ttrecsolve(h.head(m), y.head(m), l - 1);
20      Eigen::VectorXd y2 =
21          y.segment(m, m) -
22          toepmult(h.segment(m, m), h.segment(1, m).reverse(), x1);
23      Eigen::VectorXd x2 = ttrecsolve(h.head(m), y2, l - 1);
24
25      x.resize(n);
26      x.head(m) = x1;
27      x.tail(m) = x2;
28    }
29
30    return x;
31  }
```

Get it on GitLab (toeplitz.hpp).

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.

▲

(4-4.f) ⊡ (5 min.) Why is the algorithm implemented in `ttrecsolve()` called a "divide & conquer" method?

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.

▲

(4-4.g) ⚀ (30 min.) Perform a runtme comparison of `toepmatmult` vs `toepmult` and `ttmatsolve` vs `ttrecsolve` for

```
h = VectorXd::LinSpaced(n,1,n).cwiseInverse();
```

and $n = 2^l$, $l = 3, \ldots, 9$. Plot the timing results for both functions in doubly logarithmic scale. Describe your observations.

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, use the provided `timer.h`. ⌟

SOLUTION of (4-4.g):

C++11-code 4.4.8: Solution of (4-4.g)

```
2  void runtime_toeplitz() {
3    // memory allocation for plot
4    std::vector<double> vec_size;
5    std::vector<double> elap_time_matmult, elap_time_mult, elap_time_ttmat,
6        elap_time_ttrec;
7
8    // header for the results to print out
9    std::cout << std::setw(8) << "n" << std::setw(15) << "toepmatmult"
10             << std::setw(15) << "toepmult" << std::setw(20) << "ttmatsolve"
11             << std::setw(15) << "ttrecsolve" << std::endl;
12
```

```
13   for (unsigned int l = 3; l <= 9; l += 1) {
14     // vector size
15     unsigned int n = std::pow(2, l);
16     // save vector size n
17     vec_size.push_back(n);
18
19     // number of repetitions
20     unsigned int repeats = 3;
21     // TODO: (4-4.g) Perform a runtime comparison by repeating time
          computation
22     // 'repeats' times
23     // START
24     Timer tm_matmult, tm_mult, tm_ttmat, tm_ttrec;
25     // repeat test 'repeats' times
26     for (unsigned int rr = 0; rr < repeats; ++rr) {
27       // create runtime test using randome vectors and given vector h
28       Eigen::VectorXd h = Eigen::VectorXd::LinSpaced(n, 1, n).cwiseInverse();
29       Eigen::VectorXd c = Eigen::VectorXd::Random(n);
30       Eigen::VectorXd r = Eigen::VectorXd::Random(n);
31       Eigen::VectorXd x = Eigen::VectorXd::Random(n);
32       Eigen::VectorXd y = Eigen::VectorXd::Random(n);
33       r(0) = c(0);
34
35       // compute times for toepmatmult implementation
36       tm_matmult.start();
37       toepmatmult(c, r, x);
38       tm_matmult.stop();
39
40       // compute times for toepmult implementation
41       tm_mult.start();
42       toepmult(c, r, x);
43       tm_mult.stop();
44
45       // compute times for ttmatsolve implementation
46       tm_ttmat.start();
47       ttmatsolve(h, y);
48       tm_ttmat.stop();
49       // compute times
50       tm_ttrec.start();
51       ttrecsolve(h, y, l);
52       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   /* DO NOT CHANGE */
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.

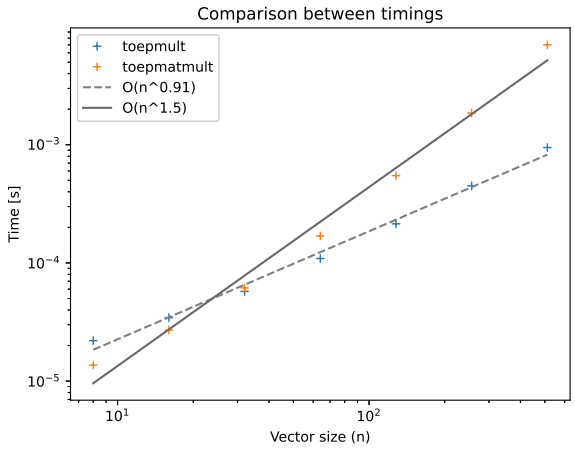


Fig. 28

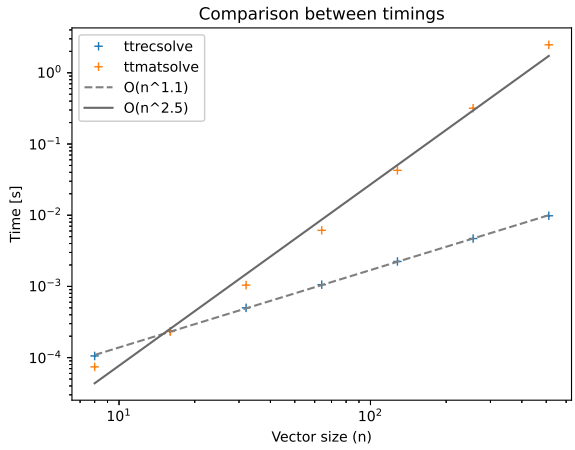


Fig. 29

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.

▲

(4-4.h) ⚀ (20 min.) Derive the asymptotic complexity of both functions `ttmatsolve()` and `ttrecsolve()` in terms of the problem size parameter $n \to \infty$.

SOLUTION 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:

$$\sum_{\tilde{l}=1}^{l} 2^{l-\tilde{l}}(l-\tilde{l}+1)2^{\tilde{l}} = 2^l \sum_{\tilde{l}=1}^{l}(l-\tilde{l}+1)$$

$$= n\left(l^2 - \frac{l(l+1)}{2} + l\right) = n\frac{l(l+1)}{2} = O\left(n\log^2(n)\right) \quad \text{for} \quad n \to \infty\,.$$

In the last step we exploited the equality $l = \log_2(n) = \frac{\log(n)}{\log(2)}$.

▲

(4-4.i) ⚀ (30 min.) For the case that `n = h.size()` is **not** a power of 2, implement a *wrapper function*

```
VectorXd ttsolve(const VectorXd &h, const VectorXd &y);
```

for `ttrecsolve` that, in the absence of roundoff errors, would behave exactly like `ttmatsolve`, i.e. `ttsolve(h,y)== ttmatsolve(h,y)`. The term "wrapper function" indicates that `ttsolve()` is supposed to call `ttrecsolve()` after preparing suitable input vectors.

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    int n = h.size();
5
6    Eigen::VectorXd x;
7
8    // TODO: wrap 'ttrecsolve' for any size n
9    // START
10   int l = std::ceil(std::log2(n));
11   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).

▲

End Problem 4-4 , 145 min.

Problem 4-5: Implementing Discrete Convolution

[Lecture → Section 4.2.2] discussed in detail the implementation of discrete convolutions of signals stored in vectors by means of the discrete Fourier transform (DFT). This problem revisits these techniques and adds a new twist to implementation.

This problem assumes solid knowledge of [Lecture → Section 4.2.2].

From [Lecture → Section 4.1.3] we recall the definitions

Definition [Lecture → Def. 4.1.3.3]. Discrete convolution

Given $\mathbf{x} = [x_0, \ldots, x_{m-1}]^\top \in \mathbb{K}^m$, $\mathbf{h} = [h_0, \ldots, h_{n-1}]^\top \in \mathbb{K}^n$ their **discrete convolution** (DCONV) $\mathbf{h} * \mathbf{x}$ is the vector $\mathbf{y} = [y_0, \ldots, y_{m+n-2}]^\top \in \mathbb{K}^{m+n-1}$ with components

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j \,, \quad k = 0, \ldots, m+n-2 \,, \qquad \text{[Lecture → Eq. (4.1.3.4)]}$$

where we have adopted the convention $h_j := 0$ for $j < 0$ or $j \geq n$.

and

Definition *cf.* [Lecture → Def. 4.1.4.7]. Discrete periodic convolution (of vectors)

The **discrete periodic convolution** $\mathbf{p} *_n \mathbf{x}$ of two vectors $\mathbf{p}, \mathbf{x} \in \mathbb{K}^n$ is the vector $\mathbf{y} \in \mathbb{K}^n$ with components (C++ indexing)

$$(\mathbf{y})_k := \sum_{j=0}^{n-1} (\mathbf{p})_{(k-j) \bmod n} (\mathbf{x})_j \,, \quad k \in \{0, \ldots, n-1\} \,.$$

($(k-j) \bmod n \in \{0, \ldots, n-1\} \;\widehat{=}\;$ `(k-j)%n` in C++ syntax.)

(4-5.a) ⚀ (15 min.) For given $\mathbf{h} \in \mathbb{K}^n$, $\mathbf{x} \in \mathbb{K}^m$, specify the vectors $\widetilde{\mathbf{h}}, \widetilde{\mathbf{x}}$, the *minimal* vector length $N \in \mathbb{N}$, and the missing index range `?:?` in the formula

$$\mathbf{h} * \mathbf{x} = \left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}} \right)_{?:?} \,.$$

C++ indexing has to be used and the notational convention is $(\mathbf{v})_{r:s} = [(\mathbf{v})_r, \ldots, (\mathbf{v})_s] \in \mathbb{K}^{s-r+1}$ for $r, s \in \mathbb{N}$, $r \leq s$.

SOLUTION of (4-5.a):

We can rewrite the discrete convolution of two vectors [Lecture → Eq. (4.1.3.4)] as (C++ indexing used)

$$(\mathbf{h} * \mathbf{x})_k = \sum_{j=\max\{0,k-n+1\}}^{\min\{m-1,k\}} (\mathbf{x})_j (\mathbf{h})_{k-j} \,, \quad k = 0, \ldots, m+n-2 \,. \tag{4.5.1}$$

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 $N \geq \max\{m,n\}$.

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

$$\begin{aligned}&\left(\widetilde{\mathbf{h}}\right)_{k+N-j} = 0 \qquad \forall j \in \{k+1, \ldots, m-1\} , \quad \forall k \in \{0, \ldots, m-2\} \\ \Leftrightarrow\ & k+N-j \geq n \qquad \forall j \in \{k+1, \ldots, m-1\} , \quad \forall k \in \{0, \ldots, m-2\} \\ \Leftrightarrow\ & N-(m-1) \geq n \\ \Leftrightarrow\ & N \geq n+m-1 .\end{aligned}$$

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 4.5.2: 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].

▲

The discrete Fourier transform in EIGEN relies on the library KISSFFT, which offers a very efficient implementation of DFT with asymptotic computational cost $O(n \log n)$ for vectors of length $n := 2^\ell$, $\ell \in \mathbb{N}$, but may not be fast for vectors of other lengths. To avoid degraded performance of a numerical code, the following rule is imposed throughout this problem:

EIGEN's built-in DFT implementations may be invoked only for vectors of length 2^ℓ for some $\ell \in \mathbb{N}$!

(4-5.b) ⚄ (40 min.) Based on EIGEN, in the file `discreteconvolution.hpp` implement an *efficient* function

```
Eigen::VectorXd pconv_fast(const Eigen::VectorXd &p,
                           const Eigen::VectorXd &x);
```

that computes the discrete periodic convolution $\mathbf{p} *_n \mathbf{x}$ as defined in [Lecture → Def. 4.1.4.7] of two real vectors $\mathbf{p}, \mathbf{x} \in \mathbb{R}^n$ of the same length.

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 $\mathbf{p} = [p_0, \ldots, p_{n-1}]^\top \in \mathbb{K}^n$, $\mathbf{x} = [x_0, \ldots, x_{n-1}]^\top \in \mathbb{K}^n$ from [Lecture → Def. 4.1.4.7], see also [Lecture → Code 4.2.2.1]:

$$(\mathbf{p} *_n \mathbf{x})_k = \sum_{j=0}^{n-1} x_j p_{(k-j) \bmod n} = \sum_{j=0}^{k} x_j p_{k-j} + \sum_{j=k+1}^{n-1} x_j p_{n+k-j}\,, \quad k = 0, \ldots, n-1\,. \tag{4.5.3}$$

Then 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

$$(\widetilde{\mathbf{x}})_j := \begin{cases} x_j & \text{for } j = 0,\ldots,n-1\,, \\ 0 & \text{for } j = 1n,\ldots,N-1 \end{cases}\,, \quad (\widetilde{\mathbf{p}})_j := \begin{cases} p_j & \text{for } j = 0,\ldots,n-1\,, \\ 0 & j = n+1,\ldots,N-n-1\,, \\ p_{j+(n-N)} & \text{for } j = n-N,\ldots,N-1\,. \end{cases}$$

In particular, this means

$$\widetilde{p}_j = p_j \quad \text{for} \quad j = 0,\ldots,n-1 \quad , \quad \widetilde{p}_{N-\ell} = p_{n-\ell} \quad \text{for} \quad \ell = 1,\ldots,n\,. \tag{4.5.4}$$

Based on (4.5.3), 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 $j \geq n$:

$$\blacktriangleright \quad (\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} x_j \widetilde{p}_{k-j} + \sum_{j=k+1}^{n-1} x_j \widetilde{p}_{N+k-j}\,. \tag{4.5.5}$$

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 (4.5.5) that $k-j \in \{0,\ldots,n-1\}$,
- and that $j-k \in \{1,\ldots,n-1\}$ in the second sum of (4.5.5), which means $\widetilde{p}_{N+k-j} = p_{n+k-j}$ thanks to (4.5.4).

$$\blacktriangleright \quad (\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} x_j p_{k-j} + \sum_{k=k+1}^{n-1} x_j p_{n+k-j} = (\mathbf{p} *_n \mathbf{x})_k\,, \quad k \in \{0,\ldots,n-1\}\,.$$

This is exploited in the following C++ function, to which the parameter `N` passes N:

C++ code 4.5.6: Resizing periodic convolution

```
Eigen::VectorXd pconvN(const Eigen::VectorXd &p, const Eigen::VectorXd &x,
                       unsigned int N) {
  const int n = x.size();
  assert(n == p.size());
  // Requirement on N to avoid overlap in periodic padding
  assert(N >= 2 * n);
  // Periodic padding of vector p
  Eigen::VectorXd pt{Eigen::VectorXd::Zero(N)};
  pt.head(n) = p;
  pt.tail(n) = p;
  // Zero padding of vector x
  Eigen::VectorXd xt{Eigen::VectorXd::Zero(N)};
  xt.head(n) = x;
  // Periodic convolution of length N
  const Eigen::VectorXd y = pconv(pt, xt);
  return y.head(n);
}
```

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 4.5.6 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 4.5.6 and [Lecture → Code 4.2.2.1].

C++ code 4.5.7: 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    // TO DO: 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.

▲

End Problem 4-5 , 55 min.

Problem 4-6: Asymptotic Cost of EIGEN-Based Functions

This short problem is about reading off the asymptotic computational cost of some C++ functions performing numerical linear algebra tasks based on EIGEN. It relies on information provided in [Lecture → Section 1.4.2], [Lecture → § 2.5.0.4], [Lecture → § 3.3.3.37], and [Lecture → Section 4.3].

Just reading of C++ code is required.

(4-6.a) ⚂ (8 min.) The function `slvtriag()` listed in Code 4.6.1 expects two vector arguments of equal length $n \in \mathbb{N}$. What is its asymptotic computational complexity for $n \to \infty$?

$$\mathrm{cost}(\texttt{slvtriag}) = O(\quad) \quad \text{for} \quad n \to \infty\,.$$

C++ code 4.6.1: Function `slvtriag()`

```
2 Eigen::VectorXd slvtriag(const Eigen::VectorXd &u, const Eigen::VectorXd &v) {
3   assert((u.size() == v.size()) && "Size mismatch");
4   const Eigen::MatrixXd A{u * v.transpose()};
5   return A.triangularView<Eigen::Upper>().solve(u);
6 }
```

SOLUTION 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

$$\texttt{slvtriag}(\mathbf{u},\mathbf{v}) = \mathrm{triu}(\mathbf{u}\mathbf{v}^\top)^{-1}\mathbf{u}\,.$$

▲

(4-6.b) ⚂ (8 min.) What is the asymptotic complexity of the C++ function `slvsymrom()` listed as Code 4.6.2 in terms of the length n of its argument vector `u`?

$$\mathrm{cost}(\texttt{slvsymrom}) = O(\quad) \quad \text{for} \quad n \to \infty\,.$$

A sharp bound is expected.

C++ code 4.6.2: Function `slvsymrom()`

```
2 Eigen::VectorXd slvsymrom(const Eigen::VectorXd &u) {
3   const unsigned int n = u.size();
4   return (Eigen::MatrixXd::Identity(n, n) + u * u.transpose()).lu().solve(u);
```

```
5 }
```

SOLUTION 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 → Eq. (2.3.2.10)]. This step determines the asymptotic computational complexity of the function.

Remark. The result of the function is

$$\texttt{slvsymrom}(\mathbf{u}) = (\mathbf{I} + \mathbf{u}\mathbf{u}^\top)^{-1}\mathbf{u} \,.$$

▲

(4-6.c) ⚀ (12 min.) Code 4.6.3 lists the C++ function `getcompbas()`, whose argument `A` is a densely populated matrix $\mathbf{A} \in \mathbb{R}^{n,k}$, $k < n$. Give a sharp asymptotic bound for the asymptotic computational cost of `getcompbas()` for $n \to \infty$ assuming k to be small and fixed.

$$\texttt{cost}(\texttt{getcompbas}) = O(\quad\quad) \quad \text{for} \quad n \to \infty \,.$$

C++ code 4.6.3: Function `getcompbas()`

```
2  Eigen::MatrixXd getcompbas(const Eigen::MatrixXd &A) {
3    const int n = A.rows();
4    const int k = A.cols();
5    Eigen::HouseholderQR<Eigen::MatrixXd> qr(A);
6    return qr.householderQ() *
7           (Eigen::MatrixXd(n, n - k) << Eigen::MatrixXd::Zero(k, n - k),
8            Eigen::MatrixXd::Identity(n - k, n - k))
9              .finished();
10 }
```

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.

▲

(4-6.d) ⚄ (12 min.) Code 4.6.4 shows the implementation of the C++ function `fft2_square()`, which requires a complex $n \times n$-matrix $\mathbf{Y} \in \mathbb{C}^{n,n}$ as argument `Y`. What is a sharp bound on the asymptotic complexity of `fft2_square()` as $n \to \infty$?

$$\text{cost}(\texttt{fft2\_square}) = O(\quad\quad) \quad \text{for} \quad n \to \infty\,.$$

C++ code 4.6.4: Function `fft2_square()`

```
2  Eigen::MatrixXcd fft2_square(const Eigen::MatrixXcd &Y) {
3    const unsigned int n = Y.rows();
4    assert((n == Y.cols()) && "Matrix Y must be square");
5    Eigen::MatrixXcd tmp(n, n);
6    Eigen::FFT<double> fft;
7    for (int k = 0; k < n; ++k) {
8      const Eigen::VectorXcd tv(Y.row(k));
9      tmp.row(k) = fft.fwd(tv).transpose();
10   }
11   for (int k = 0; k < n; ++k) {
12     const Eigen::VectorXcd tv(tmp.col(k));
13     tmp.col(k) = fft.fwd(tv);
14   }
15   return tmp;
16 }
```

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].

▲

End Problem 4-6, 40 min.

Problem 4-7: Modified Cosine Transform

Many so-called trigonometric transformations can be reduced to the discrete Fourier transform (DFT). One such example is studied in this problem.

Assumes familiarity with the discrete Fourier transform, C++, and the complex exponential

For a given vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]^\top \in \mathbb{R}^n$, $n \in \mathbb{N}$, we want to compute another vector $\mathbf{f} := [f_0, \ldots, f_{n-1}]^\top \in \mathbb{R}^n$ according to the formula

$$f_k := \sum_{j=0}^{n-1} a_j \cos\left(\pi \frac{jk}{n}\right), \quad k = 0, \ldots, n-1 . \tag{4.7.1}$$

Using the identity $\cos x = \frac{1}{2}(\exp(-\imath x) + \exp(\imath x))$, we find the alternative formula

$$f_k = \sum_{\ell=0}^{n-1} \tfrac{1}{2} a_\ell \exp\left(-2\pi\imath \frac{k\ell}{2n}\right) + \exp\left(\pi\imath \frac{k(n-1)}{n}\right) \cdot \sum_{\ell=0}^{n-1} \tfrac{1}{2} a_{n-1-\ell} \exp\left(-2\pi\imath \frac{k\ell}{2n}\right) \tag{4.7.2}$$

for $k = 0, \ldots, n-1$.

Recall the definition of the discrete Fourier transform (DFT) as multiplication of a vector with the Fourier matrix $\mathbf{F}_n$:

Definition [Lecture → Def. 4.2.1.18].

The linear map $\mathrm{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathrm{DFT}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called **discrete Fourier transform** (DFT), i.e. for $[c_0, \ldots, c_{n-1}] := \mathrm{DFT}_n(\mathbf{y})$

$$c_k := \sum_{j=0}^{n-1} y_j \exp\left(-2\pi\imath \frac{kj}{n}\right), \quad k = 0, \ldots, n-1 . \qquad [\text{Lecture} \rightarrow \text{Eq. (4.2.1.19)}]$$

(4-7.a) ⚂ (10 min.) Taking the cue from (4.7.2) we can write ("C++ indexing" throughout)

$$\mathbf{f} = (\mathrm{DFT}_{2n}(\mathbf{x}))_{0:n-1} + \mathbf{D}(\mathrm{DFT}_{2n}(\mathbf{y}))_{0:n-1} , \tag{4.7.3}$$

with suitable vectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^{2n}$ and a diagonal matrix $\mathbf{D} \in \mathbb{C}^{n,n}$. Characterize the vectors $\mathbf{x}, \mathbf{y}$ (in terms of $\mathbf{a}$) and the matrix $\mathbf{D}$.

$$(\mathbf{x})_\ell = \begin{cases} \square & \text{for } \ell \in \{\square\}, \\ \square & \text{for } \ell \in \{\square\}, \end{cases} \quad , \quad (\mathbf{y})_\ell = \begin{cases} \square & \text{for } \ell \in \{\square\}, \\ \square & \text{for } \ell \in \{\square\}, \end{cases} \quad ,$$

$$(\mathbf{D})_{k,k} = \square , \quad k = 0, \ldots, n-1 .$$

SOLUTION of (4-7.a):

Remark. The formula (4.7.2) is derived as follows

$$\begin{aligned} f_k &= \sum_{j=0}^{n-1} a_j \cos\left(\pi \frac{kj}{n}\right) = \frac{1}{2} \sum_{j=0}^{n-1} a_j \left(e^{-\pi \imath \frac{kj}{n}} + e^{\pi \imath \frac{kj}{n}} \right) \\ &= \sum_{j=0}^{n-1} \tfrac{1}{2} a_j e^{-2\pi \imath \frac{kj}{2n}} + \sum_{\ell=0}^{n-1} \tfrac{1}{2} a_{n-1-\ell} e^{2\pi \imath \frac{k(n-1-\ell)}{2n}} \\ &= \sum_{j=0}^{n-1} \tfrac{1}{2} a_j e^{-2\pi \imath \frac{kj}{2n}} + e^{\pi \imath \frac{k(n-1)}{n}} \sum_{\ell=0}^{n-1} \tfrac{1}{2} a_{n-1-\ell} e^{-2\pi \imath \frac{k\ell}{2n}} \end{aligned}$$

Matching this with (4.7.3) we find

$$(\mathbf{x})_j = \begin{cases} \frac{1}{2} a_j & \text{for } j \in \{0, \ldots, n-1\}, \\ 0 & \text{for } j \in \{n, \ldots, 2n-1\}, \end{cases}, \quad (\mathbf{y})_j = \begin{cases} \frac{1}{2} a_{n-1-j} & \text{for } j \in \{0, \ldots, n-1\}, \\ 0 & \text{for } j \in \{n, \ldots, 2n-1\}, \end{cases},$$

$$(\mathbf{D})_{k,k} = \exp\left(\pi \imath k \frac{n-1}{n}\right), \quad \ell = 0, \ldots, n-1. \tag{4.7.4}$$

▲

(4-7.b) ⚀ (10 min.) The C++ function `modcostrf()` realizes the mapping $\mathbf{a} \to \mathbf{f}$ according to (4.7.1). Supplement the missing parts of the following listing by writing valid C++ code in the boxes.

```
Eigen::VectorXd modcostrf(const Eigen::VectorXd &a) {
  using Comp = std::complex<double>;
  unsigned int n = a.size();
  Eigen::VectorXd f(n);
  Eigen::FFT<double> fft;
  Eigen::VectorXcd tmp(          );
  tmp << 
                                  , Eigen::VectorXcd::Zero(n);
  Eigen::VectorXcd v1 = fft.                ;
  tmp << 0.5 * a.reverse(),                                    ;
  Eigen::VectorXcd v2 = fft.                ;
  Comp fac = std::exp((n - 1) * M_PI * Comp(0, 1)) / (double)n);
  Comp d(1.0, 0.0);
  for (int k = 0; k < n; ++k) {
    f[k] = (v1[          ] +           * v2[          ]).real();
                *=           ;
  }
  return f;
}
```

The constant `M_PI` is the number π.

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::DFT** 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()`

```
2  Eigen::VectorXd modcostrf(const Eigen::VectorXd &a) {
3    using Comp = std::complex<double>;
4    unsigned int n = a.size();
5    Eigen::VectorXd f(n);
6    Eigen::FFT<double> fft;
7    Eigen::VectorXcd tmp(2 * n);
8    tmp << 0.5 * a, Eigen::VectorXcd::Zero(n);
9    Eigen::VectorXcd v1 = fft.fwd(tmp);
10   tmp << 0.5 * a.reverse(), Eigen::VectorXcd::Zero(n);
11   Eigen::VectorXcd v2 = fft.fwd(tmp);
12   Comp fac = std::exp(((n - 1) * M_PI * Comp(0, 1)) / (double)n);
13   Comp d(1.0, 0.0);
14   for (int k = 0; k < n; ++k) {
15     f[k] = (v1[k] + d * v2[k]).real();
16     d *= fac;
17   }
18   return f;
19 }
```

▲

End Problem 4-7, 20 min.

Chapter 5

Data Interpolation and Data Fitting in 1D

Problem 5-1: Evaluating the derivatives of interpolating polynomials

In [Lecture → Ex. 5.1.0.8] we learned about the importance of data interpolation for obtaining functor representations of constitutive relationships $t \mapsto f(t)$. Numerical methods like Newton's method [Lecture → Code 8.4.2.2] often require information about the derivative f' as well. Therfore, we need efficient ways to evaluate the derivatives of interpolants. In this problem we discuss this issue for polynomial interpolation for (i) monomial representation and (ii) "update-friendly" point evaluation. We generalize the Horner scheme [Lecture → § 5.2.1.5] and the Aitken-Neville algorithm [Lecture → § 5.2.3.8].

This problem is connected with [Lecture → Section 5.2.3], simple coding in C++/EIGEN is requested.

We first deal with polynomials in monomial representation [Lecture → Rem. 5.2.1.4].

(5-1.a) ⚄ (20 min.) Using the Horner scheme [Lecture → § 5.2.1.5], in the file `eval_deriv.hpp` write an efficient C++ implementation of a template function which returns the pair $(p(x), p'(x))$, where p is a polynomial with coefficients in `c`:

```
std::pair<double,double> evaldp(const CoeffVec &c, double x);
```

Vector `c` contains the coefficient of the polynomial in the monomial basis, following the PYTHON/MATLAB convention (leading coefficient in `c[0]`), that is,

$$p(t) = \texttt{c[0]}\,t^{k} + \texttt{c[1]}\,t^{k-1} + \cdots + \texttt{c[k-1]}\,t + \texttt{c[k]}\,, \quad t \in \mathbb{R}\,. \tag{5.1.1}$$

As indicated above, the type **CoeffVec** must provide component access via **operator** `[](int) const` and a `size()` method.

SOLUTION 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()`

```
std::pair<double, double> evaldp(const Eigen::VectorXd &c, const double x) {
  double px, dpx;
  const int s = c.size();  // degree + 1

  // TODO (5-1.a):  Implement the function evaldp().

  // START
  // Standard Horner scheme
  px = c(0);
  for (int i = 1; i < s; ++i) {
    px = x * px + c(i);
  }

  // Horner scheme for derivative
  dpx = (s - 1) * c(0);
  for (int i = 1; i < s - 1; ++i) {
    dpx = x * dpx + (s - i - 1) * c(i);
  }
  // END

  return {px, dpx};
}
```

Get it on GitLab (eval_deriv.hpp).

▲

(5-1.b) ⚀ (15 min.) For the sake of testing, write a naive C++ implementation of the evaluation requested in (5-1.a)

```
std::pair<double,double> evaldp_naive(const CoeffVec &c, double x);
```

which returns the same pair $(p(x), p'(x))$. However, this time $p(x)$ and $p'(x)$ should be calculated by means of simply summing the contributions of the monomials $t \mapsto t^k$.

SOLUTION 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);
```

```
16    for (int i = 1; i < n - 1; ++i) {
17      dpx = dpx + (n - i - 1) * c[i] * std::pow(x, n - i - 2);
18    }
19    // END
20
21    return {px, dpx};
22  }
```

Get it on GitLab (eval_deriv.hpp).

▲

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

What are the asymptotic complexities of the two functions you implemented in (5-1.a) and (5-1.b) in terms of raising the polynomial degree?

SOLUTION of (5-1.c):

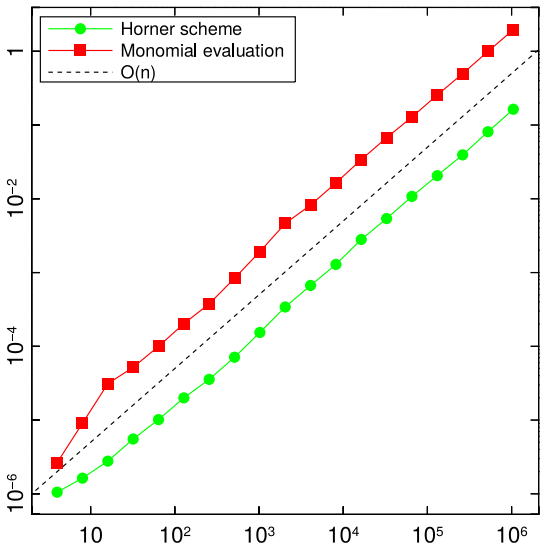


Fig. 30

The answer depends on your implementation. In any case, the code for `evaldp()` from Sub-problem (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!

▲

(5-1.d) ⚀ (15 min.) [depends on Sub-problem (5-1.a), Sub-problem (5-1.b)]

Implement a function

```
bool polyTestTime(const unsigned int d);
```

that

(i) validates your implementation of the two functions `evaldp()`/`evaldp_naive()` from (5-1.a)/(5-1.b) by comparing their results for $x = 1.23$ and a polynomial of degree n with monomial coefficients $c_0 = 1, c_1 = 2, \ldots, c_n = n$ and $n = 2, 4, 8, \ldots, 2^d$. Return **false** in case of a discrepancy.

(ii) measures the runtimes of the two functions for every degree and tabulates them. To that end use the **Timer** class defined in `timer.h`:

```
Timer t;
t.start(); /* do something */ t.stop();
double time = t.duration();
```

HINT 1 for (5-1.d): Please recall from [Lecture → Rem. 1.5.3.15] the advice on comparing numerically computed values ⌟

SOLUTION of (5-1.d):

C++-code 5.1.5: Solution of Sub-problem (5-1.d)

```
2  bool polyTestTime(const unsigned int d) {
3    bool ret = false;  // Return value
4    double x = 0.123;
5    std::pair<double, double> p, p_naive;
6    const int repeats = 10;
7
8    std::cout
9        << std::setw(10) << "n" << std::setw(25)
10       << "Horner scheme:" << std::setw(25) << "Monomial approach:"
11       << "\n"
12       << " ==========================================================\n";
13
14   // TODO (5-1.d): Compare the output of evaldp() and evaldp_naive() for
15   // x = 0.123 and tabulate their runtimes for n = 2,4,...,2^d.
16
17   // START
18   const double TOL = 1E-8;
19   const int max_n = std::pow(2, d);
20   Eigen::VectorXd c = Eigen::VectorXd::LinSpaced(max_n + 1, 1, max_n + 1);
21
22   for (int n = 2; n <= max_n; n *= 2) {
23     Timer tm_slow, tm_fast;
24
25     for (int r = 0; r < repeats; r++) {
26       tm_fast.start();
27       p = evaldp(c.head(n + 1), x);
28       tm_fast.stop();
29
30       tm_slow.start();
31       p_naive = evaldp_naive(c.head(n + 1), x);
32       tm_slow.stop();
33     }
34
35     std::cout << std::setw(10) << n << std::setw(25) << tm_fast.mean()
36               << std::setw(25) << tm_slow.mean() << "\n";
37
38     // Check correctness
39     double err_p = std::abs(p.first - p_naive.first);
40     double err_dp = std::abs(p.second - p_naive.second);
41
42     if (err_p >= TOL || err_dp >= TOL) {
43       std::cout << "Discrepancy at n = " << n << std::endl;
44
45       return false;
46     }
47   }
48
```

```
49    ret = true;
50    // END
51
52    return ret;
53  }
```

Get it on GitLab (eval_deriv.hpp).

▲

Now we revisit polynomial interpolation. In [Lecture → Section 5.2.3.2] we learned about an efficient and "update-friendly" scheme for evaluating Lagrange interpolants at a *single or a few* points. This so-called **Aitken-Neville algorithm**, see [Lecture → Code 5.2.3.10], can be extended to return the derivative value of the polynomial interpolant as well. This will be explored next.

(5-1.e) ⚀ (30 min.) In the file `eval_deriv.hpp` write an efficient C++ function

```
Eigen::VectorXd dipoleval(const Eigen::VectorXd & t,
                          const Eigen::VectorXd & y,
                          const Eigen::VectorXd & x);
```

that returns the row vector $[p'(x_1),\ldots,p'(x_m)]^\top$, when the argument `x` passes $[x_1,\ldots,x_m]$, $m \in \mathbb{N}$ small. Here, p' denotes the *derivative* of the polynomial $p \in \mathcal{P}_n$ interpolating the data points (t_i, y_i), $i = 0,\ldots,n$, for pairwise different $t_i \in \mathbb{R}$ and data values $y_i \in \mathbb{R}$.

To that end differentiate the recursion formula [Lecture → Eq. (5.2.3.9)] and devise an algorithm in the spirit of the Aitken-Neville algorithm implemented in [Lecture → Code 5.2.3.10].

HINT 1 for (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).

▲

(5-1.f) ⚀ (30 min.) For validation purposes devise an alternative, less efficient, implementation of `dipoleval()`

```
Eigen::VectorXd dipoleval_alt(const Eigen::VectorXd &t,
                              const Eigen::VectorXd &y,
                              const Eigen::VectorXd &x);
```

based on the following algorithm:

1. Use the function

   ```
   Eigen::VectorXd polyfit(const Eigen::VectorXd& t,
                           const Eigen::VectorXd& y,
                           unsigned int degree);
   ```

   (supplied in `polyfit.hpp`) to compute the monomial coefficients of the Lagrange interpolant of the data points.

2. Compute the monomial coefficients of the derivative.

3. Use the function

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

   (defined in `polyval.hpp`) to evaluate the derivative at a number of points. This function evaluates the polynomial with monomial coefficients passed in `p` at all points in `x`, *cf.* [Lecture → Code 5.2.1.7].

Use `dipoleval_alt()` to verify the correctness of your implementation of `dipoleval()` by computing the derivative of the degree-10 interpolating polynomial for the values $(t_i, sin(t_i))$, where t_i are equidistant points in $[0,3]$. This test should be conducted within the function

```
bool testDipolEval(void);
```

Return **false** in case the test fails, **true** otherwise.

SOLUTION 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) {
```

```
    ret = true;
  }
  // END

  return ret;
}
```

Get it on GitLab (eval_deriv.hpp).

▲

(5-1.g) ⚀ (20 min.) Based on your version of `dipoleval()` write a C++ function

```
void plotPolyInterpolant(const std::string &filename);
```

that saves plots the derivative of the polynomial interpolant for the test data as specified in Subproblem (5-1.f) in the file `<filename>.png`. To that end evaluate the polynomial in 100 equidistant points in the interval $[0,3]$ and, based on the obtained value, use the `plot()` function of MATPLOTLIBCPP to draw the polynomials. Do not forget to add axis labels "t" and "$p'(t)$", and a title.

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

```
void plotPolyInterpolant(const std::string &filename) {
  plt::figure();
  // TODO (5-1.g): Plot the derivative of the polynomial that
  // interpolates the data from (5-1.f).
  // Hint: matplotlibcpp (here = plt) implements Python's
  // matplotlib functions such as figure(), plot(), xlabel(), title(),...
  // You can use main.cpp of the LinearDataFit problem as a reference.

  // START
  Eigen::VectorXd dPx;
  const int n = 10;  // polynomial degree
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(100, 0, 3);
  Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(n + 1, 0, 3);
  Eigen::VectorXd y = sin(t.array()).matrix();

  dPx = dipoleval(t, y, x);

  plt::plot(x, dPx);
  plt::title("Derivative of interpolating polynomial");
  plt::xlabel("t");
  plt::ylabel("p'(t)");
  // END

  plt::savefig("./cx_out/" + filename + ".png");
}
```

Get it on GitLab (eval_deriv.hpp).

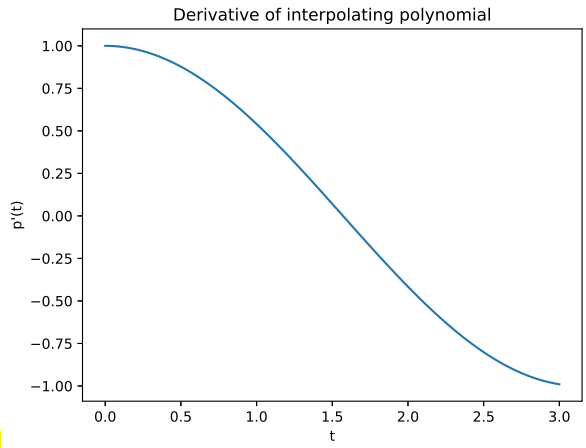


Fig. 31

▲

End Problem 5-1 , 135 min.

Problem 5-2: Lagrange Polynomials

This short problem studies a particular aspect of the **Lagrange polynomials** introduced in [Lecture → § 5.2.2.3].

This exercise just requires basic knowledge of [Lecture → Section 5.2.2]

As in [Lecture → Eq. (5.2.2.4)] we denote by L_i the i-th Lagrange polynomial for given nodes $t_j \in \mathbb{R}$, $j = 0,\ldots,n$, $t_i \neq t_j$, if $i \neq j$. Then the polynomial Lagrange interpolant p through the data points $(t_i,y_i)_{i=0}^n$ has the representation

$$p(x) = \sum_{i=0}^{n} y_i L_i(x) \quad \text{with} \quad L_i(x) := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-t_j}{t_i-t_j} \,. \tag{5.2.1}$$

(5-2.a) ⚀ (10 min.) Show that

$$\sum_{i=0}^{n} L_i(t) = 1 \quad \forall t \in \mathbb{R} \,. \tag{5.2.2}$$

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 → 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 $1 = p(x) = \sum_{i=0}^n L_i(x)$.

▲

(5-2.b) ⚁ (15 min.) Show that

$$\sum_{i=0}^{n} L_i(0) t_i^j = \begin{cases} 1 & \text{for } j = 0 \,, \\ 0 & \text{for } j = 1,\ldots,n \,. \end{cases} \tag{5.2.3}$$

(Here, read t_i^j as t_i raised to the power j.)

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 $x = 0$: $0 = \sum_{i=0}^n L_i(0) t_i^j$.

▲

(5-2.c) ⚀ (15 min.) Show that $p(x)$ can also be written as

$$p(x) = \omega(x) \sum_{i=0}^{n} \frac{y_i}{(x - t_i)\omega'(t_i)} \quad \text{with} \quad \omega(t) := \prod_{j=0}^{n} (t - t_j) \ . \tag{5.2.4}$$

Here ω' is the derivative of ω.

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:

$$\begin{aligned} \omega(x) \sum_{i=0}^{n} \frac{y_i}{(x - t_i)\omega'(t_i)} &= \sum_{i=0}^{n} \left[\frac{y_i}{(x - t_i) \prod_{\substack{j=0 \\ j \neq i}}^{n} (t_i - t_j)} \prod_{j=0}^{n} (x - t_j) \right] \\ &= \sum_{i=0}^{n} \left[\frac{y_i}{\prod_{\substack{j=0 \\ j \neq i}}^{n} (t_i - t_j)} \prod_{\substack{j=0 \\ j \neq i}}^{n} (x - t_j) \right] = \sum_{i=0}^{n} \left[y_i \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - t_j}{t_i - t_j} \right] = p(x) \ . \end{aligned}$$

▲

End Problem 5-2 , 40 min.

Problem 5-3: Generalized Lagrange polynomials for Hermite interpolation

[Lecture → Rem. 5.2.2.14] addresses a particular generalization of the Lagrange polynomial interpolation considered in [Lecture → Section 5.2.2]: **Hermite interpolation**, where beside the usual "nodal" interpolation conditions also the derivative of the interpolating polynomials is prescribed in the nodes, see [Lecture → Eq. (5.2.2.15)] (for $l_j = 1$ throughout). The associated generalized Lagrange polynomials were defined in [Lecture → Def. 5.2.2.17] and in this problem we aim to find concrete formulas for them in the spirit of [Lecture → Eq. (5.2.2.4)].

This is a purely theoretical problem.

Let $n+1$ different nodes t_i, $i = 0, \ldots, n$, $n \in \mathbb{N}$, be given. For numbers $y_j, c_j \in \mathbb{R}$, $j = 0, \ldots, n$, Hermite interpolation seeks a polynomial $p \in \mathcal{P}_{2n+1}$ satisfying $p(t_i) = y_i$ and $p'(t_i) = c_i$, $i = 0, \ldots, n$. Here p' designates the derivative of p.

The **generalized Lagrange polynomials** for Hermite interpolation $L_i, K_i \in \mathcal{P}_{2n+1}$ satisfy

$$L_i(t_j) = \delta_{ij}\,, \qquad L_i'(t_j) = 0\,, \tag{5.3.1a}$$

$$K_i(t_j) = 0\,, \qquad K_i'(t_j) = \delta_{ij}\,, \tag{5.3.1b}$$

for $i, j \in \{0, \ldots, n\}$.

(5-3.a) ⚀ (15 min.) Explicitly state the linear system of equations for the basis expansion coefficients of the Hermite interpolant with respect to the *monomial basis* $\{x \mapsto x^k,\, k = 0, \ldots, 2n+1\}$ of $\mathcal{P}_{2n+1}$.

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 $\mathbf{V}\mathbf{a} = \mathbf{b}$, 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 $p'(x) = \sum_{i=1}^{2n+1} i a_i x^{i-1}$. Using the interpolation condition $p'(t_i) = c_i$ for the derivative gives $n+1$ more rows of the linear system:

$$c_j = p'(t_j) = \sum_{i=1}^{2n+1} a_i i t_j^{i-1},\ j = 0, \ldots, n\,.$$

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

$$\mathbf{V} = \begin{bmatrix} 1 & t_0 & t_0^2 & \ddots & t_0^{2n+1} \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & t_n & t_n^2 & \ldots & t_n^{2n+1} \\ 0 & 1 & 2t_0 & \ldots & (2n+1)t_0^{2n} \\ \vdots & \vdots & \vdots & \vdots & \\ 0 & 1 & 2t_n & \ldots & (2n+1)t_n^{2n} \end{bmatrix} \in \mathbb{R}^{2n+2,2n+2}\,.$$

▲

(5-3.b) ⚀ (15 min.) [depends on Sub-problem (5-3.a)]

Give concrete formulas for K_0, K_1, L_0, L_1 for $n = 1$, $t_0 = -1$, $t_1 = 1$.

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 $[-1,1]$. ▲

(5-3.c) ⚀ (10 min.) Find the general formula describing all polynomials in t

1. of degree 2 that have a double zero in $t = a$, $a \in \mathbb{R}$,
2. of degree $n > 2$ that have a double zero in $t = a$, $a \in \mathbb{R}$.

A polynomial p is said to have a double zero in $t = a$, if $p(a) = 0$ *and* $p'(a) = 0$.

SOLUTION 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

$$p(t) = (t-a)^2 q(t) \quad \text{for some} \quad q \in \mathcal{P}_{n-2} .$$

▲

(5-3.d) ⚀ (15 min.) Let $g_1, \ldots, g_m$, $m \in \mathbb{N}$, be functions in $C^1(I)$ (space of continuously differentiable functions). Find a formula for the derivative of $h(t) = \prod_{k=1}^{m} g_k(t)$.

SOLUTION 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

$$h'(t) = \sum_{j=1}^{m} \Big\{ g_j'(t) \cdot \prod_{\substack{k=1 \\ k \neq j}}^{m} g_k(t) \Big\} . \tag{5.3.2}$$

▲

(5-3.e) ⚄ (30 min.) [depends on Sub-problem (5-3.c), Sub-problem (5-3.d)]

Find general formulas for the polynomials L_i, K_i as defined through (5.3.1). Of course, these formulas will involve the nodes t_i.

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,

$$K_i(t) = \frac{1}{h_i(t_i)}(t-t_i)h_i(t)\,.$$

▲

End Problem 5-3 , 85 min.

Problem 5-4: Piecewise linear interpolation

[Lecture → Ex. 5.1.0.15] introduced piecewise linear interpolation as a simple linear interpolation scheme. It finds an interpolant in the space spanned by the so-called tent functions, which are *cardinal basis functions*. Formulas are given in [Lecture → Eq. (5.1.0.17)].

Study [Lecture → Ex. 5.1.0.15] before tackling this problem. Problem involves coding in C++.

(5-4.a) ⚀ (20 min.)

Realize a C++ class

```
class LinearInterpolant {
public:
  LinearInterpolant(const Eigen::VectorXd &t, const
     Eigen::VectorXd &y);
  double operator(double x) const;
private:
  // Necessary data members here
};
```

representing the piecewise linear interpolant of the data points (t_i, y_i) passed to the constructor through two vectors `t` and `y` of equal length. In addition the class should provide the evaluation operator **double operator(double x)const**, which returns the value of the piecewise linear interpolant at $x \in \mathbb{R}$. Pay attention to efficient implementation.

Note that you must not assume that the nodes passed in `t` are sorted.

SOLUTION 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:

```
2  LinearInterpolant::LinearInterpolant(const Eigen::VectorXd &t,
3                                       const Eigen::VectorXd &y) {
4    assert(t.size() == y.size() && "Input sizes must match");
5    assert(t.size() > 1 && "Must specify at least two nodes");
6
7    // TODO: (6-4.a) Implement the constructor of the class.  t and y
        may not be
8    // sorted.  START
9
10   // helper lambda to transform two Eigen::VectorXd into one
11   // std::vector<std::pair<double, double»
12   auto VecToVecPair = [](const double &t_i, const double &y_i) -> pair {
13     return std::make_pair(t_i, y_i);
14   };
15
16   // perform transformation towards std::vector<std::pair<double,
        double»
17   i_points.resize(t.size());
18   std::transform(t.data(), t.data() + t.size(), y.data(), i_points.begin(),
19                  VecToVecPair);
20
21   // This is needed in std::sort
```

```
22    auto Ordering = [](const pair &P, const pair &Q) -> bool {
23      return P.first < Q.first;
24    };
25    // Sort the array
26    std::sort(i_points.begin(), i_points.end(), Ordering);
27    // END
28  }
```

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:

```
2   double LinearInterpolant::operator()(double x) {
3     double l_x = 0.;
4     // TODO: (6-4.a) Implement an efficient evaluation operator.
5     // START
6
7     // Lambda for comparison of pair type,
8     // we want to compare only the
9     // first element of the pair, i.e.  t_i
10    auto Compare = [](const pair &P, double V) -> bool { return P.first < V; };
11
12    // Find the place i (as iterator), wehre t_i >= x, notice that i_points
13    // *must* be sorted (this is needed in std::lower_bound)
14    // 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).

▲

(5-4.b) ⚂ (10 min.) Discuss the asymptotic effort involved in the constructor and the evaluation operator of the class **LinearInterpolant** as the number n of data points becomes large.

SOLUTION 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 $n \to \infty$.

▲

End Problem 5-4 , 30 min.

Problem 5-5: Cardinal basis for trigonometric interpolation

In [Lecture → Section 5.6] we learned about interpolation into the space $\mathcal{P}_{2n}^T$ or trigonometric polynomials from [Lecture → Def. 5.6.1.1]. In this task we investigate the cardinal basis functions for this interpolation operator and will also obtain a result about its stability.

Requires everything from [Lecture → Section 5.6] and involves coding in C++.

The (ordered) real trigonometric basis of $\mathcal{P}_{2n}^T$ is

$$\begin{aligned}\mathcal{B}_{\mathrm{Re}} := \Big\{ & C_0 := t \mapsto 1, S_1 := t \mapsto \sin(2\pi t), C_1 := t \mapsto \cos(2\pi t), S_2 := t \mapsto \sin(4\pi t), \\ & C_2 := t \mapsto \cos(4\pi t), \ldots, S_n := t \mapsto \sin(2n\pi t), C_n := t \mapsto \cos(2n\pi t) \Big\} .\end{aligned} \tag{5.5.1}$$

Another (ordered) basis of $\mathcal{P}_{2n}^T$ is

$$\mathcal{B}_{\exp} := \{ B_k := t \mapsto \exp(2\pi k \imath t) : k = -n, \ldots, n \} . \tag{5.5.2}$$

(Strictly speaking, we have to take the real part of these functions.) Both bases have $2n+1$ elements, which agrees with the dimension of $\mathcal{P}_{2n}^T$ [Lecture → Cor. 5.6.1.6].

(5-5.a) ⚄ (15 min.) Give the matrix $\mathbf{S} \in \mathbb{C}^{2n+1,2n+1}$ that effects the transformation of a coefficient representation with respect to $\mathcal{B}_{\mathrm{Re}}$ into a coefficient representation with respect to $\mathcal{B}_{\exp}$.

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**

$$\begin{aligned} S_k(t) &= \frac{1}{2\imath} B_{-k}(t) - \frac{1}{2\imath} B_k , \\ C_k(t) &= \frac{1}{2} B_{-k}(t) + \frac{1}{2} B_k . \end{aligned}$$

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

$$\mathbf{S} = \begin{bmatrix} 0 & 0 & 0 & \ldots & \frac{\imath}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \frac{\imath}{2} & \frac{1}{2} & \ldots & 0 & 0 \\ 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & -\frac{\imath}{2} & \frac{1}{2} & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & \ldots & -\frac{\imath}{2} & \frac{1}{2} \end{bmatrix} \in \mathbb{C}^{2n+1,2n+1} .$$

▲

(5-5.b) ⚂ (10 min.) The shift operator $S_\tau : C^0(\mathbb{R}) \to C^0(\mathbb{R})$ acts on a function according to $S_\tau f(t) = f(t-\tau)$, for $\tau \in \mathbb{R}$. If $\mathbf{c} \in \mathbb{C}^{2n+1}$ is the coefficient vector of $p \in \mathcal{P}_{2n}^T$ with respect to $\mathcal{B}_{\exp}$, what is the coefficient vector $\tilde{\mathbf{c}} \in \mathbb{C}^{2n+1}$ of $S_\tau p$?

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 $\tilde{\mathbf{c}} = [c_k \exp(-2\pi\imath k\tau)]_{k=-n}^{n}$.

▲

Now we consider the set of interpolation nodes

$$\mathcal{T}_n := \left\{ t_j := \frac{j+1/2}{2n+1} : \ j = 0,\ldots,2n \right\} . \tag{5.5.3}$$

The **cardinal basis functions** $b_0,\ldots,b_{2n} \in \mathcal{P}_{2n}^T$ of $\mathcal{P}_{2n}^T$ with respect to $\mathcal{T}_n$ are defined by

$$b_j(t_k) = \delta_{kj} \quad \text{[Kronecker symbol]} , \quad j,k = 0,\ldots,2n \tag{5.5.4}$$

(5-5.c) ⚂ (30 min.) Use the function `trigpolyvalequid`, see [Lecture → Code 5.6.3.9] to plot the cardinal basis function (implement plot_basis in trigipcardfn.hpp) $t \mapsto b_0(t)$ in $[0,1]$ for $n = 5$. To that end evaluate $b_0(t)$ in 1000 equidistant points $\frac{k}{1000}$, $k = 0,\ldots,999$.

SOLUTION 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.

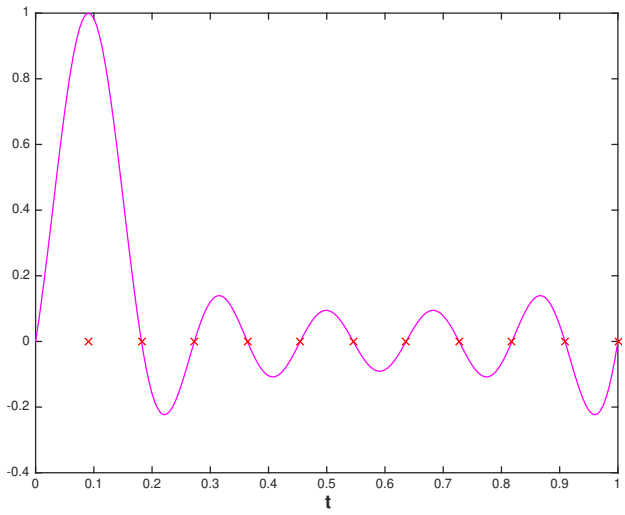


Fig. 32

C++11-code 5.5.5: Plotting of the cardinal basis function b_0.

```
2  void plot_basis(unsigned int n) {
3    // mesh size
4    constexpr unsigned int M = 1e3;
5    plt::figure();
6
7    // TODO: (5-5.c) use the function trigpolyvalequid from
8    // trigipcardfn.hpp to plot the cardinal basis function b_0(t) in
       function
9    // of t for n = 5
10   // START
11   // basis vector e_1
12   Eigen::ArrayXd e = Eigen::ArrayXd::Zero(2 * n + 1);
13   e(0) = 1;
14   Eigen::VectorXd y;
15   trigpolyvalequid(e, M, y);
16
17   Eigen::ArrayXd t = Eigen::ArrayXd::LinSpaced(M, 0, 1);
18   Eigen::ArrayXd zer = Eigen::ArrayXd::Zero(t.size());
19
20   // Shift function right a bit
21   Eigen::ArrayXd y_shift(M);
22   const unsigned int h = M / (2 * n + 1);
23   y_shift << y.tail(h), y.head(M - h);
24
25   // plotting functions using matplotlib
26   plt::title("b_0(t)");
27   plt::xlabel("t");
28   plt::ylabel("y");
29   plt::plot(t, y_shift, "r", {{"label", "b_0(t)"}});
30   plt::plot(t, zer, "--");
31   plt::legend();
32   // END
33   plt::savefig("./cx_out/basis_function.png");
34 }
```

Get it on GitLab (trigipcardfn.hpp).

▲

(5-5.d) ⚀ Show that

$$b_{k+1} = \mathsf{S}_\delta b_k \quad \text{with} \quad \delta := \frac{1}{2n+1}\,. \tag{5.5.6}$$

HINT 1 for (5-5.d): Verify (5.5.4) for $\mathsf{S}_\delta b_k$ ⌟

SOLUTION of (5-5.d):

We have (use periodicity and equidistance of the points):

$$\mathsf{S}_\delta b_k(t_j) = b_k(t_j - \frac{1}{2n+1}) = b_k(\frac{(j-1)+1/2}{2n+1}) = b_k(t_{j-1}) = \delta_{k,j-1} = \delta_{k+1,j} = b_{k+1}(t_j).$$

The claim follows from uniqueness of interpolating trigonometric polynomials.

▲

(5-5.e) ⚁ (15 min.) Describe the entries of the "interpolation matrix", that is, the system matrix of [Lecture → Eq. (5.1.0.23)], for the trigonometric interpolation problem with node set $\mathcal{T}_n$ and with respect to the basis $\mathcal{B}_{\exp}$ of $\mathcal{P}_{2n}^T$.

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:

$$\begin{aligned}(\mathbf{A})_{j,\ell} = B_{\ell-n}(t_j) &= \exp(2\pi\imath\frac{(j+1/2)(\ell-n)}{2n+1}) = \exp(2\pi\imath\frac{j\ell + \frac{1}{2}\ell - jn - \frac{1}{2}n}{2n+1}) \\ &= \underbrace{\exp(2\pi\imath\frac{j\ell}{2n+1})}_{=(\overline{\mathbf{F}}_{2n+1})_{j,\ell}} \exp(\pi\imath\frac{\ell}{2n+1}) \exp(-2\pi\imath\frac{jn}{2n+1}) \exp(-\pi\imath\frac{1}{2n+1})\,,\end{aligned}$$

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:

$$\mathbf{A} = \exp(-\pi\imath\frac{n}{2n+1})\,\mathbf{D}_1\overline{\mathbf{F}}_{2n+1}\mathbf{D}_2\,, \tag{5.5.7}$$

$$\mathbf{D}_1 := \operatorname{diag}\left(\left[e^{-2\pi\imath\frac{jn}{2n+1}}\right]_{j=0,\ldots,2n}\right) \quad , \quad \mathbf{D}_2 := \operatorname{diag}\left(\left[e^{\pi\imath\frac{\ell}{2n+1}}\right]_{\ell=0,\ldots,2n}\right) .$$

▲

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

What is the inverse of the interpolation matrix found in the previous problem?

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

$$\begin{aligned}\mathbf{A}^{-1} &= e^{\pi\imath\frac{n}{2n+1}}\,\mathbf{D}_2^{-1}\left(\overline{\mathbf{F}}_{2n+1}\right)^{-1}\mathbf{D}_1^{-1} \\ &= e^{\pi\imath\frac{n}{2n+1}}\operatorname{diag}\left(\left[e^{-\pi\imath\frac{\ell}{2n+1}}\right]_{\ell=0,\ldots,2n}\right)\frac{1}{2n+1}\mathbf{F}_{2n+1}\operatorname{diag}\left(\left[e^{2\pi\imath\frac{jn}{2n+1}}\right]_{j=0,\ldots,2n}\right) , \end{aligned} \tag{5.5.8}$$

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

$$(\mathbf{A})_{j,\ell} = \exp(2\pi\imath\frac{(j+1/2)(\ell-n)}{2n+1}) .$$

▲

(5-5.g) ⚄ (45 min.) Give a closed-form expression for the cardinal basis functions b_k defined in (5.5.4).

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:

$$(n+\tfrac{1}{2})b_k(t) = \frac{\sin(2\pi(n+\frac{1}{2})t)}{2\sin(\pi(t-t_k))}, \quad t \in \mathbb{R}. \tag{5.5.10}$$

▲

(5-5.h) ⚄ (30 min.) Write a function

```
double trigIpL(std::size_t n);
```

that approximately computes the **Lebesgue constant** $\lambda(n)$, see [Lecture → § 5.2.4.8], for trigonometric interpolation based on the node set $\mathcal{T}_n$. The Lebesgue constant is available through the formula

$$\lambda(n) = \sup_{\mathbf{y}\in\mathbb{C}^{2n+1}\setminus\{\mathbf{0}\}} \frac{\|\mathrm{I}_n\mathbf{y}\|_{L^\infty([0,1])}}{\|\mathbf{y}\|_\infty} = \sup_{t\in[0,1]} \sum_{k=0}^{2n} |b_k(t)| \,, \tag{5.5.11}$$

where $\mathrm{I}_n : \mathbb{C}^{2n+1} \to C^0(\mathbb{R})$ is the trigonometric interpolation operator.

Use sampling in 10^4 equidistant points to obtain a good guess for the supremum norm of a function on $[0,1]$.

Note. Efficiency of the implementation need not be a concern in this numerical experiment.

SOLUTION of (5-5.h):

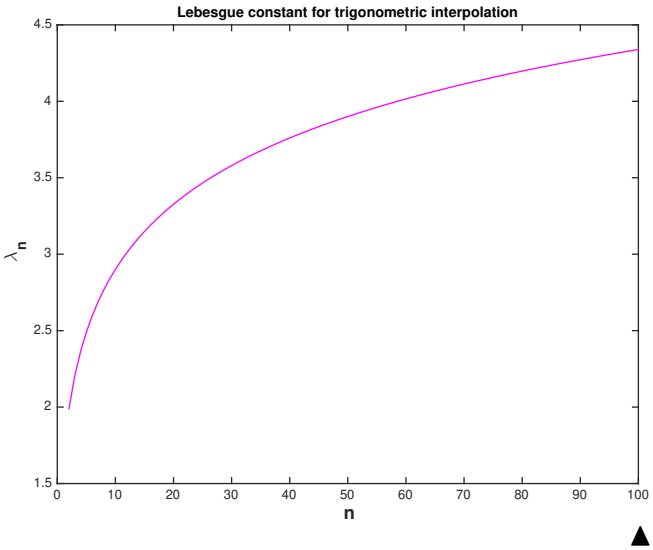
The following code is not optimal. A better, but more complicated implementation could rely on FFT.

C++11-code 5.5.12: Computation of Lebesgue constant λ for trigonometric interpolation.

```
2  double trigIpL(std::size_t n) {
3    double lambda = 0.;
4    // TODO: (5-5.h) write a function that approximately computes the
5    // Lebesgue constant λ(n)
6    // START
7    // Nodes where to evaluate
8    Eigen::ArrayXd t = Eigen::ArrayXd::LinSpaced(1e4, 0, 1);
9    // Will contain value of funtion at all nodes
10   Eigen::ArrayXd s = Eigen::ArrayXd::Zero(t.size());
11   // Sum over all basis functions
12   for (unsigned int k = 0; k <= 2 * n; ++k) {
13     // Compute the b_k using formula (6.5.8)
14     Eigen::ArrayXd b = Eigen::ArrayXd::Zero(t.size());
15     const double tk = (k + 0.5) / (2 * n + 1.);
16     b = sin(M_PI * (2 * n + 1) * (t - tk)) / sin(M_PI * (t - tk));
17
18     s += b.cwiseAbs();
19   }
20   // Find max and rescale
21   lambda = s.maxCoeff() / (2 * n + 1);
22   // END
23   return lambda;
24 }
```

Get it on GitLab (trigipcardfn.hpp).

Graph of $n \mapsto \lambda(n)$



▲

(5-5.i) ⚂ (5 min.) The following table gives the values $\lambda(n)$ for $n = 2^k$, $k = 2, 3, \ldots, 8$.

2^k	$\lambda(2^k)$
4	2.7
8	3.07
16	3.46
32	3.89
64	4.32
128	4.75
256	5.18
512	5.62

From these numbers predict the asymptotic behavior of $\lambda(n)$ for $n \to \infty$. Is it $O(\log n)$, $O(n)$, $O(n \log n)$, or $O(n^2)$?

SOLUTION 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.

▲

End Problem 5-5 , 165 min.

Problem 5-6: Quadratic Splines

[Lecture → Def. 5.4.1.1] introduces spline spaces $\mathcal{S}_{d,\mathcal{M}}$ of any degree $d \in \mathbb{N}_0$ on a node set $\mathcal{M} \subset \mathbb{R}$. [Lecture → Section 5.4.2] discusses interpolation by means of cubic splines, which is the most important case. In this problem we practise spline interpolation for quadratic splines in order to understand the general principles. We will see a situation, where knot sets and node sets have to be different.

Familiarity with [Lecture → Section 5.4.2] is required. This exercise involves substantial implementation in C++ and EIGEN.

Consider a 1-periodic function $f : \mathbb{R} \to \mathbb{R}$, that is, $f(t+1) = f(t)$ for all $t \in \mathbb{R}$, and a set of knots

$$\mathcal{M} := \{0 = t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = 1\} \subset [0,1] .$$

We want to approximate f using a *1-periodic* quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$, which interpolates f *in the midpoints* of the intervals $[t_{j-1}, t_j]$, $j = 0, \ldots, n$:

$$s\left(\tfrac{1}{2}(t_{j-1}+t_j)\right) = y_j := f\left(\tfrac{1}{2}(t_{j-1}+t_j)\right) , \quad j = 1, \ldots, n . \tag{5.6.1}$$

In analogy to the local representation of a cubic spline function according to [Lecture → Eq. (5.4.2.6)], we locally in each knot interval parameterize a quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$ according to

$$s|_{[t_{j-1},t_j]}(t) = d_j\, \tau^2 + c_j\, 4\, \tau(1-\tau) + d_{j-1}\, (1-\tau)^2 , \quad \tau := \frac{t - t_{j-1}}{t_j - t_{j-1}} , \quad j = 1, \ldots, n , \tag{5.6.2}$$

with $c_j, d_k \in \mathbb{R}$, $j = 1, \ldots, n$, $k = 0, \ldots, n$.

Remark. In this problem we see a case of spline interpolation for which the **knots** t_j, $j = 0, \ldots, n$, do not coincide with the interpolation **nodes**, which are the first coordinates of the data points to be interpolated.

(5-6.a) ⚀ (5 min.) How are the coefficients d_j, c_j, and d_{j-1} in (5.6.2) linked to the values of $s|_{[t_{j-1},t_j]}$ in the points $t = t_{j-1}, t_j, \frac{1}{2}(t_{j-1}+t_j)$?

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.

▲

(5-6.b) ⚁ (10 min.) What is the dimension of the subspace of *1-periodic* spline functions in $\mathcal{S}_{2,\mathcal{M}}$?

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:

$$\dim\{s \in \mathcal{S}_{2,\mathcal{M}} : s(0) = s(1),\ s'(0) = s'(1)\} = n \ .$$

▲

(5-6.c) ⚀ (10 min.) [depends on Sub-problem (5-6.a)]

What kind of continuity is already guaranteed by the mere use of the representation (5.6.2)?

SOLUTION 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.

▲

(5-6.d) ⚁ (45 min.) Derive a $n \times n$ linear system of equations (system matrix and right hand side) whose solution provides the coefficients c_j, $j = 1,\ldots,n$, in the representation (5.6.2) based on the function values $y_j := f(\frac{1}{2}(t_{j-1}+t_j))$, $j = 1,\ldots,n$.

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:

$$\begin{bmatrix} B_1 & C_2 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & A_0 \\ A_1 & B_2 & C_3 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_3 & C_4 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & A_{n-2} & B_{n-1} & C_n \\ C_{n+1} & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & A_{n-1} & B_n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}. \tag{5.6.8}$$

▲

(5-6.e) ⚂ (15 min.) [depends on Sub-problem (5-6.d)]

Show that the system matrix found in Sub-problem (5-6.d) can be written as a rank-1 modification of a tridiagonal matrix.

SOLUTION of (5-6.e):

Denote by $\mathbf{M}$ the system matrix from (5.6.8). Then

$$\mathbf{M} = \begin{bmatrix} B_1 - 1 & C_2 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ A_1 & B_2 & C_3 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_3 & C_4 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & A_{n-2} & B_{n-1} & C_n \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & A_{n-1} & B_n - A_0 C_{n+1} \end{bmatrix} + \mathbf{u}\mathbf{v}^\top ,$$

$$\text{with} \quad \mathbf{u} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ C_{n+1} \end{bmatrix} \in \mathbb{R}^n \quad , \quad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ A_0 \end{bmatrix} \in \mathbb{R}^n .$$

▲

(5-6.f) ⚀ (15 min.) [depends on Sub-problem (5-6.d) and Sub-problem (5-6.e)]

Outline an efficient algorithm for the solution of the linear system of equations found in Sub-problem (5-6.d). What is the asymptotic complexity of the algorithm in terms of the number n of data values for $n \to \infty$

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 $\mathbf{M} = \mathbf{Q}\mathbf{R}$ 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, $\mathbf{M} = \mathbf{L}\mathbf{U}$ 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.

▲

(5-6.g) ⚄ (25 min.) [depends on Sub-problem (5-6.d)]

Implement a C++ function

```
Eigen::VectorXd compute_c(const Eigen::VectorXd &t,
                          const Eigen::VectorXd &y);
```

that returns the coefficient vector $[c_1,\ldots,c_n]^\top \in \mathbb{R}^n$ when supplied a sorted knot vector `t` (of length $n-1$, because $t_0 = 0$ and $t_n = 1$ will be taken for granted), an n-vector `y` of data values y_j, $j = 1,\ldots,n$, that specify the function values of the quadratic spline s at the midpoints of the knot intervals.

You can immediately use a suitable sparse direct solver provided by EIGEN, which is "smart" enough to solve the linear system (5.6.8) with optimal computational effort. You need not employ the techniques of [Lecture → § 2.6.0.12].

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()`

```
2  std::pair<VectorXd, VectorXd> increments(const VectorXd &t) {
3    VectorXd dt, ddt;
4    // TO DO (5-6.g) : compute the increments t_j - t_{j-1}
5    // for j = 0,...,n+1 and t_{j+1} - t_{j-1} for j = 0,...,n
6    // Assume that t is sorted and that t does not include the endpoints.
7    // Hint: use periodic conditions to define t_{-1}, t_{n+1} for vectorization.
8
9    // START
10   // Create extended vector using the periodicity:
11   const unsigned int n = t.size() + 1;  // number of intervals
12   VectorXd ext_t(n + 3);
13   ext_t << t(n - 2) - 1, 0, t, 1, 1 + t(0);
14   // Increments in t
15   dt = ext_t.tail(n + 2).array() - ext_t.head(n + 2).array();
16   ddt = ext_t.tail(n + 1).array() - ext_t.head(n + 1).array();
17   // END
18
19   return std::make_pair(dt, ddt);
20 }
```

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()`

```
2  VectorXd compute_c(const VectorXd &t, const VectorXd &y) {
3    const unsigned int n = y.size();  // number of intervals
4    VectorXd c(n);
5    // TO DO (5-6.g) : Build the (sparse) matrix for spline interpolation
6    // at midpoints. Then compute the coefficients c for the data y.
7
8    // START
9    assert(t.size() == n - 1 && "number of intervals mismatch");
10   assert(t.minCoeff() > 0 && t.maxCoeff() < 1 && "mesh nodes out of range");
```

```
11
12    std::pair<VectorXd, VectorXd> p = increments(t);
13    VectorXd dt = p.first;
14    VectorXd ddt = p.second;
15
16    // Note: A_0 = A_n and C_1 = C_{n+1}
17    VectorXd A = dt.tail(n).cwiseQuotient(2 * ddt.tail(n));
18    VectorXd C = dt.head(n).cwiseQuotient(2 * ddt.head(n));
19
20    std::vector<Triplet<double>> triplets(3 * n);
21
22    // Build matrix as triplets
23    for (unsigned int j = 0; j < n - 1; ++j) {
24      triplets.push_back(Triplet<double>(j, j, A(j) + C(j) + 1));
25      triplets.push_back(Triplet<double>(j, j + 1, C(j + 1)));
26      triplets.push_back(Triplet<double>(j + 1, j, A(j)));
27    }
28    triplets.push_back(Triplet<double>(n - 1, n - 1, A(n - 1) + C(n - 1) + 1));
29    triplets.push_back(Triplet<double>(n - 1, 0, C(0)));
30    triplets.push_back(Triplet<double>(0, n - 1, A(n - 1)));
31
32    SparseMatrix<double> M(n, n);
33    M.setFromTriplets(triplets.begin(), triplets.end());
34    M.makeCompressed();
35    // solve linear system
36    SparseLU<SparseMatrix<double>> solver;
37    solver.compute(M);
38    c = solver.solve(y);
39    // END
40    return c;
41  }
```

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].

▲

(5-6.h) ⚄ (20 min.) [depends on Sub-problem (5-6.d)]

Realize a C++ function

```
Eigen::VectorXd compute_d ( const Eigen::VectorXd &c, const
   Eigen::VectorXd &t);
```

that determines the coefficients d_j, $j = 0, \ldots, n$, in the local representation (5.6.2) assuming that the values c_j, $j = 1, \ldots, n$, have already been computed. The argument `t` is to pass the variable knot positions t_j, $j = 1, \ldots, n-1$.

SOLUTION of (5-6.h):

We can simply rely on the formula (5.6.4).

C++ code 5.6.11: Implementation of `compute_d()`

```
2  VectorXd compute_d(const VectorXd &c, const VectorXd &t) {
3    const unsigned int n = c.size();  // number of intervals
4    VectorXd d_ext(n + 1);
5    // TO DO (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<VectorXd, VectorXd> p = increments(t);
9    VectorXd dt = p.first;
10   VectorXd ddt = p.second;
11
12   // extend c periodically for vectorization
13   VectorXd c_ext(c.size() + 1);
14   c_ext << c, c(0);
15
16   // coefficients d_j for j = 1...n
17   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).

▲

(5-6.i) ⚂ (25 min.) [depends on Sub-problem (5-6.e), Sub-problem (5-6.g), Sub-problem (5-6.h)]

Based on `compute_c()` from Sub-problem (5-6.g) implement an efficient C++ function

```
Eigen::VectorXd quadspline(const Eigen::VectorXd &t,
                           const Eigen::VectorXd &y,
                           const Eigen::VectorXd &x);
```

which takes as input a (sorted) knot vector `t` (of length $n-1$, because $t_0 = 0$ and $t_n = 1$ will be taken for granted), an n-vector `y` containing the values $y_j \in \mathbb{R}$ of a function f at the midpoints $\frac{1}{2}(t_{j-1} + t_j)$, $j = 1, \ldots, n$, and a *sorted* N-vector `x` of evaluation points in $[0, 1]$.

The function is to return the values of the interpolating quadratic spline s at the positions `x`.

SOLUTION 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()`

```
2  VectorXd quadspline(const VectorXd &t, const VectorXd &y, const VectorXd &x) {
3    VectorXd fval(x.size());
4
5    // TO DO (5-6.i): evaluate the spline at the points defined in x.
6    // Assume that x is sorted.
```

```
  assert(x.minCoeff() >= 0 && x.maxCoeff() <= 1 &&
         "evaluation samples out of range");

  // START
  VectorXd c = compute_c(t, y);
  const unsigned int n = y.size();  // number of intervals

  std::pair<VectorXd, VectorXd> p = increments(t);
  VectorXd dt = p.first;
  VectorXd ddt = p.second;

  VectorXd d_ext = compute_d(c, t);
  // extend t for vectorization
  VectorXd t_ext(n + 1);
  t_ext << 0, t, 1;

  unsigned int i = 0;
  // loop over intervals
  for (unsigned int j = 1; j <= n; ++j) {
    // loop over sample points in the interval
    while (i < x.size() && x(i) <= t_ext(j)) {
      double tau = (x(i) - t_ext(j - 1)) / dt(j);
      double uat = 1 - tau;
      fval(i) = d_ext(j) * tau * tau + 4 * c(j - 1) * tau * uat +
                d_ext(j - 1) * uat * uat;
      ++i;
    }
    if (i == x.size()) {
      break;
    }
  }
  // END
  return fval;
}
```

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 $n \to \infty$.

▲

(5-6.j) ⚂ Based on MATPLOTLIBCPP write a C++ function

```
void plotquadspline(const std::string &filename);
```

that saves a plot (in `<filename>.png`) of the interpolating 1-periodic quadratic spline s for $f(t) := \exp(\sin(2\pi t))$, $n = 10$, and $\mathcal{M} = \left\{\frac{j}{n}\right\}_{j=0}^{10}$, that is, the spline s is to fulfill $s(\frac{1}{2}(t_j + t_{j-1})) = f(\frac{1}{2}(t_j + t_{j-1}))$, $j = 1, \ldots, 10$. To that end evaluate the spline in 200 equidistant points in the interval $[0,1]$ and, based on the obtained value, use the `plot()` function of MATPLOTLIBCPP to draw it. Do not forget to add axis labels "t" and "$s(t)$", and a suitable title.

SOLUTION of (5-6.j):

C++ code 5.6.13: Implementation of `plotquadspline()`

```
2  void plotquadspline(const std::string &filename) {
3    VectorXd mesh = VectorXd::LinSpaced(9, .1, .9);
4    auto f = [](VectorXd t) { return (2 * M_PI * t).array().sin().exp(); };
5
6    plt::figure();
7    // TO DO (5-6.j) : plot the quadratic spline for the function f based on
8    // the intervals defined in t. Plot also the data (interpolation) points.
9
10   // START
11   // The interpolation points are the midpoints of the intervals
12   VectorXd t = VectorXd::LinSpaced(10, .05, .95);
13   VectorXd y = f(t);
14   // plot data points
15   plt::plot(t, y, "o", {{"label", "data"}});
16
17   // std::cout << y <<std::endl;
18   VectorXd x = VectorXd::LinSpaced(200, 0, 1);
19   VectorXd spline_val = quadspline(mesh, y, x);
20   // std::cout << spline_val <<std::endl;
21
22   plt::plot(x, spline_val, {{"label", "spline"}});
23   plt::legend();
24   plt::title("Quadratic spline");
25   plt::xlabel("t");
26   plt::ylabel("s(t)");
27   // END
28   plt::savefig("./cx_out/" + filename + ".png");
29 }
```

Get it on GitLab (quadsplines.hpp).

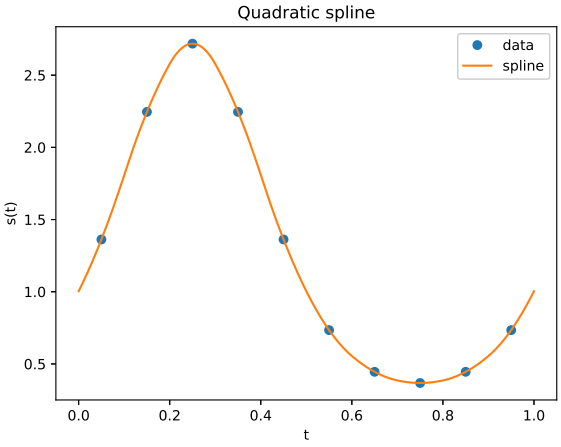


Fig. 34

◁ plot of the quadratic spline s.

▲

(5-6.k) ⚄ (30 min.) [depends on Sub-problem (5-6.i)]

Write a C++ function

```
std::vector<double> qsp_error(unsigned int q);
```

that approximately computes the $L^\infty([0,1])$-norm of the error of the approximation of $f(t) := \exp(\sin(2\pi t))$ by its 1-periodic quadratic spline interpolant s_n based on the equidistant knot

set $\mathcal{M}_n = \{j/n\}_{j=0}^n$:

$$E_n := \|f - s_n\|_{L^\infty([0,1])} := \sup_{x\in[0,1]} |f(x) - s_n(x)| \,.$$

To approximate the $L^\infty([0,1])$-norm you can evaluate the difference $|f(x) - s_n(x)|$ at $N \gg n$ equidistant points in $[0,1]$ and then take the maximum; choose $N = 10.000$.

The function is supposed to return the error norms E_n for $n = 2, 4, 8, \ldots, 2^q$, where q is passed in the argument `q`. Describe quantitatively and qualitatively the convergence of the error norms in terms of $n \to \infty$.

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

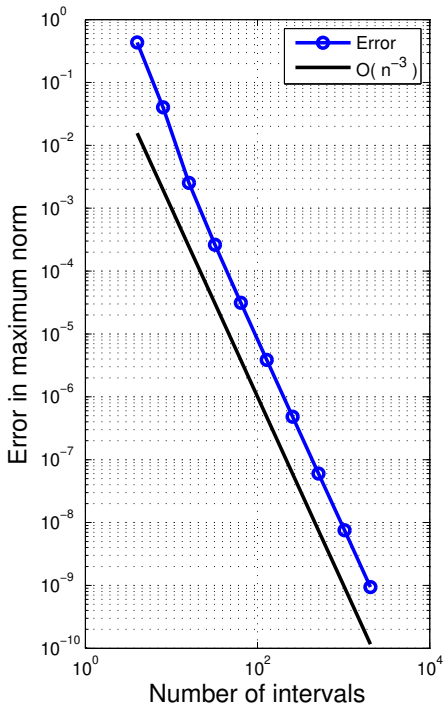
◁ Tabulated spline approximation errors in the maximum norm. For larger values of n we get $E_{2^j} : E_{2^{j+1}} \approx 8$, which indicates and asymptotic behavior of the error according to the law

$$E_n = O(n^{-3}) \,.$$

C++ code 5.6.14: Implementation of `qsq_error()`

```
2  std::vector<double> qsp_error(unsigned int q) {
3    std::vector<double> Err;
4    constexpr unsigned int N = 10000;
5    VectorXd x = VectorXd::LinSpaced(N, 0, 1);
6    auto f = [](VectorXd t) { return (2 * M_PI * t).array().sin().exp(); };
7    // TO DO (5-6.k) : compute L^infty errors for all n = 2, 4, ..., 2^q
8    // START
9    const unsigned int n_max = std::pow(2, q);
10   VectorXd f_exact = f(x);
11
12   for (unsigned int n = 2; n <= n_max; n <<= 1) {
13     VectorXd t = VectorXd::LinSpaced(n - 1, 1.0 / n, 1.0 - 1.0 / n);
14     // The interpolation points are the midpoints of the intervals
15     VectorXd interp = VectorXd::LinSpaced(n, .5 / n, 1.0 - .5 / n);
16     VectorXd y = f(interp);
17     VectorXd f_spline = quadspline(t, y, x);
18     // compute max error
19     Err.push_back((f_spline - f_exact).cwiseAbs().maxCoeff());
20   }
21   // END
22   return Err;
23 }
```

Get it on GitLab (quadsplines.hpp).



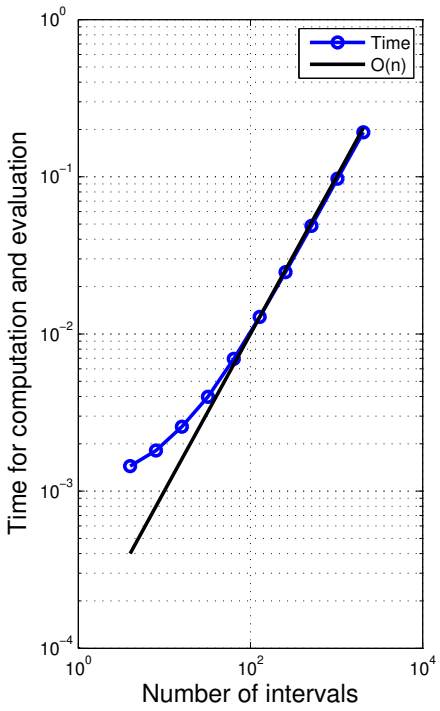


Fig. 35

◁ plot of errors E_n and runtimes as a function of the number n of knots.

In this *doubly logarithmic* plot the data points (n, E_n) are neatly aligned. This indicates asymptotic algebraic convergence $O(n^{-3})$, the rate can be obtained from the slopes of the lines.

▲

End Problem 5-6 , 200 min.

Problem 5-7: Linear monotonicity-preserving C^1-interpolation scheme

In Sections [Lecture → Section 5.3.2], [Lecture → Section 5.3.3.2], [Lecture → Section 5.3], different shape preserving operator are introduced. In the first case (piecewise linear functions) the interpolating function is continuous but not C^1, in the two remaining cases (Hermite polynomials and splines) the interpolating operator is not linear. In all these examples the monotonicity is preserved, i.e., monotonic data yield monotonic interpolants. This problem shows that monotonicity preserving interpolators that are both C^1 and linear present an unpleasant and unexpected feature.

A purely theoretical problem. As preparation study [Lecture → § 5.1.0.21] and [Lecture → Def. 5.1.0.25] together with [Lecture → Section 5.3.3.2].

The goal of this problem is to prove the following theorem.

Theorem 5.7.1. Linear monotonicity preserving interpolation operator

Given the node set $\mathcal{T} = \{t_0, \ldots, t_n\} \subset [t_0, t_n]$, with $t_0 < t_1 < \ldots < t_n$, consider an interpolation operator $I_\mathcal{T} : \mathbb{R}^{n+1} \to C^0([t_0, t_n])$, for which, by definition, we have $\big(I_\mathcal{T}(\mathbf{y})\big)(t_j) = y_j$ for every $j = 0, \ldots, n$ and every vector of data values $\mathbf{y} = [y_0, \ldots, y_n]^\top \in \mathbb{R}^{n+1}$. Assume that the operator satisfies

i) smoothness: $I_\mathcal{T}(\mathbf{y})$ belongs to $C^1([t_0, t_n])$ for any $\mathbf{y} \in \mathbb{R}^{n+1}$,

ii) linearity: $I_\mathcal{T}(\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha I_\mathcal{T}(\mathbf{y}) + \beta I_\mathcal{T}(\mathbf{z})$, for every $\alpha, \beta \in \mathbb{R}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^{n+1}$,

iii) (positive) monotonicity preserving: if $y_j \le y_{j+1}$, $\forall\, j = 0, \ldots, n-1$, then $I_\mathcal{T}(\mathbf{y})$ is a monotonic non-decreasing function.

Then, for any $\mathbf{y} \in \mathbb{R}^{n+1}$, the derivative of $I_\mathcal{T}(\mathbf{y})$ in the interior nodes vanishes:

$$\big(I_\mathcal{T}(\mathbf{y})\big)'(t_j) = 0\,, \qquad \forall j \in \{1, \ldots, n-1\}\,, \quad \forall \mathbf{y} \in \mathbb{R}^{n+1}\,. \tag{5.7.2}$$

(5-7.a) ⚀ (15 min.) Show that the assertion (5.7.2) holds for the special data vectors

$$\mathbf{y} = \mathbf{y}^{(j)} := [\underbrace{0, \ldots, 0}_{j \text{ times}}, 1, \ldots, 1]^\top\,, \quad j = 0, \ldots, n\,.$$

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 $\big(\mathbf{y}^{(j)}\big)_1 = \cdots = \big(\mathbf{y}^{(j)}\big)_j = 0$ and $\big(\mathbf{y}^{(j)}\big)_{j+1} = \cdots = \big(\mathbf{y}^{(j)}\big)_{n=1} = 1$. 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 $\big(I_\mathcal{T}(\mathbf{y}^{(j)})\big)'(t_k) = 0$.

The derivative of $I_\mathcal{T}(\mathbf{y}^{(1)})$ is not necessarily zero in t_0 as well as the derivative of $I_\mathcal{T}(\mathbf{y}^{(n)})$ in t_n.

▲

(5-7.b) ⚄ (10 min.) Prove that $\operatorname{span}\{\mathbf{y}^{(j)},\ j=0,\ldots,n\} = \mathbb{R}^{n+1}$.

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:

$$\mathbf{e}^{(j)} := (\underbrace{0,\ldots,0}_{j},1,0,\ldots,0)^T = \mathbf{y}^{(j)} - \mathbf{y}^{(j+1)}, \quad \forall\, j=0,\ldots,n-1, \quad \mathbf{e}^{(n)} := (\underbrace{0,\ldots,0}_{n},1)^T = \mathbf{y}^{(n)}\,,$$

$$\Rightarrow \quad \mathbb{R}^{n+1} = \operatorname{span}\{\mathbf{e}^{(j)},\ j=0,\ldots,n\} \subseteq \operatorname{span}\{\mathbf{y}^{(j)},\ j=0,\ldots,n\} \subseteq \mathbb{R}^{n+1}\,,$$

$$\Rightarrow \quad \operatorname{span}\{\mathbf{y}^{(j)},\ j=0,\ldots,n\} = \mathbb{R}^{n+1}\,.$$

▲

(5-7.c) ⚄ (15 min.) [depends on Sub-problem (5-7.a) and Sub-problem (5-7.b)]

Invoke the insight gained in Sub-problem (5-7.a) to complete the proof of the theorem.

SOLUTION 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

$$\big(I_{\mathcal{T}}(\mathbf{y})\big)'(t_k) \overset{\text{linearity of } I_{\mathcal{T}}}{=} \sum_{j=0}^{n} c_j \big(I_{\mathcal{T}}(\mathbf{y}^{(j)})\big)'(t_k) \overset{\text{Sub-problem (5-7.a)}}{=} 0\,, \quad k=1,\ldots,n-1\,.$$

▲

End Problem 5-7, 40 min.

Problem 5-8: Approximating π by extrapolation

In [Lecture → Section 5.2.3.3] we learned about the approximation of a limit $\lim_{h\to 0}\psi(h)$ based on the evaluation of the function ψ "far away" from 0. In this exercise we will practice the underlying technique of "Lagrangian polynomial extrapolation" for a geometric approximation problem.

Study [Lecture → Section 5.2.3.3] and, in particular [Lecture → Code 5.2.3.19], before your tackle this problem.

In this problem we encounter the situation that a quantity of interest is defined as a limit of a sequence

$$x^* = \lim_{n\to\infty} T(n)\ ,$$

where the function $T : \{n_{\min}, n_{\min}+1, \dots\} \mapsto \mathbb{R}$ may be given in procedural form as **double** `T(int n)` only. However, invoking `T(Inf)` will usually result in an error and for very large arguments n the implementation of `T()` may not yield reliable results, *cf.* [Lecture → Ex. 1.5.4.7] and [Lecture → § 5.2.3.16].

The idea of **extrapolation** is, firstly, to compute a few values $T(n_0), T(n_1), \dots, T(n_k)$, $k \in \mathbb{N}$, and to consider them as the values $g(1/n_0), g(1/n_1), \dots, g(1/n_k)$ of a continuous function

$$g : (0, 1/n_{\min}] \mapsto \mathbb{R}, \quad \text{for which, obviously} \quad x^* = \lim_{h\to 0} g(h)\ .$$

Secondly, the function g is approximated by an interpolating polynomial $p_k \in \mathcal{P}_k$ with $p_k(n_j^{-1}) = T(n_j)$, $j = 0, \dots, k$. In many cases we can expect that $p_k(0)$ will provide a good approximation for x^*.

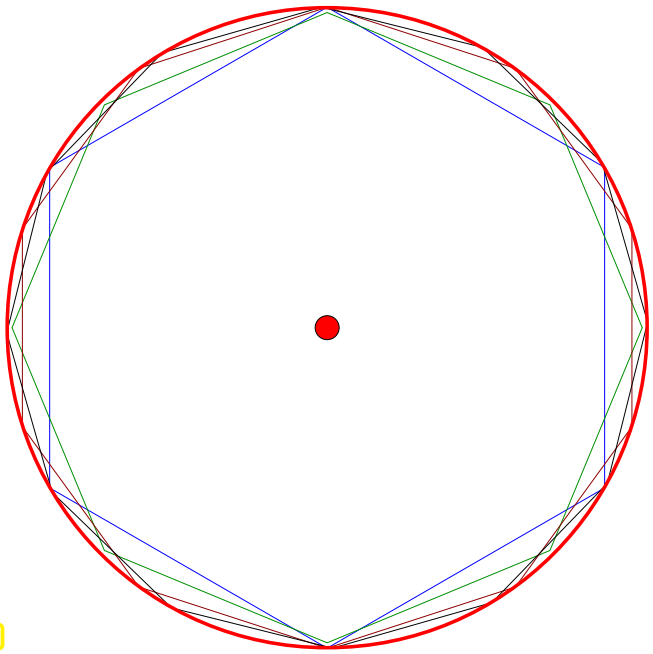


Fig. 36

The unit circle can be approximated by inscribed regular polygons with k edges. Thus, the length of half circumference (i.e., π) can be approximated by the half the length of the perimeters s_k of such polygons. These values $s_k/2$ can be calculated by elementary trigonometry and a closed form formula is

$$\tfrac{1}{2}s_k = k\sin(\pi/k)\ . \tag{5.8.1}$$

(5-8.a) ⚄ (10 min.) Show that for any $L \in \mathbb{N}$ and for $k \to \infty$

$$\tfrac{1}{2}s_k = \pi + \sum_{\ell=1}^{L} c_\ell k^{-2\ell} + O(k^{-2(L+1)}) \quad \text{for some} \quad c_\ell \in \mathbb{R} \quad \text{independent of} \quad L\ . \tag{5.8.2}$$

We say that s_k has an asymptotic expansion in k^{-2}.

Remark. The heuristics behind polynomial extrapolation in this example is that the existence of an asymptotic expansion (5.8.2) suggests that $\frac{1}{2}s_\infty$ can be well approximated by $p(0)$, where p is an interpolating polynomial in k^{-1} or k^{-2}.

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

$$\sum_{\ell=L+1}^{\infty} \frac{(-1)^\ell \pi^{2\ell+1} k^{-2\ell}}{(2\ell+1)!} \leq Ck^{-2(L+1)} \quad \text{for} \quad C > 0 \quad \text{independent of} \quad k.$$

▲

(5-8.b) ⚂ (45 min.) Implement a C++ function

```
double extrapolate_to_pi(const unsigned int k);
```

that uses the *Aitken-Neville scheme*, see [Lecture → Code 5.2.3.10], to approximate π by extrapolation from the data points $(j^{-1}, \frac{1}{2}s_j)$, for $j = 2, \ldots, k$. Use the values $\frac{1}{2}s_j$ as given in (5.8.1).

SOLUTION 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));
```

```
    }
  }
  // Now S(0) has contributions from all data points.
  pi = S(0);
  // END

  return pi;
}
```

Get it on GitLab (extrapolpi.hpp).

▲

(5-8.c) ⚀ (30 min.) Write a C++ function

```
void plotExtrapolationError(const unsigned int kmax);
```

that generates a *linear-logarithmic* plot of the extrapolation errors

$$\mathrm{err}(k) := |\pi - p_k(0)| \,, \quad k = 2, \ldots, k_{\max} \,,$$

versus k and also tabulates the errors. To create the plot use the functions of MATPLOTLIBCPP. Do not forget axis annotations and a meaningful title for the plot.

In `main.cpp`, `plotExtrapolationError(10)` is called for $k_{\max} = 10$.

SOLUTION 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);
```

```
30    plt::title("Approximation of pi");
31    plt::xlabel("Polynomial degree");
32    plt::ylabel("Error");
33    plt::grid();
34    // END
35
36    std::string path = "./cx_out/pi_error_" + std::to_string(kmax) + ".png";
37    plt::savefig(path);
38  }
```

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
```

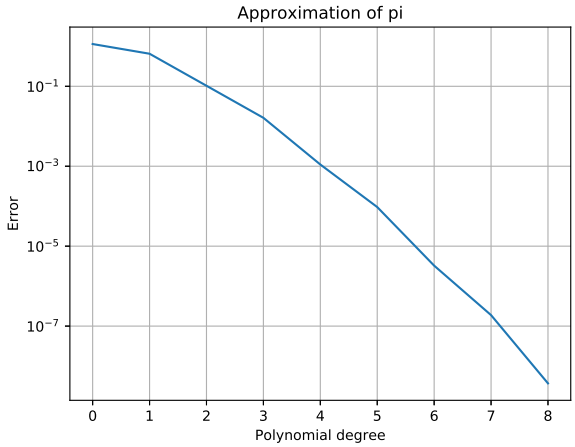


Fig. 37

Linear-logarithmic plot of error in the approximation of π by extrapolation.

▲

(5-8.d) ⚂ (15 min.) [depends on Sub-problem (5-8.c)]

Which kind of convergence of $\mathrm{err}(k) \to 0$ for $k \to \infty$ can you conclude from the data generated in Sub-problem (5-8.c)? The main types of asymptotic convergence to 0 are introduced in [Lecture → Def. 6.2.2.7].

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):

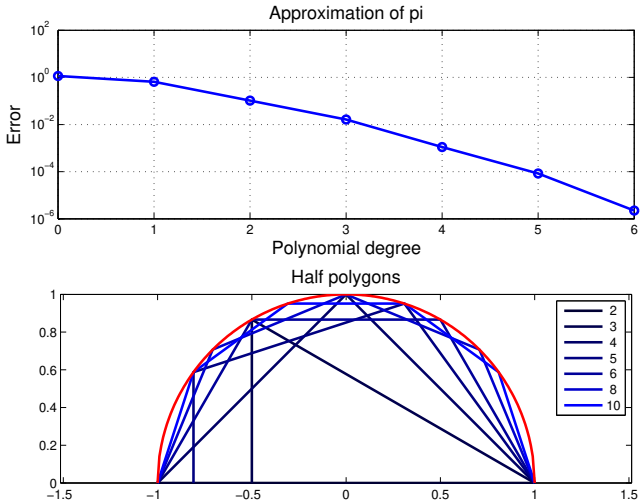


Fig. 38

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.

▲

(5-8.e) ⚄ (15 min.) In `main.cpp`, `plotExtrapolationError(30)` is called which plots and tabulates err_k for k up to 30. Describe and explain your observations of the plot found in `cx_out/pi_error_30.png`.

SOLUTION of (5-8.e):

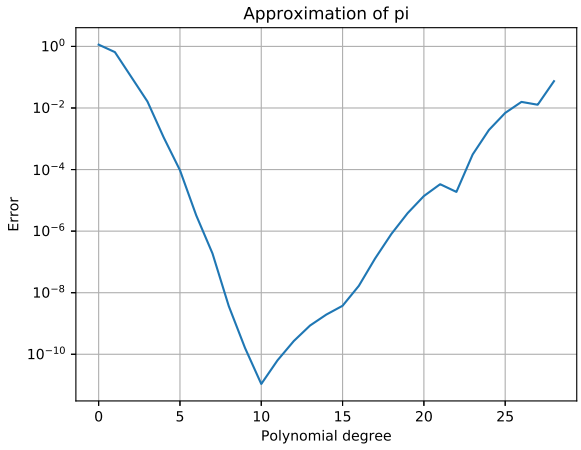


Fig. 39

For large k the error starts increasing again!
Two possible explanations:

1. High-degree polynomial interpolants display strong oscillations between the data points, which affect $p(0)$.
2. The Aitken-Neville algorithm is based on the computation of differences of similar quantities, therefore, ultimately, it will be vulnerable to cancellation errors.

Both effects may play a role here.

▲

End Problem 5-8 , 115 min.

Problem 5-9: Spaces of Piecewise Polynomial Functions

Piecewise polynomial functions, most prominently splines, are widely used for data interpolation and functions approximation. In this problem we examine some aspects of particular specimens.

A purely theoretical problem connected with [Lecture → Section 5.4].

For a mesh $\mathcal{M} := \{a = t_0 < t_1 < \ldots < t_{n-1} < t_n = b\}$ we consider the following three spaces of piecewise polynomials:

$$V_{-1} := \left\{ v : [a,b] \to \mathbb{R} : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}, \tag{5.9.1}$$

$$V_0 := \left\{ v \in C^0([a,b]) : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}, \tag{5.9.2}$$

$$V_1 := \left\{ v \in C^1([a,b]) : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}, \tag{5.9.3}$$

$$V_2 := \left\{ v \in C^2([a,b]) : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}. \tag{5.9.4}$$

Here, $\mathcal{P}_d$ stands for the space of polynomials $\mathbb{R} \to \mathbb{R}$ of degree $\leq d$.

(5-9.a) ⚄ (15 min.) Determine the dimensions of the four spaces defined above:

$$\dim V_{-1} = \boxed{},$$
$$\dim V_0 = \boxed{},$$
$$\dim V_1 = \boxed{},$$
$$\dim V_2 = \boxed{}.$$

SOLUTION of (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:
$$\blacktriangleright \quad \dim V_0 = \dim V_{-1} - (n-1) = 2n+1 .$$
- Another continuity condition for the first derivative is imposed in the case of the space V_1. This introduces another $n-1$ constraints.
$$\blacktriangleright \quad \dim V_1 = \dim V_0 - (n-1) = n+2 .$$
- Further $n-1$ continuity conditions have to take into account in the case of V_2:
$$\blacktriangleright \quad \dim V_2 = \dim V_1 - (n-1) = 3 = \dim \mathcal{P}_2 .$$
In fact, we have $V_2 = \mathcal{P}_2$!

▲

(5-9.b) ⚂ (10 min.) Which of the spaces V_k, $k = -1, 0, 1$, coincides with one of the spline spaces $\mathcal{S}_{d,\mathcal{M}}$ introduced in [Lecture → Def. 5.4.1.1] in class?

$$V_{\square} = \mathcal{S}_{\square,\mathcal{M}} .$$

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:

$$\blacktriangleright \quad V_1 = \mathcal{S}_{2,\mathcal{M}} .$$

▲

(5-9.c) ⚂ (15 min.) Which of the following conditions for a function $v \in V_1$ are *sufficient* and which are *necessary* for v being non-decreasing, that is, for

$$a \le x \le y \le b \quad \Rightarrow \quad v(x) \le v(y) \ ?$$

(i) $v(t_{j-1}) \le v(t_j)$ for all $j = 1, \ldots, n$

○ necessary ○ sufficient ○ neither

(ii) $v'(x) \ge 0$ for all $x \in [a, b]$

○ necessary ○ sufficient ○ neither

(iii) $v'(\frac{1}{2}(t_{j-1} + t_j)) \ge 0$ for all $j \in \{1, \ldots, n\}$

○ necessary ○ sufficient ○ neither

(iv) $v'(t_j) \ge 0$ for all $j \in \{0, \ldots, n\}$

○ necessary ○ sufficient ○ neither

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).

▲

In the sequel we consider the mesh (knot set) $\mathcal{M} = \{0,1,2\}$.

(5-9.d) ⚄ (10 min.) Determine the parameters $a, b \in \mathbb{R}$ in the following function definition

$$v(x) := \begin{cases} ax + b & \text{for } 0 \leq x < 1 \,, \\ 1 - x^2 & \text{for } 1 \leq x \leq 2 \,, \end{cases}$$

such that $v \in V_1$.

$$a = \boxed{} \quad , \quad b = \boxed{} \,.$$

SOLUTION 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 $b = 2$.

▲

(5-9.e) ⚄ (10 min.) Write down a $v \in V_1$ such that

$$v(0) = 0 \quad , \quad v(1) = 1 \quad , \quad v(2) = 0 \,.$$

$$v(x) = \begin{cases} \boxed{} & \text{for } 0 \leq x < 1 \,, \\ \boxed{} & \text{for } 1 \leq x \leq 2 \,. \end{cases}$$

SOLUTION 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.

▲

End Problem 5-9 , 60 min.

Problem 5-10: Adaptive Interpolation of Closed Curves

Often, for the sake of efficiency, closed curves (loops) need to be approximated by piecewise polynomials. This problem presents a simple adaptive algorithm for constructing such an approximation.

Related to [Lecture → Section 5.3.3]; the idea of adaptive approximation is borrowed from [Lecture → Section 7.6].

A *1-periodic* continuously differentiable function $\mathbf{c} : \mathbb{R} \to \mathbb{R}^2$ parameterizes a closed curve in the plane.

For instance, the "kite-shaped" curve displayed beside is described by

$$\mathbf{c}(t) = \begin{bmatrix} \cos(2\pi t) + \frac{2}{3}\cos(4\pi t) \\ \frac{3}{2}\sin(2\pi t) \end{bmatrix}. \tag{5.10.1}$$

▷

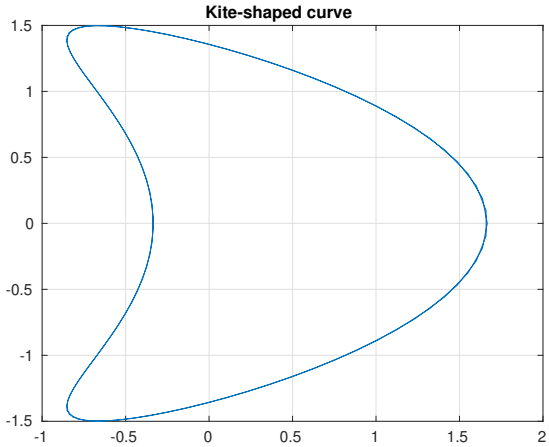


Fig. 40

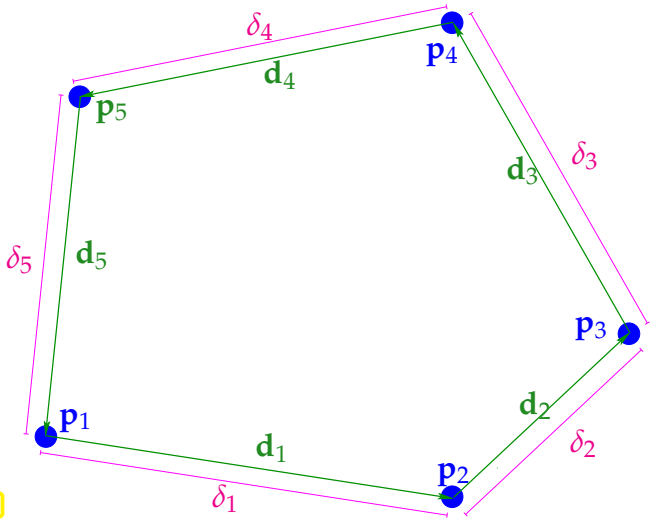


Fig. 41

Notations. Based on a sequence of $n \geq 2$ mutually distinct points $(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n)$ in the plane ($\mathbf{p}_k \in \mathbb{R}^2$) the following notations are defined:

- $\delta_i := \|\mathbf{p}_{i+1} - \mathbf{p}_i\|$, $i = 1, \ldots, n-1$,
  $\delta_n := \|\mathbf{p}_1 - \mathbf{p}_n\|$,
- $\lambda_0 := 0$,
  $\lambda_k := \sum_{j=1}^{k} \delta_j$, $k = 1, \ldots, n$,
- $\mathbf{d}_i := \mathbf{p}_{i+1} - \mathbf{p}_i$, $i = 1, \ldots, n-1$,
  $\mathbf{d}_n := \mathbf{p}_1 - \mathbf{p}_n$,
- $\mathbf{t}_i := \dfrac{\mathbf{d}_i}{\|\mathbf{d}_i\|}$, $i = 1, \ldots, n$.

We study two ways to connect points in the plane by a piecewise polynomial closed curve.

Definition 5.10.2. Closed polygonal interpolant

Given a sequence $\Sigma := (\mathbf{p}_1, \ldots, \mathbf{p}_n)$, $n \in \mathbb{N}$, of mutually distinct points $\in \mathbb{R}^2$, their **interpolating closed polygon** $\mathbf{\Pi}_\Sigma$ is a function $\mathbf{\Pi}_\Sigma : [0, \lambda_n] \to \mathbb{R}^2$ that fulfills:

(i) $\mathbf{\Pi}_\Sigma(\lambda_k) = \mathbf{p}_{k+1}$, $k = 1, \ldots, n-1$, and $\mathbf{\Pi}_\Sigma(0) = \mathbf{\Pi}_\Sigma(\lambda_n) = \mathbf{p}_1$.

(ii) $\mathbf{\Pi}_\Sigma|_{[\lambda_{k-1}, \lambda_k]}$, $k = 1, \ldots, n$, is componentwise an affine linear function, that is, $\mathbf{\Pi}_\Sigma|_{[\lambda_{k-1}, \lambda_k]} \in (\mathcal{P}_1)^2$.

Definition 5.10.3. Interpolating closed cubic Hermite curve

Given a sequence $\Sigma := (\mathbf{p}_1, \ldots, \mathbf{p}_n)$, $n \in \mathbb{N}$, of mutually distinct points $\in \mathbb{R}^2$, their **interpolating closed cubic Hermite curve** $\mathbf{H}_\Sigma$ is a function $\mathbf{H}_\Sigma : [0, \lambda_n] \to \mathbb{R}^2$ that satisfies:

(i) $\mathbf{H}_\Sigma(\lambda_k) = \mathbf{p}_{k+1}$, $k = 1, \ldots, n-1$, and $\mathbf{H}_\Sigma(0) = \mathbf{H}_\Sigma(\lambda_n) = \mathbf{p}_1$.

(ii) $\mathbf{H}'(\lambda_k) = \dfrac{\mathbf{s}_k}{\|\mathbf{s}_k\|}$, $\mathbf{s}_k := \begin{cases} \frac{\delta_{k+1}}{\delta_k+\delta_{k+1}}\mathbf{t}_k + \frac{\delta_k}{\delta_k+\delta_{k+1}}\mathbf{t}_{k+1} & \text{, if } k = 1, \ldots, n-1 \text{ ,} \\ \frac{\delta_n}{\delta_1+\delta_n}\mathbf{t}_1 + \frac{\delta_1}{\delta_1+\delta_n}\mathbf{t}_n & \text{, if } k = 0, n \text{ .} \end{cases}$

($\mathbf{H}'$ denotes the derivative of $\mathbf{H}$.)

(iii) $\mathbf{H}_\Sigma|_{[\lambda_{k-1},\lambda_k]}$, $k = 1, \ldots, n$, is componentwise a cubic polynomial, that is $\mathbf{H}_\Sigma|_{[\lambda_{k-1},\lambda_k]} \in (\mathcal{P}_3)^2$.

(5-10.a) ⚂ (20 min.)

Code an efficient C++ function

```
std::vector<Eigen::Vector2d>
  closedPolygonalInterpolant(
    std::vector<Eigen::Vector2d> &Sigma,
    const Eigen::VectorXd &x);
```

that returns the sequence of coordinate vectors $\mathbf{\Pi}_\Sigma(x_k\lambda_n)$, $k = 1, \ldots, M$, M the length of the vector `x` with *sorted* entries $0 \le x_1 < x_2 < \ldots x_M < 1$. $\mathbf{\Pi}_\Sigma$ is the interpolating closed polygon for a set Σ of points passed through the vector argument `Sigma`.

SOLUTION 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 $\mathbf{p}^* := \mathbf{\Pi}_\Sigma(x_k\lambda_n)$:

$$\ell \in \{1, \ldots, n\} : \quad \lambda_{\ell-1} \le x_k\lambda_n < \lambda_\ell \, .$$

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<Vector2d> closedPolygonalInterpolant(std::vector<Vector2d> &Sigma,
3                                                   const 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<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<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   // TO DO (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       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).

▲

(5-10.b) ⚀ (20 min.) [depends on Sub-problem (5-10.a)]

Implement an efficient C++ function

```
std::vector<Eigen::Vector2d>
  closedHermiteInterpolant(
    std::vector<Eigen::Vector2d> &Sigma,
    const Eigen::VectorXd &x);
```

that returns the M vectors $\mathbf{H}_\Sigma(x_k\lambda_n) \in \mathbb{R}^2$, $k = 1,\ldots,M$, M the length of the vector `x` with *sorted* entries $0 \le x_1 < x_2 < \cdots < x_M \le 1$. In this case $\mathbf{H}_\Sigma$ is the interpolating closed cubic Hermite curve according to Def. 5.10.3 for the points passed in the vector `Sigma`.

You may use the auxiliary function

```
double hermloceval(double t,
    double t1, double t2,
    double y1, double y2,
    double c1, double c2);
```

provided in the file `hermloceval.hpp`, which implements the formula [Lecture → Eq. (5.3.3.4)] for evaluating a cubic Hermite interpolating polynomial, see also [Lecture → Code 5.3.3.6].

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<Vector2d> closedHermiteInterpolant(std::vector<Vector2d> &Sigma,
3                                                 const 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<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<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<Vector2d> slopes(n + 1);
27   // TO DO (5-10.b)
28   // START
29   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    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));
50        double coordy =
51            hermloceval(x_scl(j), lambda[i - 1], lambda[i], Sigma[i - 1](1),
52                        Sigma[i](1), slopes[i - 1](1), slopes[i](1));
53        // store the next point
54        res[j] = {coordx, coordy};
55        ++j;
56      }
57      if (j == x.size()) break;
58    }
59    // END
60
61    // resume original Sigma value
62    Sigma.pop_back();
63
64    return res;
65  }
```

Get it on GitLab (adaptiveintp.hpp).

Some curves produced by cubic Hermite interpolation:

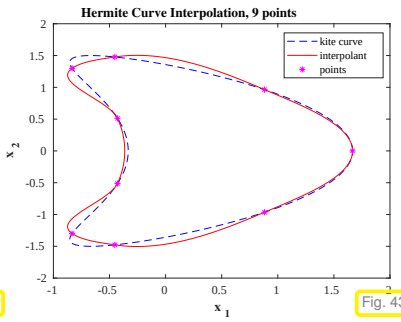


Fig. 42

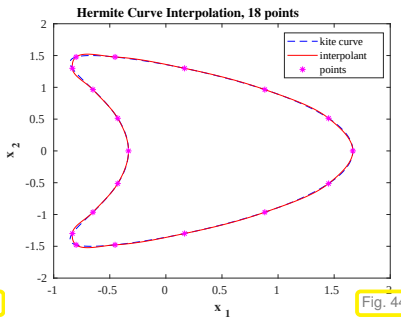


Fig. 43

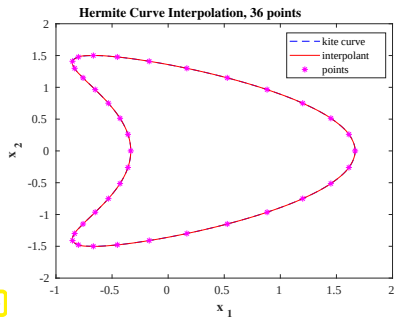


Fig. 44

▲

Now, given a 1-periodic function $\mathbf{c} : \mathbb{R} \to \mathbb{R}^2$ providing the parameterization of a closed curve, we pursue the following policy aiming for its adaptive piecewise polynomial approximation.

Pseudocode 5.10.6: Algorithm for adaptive approximation of closed curve

Given: $\texttt{tol} > 0$, $n_{\min} \in \mathbb{N} \setminus \{1\}$.
$\mathcal{M} := \{t_0 := 0, t_1 := \frac{1}{n_{\min}}, \ldots, t_{n-1} := \frac{n_{\min}-1}{n_{\min}}\}$ // An ordered set
repeat {
  $n := \sharp\mathcal{M}$; // Notation: $\mathcal{M} := \{t_0, \ldots, t_{n-1}\}$
  $\Sigma := (\mathbf{c}(t))_{t\in\mathcal{M}}$; // sequence of points; also defines λ_k
  // $\Pi_\Sigma/\mathbf{H}_\Sigma$ according to Def. 5.10.2, Def. 5.10.3
  $\sigma_k := \left\| (\mathbf{H}_\Sigma - \Pi_\Sigma)(\frac{1}{2}(\lambda_{k-1} + \lambda_k)) \right\|$, $k = 1, \ldots, n$;
  $\alpha := \frac{1}{n}\sum_{k=1}^{n} \sigma_k$;
  for $j = 1, \ldots, n$ **do** {
    if $(\sigma_j > 0.9\alpha)$ $\mathcal{M} \leftarrow \mathcal{M} \cup \{\frac{1}{2}(t_{j-1} + t_j)\}$; // Convention $t_n := 1$
    // $\mathcal{M}$ is always assumed to be sorted!
  }
}
until $(\max_{k=1,\ldots,n} \sigma_k \le \texttt{tol})$;

(5-10.c) ⚀ (30 min.) [depends on Sub-problem (5-10.a) and Sub-problem (5-10.b)]

Based on the C++ functions `closedPolygonalInterpolant()` and `closedHermiteInterpolant()` implement a C++ function

```
template <typename CurveFunctor>
  std::pair<std::vector<Eigen::Vector2d>,
            std::vector<Eigen::Vector2d> >
    adaptedHermiteInterpolant(
      CurveFunctor &&c,
      unsigned int nmin,
      const Eigen::VectorXd &x, double tol = 1.0E-3);
```

that performs the interpolation of a closed curve using the above algorithm from Code 5.10.6. The 1-periodic continuous function $\mathbf{c} : \mathbb{R} \to \mathbb{R}$ is passed through a functor object `c`, which is to feature an evaluation operator

```
Eigen::Vector2d operator () (double t);
```

The argument `tol` supplies the tolerance for the termination of the adaptive algorithm. The function is to return

1. the vectors $\mathbf{H}_\Sigma(x_k L)$, $k = 1, \ldots, M$, M the length of the vector `x` with *sorted* entries $0 \le x_1 < x_2 < \cdots < x_M \le 1$. In this case $\mathbf{H}_\Sigma$ is the interpolating closed cubic Hermite curve according to Def. 5.10.3 for the set Σ at the termination of the algorithm,
2. the coordinates of the points in the sequence Σ as a second sequence of 2-vectors.

SOLUTION 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<Vector2d>, std::vector<Vector2d>>
    adaptedHermiteInterpolant(CurveFunctor &&c, unsigned int nmin, const VectorXd
```

```
    &x, double tol = 1.0e-3) {
  // Note: we assume that x is sorted
  assert(x.minCoeff() >= 0 && x.maxCoeff() <= 1 && "x out of range");
  // will contain the result
  std::vector<Vector2d> eval;
  std::vector<Vector2d> Sigma;

  VectorXd t = VectorXd::LinSpaced(nmin + 1, 0, 1);
  // TO DO (5-10.c)
  // START
  // vector of deviations
  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
    VectorXd lambda(n + 1);

    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<Vector2d> v =
        closedPolygonalInterpolant(Sigma, midpt / lambda[n]);
    std::vector<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);
        num++;
      }
    }
    t.conservativeResize(n + 1 + t_temp.size());

    t.tail(t_temp.size()) = VectorXd::Map(t_temp.data(), t_temp.size());
    // sort vector of arc-length values
    std::sort(t.data(), t.data() + t.size());
  } while (dev.maxCoeff() > tol);

  eval = closedHermiteInterpolant(Sigma, x);
  // END
  return std::make_pair(eval, Sigma);
```

```
63 }
```

Get it on GitLab (adaptiveintp.hpp).

▲

(5-10.d) ⚄ (30 min.) [depends on Sub-problem (5-10.c)]

Using MATPLOTLIBCPP and based on your implementation of `adaptedHermiteInterpolant()`, write a C++ function

```
void plotKite(const char *filename, double tol = 1.0e-3);
```

that, for the kite-shaped curve $\mathbf{c} : [0,1] \to \mathbb{R}^2$ as defined in (5.10.1), and with the choice $n_{\min} = 6$, generates a rendering of the closed polygonal interpolant ($\to$ Def. 5.10.2) and the interpolating closed cubic Hermite curve ($\to$ Def. 5.10.3) produced by a call to the function `adaptedHermiteInterpolant()` with the given tolerance `tol`. The plots should be created using $N := 10000$ equidistant sampling points $x_k := \frac{k}{N}, k = 0, \ldots, N-1$.

The graphics is to be saved in PNG format in a file `<filename>.png`.

SOLUTION of (5-10.d):

C++ code 5.10.8: Sub-problem (5-10.c): Plotting kite-shaped curve

```
2  void plotKite(const char *filename, double tol = 1.0e-3) {
3    // TO DO (5-10.d)
4    // START
5    // preparing call to adaptedHermiteInterpolant
6    auto c = [](double t) -> Vector2d {
7      Vector2d ct;
8      ct << std::cos(2 * M_PI * t) + 2. / 3. * std::cos(4 * M_PI * t),
9          3. / 2. * std::sin(2 * M_PI * t);
10     return ct;
11   };
12   unsigned int nmin = 6;
13   unsigned int N = 1e5;
14   // initializing x, dsicarding the 1 at the end
15   VectorXd x = VectorXd::LinSpaced(N + 1, 0, 1).head(N);
16
17   std::pair<std::vector<Vector2d>, std::vector<Vector2d>> vec =
18       adaptedHermiteInterpolant(c, nmin, x, tol);
19
20   std::vector<Vector2d> eval = vec.first;
21   std::vector<Vector2d> Sigma = vec.second;
22   unsigned int M = Sigma.size();
23   VectorXd x_hermite(N);
24   VectorXd y_hermite(N);
25   VectorXd x_poly(M);
26   VectorXd y_poly(M);
27   // assembling x and y of Hermite Interpolation
28   for (unsigned int i = 0; i < N; i++) {
29     x_hermite(i) = eval.at(i)(0);
30     y_hermite(i) = eval.at(i)(1);
31   }
32   // assembling x and y of Polygonal Interpolation
```

```
33    // We do not need to call ClosedPolygonalInterpolate
34    // as matplotlibcpp will already connect the points
35    // with straight lines for us.
36    for (unsigned int i = 0; i < M; i++) {
37      x_poly(i) = Sigma.at(i)(0);
38      y_poly(i) = Sigma.at(i)(1);
39    }
40    // making the plot
41    plt::figure();
42    plt::title("Adapted Interpolation with tol=" + std::to_string(tol));
43    plt::xlabel("x");
44    plt::ylabel("y");
45    plt::plot(x_poly, y_poly, "r", {{"label", "Polygonal Interpolant"}});
46    plt::plot(x_hermite, y_hermite, {{"label", "Hermite Interpolation"}});
47    plt::legend();
48    plt::savefig(filename);
49    // END
50    return;
51  }
```

Get it on GitLab (adaptiveintp.hpp).

▲

Figures of kite-shaped curves produced by the code above code:

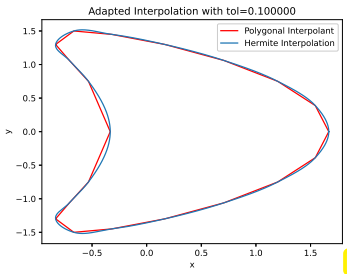


Fig. 45

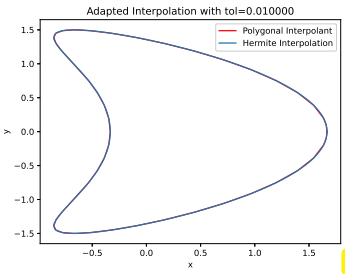


Fig. 46

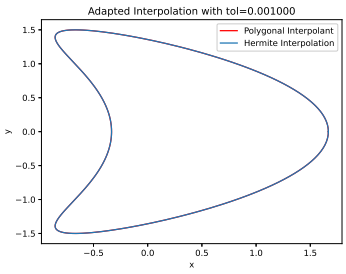


Fig. 47

End Problem 5-10 , 100 min.

Problem 5-11: A cubic spline

Since they mimic the behavior of an elastic rod pinned at fixed points, see [Lecture → § 5.4.3.1], cubic splines are very popular for creating "aesthetically pleasing" interpolating functions. However, in this problem we look at a cubic spline from the perspective of its defining properties, see [Lecture → Def. 5.4.1.1], in order to become more familiar with the concept of spline function and the consequences of the smoothness required by the definition.

Related to [Lecture → Section 5.4.1]

(5-11.a) ⚄ (15 min.) For parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ we define the function

$$s(t) = \begin{cases} -1 & \text{for} \quad -1 \le t < 0\,, \\ \alpha t^3 + \beta t^2 + \gamma t + \delta & \text{for} \quad 0 \le t < 1\,, \\ -t^3 + \sigma t^2 + \rho t + 2 & \text{for} \quad 1 \le t \le 2\,. \end{cases}$$

Determine the parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ such that s is a cubic spline with respect to the knot set $\mathcal{M} := \{-1, 0, 1, 2\}$.

$\alpha =$ ☐ , $\beta =$ ☐ $\gamma =$ ☐ ,

$\delta =$ ☐ , $\sigma =$ ☐ $\rho =$ ☐ .

SOLUTION of (5-11.a):

Since $s \in C^2([-1,2])$ is required for $s \in \mathcal{S}_{3,\mathcal{M}}$ [Lecture → 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

$$\begin{bmatrix} 1 & -1 & -1 \\ 3 & -2 & -1 \\ 6 & -2 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \sigma \\ \rho \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ -6 \end{bmatrix} \quad \Rightarrow \quad \begin{array}{l} \alpha = 2\,, \\ \sigma = 9\,, \\ \rho = -9\,. \end{array}$$

▲

(5-11.b) ⚄ (15 min.) For parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ we define the function

$$s(t) = \begin{cases} 2 & \text{for} \quad -1 \leq t < 0\,, \\ \sigma t^3 + \rho t^2 + \gamma t + \delta & \text{for} \quad 0 \leq t < 1\,, \\ -t^3 + \alpha t^2 + \beta t - 1 & \text{for} \quad 1 \leq t \leq 2\,. \end{cases}$$

Determine the parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ such that s is a cubic spline with respect to the knot set $\mathcal{M} := \{-1, 0, 1, 2\}$.

$$\alpha = \boxed{}\,, \quad \beta = \boxed{} \quad \gamma = \boxed{}\,,$$

$$\delta = \boxed{}\,, \quad \sigma = \boxed{} \quad \rho = \boxed{}\,.$$

SOLUTION of (5-11.b):

- At $t = 0$: $\delta = 2, \gamma = 0, \rho = 0$.
- At $t = 1$: From the continuity conditions

$$\begin{aligned} \sigma + 2 &= -1 + \alpha + \beta - 1 \\ 3\sigma &= -3 + 2\alpha + \beta \\ 6\sigma &= -6 + 2\alpha \end{aligned} \quad \blacktriangleright \quad \begin{bmatrix} 1 & 1 & -1 \\ 2 & 1 & -3 \\ 2 & 0 & -6 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \sigma \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 6 \end{bmatrix} \quad \Rightarrow \quad \begin{aligned} \alpha &= -9\,, \\ \beta &= 9\,, \\ \sigma &= -4\,. \end{aligned}$$

▲

(5-11.c) ⚄ (15 min.) For parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ we define the function

$$s(t) = \begin{cases} 1 & \text{for} \quad -1 \leq t < 0\,, \\ \alpha t^3 + \beta t^2 + \gamma t + \delta & \text{for} \quad 0 \leq t < 2\,, \\ -t^3 + \rho t^2 + \sigma t - 7 & \text{for} \quad 2 \leq t \leq 3\,. \end{cases}$$

Determine the parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ such that s is a cubic spline with respect to the knot set $\mathcal{M} := \{-1, 0, 2, 3\}$.

$$\alpha = \boxed{}\,, \quad \beta = \boxed{} \quad \gamma = \boxed{}\,,$$

$$\delta = \boxed{}\,, \quad \sigma = \boxed{} \quad \rho = \boxed{}\,.$$

SOLUTION of (5-11.c):

- At $t = 0$: $\delta = 1, \gamma = 0, \beta = 0$
- At $t = 2$:

$$\begin{aligned} 8\alpha + 1 &= -8 + 4\rho + 2\sigma - 7 \\ 12\alpha &= -12 + 4\rho + \sigma \\ 12\alpha &= -12 + 2\rho \end{aligned} \quad \blacktriangleright \quad \begin{bmatrix} 8 & -4 & -2 \\ 12 & -4 & -1 \\ 12 & -2 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \rho \\ \sigma \end{bmatrix} = \begin{bmatrix} -16 \\ -12 \\ -12 \end{bmatrix} \Rightarrow \quad \begin{aligned} \alpha &= -2\,, \\ \rho &= -6\,, \\ \sigma &= 12\,. \end{aligned}$$

▲

End Problem 5-11 , 45 min.

Problem 5-12: Not-a-knot Cubic Splines

As we have seen in [Lecture → Section 5.4.2] cubic splines can be used for one-dimensional data interpolation, which is a popular technique in computer aided geometric design (CAGD). Unfortunately, extra conditions have to be imposed to render the cubic-spline interpolant unique as discussed in [Lecture → § 5.4.2.11]. Not-a-knot cubic splines are another way to arrive at a unique cubic-spline interpolant.

This problem builds on [Lecture → Section 5.4.2] and contents of [Lecture → § 5.4.2.5] should be studied in detail before tackling this problem.

Recall what is a cubic spline interpolant,

Definition [Lecture → Def. 5.4.2.3]. Cubic-spline interpolant

Given a node set/knot set $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$, $n \in \mathbb{N}$, and data values $y_0, \ldots, y_n \in \mathbb{R}$, an associated **cubic spline interpolant** is a function $s \in \mathcal{S}_{3,\mathcal{M}}$ that complies with the interpolation conditions

$$s(t_j) = y_j \quad , \quad j = 0, \ldots, n \, . \tag{5.12.1}$$

and, more fundamentally, the definition of spline functions as special piecewise polynomial functions:

Definition [Lecture → Def. 5.4.1.1]. Splines

Given an interval $I := [a,b] \subset \mathbb{R}$ and a **knot sequence** $\mathcal{M} := \{a = t_0 < t_1 < \ldots < t_{n-1} < t_n = b\}$, $n \in \mathbb{N}$, the vector space $\mathcal{S}_{d,\mathcal{M}}$ of the **spline functions** of degree d (or order $d+1$), $d \in \mathbb{N}_0$, is defined by

$$\mathcal{S}_{d,\mathcal{M}} := \{s \in C^{d-1}(I) \colon s_j := s_{|[t_{j-1},t_j]} \in \mathcal{P}_d \; \forall j = 1, \ldots, n\} \, .$$

Now we study a specialization of [Lecture → Def. 5.4.2.3]:

Definition 5.12.2. Not-a-knot cubic spline

Given a node set/knot set $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$, $n \in \mathbb{N}$, $n \geq 2$, and data values $y_0, \ldots, y_n \in \mathbb{R}$, an associated **not-a-knot cubic spline interpolant** is a cubic spline interpolant s for the data points (t_i, y_i), $i = 0, \ldots, n$, that satisfies the extra conditions that

the *third derivative* of s is *continuous at* t_1 *and* t_{n-1}.

The following theorem of De Boor confirms that the not-a-knot constructions resolves the non-uniqueness of cubic spline interpolants.

Theorem 5.12.3. Existence and uniqueness of not-a-knot cubic spline interpolant

For any $n \in \mathbb{N}$, $n > 2$, and any node set/knot set $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$ and data values $y_i \in \mathbb{R}$, $i = 0, \ldots, n$, there exists a unique not-a-knot cubic spline interpolant according to Def. 5.12.2.

(5-12.a) ⚂ (10 min.) Why is the assumption $n > 2$ necessary in the statement of Thm. 5.12.3?

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 → Thm. 5.2.2.7] such an interpolating polynomial is not unique, because it would take four interpolation conditions in order to ensure uniqueness.

▲

You are supposed to implement a C++ class

```
class NotAKnotCubicSpline {
  public:
  NotAKnotCubicSpline(Eigen::VectorXd t,
  Eigen::VectorXd y);
  ~NotAKnotCubicSpline() = default;
  [[nodiscard]] double eval(double t) const;
  private:
  [[nodiscard]] int getPtIdx(double t) const;
  Eigen::VectorXd t_;   // knot sequence
  Eigen::VectorXd y_;  // data values
  Eigen::VectorXd c_;  // slopes
};
```

The public member function `eval()` is supposed to evaluate the not-a-knot cubic spline interpolant at a single point $t \in [t_0, t_n]$.

Fortunately, you found a complete implementation of a class **NaturalCubicSpline** with a 100% analogous definition for natural cubic spline interpolation. You stored it in the files `notaknotcubicspline.hpp` and ran a text replacement **NaturalCubicSpline** → **NotAKnotCubicSpline** on it. And this is where you are standing.

(5-12.b) ⚄ (30 min.) Each of the continuity requirements for the third derivative of the cubic spline interpolant at the knots t_1 and t_{n-1} (assume $n > 2$, see Def. 5.12.2 for notations) induces one linear relationship between the components of the vectors `y_` and `c_`. Derive and state those relationships.

SOLUTION 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 $[t_{j-1}, t_j]$, $j = 1, \ldots, n$:

$$\begin{aligned} s_{|[t_{j-1},t_j]}(t) \; = & y_{j-1}(1 - 3\tau^2 + 2\tau^3) + y_j(3\tau^2 - 2\tau^3) + \\ & h_j c_{j-1}(\tau - 2\tau^2 + \tau^3) + h_j c_j(-\tau^2 + \tau^3) \, , \end{aligned} \tag{5.12.4}$$

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

$$\frac{12}{h_1^3}(y_0-y_1) + \frac{6}{h_1^2}(c_0+c_1) = \frac{12}{h_2^3}(y_1-y_2) + \frac{6}{h_2^2}(c_1+c_2)\,, \tag{5.12.6}$$

$$\frac{12}{h_{n-1}^3}(y_{n-2}-y_{n-1}) + \frac{6}{h_{n-1}^2}(c_{n-2}+c_{n-1}) = \frac{12}{h_n^3}(y_{n-1}-y_n) + \frac{6}{h_n^2}(c_{n-1}+c_n)\,. \tag{5.12.7}$$

▲

(5-12.c) ⚄ (30 min.) [depends on Sub-problem (5-12.b)]

In the file `notaknotcubicspline.hpp` write *efficient* code for the constructor

```
NotAKnotCubicSpline(Eigen::VectorXd t,
                    Eigen::VectorXd y);
```

that is supposed to initialize the data members `y_` and `c_` in a way that does not break the already implemented `eval()` member function.

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

```
2  NotAKnotCubicSpline::NotAKnotCubicSpline(Eigen::VectorXd t, Eigen::VectorXd y)
3      : t_(std::move(t)), y_(std::move(y)), c_(y_.size()) {
4    // TODO: (5-12.c) Modify the given constructor of a Natural Cubic
       Spline into
5    // an efficient constructor for Not-a-knot Cubic Spline START
6    const long n = t_.size() - 1;
7    assert(n > 2);
8    assert(n == y_.size() - 1);
9    // Sequence (h_i)_{i=1}^n, knots have to be sorted
10   auto h = (t_.tail(n) - t_.head(n)).array();
11   auto hs = h.pow(2);                                  // h_i^2
```

```
12    auto b = 1.0 / h;                                    // b_i := h_i^{-1}
13    auto a = 2.0 * (b.head(n - 1) + b.tail(n - 1));  // a_i
14    // Initialize sparse coefficient matrix for the linear system of
          equations
15    // needed to compute the unknown slopes.  Compared to [Lecture →
          Eq. (5.4.2.13)] only
16    // the first and last row has to be adjusted, see (5.12.8).
17    Eigen::SparseMatrix<double> M(n + 1, n + 1);
18    M.reserve(Eigen::RowVectorXi::Constant(n + 1, 3));
19    // Matrix row arising from the continuity of
20    M.insert(0, 0) = 6.0 / hs[0];
21    M.insert(0, 1) = (6.0 / hs[0] - 6.0 / hs[1]);
22    M.insert(0, 2) = -6.0 / hs[1];
23    for (int i = 1; i < n; ++i) {
24      M.insert(i, i - 1) = b[i - 1];
25      M.insert(i, i) = a[i - 1];
26      M.insert(i, i + 1) = b[i];
27    }
28    // Matrix due to continuity of s^(3) at t_{n-1}
29    M.insert(n, n - 2) = 6.0 / hs[n - 2];
30    M.insert(n, n - 1) = 6.0 / hs[n - 2] - 6.0 / hs[n - 1];
31    M.insert(n, n) = -6.0 / hs[n - 1];
32    // Initialize right-hand-side vector, see (5.12.8)
33    Eigen::VectorXd rhs(n + 1);
34    rhs[0] = -12.0 / (hs[0] * h[0]) * (y_[0] - y_[1]) +
35             12.0 / (hs[1] * h[1]) * (y_[1] - y_[2]);
36    for (int i = 1; i < n; ++i) {
37      rhs[i] =
38          3.0 * ((y_[i] - y_[i - 1]) / hs[i - 1] + (y_[i + 1] - y_[i]) / hs[i]);
39    }
40    rhs[n] = -12.0 / (hs[n - 2] * h[n - 2]) * (y_[n - 2] - y_[n - 1]) +
41             12.0 / (hs[n - 1] * h[n - 1]) * (y_[n - 1] - y_[n]);
42    // Compute slopes by solving an (almost) tridiagonal linear system
43    // with asymptotic effort O(n) for n → ∞.
44    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
45    solver.compute(M);
46    if (solver.info() != Eigen::Success) {
47      throw std::runtime_error("LU factorization failed");
48    }
49    c_ = solver.solve(rhs);
50    if (solver.info() != Eigen::Success) {
51      throw std::runtime_error("Elimination failed");
52    }
53    if (checkC2()) {
54      std::cout << "C2 continuity satisfied" << std::endl;
55    }
56    // END
57  }
```

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].

▲

(5-12.d) ⚄ (20 min.) An annoying client asks you to extend the functionality of **NotAKnotCubicSpline** to include a member function

```
double evalderivative(double t) const;
```

that returns the value of the first derivative of the not-a-knot cubic spline interpolant at a single point $t \in]t_0, t_n[$.

Add this member function to the class definition and provide an implementation in the file `notaknotcubicspline.hpp`.

SOLUTION 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.

▲

End Problem 5-12, 90 min.

Problem 5-13: Various Aspects of Interpolation by Global Polynomials

[Lecture → Chapter 5] teaches the mathematical foundations and algorithmic realizations of interpolation by means of global polynomials. This problems reviews some of these aspects.

This is a purely theoretical problem connected with [Lecture → Section 5.2]

Throughout this problem we write $\mathcal{P}_d(\mathbb{R})$ for the space of uni-variate polynomials of degree $\leq d$:

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_1 t + \alpha_0 \cdot 1\,,\ \alpha_j \in \mathbb{R}\}\,. \qquad \text{[Lecture → Eq. (5.2.1.1)]}$$

(5-13.a) ⚀ (6 min.) What are the dimensions of the following subspaces of $\mathcal{P}_d(\mathbb{R})$, $d \in \mathbb{N}$?

(i) $V_1 := \{p \in \mathcal{P}_d(\mathbb{R}) : \int_{-1}^{1} p(t)\,\mathrm{d}t = 0\}$: $\dim V_1 =$ ☐.

(ii) $V_2 := \{p \in \mathcal{P}_d(\mathbb{R}) : p(1) = p(-1) = 0\}$: $\dim V_2 =$ ☐.

(iii) $V_3 := \{p \in \mathcal{P}_d(\mathbb{R}) : p^{(3)}(t) = 0\ \forall t \in [-1,1]\}$: $\dim V_3 =$ ☐ for ☐, ☐ for ☐.

Here $p^{(3)}$ stands for the third derivative of p.

(iv) $V_4 := \{p \in \mathcal{P}_d(\mathbb{R}) : p(t) = p(-t)\}$: $\dim V_4 =$ ☐ for ☐, ☐ for ☐.

SOLUTION 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

$$V_4 \subset \mathcal{P}_n^{\text{even}} := \{t \mapsto a_0 + a_2 t^2 + a_4 t^4 + \cdots + a_{2n} t^{2n}\}\,, \quad n \in \mathbb{N}\,.$$

Elements of space $\mathcal{P}_n^{\text{even}}$ are defined by $n+1$ parameters, which means $\dim \mathcal{P}_n^{\text{even}} = n+1$.

Note that $n = \frac{d}{2}$ for even d and $n = \frac{d-1}{2}$ for odd d.

$$\blacktriangleright \quad \dim V_4 = \begin{cases} \frac{d}{2}+1 & \text{for even } d\,, \\ \frac{d-1}{2}+1 & \text{for odd } d. \end{cases}$$

▲

(5-13.b) ⚂ (3 min.) Let $\{t_0, t_1, \ldots, t_n\} \subset \mathbb{R}$, $n \in \mathbb{N}$. The following sets of functions provide bases for $\mathcal{P}_d$:

$$\mathfrak{B}_1 := \left\{ t \mapsto \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{t - t_j}{t_i - t_j}\,, \quad i = 0, \ldots, n \right\}\,, \qquad \square$$

$$\mathfrak{B}_2 := \left\{ t \mapsto \prod_{j=0}^{i-1} (t - t_j)\,, \quad i = 0, \ldots, n \right\}\,, \qquad \square$$

$$\mathfrak{B}_3 := \left\{ t \mapsto t^i\,, \quad i = 0, \ldots, n \right\}\,. \qquad \square$$

Note that an empty product is defined to be $\equiv 1$.

Put the right letter in the box indicating the name of the basis:

A ≙ monomial basis , **B** ≙ Newton basis , **C** ≙ Lagrangian basis .

SOLUTION of (5-13.b):

- $\mathfrak{B}_1$ is the Lagrangian basis **C** [Lecture → 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 → Eq. (5.2.3.23)], used for update-friendly polynomial interpolation
- $\mathfrak{B}_3$ is the monomial basis **A** [Lecture → Section 5.2.1].

▲

(5-13.c) ⚂ (6 min.) Given data points (t_i, y_i), $i = 0, \ldots, n$, $n \in \mathbb{N}$, we write $p_{k,\ell}$, $0 \le k \le \ell \le n$, for the polynomial $p_{k,\ell} \in \mathcal{P}_{\ell-k}$ interpolating through $(t_i, y_i)_{i=k}^{\ell}$: $p_{k,\ell}(t_i) = y_i$, $i = k, \ldots, \ell$. Supplement the missing parts of the following recursion:

$$p_{k,\ell}(t) = \frac{\left(\square\right) p_{k+1,\ell}(t) + \left(\square\right) p_{k,\ell-1}(t)}{\square - \square}\,, \quad t \in \mathbb{R}\,, \quad 0 \le k < \ell \le n\,.$$

SOLUTION of (5-13.c):

This is the recursion [Lecture → Eq. (5.2.3.9)] underlying the **Aitken-Neville** scheme:

$$p_{k,\ell}(t) = \frac{\big(t - t_k\big)p_{k+1,\ell}(t) + \big(t_\ell - t\big)p_{k,\ell-1}(t)}{t_\ell - t_k}, \quad t \in \mathbb{R}\,.$$

The correctness of this recursion is shown by induction, verifying the interpolation conditions.

▲

End Problem 5-13, 15 min.

Problem 5-14: Local Representations of Splines

Locally on knot intervals splines coincide with polynomials of a fixed degree. In this problem we compute those local polynomial representations in special cases.

This is a purely theoretical problem based on [Lecture → Section 5.4.1] and [Lecture → Section 5.4.2].

(5-14.a) ⚄ (6 min.)

Let s denote a *quadratic spline*, $s \in \mathcal{S}_{2,\mathcal{M}}$ with respect to the knot set $\mathcal{M} := \{-3,-2,-1,1,2,3\}$ and satisfying

$$s \equiv 0 \quad \text{on} \quad [-3,-2] \cup [2,3]\,,$$
$$s(-1) = s(1) = 1\,.$$

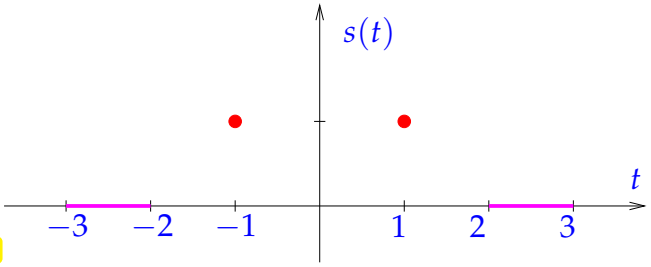


Fig. 48

Determine the unknown real coefficients in the local representations

$$s(t) = \square\, t^2 + \square\, t + \square \quad \text{for} \quad t \in [1,2]\,,$$

$$s(t) = \square\, t^2 + \square\, t + \square \quad \text{for} \quad t \in [-1,1]\,.$$

SOLUTION 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

$$s(t) = \boxed{1}\cdot t^2 - \boxed{4}\, t + \boxed{4} \quad \text{on} \quad [1,2]\,.$$

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

$$s(t) = \boxed{-1}\cdot t^2 + \boxed{0}\, t + \boxed{2} \quad \text{on} \quad [-1,1]\,.$$

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.

▲

(5-14.b) ⚄ (12 min.) Let the knot set $\mathcal{M} := \{t_0 < t_1 < t_2 < \cdots < t_n\} \subset \mathbb{R}$ be given. The local representation of a cubic spline $s \in \mathcal{S}_{3,\mathcal{M}}$ is given by ($h_j := t_j - t_{j-1}$, $j \in \{1,\ldots,n\}$)

$$s|_{[t_{j-1},t_j]}(t) = \alpha_j H_1(\tau) + \beta_j H_2(\tau) + h_j\gamma_j H_3(\tau) + h_j\delta_j H_4(\tau)\,, \quad \tau := \frac{t - t_{j-1}}{h_j}\,, \quad \alpha_j,\beta_j,\gamma_j,\delta_j \in \mathbb{R}\,,$$

where H_1, H_2, H_3, H_4 are the cardinal basis functions for Hermite interpolation on $[0,1]$,

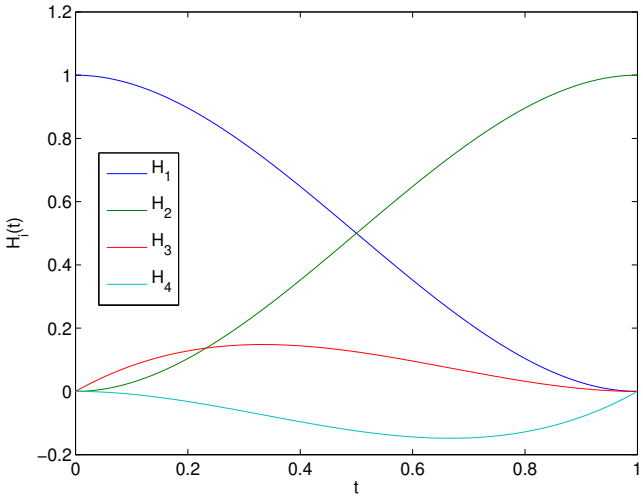


Fig. 49

$$\begin{aligned} H_1(\tau) &= 1 - 3\tau^2 + 2\tau^3\,,\\ H_2(\tau) &= 3\tau^2 - 2\tau^3\,,\\ H_3(\tau) &= \tau - 2\tau^2 + \tau^3\,,\\ H_4(\tau) &= -\tau^2 + \tau^3\,. \end{aligned}$$

Inserting the midpoints of knot intervals gives us the extended knot set

$$\widetilde{\mathcal{M}} := \{\widetilde{t}_0 < \widetilde{t}_1 < \widetilde{t}_2 < \cdots < \widetilde{t}_{2n}\}\,, \quad \begin{aligned} \widetilde{t}_{2j} &:= t_j\,, & j &\in \{0,\ldots,n\}\,,\\ \widetilde{t}_{2j-1} &:= \tfrac{1}{2}(t_j + t_{j-1})\,, & j &\in \{1,\ldots,n\}\,. \end{aligned}$$

On the knot intervals of $\widetilde{\mathcal{M}}$ the spline s has the local representation ($\widetilde{h}_\ell := \widetilde{t}_\ell - \widetilde{t}_{\ell-1}$, $\ell \in \{1,\ldots,2n\}$)

$$s|_{[\widetilde{t}_{\ell-1},\widetilde{t}_\ell]}(t) = \widetilde{\alpha}_\ell H_1(\tau) + \widetilde{\beta}_\ell H_2(\tau) + \widetilde{h}_\ell\widetilde{\gamma}_\ell H_3(\tau) + \widetilde{h}_\ell\widetilde{\delta}_\ell H_4(\tau)\,,\ \tau := \frac{t - \widetilde{t}_{\ell-1}}{\widetilde{h}_\ell}\,,\ \widetilde{\alpha}_\ell,\widetilde{\beta}_\ell,\widetilde{\gamma}_\ell,\widetilde{\delta}_\ell \in \mathbb{R}\,.$$

Express the coefficients $\widetilde{\alpha}_\ell, \widetilde{\beta}_\ell, \widetilde{\gamma}_\ell, \widetilde{\delta}_\ell$ in terms of $\alpha_j, \beta_j, \gamma_j, \delta_j$:

$$\widetilde{\alpha}_\ell = \begin{cases} \square & \text{for even } \ell = 2j\,, \\ \square & \text{for odd } \ell = 2j-1\,, \end{cases}$$

$$\widetilde{\beta}_\ell = \begin{cases} \square & \text{for even } \ell = 2j, \\ \square & \text{for odd } \ell = 2j-1\,, \end{cases}$$

$$\widetilde{\gamma}_\ell = \begin{cases} \square & \text{for even } \ell = 2j, \\ \square & \text{for odd } \ell = 2j-1\,, \end{cases}$$

$$\widetilde{\delta}_\ell = \begin{cases} \square & \text{for even } \ell = 2j, \\ \square & \text{for odd } \ell = 2j-1\,. \end{cases}$$

SOLUTION of (5-14.b):

By straightforward computations:

$$\begin{aligned} &H_1(0)=1\,,\quad H_1(1)=0\,,\quad H_1'(0)=0\,,\quad H_1'(1)=0\,,\quad H_1(\tfrac{1}{2})=\tfrac{1}{2}\,,\quad H_1'(\tfrac{1}{2})=-\tfrac{3}{2}\,,\\ &H_2(0)=0\,,\quad H_2(1)=1\,,\quad H_2'(0)=0\,,\quad H_2'(1)=0\,,\quad H_2(\tfrac{1}{2})=\tfrac{1}{2}\,,\quad H_2'(\tfrac{1}{2})=\tfrac{3}{2}\,,\\ &H_3(0)=0\,,\quad H_3(1)=0\,,\quad H_3'(0)=1\,,\quad H_3'(1)=0\,,\quad H_3(\tfrac{1}{2})=\tfrac{1}{8}\,,\quad H_3'(\tfrac{1}{2})=-\tfrac{1}{4}\,,\\ &H_4(0)=0\,,\quad H_4(1)=0\,,\quad H_4'(0)=0\,,\quad H_4'(1)=1\,,\quad H_4(\tfrac{1}{2})=-\tfrac{1}{8}\,,\quad H_4'(\tfrac{1}{2})=-\tfrac{1}{4}\,. \end{aligned} \tag{5.14.1}$$

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

$$\begin{aligned} &H_1(1/2)=\tfrac{1}{2}\,,\quad H_1'(1/2)=-\tfrac{3}{2}\,,,\\ &H_2(1/2)=\tfrac{1}{2}\,,\quad H_2'(1/2)=\tfrac{3}{2}\,,\,,\\ &H_3(1/2)=\tfrac{1}{8}\,,\quad H_3'(1/2)=-\tfrac{1}{4}\,,,\\ &H_4(1/2)=-\tfrac{1}{8}\,,\quad H_4'(1/2)=-\tfrac{1}{4}\,,, \end{aligned} \tag{5.14.4}$$

which implies

$$s(\tfrac{1}{2}(t_j+t_{j-1}))=\tfrac{1}{2}(\alpha_j+\beta_j)+\tfrac{1}{8}(\gamma_j-\delta_j)\,, \tag{5.14.5}$$

$$s'(\tfrac{1}{2}(t_j + t_{j-1})) = \tfrac{3}{2}(-\alpha_j + \beta_j) - \tfrac{1}{4}(\gamma_j + \delta_j) \,. \tag{5.14.6}$$

I. For **even** $\ell = 2j$ we have $\widetilde{t}_\ell = t_j$, $j = 1, \ldots, n$ and the associated knot interval for $\ell, j > 0$ is

$$[\widetilde{t}_{\ell-1}, \widetilde{t}_\ell] = [\tfrac{1}{2}(t_j + t_{j-1}), t_j] \,.$$

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)

$$\begin{aligned}
\widetilde{\alpha}_\ell &= s(\widetilde{t}_{\ell-1}) = s(\tfrac{1}{2}(t_j + t_{j-1})) = \tfrac{1}{2}(\alpha_j + \beta_j) + \tfrac{1}{8}h_j(\gamma_j - \delta_j) \,, \\
\widetilde{\beta}_\ell &= s(\widetilde{t}_\ell) = s(t_j) = \beta_j \,, \\
\widetilde{\gamma}_\ell &= s'(\widetilde{t}_{\ell-1}) = s'(\tfrac{1}{2}(t_j + t_{j-1})) = \tfrac{3}{2}h_j^{-1}(-\alpha_j + \beta_j) - \tfrac{1}{4}(\gamma_j + \delta_j) \,, \\
\widetilde{\delta}_\ell &= s'(\widetilde{t}_\ell) = s'(t_j) = \delta_j \,.
\end{aligned}$$

The factor h_j^{-1} is introduced by applying the chain rule.

II. For **odd** $\ell = 2j - 1$, $j = 1, \ldots, n$, 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

$$\widetilde{\alpha}_\ell = \begin{cases} \tfrac{1}{2}(\alpha_j + \beta_j) + \tfrac{1}{8}h_j(\gamma_j - \delta_j) & \text{for even } \ell \,, \\ \alpha_j & \text{for odd } \ell \,, \end{cases} \quad \widetilde{\beta}_\ell = \begin{cases} \beta_j & \text{for even } \ell \,, \\ \tfrac{1}{2}(\alpha_j + \beta_j) + \tfrac{1}{8}h_j(\gamma_j - \delta_j) & \text{for odd } \ell \,, \end{cases}$$

$$\widetilde{\gamma}_\ell = \begin{cases} \tfrac{3}{2}h_j^{-1}(-\alpha_j + \beta_j) - \tfrac{1}{4}(\gamma_j + \delta_j) & \text{for even } \ell \,, \\ \gamma_j & \text{for odd } \ell \,, \end{cases} \quad \widetilde{\delta}_\ell = \begin{cases} \delta_j & \text{for even } \ell \,, \\ \tfrac{3}{2}h_j^{-1}(-\alpha_j + \beta_j) - \tfrac{1}{4}(\gamma_j + \delta_j) & \text{for odd } \ell \,. \end{cases}$$

▲

End Problem 5-14 , 18 min.

Problem 5-15: Closed Curve Interpolation with Quadratic Splines

The reconstruction of curves and surfaces from given points is a central task in computer-aided design (CAD). Here we focus on the problem of finding a closed and C^1-smooth curve running through a sequence of given points. That curve should be modeled by means of a periodic quadratic spline function.

This problem relies on [Lecture → Section 5.4] and has a focus on implementation in C++ using EIGEN.

We are given n, $n \in \mathbb{N}$, mutually different points $\mathbf{p}^1, \ldots, \mathbf{p}^n \in \mathbb{R}^2$ in the plane. Defining a knot sequence $\mathcal{M} := \{0 = t_0 < t_1 < \ldots < t_{n-1} < t_n = 1\}$ as [1]

$$t_j := \frac{\sum_{k=1}^{j} \left\| \mathbf{p}^k - \mathbf{p}^{k-1} \right\|_2}{\sum_{k=1}^{n} \left\| \mathbf{p}^k - \mathbf{p}^{k-1} \right\|_2} \quad (\, \mathbf{p}^0 := \mathbf{p}^n \,) \,, \quad j = 0, \ldots, n \,, \tag{5.15.1}$$

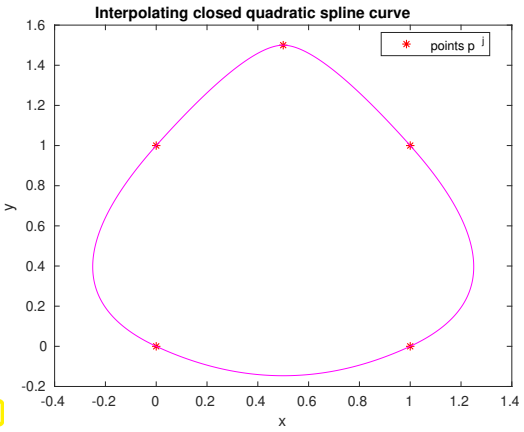


Fig. 50

we want to find a spline curve $\mathbf{s} : [0,1] \to \mathbb{R}^2$ satisfying

$$\mathbf{s} \in (\mathcal{S}_{2,\mathcal{M}})^2 \,, \quad \textit{c.f.} \ [\text{Lecture} \to \text{Def. 5.4.1.1}], \tag{5.15.2}$$

$$\mathbf{s}(t_j) = \mathbf{p}^j \quad \text{for all} \quad j = 1, \ldots, n \,, \tag{5.15.3}$$

$$\mathbf{s}(t_0) = \mathbf{p}^n \quad \text{and} \quad \mathbf{s}'(t_0) = \mathbf{s}'(t_n) \,, \tag{5.15.4}$$

where $'$ indicates differentiation with respect to the curve parameter. The requirement (5.15.4) ensures that $\mathbf{s}$ describes a *closed C^1-smooth* curve.

Throughout, for $\mathbf{S}$ use the following local representation on knot intervals:

$$\mathbf{s}|_{]t_{j-1},t_j[}(t) = \mathbf{y}^{j-1}(1-\tau) + \mathbf{x}^j h_j \tau(1-\tau) + \mathbf{y}^j \tau \,, \quad \tau := \frac{t - t_{j-1}}{h_j} \,, \quad \begin{matrix} t_{j-1} < t < t_j \,, \\ h_j := t_j - t_{j-1} \,, \end{matrix} \tag{5.15.5}$$

with unknown vector-valued coefficients $\mathbf{y}^j \in \mathbb{R}^2$, $j = 0, \ldots, n$, and $\mathbf{x}^j \in \mathbb{R}^2$, $j = 1, \ldots, n$.

(5-15.a) ⚀ (6 min.) What are the benefits of using the local representation (5.15.5)?

SOLUTION 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

[1] Sums whose lower summation bound exceeds the upper are supposed to evaluate to zero.

- the interpolation conditions (5.15.3)/(5.15.4) can easily be enforced by setting

$$\mathbf{y}^j := \mathbf{p}^j\,, \quad j = 1,\dots,n \quad , \quad \mathbf{y}^0 := \mathbf{p}^n\,. \tag{5.15.8}$$

As a consequence of (5.15.8) the vector coefficients $\mathbf{y}^j$, $j = 0,\dots,n$, can be assumed to be known.

▲

(5-15.b) (30 min.)

In explicit form state the linear system of equations

- whose coefficient matrix and right-hand side vector are to be given in terms of the point locations $\mathbf{p}^j \in \mathbb{R}^2$, $j = 1,\dots,n$ and the knot positions t_k, $k = 0,\dots,n$, and
- whose solution provides the vector $\left[(\mathbf{x}^1)_1, (\mathbf{x}^2)_1, \dots, (\mathbf{x}^n)_1\right]^\top \in \mathbb{R}^n$ of first components of the vector-valued coefficients $\mathbf{x}^j \in \mathbb{R}^2$, $j = 1,\dots,n$, in (5.15.5).

$$\begin{bmatrix} \quad & & \\ & & \\ & & \end{bmatrix} \begin{bmatrix} (\mathbf{x}^1)_1 \\ \vdots \\ \\ \vdots \\ (\mathbf{x}^n)_1 \end{bmatrix} = \begin{bmatrix} \quad \\ \\ \\ \end{bmatrix}\,.$$

SOLUTION of (5-15.b):

We start with the important observation that, assuming the knot sequence $\mathcal{M} := \{0 = t_0 < t_1 < \dots < t_{n-1} < t_n = 1\}$ 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

$$s(t) := (\mathbf{s}(t))_1\,, \quad x_j := \left(\mathbf{x}^j\right)_1\,, \quad j = 1,\dots,n\,, \quad p_j := \begin{cases} (\mathbf{p}^j)_1 & \text{for } j = 1,\dots,n\,, \\ (\mathbf{p}_n)_1 & \text{for } j = 0\,. \end{cases}$$

In light of (5.15.8) we get the local representation

$$s|_{]t_{j-1},t_j[}(t) = p_{j-1}(1-\tau) + x_j h_j \tau(1-\tau) + p_j\tau\,, \quad \tau := \frac{t - t_{j-1}}{h_j}\,, \quad \begin{array}{l} t_{j-1} < t < t_j\,, \\ h_j := t_j - t_{j-1}\,. \end{array} \tag{5.15.9}$$

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

$$s'|_{]t_{j-1},t_j[}(t_j) = -h_j^{-1}p_{j-1} - x_j + h_j^{-1}p_j\,, \qquad s'|_{]t_{j-1},t_j[}(t_{j-1}) = -h_j^{-1}p_{j-1} + x_j + h_j^{-1}p_j\,, \tag{5.15.12}$$

$$s'|_{]t_j,t_{j+1}[}(t_{j+1}) = -h_{j+1}^{-1}p_j - x_{j+1} + h_{j+1}^{-1}p_{j+1}\,, \quad s'|_{]t_j,t_{j+1}[}(t_j) = -h_{j+1}^{-1}p_j + x_{j+1} + h_{j+1}^{-1}p_{j+1}\,. \tag{5.15.13}$$

We plug these expressions into (5.15.10) and obtain linear equations

$$\text{(5.15.10b)} \;\Rightarrow\; -h_1^{-1}p_0 + x_1 + h_1^{-1}p_1 = -h_n^{-1}p_{n-1} - x_n + h_n^{-1}p_n\,, \tag{5.15.14}$$

$$\text{(5.15.10a)} \;\Rightarrow\; -h_j^{-1}p_{j-1} - x_j + h_j^{-1}p_j = -h_{j+1}^{-1}p_j + x_{j+1} + h_{j+1}^{-1}p_{j+1},\ j = 1,\ldots,n-1\,. \tag{5.15.15}$$

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:

$$\begin{bmatrix} 1 & 0 & \ldots & \ldots & 0 & 1 \\ 1 & 1 & 0 & \ldots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \ldots & \ldots & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} (\mathbf{x}^1)_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ (\mathbf{x}^n)_1 \end{bmatrix} = \begin{bmatrix} \frac{(\mathbf{p}^n)_1 - (\mathbf{p}^{n-1})_1}{h_n} - \frac{(\mathbf{p}^1)_1 - (\mathbf{p}^n)_1}{h_1} \\ \frac{(\mathbf{p}^1)_1 - (\mathbf{p}^n)_1}{h_1} - \frac{(\mathbf{p}^2)_1 - (\mathbf{p}^1)_1}{h_2} \\ \frac{(\mathbf{p}^2)_1 - (\mathbf{p}^1)_1}{h_2} - \frac{(\mathbf{p}^3)_1 - (\mathbf{p}^2)_1}{h_3} \\ \vdots \\ \vdots \\ \frac{(\mathbf{p}^{n-1})_1 - (\mathbf{p}^{n-2})_1}{h_{n-1}} - \frac{(\mathbf{p}^n)_1 - (\mathbf{p}^{n-1})_1}{h_n} \end{bmatrix}. \tag{5.15.19}$$

▲

In an object-oriented C++ code, for handling interpolating closed C^1-smooth curves as specified above we rely on the following data type:

C++ code 5.15.20: Class definition of ClosedQuadraticSplineCurve

```
class ClosedQuadraticSplineCurve {
 public:
  // Constructor
  explicit ClosedQuadraticSplineCurve(
      const Eigen::Matrix<double, 2, Eigen::Dynamic> &p);
  ~ClosedQuadraticSplineCurve() = default;
  ClosedQuadraticSplineCurve() = delete;
  ClosedQuadraticSplineCurve(const ClosedQuadraticSplineCurve &) = delete;
  ClosedQuadraticSplineCurve(const ClosedQuadraticSplineCurve &&) = delete;
  ClosedQuadraticSplineCurve &operator=(const ClosedQuadraticSplineCurve &) =
      delete;
  // Point evaluation operator for sorted parameter arguments
  [[nodiscard]] Eigen::Matrix<double, 2, Eigen::Dynamic> curve_points(
      const Eigen::VectorXd &v) const;
  // Curvature evaluation operator for sorted parameter arguments
  [[nodiscard]] Eigen::VectorXd local_curvatures(
      const Eigen::VectorXd &v) const;
  // Approximate computation of the length of the curve
  [[nodiscard]] double length(double rtol = 1E-6) const;

 private:
  [[nodiscard]] bool checkC1(double tol = 1.0E-4) const;
  // Number of points to be interpolated
  unsigned int n_;
  // Coordinates of interpolated points, duplicate:  p^0 = p^n
  Eigen::Matrix<double, 2, Eigen::Dynamic> p_;
  // Knot sequence
  Eigen::VectorXd t_;
  // Coefficients in local representation
  Eigen::Matrix<double, 2, Eigen::Dynamic> x_;
};
```

Get it on GitLab (periodicquadraticsplines.hpp).

(5-15.c) ⚀ (20 min.) [depends on Sub-problem (5-15.b)]

In the file `periodicquadraticsplines.hpp` provide an *efficient* implementation of the constructor

```
ClosedQuadraticSplineCurve::ClosedQuadraticSplineCurve(
const Eigen::Matrix<double, 2, Eigen::Dynamic> &p);
```

which is supposed to initialize the member variable `t_` with the elements of the knot sequence $\mathcal{M}$ and the columns of `x_` with the vector coefficients $\mathbf{x}^j$, $j = 1, \ldots, n$. The matrix argument `p` passes the coordinate vectors $\mathbf{p}^j$, $j = 1, \ldots, n$, in its columns.

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::Matrix<double, 2, Eigen::Dynamic> &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) && "Number of points must be odd!"); //
7    // Save point coordinates, duplicate endpoints
8    p_.col(0) = p.col(n_ - 1);
9    p_.rightCols(n_) = p;
10   // START Student code
11   // Compute knot set by accumulating length of polygon segments
12   Eigen::VectorXd lensum{n_};
13   lensum[0] = (p_.col(1) - p_.col(0)).norm();
14   for (unsigned int l = 1; l < n_; ++l) {
15     lensum[l] = lensum[l - 1] + (p_.col(l + 1) - p_.col(l)).norm();
16   }
17   t_[0] = 0.0;
18   for (unsigned int l = 1; l <= n_; ++l) {
19     t_[l] = lensum[l - 1] / lensum[n_ - 1];
20   }
21   // Lengths of knot intervals
22   const Eigen::VectorXd h = t_.tail(n_) - t_.head(n_);
23 // Compute vector coefficients x^j
24 #ifndef SMART_LSE_SOLUTION
25   // Initialize sparse matrix efficiently, see
26   Eigen::SparseMatrix<double> A(n_, n_);
27   A.reserve(Eigen::RowVectorXi::Constant(n_, 2));
28   A.insert(0, 0) = 1.0;
29   A.insert(0, n_ - 1) = 1.0;
30   for (unsigned int k = 1; k < n_; ++k) {
31     A.insert(k, k) = 1.0;
32     A.insert(k, k - 1) = 1.0;
33   }
34   A.makeCompressed();
35   // Compute LU-decomposition
36   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver(A);
37   if (solver.info() != Eigen::Success) {
38     throw std::runtime_error("Matrix factorization failed!");
39   }
40 #endif
41   // Right-hand side vector for linear system of equations
42   Eigen::VectorXd b(n_);
43   // Treat first and second component
44   for (unsigned int d = 0; d <= 1; ++d) {
45     // Compile right-hand side vector
46     const auto c{p_.row(d)};
47     b[0] = (c[n_] - c[n_ - 1]) / h[n_ - 1] - (c[1] - c[0]) / h[0];
48     for (unsigned int k = 1; k < n_; ++k) {
49       b[k] = (c[k] - c[k - 1]) / h[k - 1] - (c[k + 1] - c[k]) / h[k];
50     }
51 #ifdef SMART_LSE_SOLUTION
52     // Forward elimination in-situ in b
53     for (unsigned int j = 1; j < n_; ++j) {
54       b[j] -= b[j - 1];
55     }
```

```
56      // Backward substitution in-situ in b
57      b[n_ - 1] *= 0.5;
58      int sign = 1;
59      for (unsigned int j = 0; j < n_ - 1; ++j) {
60        b[j] -= (sign * b[n_ - 1]);
61        sign *= -1;
62      }
63      x_.row(d) = b.transpose();
64  #else
65      // Solve sparse linear system
66      x_.row(d) = solver.solve(b).transpose();
67      if (solver.info() != Eigen::Success) {
68        throw std::runtime_error("Elimination failed");
69      }
70  #endif
71    }
72    // END Student code
73  }
```

Get it on GitLab (periodicquadraticsplines.hpp).

The status checks on successful execution of the sparse solver are optional.

Remark. Why does, in Line 6, 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 $\mathbf{b} = [b_1, \dots, b_n]$ and $\mathbf{z} = [z_1, \dots, z_n]$, boils down to

$$z_1 = b_1 \quad , \quad z_j = b_j - z_{j-1} \,, \quad j = 2, \dots, n \,. \quad (5.15.23)$$

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.

▲

(5-15.d) ⚀ (15 min.) Complete the code in `periodicquadraticsplines.hpp` to achieve an *efficient* implementation, that is, with cost scaling linearly in the number of data and in the number n of points, of the member function

```
Eigen::Matrix<double, 2, Eigen::Dynamic>
ClosedQuadraticSplineCurve::curve_points(const Eigen::VectorXd &v)
    const;
```

that takes a *sorted* sequence $(0 \leq v_1 \leq v_2 \leq \cdots \leq v_{N-1} \leq v_N \leq 1)$, $N \in \mathbb{N}$, of parameter values as `v` and returns the sequence $(\mathbf{s}(v_1), \mathbf{s}(v_2), \ldots, \mathbf{s}(v_N))$ as the columns of a $2 \times N$-matrix.

SOLUTION 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::Matrix<double, 2, Eigen::Dynamic>
3  ClosedQuadraticSplineCurve::curve_points(const Eigen::VectorXd &v) const {
4    unsigned int N = v.size();
5    assert(N > 0);
6    // Matrix containing points to be computed
7    Eigen::Matrix<double, 2, Eigen::Dynamic> s(2, N);
8    // Lengths of knot intervals
9    const Eigen::VectorXd h = t_.tail(n_) - t_.head(n_);
10   // START Student code
11   // Run through all the provided parameter values
12   unsigned int knot_idx = 0;
13   for (unsigned int k = 0; k < N; ++k) {
14     assert(((v[k] >= 0.0) && (v[k] < 1.0)) && "Out of range!");
15     if (k > 0) {
16       assert(v[k] >= v[k - 1] && "Parameter values not sorted!");
17     }
18     // Find right knot interval:  knot_idx stores index of knot to the
           right of
19     // current parameter value
20     while ((knot_idx < n_) && (v[k] >= t_[knot_idx])) {
21       knot_idx++;
22     }
23     assert(knot_idx > 0);
24     const double tau = (v[k] - t_[knot_idx - 1]) / h[knot_idx - 1];
25     s.col(k) = ((1.0 - tau) * p_.col(knot_idx - 1)) +
26                (h[knot_idx - 1] * tau * (1 - tau)) * x_.col(knot_idx - 1) +
27                (tau * p_.col(knot_idx));
28   }
```

```
  // END Student code
  return s;
}
```

Get it on GitLab (periodicquadraticsplines.hpp).

The consistency checks for the parameter values `v[k]` are optional.

▲

(5-15.e) ⚄ (20 min.) Devise an *efficient* implementation of the member function

```
Eigen::VectorXd ClosedQuadraticSplineCurve::local_curvatures(
const Eigen::VectorXd &v) const;
```

that takes a *sorted* sequence $(0 \leq v_1 < v_2 < \cdots < v_{N-1} < v_N \leq 1)$, $N \in \mathbb{N}$, of parameter values as `v` and returns a vector of local curvatures of the curve $\mathbf{s}$.

The **curvature** of $\mathbf{s} : [0,1] \to \mathbb{R}^2$ is a function $\kappa : [0,1] \to \mathbb{R}$ given by

$$\kappa(t) := \frac{\det\left[\mathbf{s}'(t), \mathbf{s}_j''(t)\right]}{\|\mathbf{s}'(t)\|_2}, \quad \mathbf{s}_j := \mathbf{s}|_{]t_{j-1},t_j]}, \quad t_{j-1} < t \leq t_j, \quad j = 1,\ldots,n.$$

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 (5-15.d). The $\mathcal{M}$-piecewise constant second derivative is obtained by differentiating (5.15.11) once more using the chain rule:

$$\mathbf{s}''|_{]t_{j-1},t_j[}(t) = -2\mathbf{x}^j, \quad t_{j-1} < t < t_j.$$

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_);
  // START Student code
  // 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) {
```

```
15      assert(((v[k] >= 0.0) && (v[k] < 1.0)) && "Out of range!");
16      if (k > 0) {
17        assert(v[k] >= v[k - 1] && "Parameter values not sorted!");
18      }
19      // Find right knot interval:  knot_idx stores index of knot to the
            right of
20      // current parameter value
21      while ((v[k] >= t_[knot_idx]) && (knot_idx < n_)) {
22        knot_idx++;
23      }
24      assert(knot_idx > 0);
25      const double tau = (v[k] - t_[knot_idx - 1]) / h[knot_idx - 1];
26      // Derivative vector
27      const Eigen::Vector2d ds =
28          (p_.col(knot_idx) - p_.col(knot_idx - 1)) / h[knot_idx - 1] +
29          (1.0 - 2 * tau) * x_.col(knot_idx - 1);
30      // Second derivative vector
31      const Eigen::Vector2d dds = -2.0 * x_.col(knot_idx - 1);
32      // Evaluate formula for signed curvature
33      const double determinant = ds[0] * dds[1] - ds[1] * dds[0];
34      c[k] = determinant / ds.norm();
35    }
36    // END Student code
37    return c;
38  }
```

Get it on GitLab (periodicquadraticsplines.hpp).

▲

(5-15.f) ⚀ Based on approximation of the curve $t \mapsto \mathbf{s}(t)$ by means of polygons, implement the following member function of **ClosedQuadraticSplineCurve**:

```
double ClosedQuadraticSplineCurve::length(double rtol) const;
```

It is supposed to return an approximation of the length of the curve with a relative error bounded by `rtol`.

SOLUTION of (5-15.f):

▲

End Problem 5-15 , 91 min.

Chapter 6

Approximation of Functions in 1D

Problem 6-1: Piecewise linear approximation on graded meshes

One of the main point made in [Lecture → Section 6.2.3] is that the quality of an interpolant depends heavily on the choice of the interpolation nodes. If the function to be interpolated has a "bad behavior" in a small part of the domain, for instance it has very large derivatives of high order, more interpolation points are required in that area of the domain. Commonly used tools to cope with this task are *graded meshes*, which will be the topic of this problem.

Substantial implementation in C++ is requested.

Given a mesh $\mathcal{M} = \{0 = t_0 < t_1 < \cdots < t_{n-1} < t_n = 1\}$ on the unit interval $I = [0,1]$, $n \in \mathbb{N}$, we define the *piecewise linear interpolant*:

$$\mathsf{I}_{\mathcal{M}} : C^0(I) \to \mathcal{P}_{1,\mathcal{M}} = \{s \in C^0(I),\ s_{|[t_{j-1},t_j]} \in \mathcal{P}_1 \ \forall\, j\}, \quad \text{s.t.} \quad (\mathsf{I}_{\mathcal{M}} f)(t_j) = f(t_j), \quad j = 0,\ldots,n \tag{6.1.1}$$

See also [Lecture → Section 5.3.2].

(6-1.a) ⚀ (10 min.) If we choose the uniform mesh $\mathcal{M} = \{t_j\}_{j=0}^n$ with $t_j = j/n$, given a function $f \in C^2(I)$ what is the asymptotic behavior of the error $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ when $n \to \infty$?

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):

$$\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)} \le \frac{1}{2n^2} \|f''\|_{L^\infty(I)}\,, \tag{6.1.2}$$

because the meshwidth is $h = 1/n$. Thus, the convergence is quadratic, i.e. algebraic with order 2:

$$\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)} = O(n^{-2}) \quad \text{for} \quad n \to \infty\,. \tag{6.1.3}$$

▲

(6-1.b) ⚀ (5 min.) What is the smoothness class of the function $f : I := [0,1] \to \mathbb{R}$, $f(t) = t^\alpha$, $0 < \alpha < 2$? In other words, for which $k \in \mathbb{N}$ do we have $f \in C^k(I)$?

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.

▲

(6-1.c) ⚀ (30 min.) Implement a templated C++ function

```
template <typename FUNCTION>
double pwlintpMaxError(FUNCTION &&f, const Eigen::VectorXd &t);
```

that computes $\|f - I_{\mathcal{M}} f\|_{L^\infty(I)}$ when f is passed through the functor `f` (with evaluation operator **double operator** `()(double)const`) and the mesh $\mathcal{M}$ through its *sorted* vector $[t_0, t_1, \ldots, t_n]^\top \in \mathbb{R}^{n+1}$, $n \in \mathbb{N}$, of node positions, which also defines the interval $I := [t_0, t_n]$.

The maximum norm should be approximated by sampling in 10^5 equidistant points in I, see, e.g., [Lecture → Ex. 6.6.1.7].

SOLUTION 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,
```

```
    // then find the interval containing x.
    while (x > tj1) {  // x >= tj always satisfied.
      tj = tj1;
      fj = fj1;
      j++;
      tj1 = t[j + 1];
      fj1 = f(tj1);
    }
    // Now we have tj <= x <= tj1.

    // Evaluate linear interpolant [tj,tj1] at x.
    double lfx = fj + (fj1 - fj) / (tj1 - tj) * (x - tj);
    // Update L-infinity error on [0,x]
    double err = std::abs(f(x) - lfx);
    if (err > maxerr) {
      maxerr = err;
    }
  }
  // END
  return maxerr;
}
```

Get it on GitLab (gradedmeshes.hpp).

▲

(6-1.d) ⚂ (15 min.) [depends on Sub-problem (6-1.c)]

We consider the root functions $f(t) := t^\alpha$, $\alpha > 0$. Implement a C++ function

```
void cvgplotEquidistantMesh(const Eigen::VectorXd &alpha);
```

that uses MATPLOTLIBCPP to create log-log plots of the the error norms $\|f - I_{\mathcal{M}} f\|_{L^\infty(I)}$ as a function of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *equidistant meshes*. Use meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$, and values for α passed in `alpha`. Do not forget axis labels, plot title, and a meaningful legend. Save the plot in `cvgplotequidistand.png`.

Describe your observations qualitatively.

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)
```

```
16   // using a mesh M with #M = N(i)+1.
17   Eigen::MatrixXd Errors(n_meshes, n_alphas);
18   for (unsigned int i = 0; i < n_meshes; ++i) {
19     const unsigned int n = N(i);  // The mesh has #M = (n+1) = (N(i)+1) nodes.
20     Eigen::VectorXd mesh = Eigen::VectorXd::LinSpaced(n + 1, 0.0, 1.0);
21     for (unsigned int j = 0; j < n_alphas; ++j) {
22       // Compute L-infinity error for f_j(t) = t^alpha(j).
23       const double alphaj = alpha(j);
24       auto fj = [alphaj](double t) { return std::pow(t, alphaj); };
25       Errors(i, j) = pwlintpMaxError<std::function<double(double)>>(fj, mesh);
26     }
27   }
28
29   // Define a few line-styles to distinguish plots.
30   // Colours change automatically.
31   constexpr unsigned int n_styles = 4;
32   std::vector<std::string> style;
33   style.push_back("o--");
34   style.push_back("v-.");
35   style.push_back("s:");
36   style.push_back("D-");
37   // Plot errors
38   for (unsigned int j = 0; j < n_alphas; j++) {
39     char lab[20];
40     sprintf(lab, "alpha = %.2f", alpha(j));
41     plt::loglog(N, Errors.col(j), style[j % n_styles], {{"label", lab}});
42   }
43   plt::title("Error of pw. lin. intp. of f(t)=t^alpha on uniform meshes");
44   plt::ylabel("Error");
45   plt::xlabel("Number of mesh intervals");
46   plt::legend("lower left", std::vector<double>());
47   // END
48   plt::savefig("./cx_out/cvgplotequidistant.png");
49 }
```

Get it on GitLab (gradedmeshes.hpp).

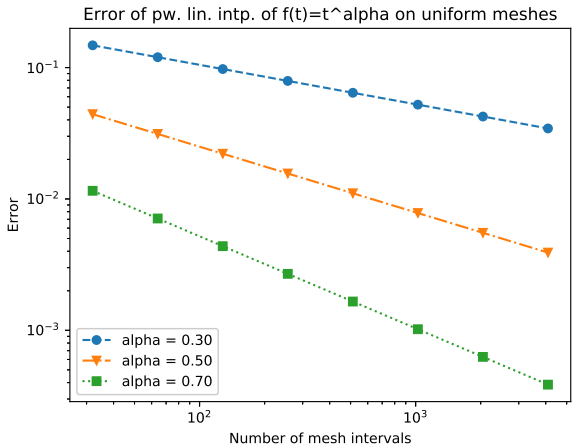


Fig. 51

◁ 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.

▲

(6-1.e) (30 min.) [depends on Sub-problem (6-1.c)]

Create a C++ function

```
Eigen::VectorXd cvgrateEquidistantMesh(const Eigen::VectorXd
```

```
    &alpha);
```

that estimates rates of asymptotic algebraic convergence (→ [Lecture → § 6.2.2.9]) of $\|f - \mathrm{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$ in terms of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *equidistant meshes* for the passed values of the parameter α and returns the estimates. These regression-based estimates should rely on meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$.

HINT 1 for (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()`

```
2  Eigen::VectorXd cvgrateEquidistantMesh(const Eigen::VectorXd &alpha) {
3    const unsigned int n_alphas = alpha.size();
4    Eigen::VectorXd Rates(n_alphas);
5
6    // TODO: (6-1.e) Fill in the entries of Rates.
7    // Hint: For each alpha(j), use polyfit() to estimate convergence
8    // rates of the maximum norm (obtained by pwlintMaxError()) with
9    // number of mesh intervals 32, 64, ..., 4096.
10   // START
11   constexpr unsigned int kmin = 5, kmax = 12;
12   const unsigned int n_meshes = kmax - kmin + 1;
13   // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
14   Eigen::ArrayXd N =
15       Eigen::pow(2, Eigen::ArrayXd::LinSpaced(n_meshes, kmin, kmax));
16
17   // Errors(i,j) = L-infinity error of interpolation of t^alpha(j)
18   // using a mesh M with #M = N(i)+1.
19   Eigen::MatrixXd Errors(n_meshes, n_alphas);
20   for (unsigned int i = 0; i < n_meshes; ++i) {
21     const unsigned int n = N(i);  // The mesh has #M = (n+1) = (N(i)+1) nodes.
22     Eigen::VectorXd mesh = Eigen::VectorXd::LinSpaced(n + 1, 0.0, 1.0);
23     for (unsigned int j = 0; j < n_alphas; ++j) {
24       // Compute L-infinity error for f_j(t) = t^alpha(j).
25       const double alphaj = alpha(j);
26       auto fj = [alphaj](double t) { return std::pow(t, alphaj); };
27       Errors(i, j) = pwlintpMaxError<std::function<double(double)>>(fj, mesh);
28     }
29   }
30
31   for (unsigned int j = 0; j < n_alphas; ++j) {
32     // Estimate convergence rate for f_j(t) = t^alpha(j) by the slope
33     // of line fitted through the errors on a log-log scale.
34     Eigen::VectorXd coeff =
35         polyfit(N.array().log(), Errors.col(j).array().log(), 1);
36     Rates(j) = -coeff[1];
37   }
38   // END
39   return Rates;
40 }
```

Get it on GitLab (gradedmeshes.hpp).

▲

(6-1.f) ⚂ (15 min.) Tabulate and plot the empiric rate of asymptotic n-convergence (in maximum norm) of the piecewise linear interpolant of $f(t) = t^\alpha$, $0 < \alpha < 2$, on *equidistant meshes*. To conduct this experiment implement a C++ function

```
void testcvgEquidistantMesh(void);
```

that plots (use MATPLOTLIBCPP) and tabulates the measured convergence rates based on $\alpha = 0.05, 0.15, 0.25, \ldots, 2.95$. Save the plot in the file `cvgRateEquidistant.png`.

SOLUTION of (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).

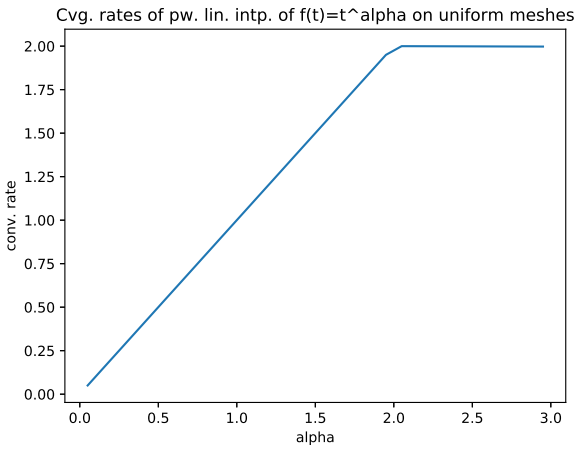


Fig. 52

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.

▲

(6-1.g) ⚂ (5 min.) In which mesh interval do you expect $|f - \mathrm{I}_{\mathcal{M}} f|$ to attain its maximum?

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 → 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.

▲

(6-1.h) ⚄ (20 min.) [depends on Sub-problem (6-1.g)]

Compute ("on paper") the exact value of $\|f - \mathrm{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$, $0 < \alpha < 2$. Compare the order of convergence obtained with the one observed numerically in (6-1.b).

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:

$$\varphi(t) = f(t) - \left(\mathrm{I}_{\mathcal{M}} f\right)(t) = t^\alpha - t\,\frac{1}{n^{\alpha-1}} \quad , \quad (\varphi(0) = \varphi(1/n) = 0)\,,$$

$$\varphi'(t) = \alpha t^{\alpha-1} - \frac{1}{n^{\alpha-1}}\,,$$

$$\blacktriangleright \quad \varphi'(t^*) = 0 \quad \text{if} \quad t^* = \frac{1}{n}\alpha^{-1/(\alpha-1)}\,.$$

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.

▲

(6-1.i) ⚂ (20 min.) Since the interpolation error is concentrated in the left part of the domain, it seems reasonable to use a finer mesh only in this part.

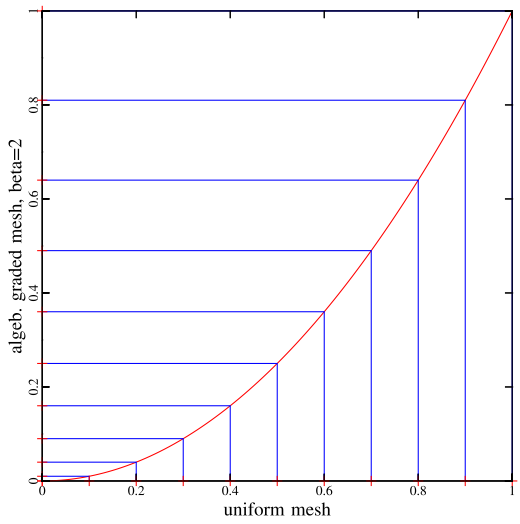


Fig. 53

A suitable choice is an **algebraically graded mesh**, defined as

$$\mathcal{G} = \left\{ t_j = \left(\frac{j}{n}\right)^{\beta}, j = 0, \ldots, n \right\} \tag{6.1.8}$$

for the **grading parameter** $\beta > 1$.

◁ Algebraically graded mesh for $\beta = 2$.

Realize a C++ function

```
Eigen::MatrixXd cvgrateGradedMesh(const Eigen::VectorXd &alpha,
                                  const Eigen::VectorXd &beta);
```

that estimates rates of asymptotic algebraic convergence (→ [Lecture → § 6.2.2.9]) of $\|f - \mathrm{I}_{\mathcal{G}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$ in terms of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *graded meshes* for the passed values of the parameter α (argument `alpha`) and of the grading parameter $\beta > 0$ (argument `beta`). The function should return the measured rates as entries of a matrix, a column for each α-value, a row for each β-value. The regression-based estimates should rely on meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$.

SOLUTION of (6-1.i):

C++ code 6.1.9: Implementation of `cvgrateGradedMesh()`

```
2  Eigen::MatrixXd cvgrateGradedMesh(const Eigen::VectorXd &alpha,
3                                    const Eigen::VectorXd &beta) {
4    const unsigned int n_alphas = alpha.size();
5    const unsigned int n_betas = beta.size();
6    Eigen::MatrixXd Rates(n_betas, n_alphas);
7
8    // TODO: (6-1.i) Fill in the entries of Rates.
9    // START
10   constexpr unsigned int kmin = 5, kmax = 12;
11   const unsigned int n_meshes = kmax - kmin + 1;
12   // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
13   Eigen::ArrayXd N =
14       Eigen::pow(2, Eigen::ArrayXd::LinSpaced(n_meshes, kmin, kmax));
15
16   // Errors(i,k*n_alphas+j) = L-infinity error of interpolation of t^alpha(j)
17   // using a mesh M with #M = N(i)+1 and grading parameter beta(k).
18   Eigen::MatrixXd Errors(n_meshes, n_alphas * n_betas);
19   for (unsigned int i = 0; i < n_meshes; ++i) {
20     const unsigned int n = N(i);  // The mesh has #M = (n+1) = (N(i)+1) nodes.
21     Eigen::VectorXd mesh = Eigen::VectorXd::LinSpaced(n + 1, 0.0, 1.0);
```

```
22      for (unsigned int k = 0; k < n_betas; ++k) {
23        // Transform the mesh using the grading parameter beta.
24        Eigen::VectorXd gradedMesh = pow(mesh.array(), beta(k)).matrix();
25        for (unsigned int j = 0; j < n_alphas; ++j) {
26          // Compute L-infinity error for f_j(t) = t^alpha(j).
27          const double alphaj = alpha(j);
28          auto fj = [alphaj](double t) { return std::pow(t, alphaj); };
29          const unsigned int ind = k * n_alphas + j;
30          Errors(i, ind) =
31              pwlintpMaxError<std::function<double(double)>>(fj, gradedMesh);
32        }
33      }
34    }
35
36    for (unsigned int k = 0; k < n_betas; ++k) {
37      for (unsigned int j = 0; j < n_alphas; ++j) {
38        // Estimate convergence rate for f_j(t) = t^alpha(j) by the slope
39        // of line fitted through the errors on a log-log scale.
40        const unsigned int ind = k * n_alphas + j;
41        Eigen::VectorXd coeff =
42            polyfit(N.array().log(), Errors.col(ind).array().log(), 1);
43        Rates(k, j) = -coeff[1];
44      }
45    }
46    // END
47    return Rates;
48  }
```

Get it on GitLab (gradedmeshes.hpp).

▲

(6-1.j) ⚄ (20 min.) [depends on Sub-problem (6-1.i)]

When using algebraically graded meshes for the approximation of $t \mapsto t^\alpha$, $0 < \alpha < 2$, by means of piecewise linear interpolation, how do you have to choose the grading parameter $\beta = \beta(\alpha)$ as a function of α in order to recover an asymptotic algebraic convergence like $O(n^{-2})$ of $\|f - \mathrm{I}_{\mathcal{G}} f\|_{L^\infty(I)}$ for $\sharp\mathcal{G} \sim n \to \infty$?

To arrive at a conjecture rely on the function `cvgrateGradedMesh()` from Sub-problem (6-1.i) and write another function

```
void testcvgGradedMesh(void);
```

that calls `cvgrateGradedMesh()` for

$$\alpha \in \{0.05, 0.15, 0.25, \ldots, 2.95\}\,,$$
$$\beta \in \{0.1, 0.2, 0.3, \ldots, 2.0\}\,,$$

and creates a useful plot of the data (unsing MATPLOTLIBCPP) and stores it in the file `alphabeta.png`.

SOLUTION of (6-1.j):

C++ code 6.1.10: Implementation of `testcvgGradedMesh()`

```
2  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).

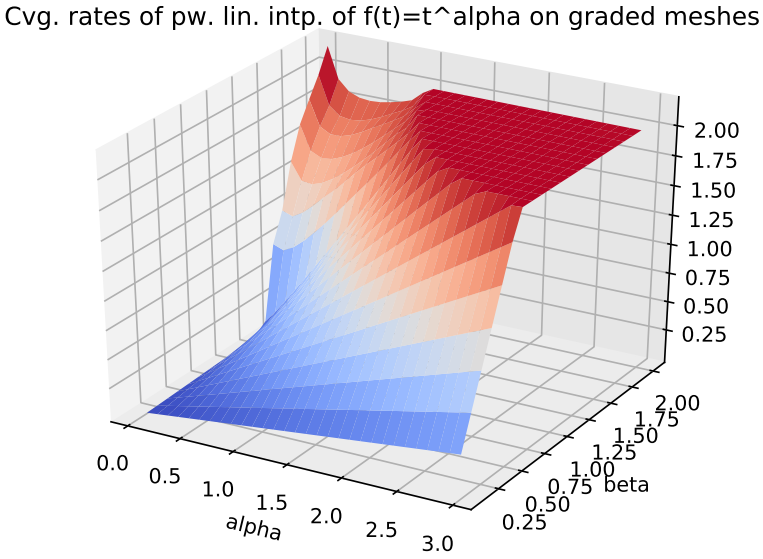


Fig. 54

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.

▲

End Problem 6-1 , 170 min.

Problem 6-2: Piecewise linear interpolation with knots different from nodes

In [Lecture → Ex. 5.1.0.15] we examined piecewise linear interpolation in the case where the interpolation nodes coincide with the abscissas of the corners of the interpolating polygon. However, that one is a very special situation. In this problem we consider a more general setting for piecewise linear interpolation.

Involves moderate implementation in C++.

We are given two ordered sets of real numbers to be read as points on the real line:

(I) the knots $x_0 < x_1 < x_2 < \cdots < x_n$,

(II) the (interpolation) nodes $t_0 < t_1 < \cdots < t_n$, $n \in \mathbb{N}$, satisfying $x_0 \le t_j \le x_n$, $j = 0, \ldots, n$.

The space of piecewise linear continuous functions with respect to the knot set $\mathcal{N} := \{x_j\}_{j=0}^n$ is defined as:

$$\mathcal{S}_{1,\mathcal{N}} := \{s \in C^0([x_0, x_n]) : \ s(t) = \gamma_j t + \beta_j \text{ for } t \in]x_{j-1}, x_j], \ \gamma_j, \beta_j \in \mathbb{R} \ i = 1, \ldots, n\} \ . \tag{6.2.1}$$

Compare [Lecture → Ex. 5.1.0.15] for the special case $x_j := t_j$.

On the function space $\mathcal{S}_{1,\mathcal{N}} \subset C^0(I)$, $I := [x_0, x_n]$, we consider the **interpolation problem**:

Find $s \in \mathcal{S}_{1,\mathcal{N}}$ such that $s(t_i) = y_i$, $i = 0, \ldots, n$, for given values $y_i \in \mathbb{R}$.

This fits the general framework of [Lecture → § 5.1.0.21].

Below we set $I_0 := [x_0, x_1]$, $I_j :=]x_{j-1}, x_{j+1}]$, $j = 1, \ldots, n-1$, $I_n := [x_{n-1}, x_n]$.

(6-2.a) ⚀ Show that the interpolation problem may not have a solution for some values y_j if a single interval $[x_j, x_{j+1}]$, $j = 0, \ldots, n-1$, contains more than 2 nodes t_i.

SOLUTION of (6-2.a):

In general a linear function cannot interpolate three data points.

▲

(6-2.b) ⚁ Show that the interpolation problem can have a unique solution for any data values y_j *only if* each interval I_j, $j = 0, \ldots, n$, contains at least one node.

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 → Fig. 149]. ⌟

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.

▲

(6-2.c) ⚄ Show that the interpolation problem has a unique solution for all data values y_j if each of the closed intervals $]x_0, x_1[,]x_1, x_2[,]x_2, x_3[, \ldots,]x_{n-1}, x_n[$ contains at least one node t_j.

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.

▲

(6-2.d) ⚄ Implement a C++ function

```
VectorXd tentBasCoeff(const VectorXd &x,const VectorXd &t,
                      const VectorXd &y);
```

that returns the vector of values $s(x_j)$ of the interpolant s of the data points (t_i, y_i), $i = 0, \ldots, n$ in $\mathcal{S}_{1,\mathcal{N}}$ in the knots x_j, $j = 0, \ldots, n$. The argument vectors `x`, `t`, and `y` pass the x_j, t_j, and values y_j, respectively.

The function should first check whether the condition formulated in Sub-problem (6-2.c) is satisified. You must not take for granted that knots or nodes are sorted already.

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)

```
2  VectorXd PwLinIP::tentBasCoeff(const VectorXd &x, const VectorXd &t,
3                                 const VectorXd &y) const
4
5  {
6      // Initialization
7      size_t n = t.size();
8      auto x_indices = order(x);
9      auto t_indices = order(t);
10     // You can also implement a solution which does not need
11    // sorted vectors and e.g. for each knot x_j looks
12     // for the closest node t_{i1} and the next closest node t_{i2}.
13     // However, such solution will not become more efficient
14     // if you give as input already sorted vectors:  for each knot x_j
15     // you will always have to iterate along the sorted vector t
16     // to find the included node t_i.
17
18     VectorXd s = VectorXd::Zero(n);
19
20     // Check condition of subproblem 5.5.c
```

```
21      size_t i = 0;
22      size_t k = 0;
23      for(size_t j=0; j<(n-1); ++j) {
24
25         bool intervalOK = false;
26           while(i < n) {
27
28             bool inInterval = (x(x_indices[j]) < t(t_indices[i])) &&
29                               (t(t_indices[i]) < x(x_indices[j+1]));
30
31                if(inInterval) {
32                 intervalOK = true;
33                    if(i == j) { // Index of interval which contains 2
                                   nodes
34                                 // t_i and t_{i+1}.  After that, we have
35                                 // i > j...
36                       k = j;
37                    }
38                    break;
39                } else {
40                    ++i;
41                }
42           }
43         if(!intervalOK) {
44            std::exit(EXIT_FAILURE);
45         }
46      }
47
48      // 1.  Find slope γ and intercept β
49      // in interval with 2 nodes k
50      // 2.  Find s_k and s_{k+1}
51      double gamma = (y(t_indices[k+1]) - y(t_indices[k])) /
52                        (t(t_indices[k+1]) - t(t_indices[k]));
53      double beta = y(t_indices[k]) - gamma * t(t_indices[k]);
54
55      s(x_indices[k])   = gamma * x(x_indices[k])   + beta;
56      s(x_indices[k+1]) = gamma * x(x_indices[k+1]) + beta;
57
58      // Find intercept, slope and value s at the lower bound x
59      // for all intervals on the left of interval with 2 nodes k
60      for(int j=k-1; j>=0; --j) {
61           gamma = (s(x_indices[j+1]) - y(t_indices[j])) /
62                     (x(x_indices[j+1]) - t(t_indices[j]));
63           beta = y(t_indices[j]) - gamma * t(t_indices[j]);
64
65           s(x_indices[j]) = gamma * x(x_indices[j]) + beta;
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
77      return s;
78   }
```

Get it on GitLab (interleaveLIP.cpp).

▲

(6-2.e) ⚂ Implement a C++ class

```
class PwLinIP {
public:
  PwLinIP(const VectorXd &x, const VectorXd &t, const VectorXd &y);
  double operator()(double arg) const;
private:
  ...
};
```

that realizes an interpolator class in the spirit of [Lecture → Rem. 5.1.0.11]. The meanings of the arguments of the constructor are the same as for the function `tentBasCoeff` from Sub-problem (6-2.d). Pay attention to the efficient implementation of the evaluation operator!

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
6       * \param[in] x Vector of knots
7       * \param[in] t Vector of nodes
8       * \param[in] y Vector of values of interpolant in nodes
9       */
10     PwLinIP(const VectorXd &x, const VectorXd &t, const VectorXd &y);
11
12     /*!
13      * \brief operator() evaluate interpolant at arg.
14      */
15     double operator()(double arg) const;
16 private:
17     VectorXd x_;
18     VectorXd t_;
19     VectorXd y_;
20     VectorXd s_;
21     /*!
22      * \brief Compute values of interpolant in knots x from (t_i, y_i)
23      * \param[in] x Vector of knots
24      * \param[in] t Vector of nodes
25      * \param[in] y Vector of values of interpolant in nodes t
26      * \param[out] s Vector of values of interpolant in knots x
27      */
28     VectorXd tentBasCoeff(const VectorXd &x, const VectorXd &t,
29                           const VectorXd &y) const;
```

```
30 };
```

Get it on GitLab (interleaveLIP.cpp).

C++11-code 6.2.4: Implementation of `PwLinIP::PwLinIP` (constructor)

```
2  PwLinIP::PwLinIP(const VectorXd &x, const VectorXd &t,
3                   const VectorXd &y)
4  {
5      assert(t.size() == y.size() && t.size() == x.size() &&
6             "x, t, y must have same size!");
7
8      size_t n = t.size();
9      x_.resize(n);
10     t_.resize(n);
11     y_.resize(n);
12
13     auto x_indices = order(x);
14     for(size_t i=0; i<n; ++i) {
15         x_(i) = x[x_indices[i]];
16     }
17
18     auto t_indices = order(t);
19     for(size_t i=0; i<n; ++i) {
20         t_(i) = t[t_indices[i]];
21         y_(i) = y[t_indices[i]];
22     }
23
24     s_ = tentBasCoeff(x_, t_, y_);
25 }
```

Get it on GitLab (interleaveLIP.cpp).

C++11-code 6.2.5: Implementation of `PwLinIP::operator()`

```
2  double PwLinIP::operator()(double arg) const
3  {
4      if(arg < x_(0) || arg > x_(x_.size()-1)) {
5
6          return 0;
7      } else {
8         std::vector<double> x(x_.size()-1); // Exclude x_0 from search of
              interval including arg
9         VectorXd::Map(&x.front(), x_.size()-1) = x_.tail(x_.size()-1);
10        size_t j = std::lower_bound(x.begin(), x.end(), arg) - x.begin() + 1; //
              Binary search
11        // '+1' is needed to restore indexing from x_0
12
13        double gamma = (s_(j) - s_(j-1)) / (x_(j) - x_(j-1));
14        double beta = s_(j-1) - gamma * x_(j-1);
15
16        return gamma * arg + beta;
17     }
18 }
```

Get it on GitLab (interleaveLIP.cpp).

(6-2.f) ⚄ Implement a C++ code that creates plots of the cardinal basis functions for interpolation in $\mathcal{S}_{1,\mathcal{N}}$

- with know set $\mathcal{N} := \{0,1,2,3,\ldots,9,10\}$
- and for interpolation nodes $t_0 := 0$, $t_j = j - \frac{1}{2}$, $j = 1,\ldots,10$.

Recall that the cardinal basis function $b_k \in \mathcal{S}_{1,\mathcal{N}}$ is characterized by the conditions:

$$b_k(t_j) = \delta_{kj}\,, \quad k,j = 0,\ldots,10 \tag{6.2.6}$$

HINT 1 for (6-2.f): You should use the **Figure** class for plotting. ⌟

SOLUTION of (6-2.f):

C++11-code 6.2.7: Solution of (6-2.f)

```
2  int main() {
3      // Initialization
4      size_t n = 11;
5
6     // Nodes
7      VectorXd x = VectorXd::LinSpaced(n,0,10);
8      VectorXd t(n);
9      t(0) = 0; t.tail(n-1).setLinSpaced(n-1,0.5,9.5);
10
11    mgl::Figure fig;
12    fig.xlabel("t");
13    fig.ylabel("y");
14
15    // Plot colors
16    const unsigned int M = 3;
17    std::vector<std::string> C = {"b", "r", "m"};
18
19    // Loop over basis functions
20    for(unsigned int i = 0; i < M; ++i) {
21        // Basis function e_i
22        VectorXd y = VectorXd::Zero(n);
23        y(i) = 1;
24
25        // Create interpoland with values y
26        PwLinIP cardinalBasis(x, t, y);
27
28        // Plotting values
29        unsigned int N = 1000;
30        VectorXd xval = VectorXd::LinSpaced(N,0,10);
31        VectorXd s(N);
32        for(size_t j=0; j<N; ++j) {
33            s(j) = cardinalBasis(xval(j));
34        }
35
36        std::stringstream ss;
37        ss << "Basis k = " << i;
38        fig.plot(xval, s, C[i].c_str()).label(ss.str().c_str());
39        fig.legend();
40
41        fig.title("Tent basis functions");
42        fig.save("tent_basis_functions.eps");
43    }
44
```

```
45 }
```

Get it on GitLab (interleaveLIP.cpp).

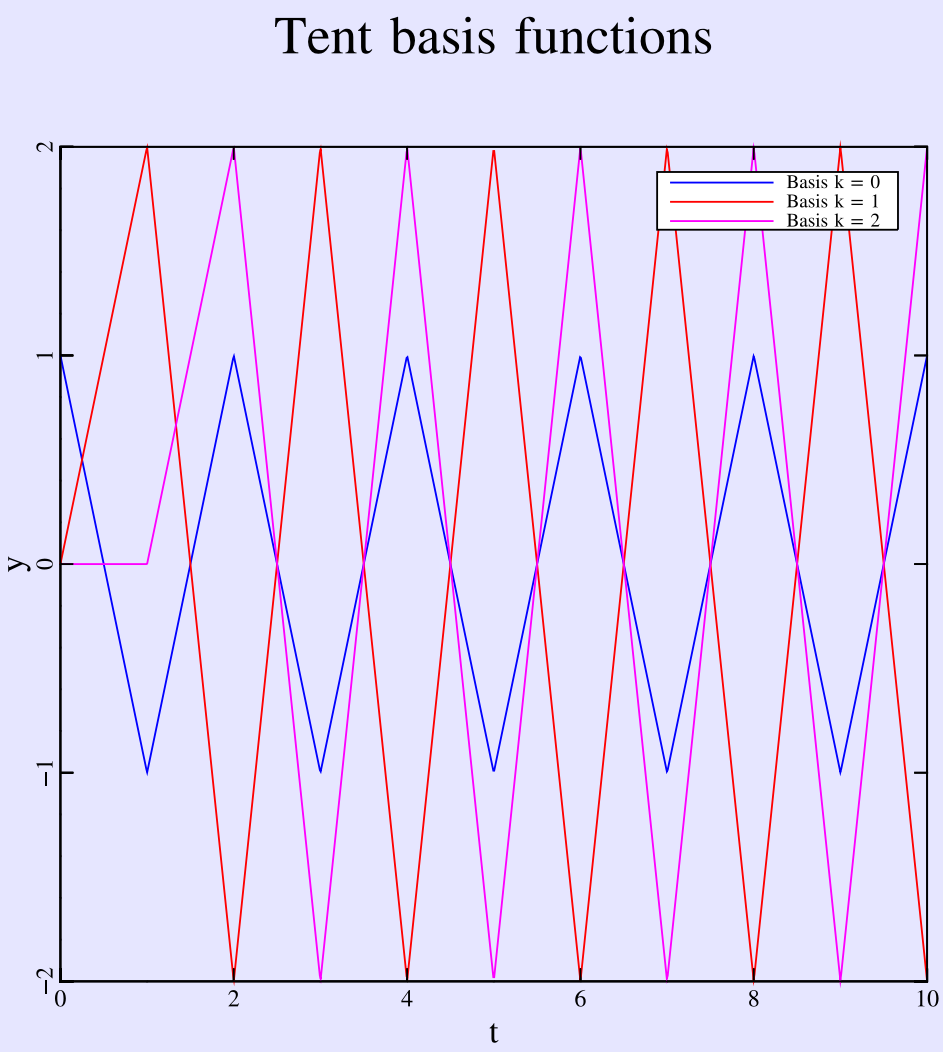


Fig. 55

▲

End Problem 6-2

Problem 6-3: Adaptive polynomial interpolation

In [Lecture → Section 6.2.3] we have seen that the *a priori* placement of interpolation nodes is key to a good approximation by a polynomial interpolant. This problem deals with an *a posteriori* adaptive strategy that controls the placement of interpolation nodes depending on the interpolant. It employs a **greedy algorithm** to grow the node set based on an intermediate interpolant.
This strategy has recently gained prominence for a variety prominence for a wide range of approximation problems, see [**TEM08**].

Considerable implementation in C++ is requested in this problem.

A description of the greedy algorithm for adaptive polynomial interpolation is as follows:

Given a function $f : [a,b] \mapsto \mathbb{R}$ one starts $\mathcal{T}_0 := \{\frac{1}{2}(b+a)\}$. Based on a fixed finite set $\mathcal{S} \subset [a,b]$ of *sampling points* one augments the set of nodes according to

$$\mathcal{T}_{n+1} = \mathcal{T}_n \cup \left\{ \operatorname*{argmax}_{t\in\mathcal{S}} |f(t) - \mathrm{I}_{\mathcal{T}_n}(t)| \right\} , \tag{6.3.1}$$

where $\mathrm{I}_{\mathcal{T}_n}$ is the polynomial interpolation operator (→ [Lecture → Cor. 5.2.2.8]) for the node set $\mathcal{T}_n$, until the relative interpolation error (in maximum norm) drops below a prescribed tolerance $\texttt{tol} > 0$:

$$\max_{t\in\mathcal{S}} |f(t) - \mathrm{I}_{\mathcal{T}_n}(t)| \leq \texttt{tol} \cdot \max_{t\in\mathcal{S}} |f(t)| . \tag{6.3.2}$$

Throughout this exercise you may rely on a C++ function

```
// IN: t:  vector of nodes t_0,...,t_n
// y:  vector of data y_0,...,y_n
// x:  vector of evaluation points x_1,...,x_N
// OUT: p:  interpolant evaluated at x
Eigen::VectorXd intpolyval(const Eigen::VectorXd& t,
                           const Eigen::VectorXd& y,
                           const Eigen::VectorXd& x);
```

that computes the values $p(x_i)$, $i = 1,\ldots,N$, where $p \in \mathcal{P}_n$ is the global polynomial interpolant through the data points (t_j, y_j), $j = 0,\ldots,n$, see [Lecture → Code 5.2.3.7].

C++ code 6.3.3: Function `intpolyval()`

```
2  Eigen::VectorXd intpolyval(const Eigen::VectorXd& t, const Eigen::VectorXd& y,
3                             const Eigen::VectorXd& x) {
4    const unsigned int
5        n = t.size(), // no. of interpolation nodes = deg. of polynomial
             -1
6        N = x.size(); // no. of evaluation points
7
8    Eigen::VectorXd p = Eigen::VectorXd(N);
9
10   // Precompute the weights λ_i with effort O(n^2)
11   Eigen::VectorXd lambda(n);
12   for (unsigned int k = 0; k < n; ++k) {
13     // little workaround: cannot subtract a vector from a scalar
14     // -> multiply scalar by vector of ones
15     lambda(k) =
```

```
16          1. /
17          ((t(k) * Eigen::VectorXd::Ones(k) - t.head(k)).prod() *
18           (t(k) * Eigen::VectorXd::Ones(n - k - 1) - t.tail(n - k - 1)).prod());
19    }
20    // Compute quotient of weighted sums of λ_i/(t-t_i),
21    // effort O(n)
22    for (unsigned int i = 0; i < N; ++i) {
23      Eigen::VectorXd z = (x(i) * Eigen::VectorXd::Ones(n) - t);
24
25      // check if we want to evaluate at a node <-> avoid division by zero
26      double* ptr = std::find(z.data(), z.data() + n, 0.0);
27      if (ptr != z.data() + n) {  // if ptr = z.data + n = z.end no zero was
            found
28        p(i) = y(ptr - z.data());  // ptr - z.data gives the position of the
              zero
29      } else {
30        Eigen::VectorXd mu = lambda.cwiseQuotient(z);
31        p(i) = (mu.cwiseProduct(y)).sum() / mu.sum();
32      }
33    }
34    return p;
35  }
```

(6-3.a) ⚀ (60 min.) Write a C++ function

```
template <class Function>
Eigen::VectorXd adaptivepolyintp(Function&& f,
                                 double a, double b,
                                 double tol, int N);
```

that implements the algorithm described above.

The function arguments are: the functor object `f`, the interval bounds `a`, `b`, the relative tolerance `tol`, the number `N` of *equidistant* sampling points, that is

$$\mathcal{S} := \{a + (b-a)j/N : j = 0, \ldots, N\}\,, \tag{6.3.4}$$

and `t` will be used as return parameter for interpolation nodes (i.e. the final set $\mathcal{T}_n$). The type **Function** defines a function and is supposed to provide an operator **double operator**`()(`**double**`)` **const**. A suitable lambda function can satisfy this requirement. The function should return the final set of interpolation nodes as proposed by the greedy algorithm.

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`.

▲

(6-3.b) ⚀ (15 min.) [depends on Sub-problem (6-3.a)]

Extend the function `adaptivepolyintp()` from Sub-problem (6-3.a) to

```
template <class Function>
  Eigen::VectorXd
    adaptivepolyintp(Function &&f,
      double a, double b,
      double tol, int N,
      std::vector<double> *errortab = nullptr);
```

so that it also reports (and returns in `errortab`) the quantity

$$\epsilon_n := \max_{t\in\mathcal{S}} |f(t) - \mathsf{T}_{\mathcal{T}_n}(t)| \tag{6.3.5}$$

for each intermediate node set $\mathcal{T}_n$.

SOLUTION of (6-3.b):

C++ code 6.3.6: Implementation of function `adaptivepolyintp()`

```
2  template <class Function>
3  Eigen::VectorXd adaptivepolyintp(Function&& f, double a, double b, double tol,
4                                   int N,
5                                   std::vector<double>* errortab = nullptr) {
6    // Generate sampling points and evaluate f there
7    Eigen::VectorXd sampling_points = Eigen::VectorXd::LinSpaced(N, a, b),
8                    fvals_at_sampling_points = sampling_points.unaryExpr(f);
9
10   // TODO: (6-3.a) Implement the greedy algorithm for adaptive interpolation.
11   // Ignore the errortab part of this function for now.
12   // TODO: (6-3.b) save the error in errortab
13   // START
14   // Approximate max|f(x)|
15   const double maxf = fvals_at_sampling_points.cwiseAbs().maxCoeff();
16
17   // Adaptive mesh (initial node)
18   std::vector<double> t{(a + b) / 2.},          // Set of interpolation nodes
19       y{static_cast<double>(f((a + b) / 2.))},  // Values at nodes
20       errors;                                   // Error at nodes
21
22   for (std::size_t i = 0; i < N; ++i) {
23     // *** Step 1: interpolate with current nodes
24     //   need to convert std::vector to
25     //   Eigen::VectorXd to use the function intpolyval
26     Eigen::Map<Eigen::VectorXd> te(t.data(), t.size());
27     Eigen::Map<Eigen::VectorXd> ye(y.data(), y.size());
28     Eigen::VectorXd intpolyvals_at_sampling_points =
29         intpolyval(te, ye, sampling_points);
30
31     // *** Step 2: find node where error is the largest
32     Eigen::VectorXd err =
33         (fvals_at_sampling_points - intpolyvals_at_sampling_points).cwiseAbs();
34
35     // We use an Eigen "Visitor"
36     //
       https://eigen.tuxfamily.org/dox/group__TutorialReductionsVisitorsBroadcasting.html
37     int idx = 0;
38     const double max = err.maxCoeff(&idx);
39
40     // Step 3: check termination criteria
41     if (max < tol * maxf) {
```

```
42        return te;
43      }
44      // Step 4: add this node to our set of nodes
45      t.push_back(sampling_points(idx));
46      y.push_back(fvals_at_sampling_points(idx));
47      // (7-3.b): save error
48      if (errortab != nullptr) {
49        errortab->push_back(max);
50      }
51    }
52    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).

▲

(6-3.c) ⚀ (30 min.) [depends on Sub-problem (6-3.b)]

For $f_1(t) := \sin(e^{2t})$ and $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$ write a C++ function

```
void plotInterpolationError();
```

that creates (employing MATPLOTLIBCPP) and stores (in `intperrplot.png`) a plot of ϵ_n versus n (the number of interpolation nodes). Choose axis scales that reveal the qualitative decay (types of convergence as given in [Lecture → Def. 6.2.2.7]) of this error norm as the number of interpolation nodes is increased. Use interval $[a,b] = [0,1]$, `N=1000` sampling points, tolerance `tol = 1e-6`.

SOLUTION of (6-3.c):

C++ code 6.3.7: Function `plotInterpolationError()`

```
2  void plotInterpolationError() {
3    // Declare test functions
4    auto f1 = [](double t) { return std::sin(std::exp(2 * t)); };
5    auto f2 = [](double t) { return std::sqrt(t) / (1 + 16 * t * t); };
6
7    // Test interval
8    constexpr double a = 0, b = 1;
9    // Get interpolation nodes
10   constexpr unsigned int N = 1000;  // no. of sampling points
11   constexpr double tol = 1e-6;      // tolerance
12
13   plt::figure();
14
15   // TODO: (6-3.c) generate the plots of error vs number of nodes for f1, f2
16   // START
17   Eigen::VectorXd tf1, tf2;       // nodes for f1 resp. f2
18   std::vector<double> ef1, ef2;  // errors for f1 resp. f2
19
20   tf1 = adaptivepolyintp(f1, a, b, tol, N, &ef1);
21   tf2 = adaptivepolyintp(f2, a, b, tol, N, &ef2);
22   Eigen::VectorXd n1 = Eigen::VectorXd::LinSpaced(ef1.size(), 1, ef1.size());
23   Eigen::VectorXd n2 = Eigen::VectorXd::LinSpaced(ef2.size(), 1, ef2.size());
```

```
24    // Plot
25    plt::title("Error VS step");
26    plt::xlabel("No. of interpolation nodes");
27    plt::ylabel("max|f(t) - I_Tf(t)|");
28    plt::semilogy(n1, ef1, "ro", {{"label", "f_1(t) = sin(e^{2t})"}});
29    plt::semilogy(n2, ef2, "bo", {{"label", "f_2(t) = sqrtt/(1+16t^2)"}});
30    plt::legend();
31    // END
32
33    plt::savefig("./cx_out/intperrplot.png");
34  }
```

Get it on GitLab (adaptivepolyintp.hpp).

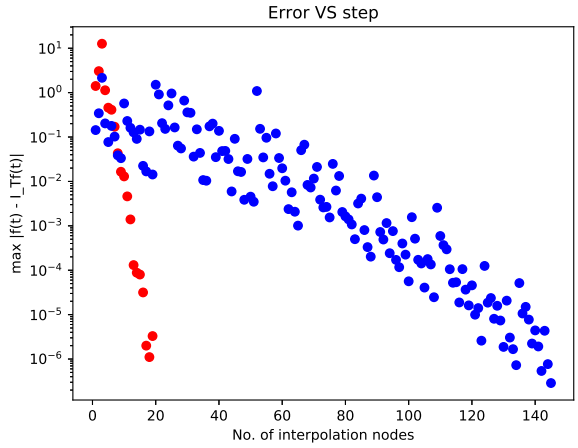


Fig. 56

- $\hat{=} f_1(t) := \sin(e^{2t})$
- $\hat{=} f_2(t) = \frac{\sqrt{t}}{1+16t^2}$

The error data points are approximately located on straight lines in the linear-logaritmic error plots.



Exponential convergence

▲

End Problem 6-3 , 105 min.

Problem 6-4: Chebyshev polynomials and their properties

Chebyshev polynomials as defined in [Lecture → Def. 6.2.3.3] are a sequence of orthogonal polynomials, which can be defined recursively, see [Lecture → Thm. 6.2.3.4]. In this problem we will examine these polynomials and a few of their many properties.

Involves a little implementation in C++

Let $T_n \in \mathcal{P}_n$ be the n-th Chebyshev polynomial, as defined in [Lecture → Def. 6.2.3.3] and $\xi_0^{(n)}, \ldots, \xi_{n-1}^{(n)}$ be the n zeros of T_n, the Chebychev nodes. According to [Lecture → Eq. (6.2.3.10)], these are given by:

$$\xi_j^{(n)} = \cos\left(\frac{2j+1}{2n}\pi\right), \quad j = 0, \ldots, n-1 . \tag{6.4.1}$$

Recall the notion of a "almost" inner product.

Definition [Lecture → Def. 6.3.1.1]. (Semi-)inner product [NIS02]

Let V be a vector space over the field $\mathbb{K}$. A mapping $b : V \times V \to \mathbb{K}$ is called an inner product on V, if it satisfies

(i) b is linear in the first argument: $b(\alpha v + \beta w, u) = \alpha b(v,u) + \beta b(w,u)$ for all $\alpha, \beta \in \mathbb{K}$, $u, v, w \in V$,
(ii) b is (anti-)symmetric: $b(v,w) = \overline{b(w,v)}$ ($\overline{\ } \hat{=}$ complex conjugation),
(iii) b is positive definite: $v \neq 0 \quad \Leftrightarrow \quad b(v,v) > 0$.

b is a semi-inner product, if it still complies with (i) and (ii), but is only positive semi-definite: $b(v,v) \geq 0$ for all $v \in V$.

We define the family of discrete L^2 semi-inner products [Lecture → Eq. (6.3.2.4)]:

$$(f,g)_n := \sum_{j=0}^{n-1} f(\xi_j^{(n)}) g(\xi_j^{(n)}), \quad f,g \in C^0([-1,1]) . \tag{6.4.2}$$

We also define the special weighted L^2 inner product:

$$(f,g)_w := \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} f(t)g(t)\,\mathrm{d}t \quad f,g \in C^0([-1,1]) . \tag{6.4.3}$$

(6-4.a) ⚄ (10 min.) Show that the weight function in (6.4.3) $w(t) := \left(1-t^2\right)^{-\frac{1}{2}}$ satisfies $\int_{-1}^{1} |w(t)|\,\mathrm{d}t < \infty$.

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

$$\int_{-1}^{1} \frac{1}{\sqrt{1-t^2}}\,\mathrm{d}t = \int_{-\frac{1}{2}\pi}^{\frac{1}{2}\pi} \frac{\cos s}{|\cos s|}\,\mathrm{d}s = \pi .$$

▲

(6-4.b) ⚀ (15 min.) [depends on Sub-problem (6-4.a)]

Show that the Chebyshev polynomials are an orthogonal family of polynomials with respect to the inner product defined in (6.4.3) in the sense of [Lecture → Def. 6.3.2.7]:

$$(T_k, T_\ell)_w = 0 \quad \text{for all} \quad k, \ell \in \mathbb{N}_0 \,, \quad k \neq \ell \,. \tag{6.4.4}$$

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 $k - l \neq 0$.

▲

Recall the concept of an orthonormal basis:

Definition [Lecture → Def. 6.3.1.14]. Orthonormal basis

A subset $\{b_1, \ldots, b_N\}$ of an N-dimensional vector space V with inner product (→ **??**) $(\cdot,\cdot)_V$ is an orthonormal basis (ONB), if $\left(b_k, b_j\right)_V = \delta_{kj}$.

A basis of $\{b_1, \ldots, b_N\}$ of V is called orthogonal, if $\left(b_k, b_j\right)_V = 0$ for $k \neq j$.

The following is a surprising result.

Theorem 6.4.5. Discrete orthogonality of Chebychev polynomials

The family of polynomials $\{T_0, \ldots, T_n\}$ is an orthogonal basis [Lecture → Def. 6.3.1.14] of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$ defined in (6.4.2).

(6-4.c) ⚀ (10 min.) Show that for every $n \in \mathbb{N}_0$ $(\cdot,\cdot)_{n+1}$ as defined in (6.4.2) is an inner product on the (real) vector space $\mathcal{P}_n$ of polynomials of degree $\leq n$.

SOLUTION 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 → Thm. 5.2.2.7] a polynomial of degree $\leq n$ that vanishes in $n+1$ distinct points must be zero.

▲

(6-4.d) ⚀ Write a C++ function

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

meant to test the assertion of Thm. 6.4.5 numerically.

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].

C++11-code 6.4.6: Code [Lecture → Code 6.2.3.6]

```
2 vector<double> chebpolmult(const int &n,const double &x)
3 {
4     vector<double> V={1,x};
5     for (int k=1; k<n; k++)
6         V.push_back(2*x*V[k]-V[k-1]);
7     return V;
8 }
```

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 (**??**).

C++ code 6.4.7: Solution of (6-4.d)

```
2  // Check the orthogonality of Chebyshev polynomials
3      n=10;
4      vector<double> V;
5      Eigen::MatrixXd scal(n+1,n+1);
6      for (int j=0; j<n+1; j++) {
7          V=chebpolmult(n,cos(M_PI*(2*j+1)/2/(n+1)));
8          for (int k=0; k<n+1; k++) scal(j,k)=V[k];
9      }
10     cout<<"Scalar products: "<<endl;
11     for (int k=0; k<n+1; k++)
12         for (int l=k+1; l<n+1; l++)
13             cout<<scal.col(k).dot(scal.col(l))<<endl;
```

Get it on GitLab (chebpolyproperties.cpp).

▲

(6-4.e) ⚄ Prove Thm. 6.4.5.

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 $k, l = 0, \ldots, n+1$ with $k \neq l$, by (6.4.1) we have:

$$\begin{aligned}(T_k, T_l)_{n+1} &= \sum_{j=0}^{n} T_k\left(\zeta_j^{(n)}\right) T_l\left(\zeta_j^{(n)}\right) \\ &= \sum_{j=0}^{n} \cos\left(k\frac{2j+1}{2(n+1)}\pi\right)\cos\left(l\frac{2j+1}{2(n+1)}\pi\right) \\ &= \frac{1}{2}\sum_{j=0}^{n}\left[\cos\left((k+l)\frac{2j+1}{2(n+1)}\pi\right) + \cos\left((k-l)\frac{2j+1}{2(n+1)}\pi\right)\right]\end{aligned}$$

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:

$$\sum_{j=0}^{n} \cos\left(m\frac{2j+1}{2(n+1)}\pi\right) = \operatorname{Re}\left(\sum_{j=0}^{n} e^{\imath m\frac{2j+1}{2(n+1)}\pi}\right) = \operatorname{Re}\left(e^{\imath m\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{\imath m\frac{j}{n+1}\pi}\right).$$

Finally, by the standard formula for the geometric sum we have:

$$\begin{aligned}e^{\imath m\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{\imath m\frac{j}{n+1}\pi} &= e^{\imath m\frac{1}{2(n+1)}\pi}\frac{1-e^{\imath m\frac{n+1}{n+1}\pi}}{1-e^{\imath m\frac{1}{n+1}\pi}} \\ &= \frac{1-e^{\imath m\pi}}{e^{-\imath m\frac{1}{2(n+1)}\pi} - e^{im\frac{1}{2(n+1)}\pi}} = -\frac{1-e^{\imath m\pi}}{2}\frac{1}{\imath\sin\left(m\frac{1}{2(n+1)}\pi\right)} \in \imath\cdot\mathbb{R}\,.\end{aligned}$$

This implies that

$$\operatorname{Re}(e^{\imath m\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{\imath m\frac{j}{n+1}\pi}) = 0\,,$$

as desired.

▲

(6-4.f) ⚀ (20 min.) [depends on Sub-problem (6-4.c) and Thm. 6.4.5]

Given $n, m \in \mathbb{N}$, $m \leq n$, and a function $f \in C^0([-1,1])$, find an expression for the best approximant $q_m \in \mathcal{P}_m$ of f in the discrete L^2-norm:

$$q_m = \operatorname{argmin}_{p \in \mathcal{P}_m} \|f - p\|_{n+1} , \tag{6.4.9}$$

where $\|\cdot\|_{n+1}$ is the norm induced by the scalar product $(\cdot,\cdot)_{n+1}$ from (6.4.2). You should represent q_m through a Chebychev expansion of the form:

$$q_m = \sum_{j=0}^{m} \alpha_j T_j , \quad \alpha_j \in \mathbb{R} , \tag{6.4.10}$$

see also [Lecture → Section 6.2.3.3].

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 (→ 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 → 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

$$q_m = \sum_{j=0}^{m} (f, T_j/\lambda_j)_{n+1} \frac{T_j}{\lambda_j} = \sum_{j=0}^{m} \alpha_j T_j ,$$

$$\blacktriangleright \qquad \alpha_j := \frac{1}{n+1} \begin{cases} (f, T_0)_{n+1} & \text{if } j = 0 , \\ 2(f, T_j)_{n+1} & \text{for } j = 1, \ldots, m . \end{cases} \tag{6.4.12}$$

▲

(6-4.g) ⚂ (10 min.) [depends on Sub-problem (6-4.f)]

Give an interpretation of q_m as defined through (6.4.9) in the case $m = n$.

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,

$$q_n \in \mathcal{P}_n: \quad q_n(\xi_j^{(n+1)}) = f(\xi_j^{(n+1)}) \quad \forall j = 0, \ldots, n \,.$$

▲

(6-4.h) ⚂ (20 min.) [depends on Sub-problem (6-4.f)]

Write a C++ function

```
template <typename Function>
  Eigen::VectorXd
    bestpolchebnodes(const Function &f,
                     unsigned int n, unsigned int m);
```

that returns the vector of coefficients α_j, $j = 0, \ldots, m$, in (6.4.10) given the function f. The degree m of the polynomial and the number $n+1$ of the Chebychev nodes entering the inner product are passed through the arguments `m` and `n`. The input `f` is a lambda-function of signature `std::function<double(double)>`, for instance,

```
auto f = [] (double & x) {return 1/(pow(5*x,2)+1);};
```

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  void bestpolchebnodes(const Function &f, Eigen::VectorXd &alpha) {
4      int n=alpha.size()-1;
5      Eigen::VectorXd fn(n+1);
6      for (int k=0; k<n+1; k++) {
7          double temp=cos(M_PI*(2*k+1)/2/(n+1));
8          fn(k)=f(temp);
9      }
10
11     vector<double> V;
12     Eigen::MatrixXd scal(n+1,n+1);
13     for (int j=0; j<n+1; j++) {
14         V=chebpolmult(n,cos(M_PI*(2*j+1)/2/(n+1)));
15         for (int k=0; k<n+1; k++) scal(j,k)=V[k];
16     }
17
18     for (int k=0; k<n+1; k++) {
19         alpha(k)=0;
20         for (int j=0; j<n+1; j++) {
21             alpha(k)+=2*fn(j)*scal(j,k)/(n+1);
22         }
```

```
    }
    alpha(0)=alpha(0)/2;
}
```

Get it on GitLab (chebpolyproperties.cpp).

▲

(6-4.i) ⚀ (20 min.) [depends on Sub-problem (6-4.h)]

We test the implementation of `bestpolchebnodes()` with the function $f(x) = \frac{1}{(5x)^2+1}$. To that end write a C++ function

```
std::vector<double> testBestPolyChebNodes(unsigned int n);
```

that takes the integer n as argument and both tabulates and returns the sequence of the approximation error norms $\|f - q_m\|_{L^\infty(]-1,1[)}$, $m = 0,\ldots,n$, q_m as in (6.4.9).

Approximate the supremum norm of the approximation error by sampling on an equidistant grid with 10^6 points.

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)

```
// Test the implementation
    auto f = [] (double & x) {return 1/(pow(5*x,2)+1);};
    n=20;
    Eigen::VectorXd alpha(n+1);
    bestpolchebnodes(f, alpha);

    // Compute the error
    Eigen::VectorXd X = Eigen::VectorXd::LinSpaced(1e6,-1,1);
    auto qn = [&alpha,&n] (double & x) {
        double temp=0;
        vector<double> V=chebpolmult(n,x);
        for (int k=0; k<n+1; k++) temp+=alpha(k)*V[k];
        return temp;
    };
    double err_max=abs(f(X(0))-qn(X(0)));
    for (int i=1; i<1e6; i++) err_max=std::max(err_max,abs(f(X(i))-qn(X(i))));
    cout<<"Error: "<< err_max <<endl;
```

Get it on GitLab (chebpolyproperties.cpp).

The output (plotted in MATLAB) is shown in Fig. 57.

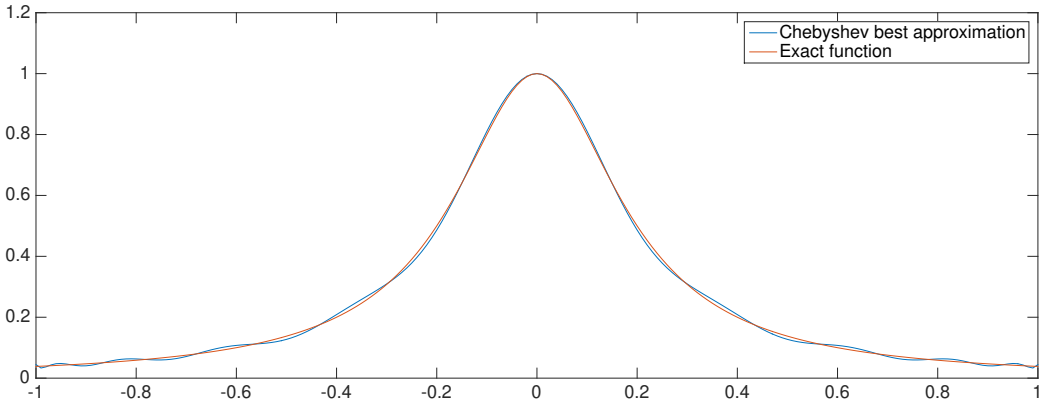


Fig. 57

▲

(6-4.j) ⚀ (20 min.) [depends on Sub-problem (6-4.f)]

Let $L_j \in \mathcal{P}_n$, $j = 0,\ldots,n$, be the Lagrange polynomials [Lecture → § 5.2.2.3] associated with the Chebychev nodes $t_j := \xi_j^{(n+1)}$ in $[-1,1]$. Show that:

$$L_j(t) = \frac{1}{n+1} + \frac{2}{n+1}\sum_{l=1}^{n} T_l(\xi_j^{(n+1)})T_l(t) \quad , \quad j = 0,\ldots,n \, . \tag{6.4.15}$$

HINT 1 for (6-4.j): Note that (6.4.15) is a Chebychev expansion. Then use the result of Sub-problem (6-4.f) to express the coefficients of the Chebychev expansions of L_j. ⌟

SOLUTION of (6-4.j):

We have already seen that $\{T_l/\lambda_l : l = 0,\ldots,n\}$ is an $(\cdot,\cdot)_{n+1}$-ONB of $\mathcal{P}_n$. Exploiting the defining property of Lagrange polynomials

$$L_j(\xi_\ell^{(n+1)}) = \delta_{k,\ell} \, , \quad k,\ell \in \{0,\ldots,n\} \, ,$$

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).

▲

End Problem 6-4 , 125 min.

Problem 6-5: Chebychev interpolation of analytic functions

This problem concerns Chebychev interpolation as discussed in [Lecture → Section 6.2.3] and takes a close look at the interpolation error estimates derived for analytic interpolands in [Lecture → Section 6.2.2.3].

This problem is closely connected to [Lecture → Rem. 6.2.3.26] and involves implementation in C++ based on EIGEN. You are expected to be able to compute with complex numbers.

Using techniques from complex analysis, notably the residue theorem [Lecture → Thm. 6.2.2.50], in class we derived an expression for the interpolation error [Lecture → Eq. (6.2.2.57)] and from it an error bound [Lecture → Eq. (6.2.2.58)], as much sharper alternative to [Lecture → Thm. 6.2.2.15] and [Lecture → Lemma 6.2.2.20] for *analytic* interpolands. The bound tells us that for all $t \in [a,b]$

$$|f(t) - \mathsf{L}_{\mathcal{T}} f(t)| \leq \left| \frac{w(x)}{2\pi i} \int_\gamma \frac{f(z)}{(z-t)w(z)} dz \right| \leq \frac{|\gamma|}{2\pi} \frac{\max_{a\leq\tau\leq b}|w(\tau)|}{\min_{z\in\gamma}|w(z)|} \frac{\max_{z\in\gamma}|f(z)|}{d([a,b],\gamma)} ,$$

where $d([a,b],\gamma)$ is the geometric distance of the integration contour $\gamma \subset \mathbb{C}$ from the interval $[a,b] \subset \mathbb{C}$ in the complex plane. The contour γ must be contractible in the domain D of analyticity of f and must wind around $[a,b]$ exactly once, see [Lecture → Fig. 212].

Now we consider the interval $[-1,1]$. Following [Lecture → Rem. 6.2.3.26], our task is to find an upper bound for this expression, in the case where f possesses an analytical extension to a complex neighbourhood of $[-1,1]$.

For the analysis of the Chebychev interpolation of analytic functions we used the elliptical contours, see [Lecture → Fig. 226],

$$\gamma_\rho(\theta) := \cos(\theta - i\log(\rho)) , \quad \forall 0 \leq \theta \leq 2\pi , \quad \rho > 1 . \tag{6.5.1}$$

(6-5.a) ⚂ (20 min.) Implement a C++ function

```
double lengthEllipse(double rho, unsigned int N);
```

that approximates the length of the ellipse γ_ρ by sampling (6.5.1) for $N \in \mathbb{N}$ (equidistant) arguments θ and returns the result.

SOLUTION 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_ρ
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);
```

```
  // Calculating distance between last and current point on ellipse for all
  // points
  complex last = f(eval_points(0));
  for (unsigned int i = 0; i < N; ++i) {
    complex current = f(eval_points(i + 1));
    length += std::abs(current - last);
    last = current;
  }

  // Alternative method using an approximation over the integral of the length
  // of \gamma_p'.
  // df = |\gamma_p'(\theta)|
  double length_alt = 0.0;
  auto df = [rho](long double theta) {
    return 0.5 * sqrt(rho * rho + 1 / (rho * rho) - 2 * cos(2 * theta));
  };
  double dtheta = 2 * M_PI / N;
  double theta_i = 0;
  for (unsigned int i = 0; i < N; ++i) {
    length_alt += df(theta_i);
    theta_i += dtheta;
  }
  length_alt *= dtheta;
  // END
  return length;
}
```

Get it on GitLab (chebyApprox.hpp).

▲

From now on we consider the S-curve function (the logistic function):

$$f(t) := \frac{1}{1+e^{-3t}}\,, \quad t \in \mathbb{R}\ bdot \tag{6.5.3}$$

(6-5.b) ⚀ (20 min.) Determine the maximal domain of analyticity of the extension of f to the complex plane $\mathbb{C}$.

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 → 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

$$\begin{aligned} -1 = \exp(z) = \exp(a+ib) = \exp(a)\exp(ib) \quad &\Leftrightarrow \quad a = 0,\ b \in 2\pi\mathbb{Z} + \pi \\ -1 = \exp(z) \quad &\Leftrightarrow \quad z \in i(2\pi\mathbb{Z} + \pi) \\ -1 = \exp(-3z) \quad &\Leftrightarrow \quad z \in \frac{i(2\pi\mathbb{Z} - \pi)}{3} . \end{aligned}$$

▲

(6-5.c) ⚁ (15 min.) [depends on Sub-problem (6-5.b)]

Find the least upper bound for the set of ellipse parameters

$$\{\rho > 1 : \gamma_\rho \subset D\} ,$$

where $D \subset \mathbb{C}$ is the domain of analyticity found in Sub-problem (6-5.b).

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:

$$\begin{aligned} &\cos(\pi/2 - i\log(\rho_{\max})) = 1/3\pi i \\ &\Updownarrow \\ &\sinh(\log(\rho_{\max})) = \pi/3 \quad \Leftrightarrow \quad \rho_{\max} = \exp(\sinh^{-1}(\pi/3)) . \end{aligned}$$

▲

(6-5.d) ⚂ (20 min.) [depends on Sub-problem (6-5.c)]

Write a C++ function

```
std::pair<double, double> bestBound();
```

that computes an approximation M of

$$\min_{\rho>1} \frac{|\gamma_\rho| \max_{z\in\gamma_\rho} |f(z)|}{\pi\, d([-1,1], \gamma_\rho)} , \tag{6.5.4}$$

by sampling 1000 equidistant ρ-values in the interval $(1, \rho_{\max}]$, where $\rho_{\max}$ is the value obtained in Sub-problem (6-5.c). For sampling along the integration path γ_ρ use 1000 points corresponding to equidistant θ-values, *cf.* Sub-problem (6-5.a). The function should return the pair $(M, \rho) \in \mathbb{R}^2$, where ρ stands for the ρ-value for which the minimum is attained.

The distance of $[a, b]$ from γ_ρ is formally defined as

$$d([a,b], \gamma) := \inf\{|z - t| \mid z \in \gamma, t \in [a,b]\} . \tag{6.5.5}$$

HINT 1 for (6-5.d): Apply geometric reasoning to establish that the distance of γ_ρ and $[-1,1]$ is $\frac{1}{2}(\rho+\rho^{-1})-1$. ⌟

SOLUTION of (6-5.d):

C++11-code 6.5.6: Code for `bestBound`.

```
2  std::pair<double, double> bestBound() {
3    const double maxRho = exp(asinh(M_PI / 3.));
4    constexpr unsigned int N = 1000;
5    double m = 0.0, r = 0.0;  // min and rho for output
6    // TODO: (6-5.d) Compute (6.5.4) with d from (6.5.5) and
7    // |γ_ρ| from (6-5.a)
8    // START
9    // define f and gamma
10   const complex i(0, 1);  // imaginary unit
11   auto f = [](complex z) { return 1. / (1. + std::exp(-3. * z)); };
12   auto gamma = [i](double rho, double theta) {
13     return std::cos(theta - i * std::log(rho));
14   };
15
16   m = std::numeric_limits<double>::max();  // save min in here
17   // discretize rhos and remove first (open interval!)
18   Eigen::VectorXd rhos = Eigen::VectorXd::LinSpaced(N + 1, 1, maxRho).head(N);
19   r = rhos(0);  // save argmin here
20
21   // discretize the ellipse
22   Eigen::VectorXd thetas = Eigen::VectorXd::LinSpaced(N, 0, 2 * M_PI);
23
24   // loop over all rhos and get compute expression
25   for (unsigned int k = 0; k < rhos.size(); ++k) {
26     const double rho = rhos(k);  // current rho
27     // get values on the ellipse
28     Eigen::VectorXcd z(thetas.size());
29     for (unsigned int j = 0; j < thetas.size(); ++j) {
30       z(j) = gamma(rho, thetas(j));
31     }
32
33     // sup of f of γ_ρ
34     const Eigen::VectorXcd fz = z.unaryExpr(f);
35     const double M = lengthEllipse(rho, N) * fz.cwiseAbs().maxCoeff() *
36                      M_1_PI;  // nominator of expression
37
38     // Distance is the one between -1 and the minimal real part of the
       ellipse
39     // (i.e.  semi-major-axis - 1):
40     const double d = (rho + 1. / rho) / 2. - 1;  // denominator of expression
41
42     if (M / d < m) {
43       r = rho;  // update rho
44     }
45     m = std::min(m, M / d);  // update minimum
46   }
47   // END
48   return std::make_pair(m, r);
49 }
```

Get it on GitLab (chebyApprox.hpp).

▲

(6-5.e) ⚀ (15 min.) [depends on Sub-problem (6-5.d)]

Based on the result of Sub-problem (6-5.d) and [Lecture → Rem. 6.2.3.26], give an "optimal" bound for

$$\|f - L_n f\|_{L^\infty([-1,1])},$$

where $L_n : C^0([-1,1]) \to \mathcal{P}_n$ is the operator of Chebychev interpolation, that is, polynomial Lagrange interpolation based on Chebychev nodes [Lecture → Eq. (6.2.3.10)], on $[-1,1]$ into the space of polynomials of degree $\leq n$.

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 → 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

$$\|f - L_n f\|_{L^\infty([-1,1])} \lesssim \frac{10.0753}{1.91355^{n+1} - 1} . \tag{6.5.7}$$

▲

(6-5.f) ⚄ (20 min.) [depends on Sub-problem (6-5.e)]

Implement a C++ function

```
std::pair<std::vector<double>, std::vector<double> >
compareErrorAndBound(unsigned int n_max);
```

that computes and tabulates

1. estimates for $\|f - L_n f\|_{L^\infty([-1,1])}$ obtained by sampling in 10000 equidistant points $\in [-1,1]$,
2. the bounds derived in Sub-problem (6-5.e),

for polynomial degrees $n = 4, \ldots, n_{\max}$, where $n_{\max}$ is passed through the argument `n_max`.

You may rely on the provided function `intpolyval` (cf. [Lecture → **??**]).

HINT 1 for (6-5.f): The function `intpolyval` has the signature:

```
void intpolyval(const VectorXd& t, const VectorXd& y, const
  VectorXd& x, VectorXd& p);
```

Where (t,y) is the interpolation data and p is used to save the values of the interpolant evaluated in x.

⌟

SOLUTION of (6-5.f):

C++11-code 6.5.8: Code for `compareErrorAndBound`.

```
2  std::pair<std::vector<double>, std::vector<double>> compareErrorAndBound(
3      unsigned int n_max) {
4    std::vector<double> upperBound, errInf;
5    constexpr unsigned N = 10000;
6
7    // TODO: (6-5.f) Tabulate the L-inf error and the bounds.
8    // START
9    auto m_rho = bestBound();
10   const double m = m_rho.first;
11   const double rho = m_rho.second;
12
13   // Test for degrees 4 to n_max
14   Eigen::VectorXd degrees = Eigen::VectorXd::LinSpaced(n_max - 3, 4, n_max);
15
16   for (unsigned int j = 0; j < degrees.size(); ++j) {
17     upperBound.emplace_back(m / (std::pow(rho, degrees(j) + 1) - 1));
18   }
19
20   // Compute actual error for any degree
21   auto f = [](double x) { return 1. / (1 + std::exp(-3 * x)); };
22
23   for (unsigned int l = 0; l < degrees.size(); ++l) {
24     const int n = degrees(l);
25     // Cheby nodes and value of f at the nodes
26     Eigen::ArrayXd k = Eigen::ArrayXd::LinSpaced(n + 1, 0, n);
27     Eigen::VectorXd t = ((2 * k + 1) / (2 * (n + 1)) * M_PI).cos().matrix(),
28                     y = t.unaryExpr(f);
29     // Evaluate error at a couple of points
30     Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N, -1, 1),
31                     F = x.unaryExpr(f);
32
33     // Use barycentric formula (in intpolyval.hpp)
34     Eigen::VectorXd LnF;
35     intpolyval(t, y, x, LnF);
36     errInf.push_back((F - LnF).cwiseAbs().maxCoeff());
37   }
38   // END
39   return {errInf, upperBound};
40 }
```

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.

▲

(6-5.g) ⚄ (20 min.) Rely on pullback to $[-1,1]$ to discuss how the error bounds in [Lecture → Eq. (6.2.3.28)] will change qualitiatively when we consider Chebychev interpolation on $[-a,a], a > 0$, instead of $[-1,1]$, whilst keeping the function f fixed.

SOLUTION 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.

▲

End Problem 6-5 , 130 min.

Problem 6-6: Polynomial Best Approximation

The famous Jackson Theorem [Lecture → Thm. 6.2.1.11] gives detailed information about the best approximation of C^r-functions by means of polynomials. In this exercise we recall that theorem and study best-approximation errors in general.

Purely theoretical problem related to [Lecture → Section 6.2.1]

Throughout this problems we write Here

(6-6.a) ⚀ (7 min.) The following is the assertion of Jackson's theorem of [Lecture → Thm. 6.2.1.11] with some parts missing. Fill either n or r in the green boxes and supplement the right subscript for the norm in the white boxes.

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{\square} \leq \left(1+\frac{\pi^2}{2}\right)^{\square} \frac{(\square-\square)!}{\square!} \left\|f^{(\square)}\right\|_{\square}$$

for all $n, r \in \mathbb{N}$, $n \geq r$, and $f \in C^r([-1,1])$. Here $\mathcal{P}_n$ designates the space of univariate polynomials of degree $\leq n$ and $f^{(k)}$ is a short notation for the k-th derivative of a function.

SOLUTION 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

$$\|\cdot\|_{\infty,-1,1} \quad \text{or} \quad \|\cdot\|_{\infty,[-1,1]} \quad \text{or} \quad \|\cdot\|_{\infty,]-1,1[}.$$

▲

(6-6.b) ⚄ (8 min.) For $n \in \mathbb{N}$ we abbreviate

$$E_n(f) := \inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} \quad \text{for} \quad f \in C^0([-1,1]). \tag{6.6.2}$$

Which of the following assertions on E_n are correct, which are wrong? Throughout, f and g denote arbitrary continuous functions on $[-1,1]$ and $n \in \mathbb{N}$

		correct	wrong
(i)	$E_n(f+g) \leq E_n(f) + E_n(g)$	○	○
(ii)	$E_n(\lambda f) = \lambda E_n(f)$ for all $\lambda \in \mathbb{R}$	○	○
(iii)	E_n is a norm on $C^0([-1,1])$	○	○
(iv)	$E_n(f) - E_n(g) \leq E_n(f-g)$	○	○

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):

$$E_n(f) = E_n(f - g + g) \overset{\text{(i)}}{\leq} E_n(f-g) + E_n(g) \,.$$

▲

End Problem 6-6 , 15 min.

Problem 6-7: Piecewise cubic Hermite Interpolation of Functions

The variant of piecewise cubic Hermite interpolation studied in [Lecture → Section 6.6.2] was a generalized interpolation scheme in the sense that it required point values of the derivative of the interpoland. Conversely, for piecewise cubic Hermite data interpolation we had to use reconstructed slopes. This problem examines the use of reconstructed slopes also for the purpose of function approximation.

Relies on [Lecture → Section 5.3.3] and [Lecture → Section 6.6.2] and involves C++ coding.

Get it on GitLab (pchi.hpp).

Piecewise cubic Hermite interpolation of a function $f \in C^1([a,b])$ with exact slopes on a mesh

$$\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$$

was defined in [Lecture → Def. 6.6.2.1]. For $f \in C^4([a,b])$ it enjoys h-convergence with rate 4 as we have seen in [Lecture → Exp. 6.6.2.2] and which is stated in [Lecture → Thm. 6.6.2.3].

Theorem [Lecture → Thm. 6.6.2.3]. Convergence of approximation by cubic Hermite interpolation

Let s be the cubic Hermite interpolant of $f \in C^4([a,b])$ on a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ according to [Lecture → Def. 6.6.2.1]. Then

$$\|f - s\|_{L^\infty([a,b])} \leq \frac{1}{4!} h_{\mathcal{M}}^4 \left\| f^{(4)} \right\|_{L^\infty([a,b])} ,$$

with the meshwidth $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}|$.

Now we consider cases, where perturbed or reconstructed slopes are used. For instance, this was done in the context of monotonicity preserving piecewise cubic Hermite interpolation as discussed in [Lecture → Section 5.3.3.2].

(6-7.a) ⚄ (30 min.) Assume that piecewise cubic Hermite interpolation is based on perturbed slopes, that is, the piecewise cubic function s on the mesh $\mathcal{M}$ satisfies:

$$s(x_j) = f(x_j) \quad , \quad s'(x_j) = f'(x_j) + \delta_j \, , \tag{6.7.1}$$

where the perturbations $\delta_j \in \mathbb{R}$ may depend on $\mathcal{M}$, too.

Now we consider a sequence of meshes with meshwidth $h_{\mathcal{M}} \to 0$. Which rate of asymptotic h-convergence of the maximum norm $\|f - s\|_{L^\infty(]a,b[)}$ of the approximation error can be expected, if we know that for all j that

$$|\delta_j| = O(h_{\mathcal{M}}^\beta) \quad \text{as} \quad h_{\mathcal{M}} \to 0 \quad \text{for some} \quad \beta \in \mathbb{N}_0 \, , \tag{6.7.2}$$

for mesh-width $h_{\mathcal{M}} \to 0$.

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. 161].

▲

(6-7.b) ⚄ We define a "strange" piecewise cubic interpolant:

Definition 6.7.3. Flat-point piecewise cubic interpolant

Given $f \in C^0([a,b])$ and a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ the flat-point piecewise cubic interpolant $s : [a,b] \to \mathbb{R}$ is defined as

$$s_{|[x_{j-1},x_j]} \in \mathcal{P}_3 , \quad j = 1,\ldots,m \quad , \quad s(x_j) = f(x_j) , \quad s'(x_j) = 0 , \quad j = 0,\ldots,m .$$

Implement a C++ function

```
template <typename Function>
Eigen::VectorXd fppchip(Function &&f,
  const Eigen::VectorXd &x, const Eigen::VectorXd &t);
```

that returns the vector $[s(t_k)]_{k=1}^N$, where

- the argument `x` passes the $m+1$ (ordered) nodes of a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$,
- s is the flat-point piecewise cubic interpolant of f according to Def. 6.7.3,
- the (sorted!) points $t_k \in [x_0, x_m]$ are supplied as components of the vector `t`.

Remark. Please refer to [Lecture → Thm. 5.3.3.16] to understand the motivation for introducing the "flat-point piecewise cubic interpolant". This is the only linear interpolation operator into C^1-functions that is locally monotonicity preserving.

SOLUTION of (6-7.b):

C++ code 6.7.4: Sub-problem (6-7.b): Computation of cubic Hermite interpolant

```
2  template <typename Function>
3  Eigen::VectorXd fppchip(Function &&f, const Eigen::VectorXd &t,
4                          const Eigen::VectorXd &x) {
5    Eigen::VectorXd ret{x.size()};
6    // TO DO: compute ret.
7    // START
8
9    Eigen::VectorXd y = t.unaryExpr(f);
10   int i_star = 0;
11   double tmp, h_tmp, t1, t2, y1, y2, a1, a2, a3;
12   for (int j = 0; j < x.size();) {
13     // Stores the current interval index and some temporary variable
14     // Find the interval for x_j while assuming that x values are sorted
15     // and within [t_min, t_max]
16     if (t(i_star) <= x(j) && x(j) <= t(i_star + 1)) {
17       t1 = t(i_star);
18       t2 = t(i_star + 1);
19       h_tmp = t2 - t1;  // h\_tmp = h\_ for equidistant grid
20       y1 = y(i_star);
21       y2 = y(i_star + 1);
22       a1 = y2 - y1;
23       a2 = a1;
24       a3 = -a1 - a2;
25
26       // Compute s(x(j))
27       tmp = (x(j) - t1) / h_tmp;
28       ret(j) = y1 + (a1 + (a2 + a3 * tmp) * (tmp - 1.)) * tmp;
29
30       ++j;
31     } else {
32       // Otherwise, go to next interval
33       ++i_star;
34       // Terminate if outside any interval
35       if (i_star >= t.size() - 1) break;
36       continue;
37     }
38   }
39
40   // END
41
42   return ret;
43 }
```

Get it on GitLab (pchi.hpp).

▲

(6-7.c) ⚂ (20 min.) [depends on Sub-problem (6-7.b)]

For the function $f(t) = \frac{1}{1+t^2}$ and the interval $[-5,5]$ empirically determine (qualitatively and quantitatively) the asymptotic convergence (in terms of meshwidth $h_{\mathcal{M}} \to 0$) of the flat-point piecewise cubic interpolants according to Def. 6.7.3 on sequences of *equidistant meshes*.

To that end study the maximum norm of the interpolation error in numerical experiments. Approximate those maximum norms by sampling in 10000 equidistant points. Implement a C++ function

```
std::vector<double> fppchipConvergence(void);
```

that tabulates and returns the error norms for meshes with $4, 8, 16, 32, 64, 128, 256, 512$ nodes.

SOLUTION of (6-7.c):

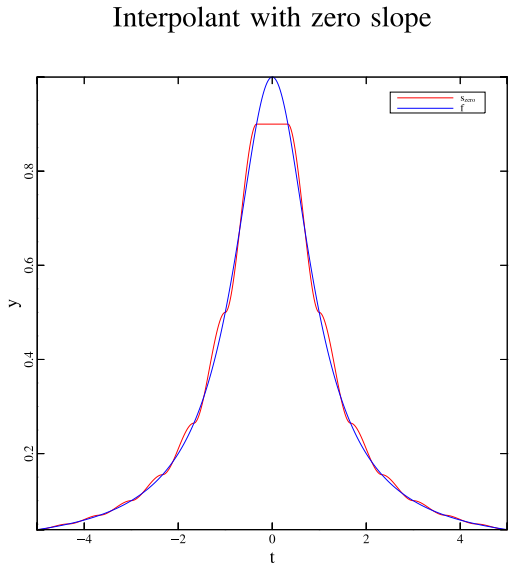


Fig. 58

◁ Flat-point piecewise cubic interpolant of the Runge function $f(t) = \frac{1}{1+t^2}$ 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    // Interpoland
4    auto f = [](double x) { return 1. / (1. + x * x); };
5
6    double a = 5;  // Interval bounds will be (-a,a)
7    int M = 1000;  // Number of  points in which to evaluate the interpoland
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   // TO DO: compute err\_zero.
17   // START
```

```
18
19   for (int i = 4; i <= 512; i = i << 1) {
20     // Define subintervals and evaluate f there (find pairs (t,y))
21     Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(i, -a, a);
22
23     // compute reconstructed slopes
24     Eigen::VectorXd s_zero_x = fppchip(f, t, x);
25
26     // Compute infinity norm of error
27     err_zero.push_back((s_zero_x - fx).lpNorm<Eigen::Infinity>());
28     N_nodes.push_back(i);
29
30     if (i == 16) {
31       plt::figure();
32       plt::title("Interpolant with zero slope");
33       plt::xlabel("t");
34       plt::ylabel("y");
35       plt::plot(x, s_zero_x, "r", {{"label", "s_zero"}});
36       plt::plot(x, fx, "b--", {{"label", "f"}});
37       plt::legend();
38       plt::savefig("./cx_out/p_zero.png");
39     }
40   }
41
42   // END
43
44   return err_zero;
45 }
```

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.

▲

(6-7.d) ⚂ (10 min.) [depends on Sub-problem (6-7.a) and Sub-problem (6-7.c)]

Invoke the result of Sub-problem (6-7.a) to give a theoretical explanation for the observations made in Sub-problem (6-7.c).

SOLUTION 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 $O(h)$.

▲

We now define a piecewise cubic Hermite interpolant with slopes reconstructed from sampled function values.

Definition 6.7.6. Piecewise cubic Hermite interpolant with reconstructed slopes

Given $f \in C^0([a,b])$ and a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ the piecewise cubic Hermite interpolant with reconstructed slopes $s : [a,b] \to \mathbb{R}$ is defined as

$$s_{|[x_{j-1},x_j]} \in \mathcal{P}_3 , \quad j = 1,\ldots,m \quad , \quad s(x_j) = f(x_j) , \quad j = 0,\ldots,m ,$$

$$s'(x_j) = \begin{cases} \dfrac{-f(x_2) + 4f(x_1) - 3f(x_0)}{2h} & \text{for } j = 0 , \\ \dfrac{f(x_{j+1}) - f(x_{j-1})}{2h} & \text{for } j = 1,\ldots,m-1 , \\ \dfrac{3f(x_m) - 4f(x_{m-1}) + f(x_{m-2})}{2h} & \text{for } j = m . \end{cases}$$

A class for piecewise cubic Hermite interpolation of continuous functions according to Def. 6.7.6 is defined as follows:

```
class CubicHermiteInterpolant {
public:
  template <typename Function>
  CubicHermiteInterpolant(Function &&f, const Eigen::VectorXd &t);
  virtual ~CubicHermiteInterpolant(void) = default;
  Eigen::VectorXd eval(const Eigen::VectorXd &x) const;
private:
  Eigen::VectorXd t_; // sorted mesh nodes
  Eigen::VectorXd y_; // function values at mesh nodes
  Eigen::VectorXd c_; // slopes at mesh nodes
};
```

The constructor expects a functor `f` with signature `std::function<double(double)>` and a (sorted!) vector `t` of nodes. Its evaluation method `CubicHermiteInterpolant::eval()` returns $s(x_k)$ for the (sorted!) components of the vector passed in `x`. Here $s \in C^1([a,b])$ is the piecewise cubic Hermite interpolant as defined in Def. 6.7.6.

(6-7.e) ⚄ (25 min.) Provide an implementation of the class **CubicHermiteInterpolant**.

SOLUTION of (6-7.e):

C++ code 6.7.7: Sub-problem (6-7.e): Definition of Hermite interpolant-class

```
2  class CubicHermiteInterpolant {
3  public:
4    /*!
5     * \brief Construct the slopes from the data.
6     * Use finite-differences or setting s'(x_j) = 0.
7     * \param[in] t Vector of nodes (assumed equidistant and sorted).
8     * \param[in] y Vector of values at nodes t.
9     * \param[in] s Flag to set if you want to reconstruct or set the slopes to
10    *zero.
11    */
12   template <typename Function>
13   CubicHermiteInterpolant(Function &&f, const Eigen::VectorXd &t);
```

```
14    virtual ~CubicHermiteInterpolant() = default;
15
16    /*!
17     * \brief Evaluate the intepolant at the nodes x.
18     * The input is assumed sorted, unique and inside the interval
19     * \param[in] x The vector of points x_i where to compute s(x_i).
20     * \return Values of interpolant at x (vector)
21     */
22    Eigen::VectorXd eval(const Eigen::VectorXd &x) const;
23
24   private:
25    // Provided nodes and values (t,y) to compute spline,
26    // Eigen vectors, c contains slopes
27    // All have the same size n
28    Eigen::VectorXd t_, y_, c_;
29    // Difference t(i) - t(i-1)
30    double h_;
31    // Size of t, y and c.
32    int n_;
33  };
```

Get it on GitLab (pchi.hpp).

C++ code 6.7.8: Sub-problem (6-7.e): Constructor of Hermite interpolant-class

```
2   template <typename Function>
3   CubicHermiteInterpolant::CubicHermiteInterpolant(Function &&f,
4                                                    const Eigen::VectorXd &t)
5       : t_(t), y_(t.unaryExpr(f)), c_(t.size()) {
6     // Sanity check
7     n_ = t_.size();
8     assert(n_ == y_.size() && "t and y must have same dimension.");
9     assert(n_ >= 3 && "Need at least three nodes.");
10    h_ = t_(1) - t_(0);
11
12    //// Reconstruction of the slope,
13    // TODO: implement reconstruction of slopes.
14    // START
15    // First order alternative:
16    c_(0) = (-1 * y_(2) + 4 * y_(1) - 3 * y_(0)) / 2 / h_;
17    for (int i = 1; i < n_ - 1; ++i) {
18      c_(i) = (y_(i + 1) - y_(i - 1)) / 2 / h_;
19    }
20
21    // First order alternative:
22    c_(n_ - 1) = (3 * y_(n_ - 1) - 4 * y_(n_ - 2) + 1 * y_(n_ - 3)) / 2 / h_;
23    // END
24  }
```

Get it on GitLab (pchi.hpp).

C++ code 6.7.9: Sub-problem (6-7.e): Implementation of constructor of evaluation operator

```
2   Eigen::VectorXd CubicHermiteInterpolant::eval(const Eigen::VectorXd &x) const {
3     Eigen::VectorXd ret(x.size());
4
5     // TO DO: implement evaluation function.
6     // START
```

```
7
8   int i_star = 0;
9   double tmp, h_tmp, t1, t2, y1, y2, c1, c2, a1, a2, a3;
10  for (int j = 0; j < x.size();) {
11    // Stores the current interval index and some temporary variable
12    // Find the interval for x_j while assuming that x values are sorted
13    // and within [t_min, t_max]
14    if (t_(i_star) <= x(j) && x(j) <= t_(i_star + 1)) {
15      t1 = t_(i_star);
16      t2 = t_(i_star + 1);
17      h_tmp = t2 - t1;  // h_tmp = h_ for equidistant grid
18      y1 = y_(i_star);
19      y2 = y_(i_star + 1);
20      c1 = c_(i_star);
21      c2 = c_(i_star + 1);
22      a1 = y2 - y1;
23      a2 = a1 - h_tmp * c1;
24      a3 = h_tmp * c2 - a1 - a2;
25
26      // Compute s(x(j))
27      tmp = (x(j) - t1) / h_tmp;
28      ret(j) = y1 + (a1 + (a2 + a3 * tmp) * (tmp - 1.)) * tmp;
29
30      ++j;
31    } else {
32      // Otherwise, go to next interval
33      ++i_star;
34      // Terminate if outside any interval
35      if (i_star >= t_.size() - 1) break;
36      continue;
37    }
38  }
39
40  // END
41
42  return ret;
43 }
```

Get it on GitLab (pchi.hpp).

▲

(6-7.f) ⚂ (10 min.) [depends on Sub-problem (6-7.e)]

Based on the class **CubicHermiteInterpolant** realize a C++ function

```
std::vector<double> rspchipConververgence(void);
```

that answers the questions of Sub-problem (6-7.c) for the interpolant according to Def. 6.7.6.

SOLUTION of (6-7.f):

C++ code 6.7.10: Sub-problem (6-7.f): Convergence of Hermite interpolation with reconstructed slopes

```
2 std::vector<double> rspchipConververgence() {
3   // Interpoland
```

```
4   auto f = [](double x) { return 1. / (1. + x * x); };
5
6   double a = 5;  // Interval bounds will be (-a,a)
7   int M = 1000;  // Number of  points in which to evaluate the interpoland
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_reconstr;
15
16  // TO DO: compute err\_reconstr.
17  // START
18
19  for (int i = 4; i <= 512; i = i << 1) {
20    // Define subintervals and evaluate f there (find pairs (t,y))
21    Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(i, -a, a);
22
23    // Construct PCHI with zero and reconstructed slopes
24    CubicHermiteInterpolant s_reconstr(f, t);
25
26    // Evaluate interpolant
27    Eigen::VectorXd s_reconstr_x = s_reconstr.eval(x);
28
29    // Compute infinity norm of error
30    err_reconstr.push_back((s_reconstr_x - fx).lpNorm<Eigen::Infinity>());
31    N_nodes.push_back(i);
32
33    if (i == 16) {
34      plt::figure();
35      plt::title("Interpolant with reconstructed slope");
36      plt::xlabel("t");
37      plt::ylabel("y");
38      plt::plot(x, s_reconstr_x, "r", {{"label", "$s_{reconstr}$"}});
39      plt::plot(x, fx, "b--", {{"label", "$f$"}});
40      plt::legend();
41      plt::savefig("./cx_out/p_reconstr.png");
42    }
43  }
44  // END
45
46  return err_reconstr;
47 }
```

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 $h_{\mathcal{M}} \to 0$.

▲

(6-7.g) ⚄ (30 min.) Write $\mathsf{I}_H : C^0([a,b]) \to C^1([a,b])$ for the piecewise cubic Hermite interpolation operator with reconstructed slopes introduced in Def. 6.7.6.

Based on the result of Sub-problem (6-7.a) prove that

$$f \in C^3([a,b]) \quad \Rightarrow \quad \exists C > 0: \quad \|f - \mathsf{I}_H f\|_{L^\infty(]a,b[)} \leq C h_{\mathcal{M}}^3 ,$$

with $C > 0$ independent of the mesh and $h_{\mathcal{M}}$ standing for the mesh width of $\mathcal{M}$.

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.

▲

End Problem 6-7 , 125 min.

Problem 6-8: Monomial representation of Chebychev polynomials

Chebychev polynomials are important both for theoretical analysis and algorithm design. For instance, as we have seen in [Lecture → Section 6.2.3.3] Chebyshev interpolants should internally be represented through their Chebychev expansion. In this problem we will study an algorithm for converting Chebychev expansions into monomial expansions.

This problem relies about elementary facts about Chebychev polynomials as introduced in [Lecture → Section 6.2.3.1].

On $[-1,1]$ the n-th Chebychev polynomial T_n, $n \in \mathbb{N}_0$ is defined as [Lecture → Def. 6.2.3.3]

$$T_n(t) := \cos(n \arccos t) \quad -1 \le t \le 1 . \tag{6.8.1}$$

That T_n is a polynomial of degree $\le n$ is clear from the 3-term recursion

$$T_{n+1}(t) = 2t\,T_n(t) - T_{n-1}(t) \quad , \quad T_0 \equiv 1 , \quad T_1(t) = t , \quad n \in \mathbb{N} . \qquad \text{[Lecture → Eq. (6.2.3.5)]}$$

The set $\{T_0, \dots, T_n\}$ forms a basis of $\mathcal{P}_n$.

(6-8.a) ⚀ (8 min.) Below is an incomplete implementation of a C++ function

```
Eigen::VectorXd chebexpToMonom(const Eigen::VectorXd &a);
```

that expects the coefficients $a_j \in \mathbb{R}$, $j = 0, \dots, n$, of the Chebyshev expansion

$$p(t) = \sum_{j=0}^{n} a_j T_j(t) , \quad t \in \mathbb{R} ,$$

of a polynomial $p \in \mathcal{P}_n$ to be passed in the vector `a` and returns the coefficients r_j, $j = 0, \dots, n$ with respect to the monomial basis,

$$p(t) = r_0 + r_1 t + r_2 t^2 + \cdots + r_n t^n ,$$

collected in a vector $[r_0, r_1, \dots, r_n]^\top$.

Fill in the missing pieces of the following code, which relies on [Lecture → Eq. (6.2.3.5)].

```
Eigen::VectorXd chebexpToMonom(const Eigen::VectorXd &a) {
  const int n = a.size() -1; // degree of polynomial
  assert(n >= 0);
  Eigen::VectorXd r{ Eigen::VectorXd::Zero(n+1) };
  r[0] = A ;
  if (n > 0) {
    r[1] = B ;
    if (n > 1) {
      // two vectors temporarily storing the monomial coefficients
      // of Chebychev polynomials of two consecutive degrees
      std::array<Eigen::VectorXd, 2> c
       { Eigen::VectorXd::Zero(n+1), Eigen::VectorXd::Zero(n+1) };
      c[0][0] = 1.0; c[1][1] = 1.0;
      for (int j=2; j<= C ; ++j) {
        c[j%2][0] *= -1.0;
```

```
            for (int k= D ; k <= j; ++k)
              c[j%2][k] = E *c[ F ][ G ] - c[j%2][k];
            r += H *c[j%2];
        }
      }
    }
    return r;
}
```

- A ≙ ☐
- B ≙ ☐
- C ≙ ☐
- D ≙ ☐
- E ≙ ☐
- F ≙ ☐
- G ≙ ☐
- H ≙ ☐

SOLUTION 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;
}
```

▲

End Problem 6-8 , 8 min.

Problem 6-9: Finding domains of analyticity

The developments of [Lecture → Lemma 6.2.2.53], [Lecture → Rem. 6.2.3.26], and [Lecture → Section 6.5.3] make clear that knowledge about that maximal subdomain $D \subset \mathbb{C}$ of the complex plane to which an analytic function can be extended from its original real interval of definition is key to predicting the speed of exponential convergence of approximation schemes like Chebychev interpolation and trigonometric interpolation. In this problem we discuss the analytic extension of 1-periodic functions.

Depends on [Lecture → Rem. 6.2.2.67] and is related to [Lecture → § 6.5.3.16].

In order to analyze the trigonometric functions occurring in this problem, the following identities can be useful:

$$\begin{aligned} \sin(x+\imath y) &= \sin(x)\cosh(y) + \imath\cos(x)\sinh(y) \quad \forall x,y \in \mathbb{R}\,, \\ \cos(x+\imath y) &= \cos(x)\cosh(y) - \imath\sin(x)\sinh(y) \quad \forall x,y \in \mathbb{R}\,, \\ \sin^2(z) + \cos^2(z) &= 1 \quad \forall z \in \mathbb{Z}\,. \end{aligned} \tag{6.9.1}$$

The following result gives domains of analyticity of special functions:

Theorem 6.9.2. domains of analyticity of special functions

- *Every polynomial is analytic on* $\mathbb{C}$.
- *Every rational function, that is, a quotient of two polynomials, is analytic in the complement of the set of zeros of its denominator.*
- *The functions* $\exp$, $\sin$, *and* $\cos$ *are analytic on* $\mathbb{C}$.
- *The functions* $\log : \mathbb{R}^+ \to \mathbb{R}$ *and* $\sqrt{} : \mathbb{R}_0^+ \to \mathbb{R}$ *can be extended to analytic functions on* $\mathbb{C} \setminus \mathbb{R}_0^-$.

(6-9.a) ⚄ (6 min.) We consider the 1-periodic function

$$f(t) = b\log(a + \sin(2\pi t))\,, \quad t \in \mathbb{R}\,, \quad b > 0,\ a > 1\,. \tag{6.9.3}$$

Determine the largest possible subset D of $\mathbb{C}$ to which f can be extended analytically.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \boxed{}\,, \\ \mathrm{Im}(z) \in \boxed{} \end{array} \right\}.$$

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 $z = x + \imath y$

$$\operatorname{Im}(a + \sin(2\pi z)) = \cos(2\pi x)\sinh(2\pi y) \quad , \quad \operatorname{Re}(a + \sin(2\pi z)) = a + \sin(2\pi x)\cosh(2\pi y) \, .$$

We first examine $\operatorname{Im}(a + \sin(2\pi z)) = 0 \Leftrightarrow \cos(2\pi x)\sinh(2\pi y) = 0$:

$$\operatorname{Im}(a + \sin(2\pi z)) = 0 \quad \Leftrightarrow \quad y = 0 \quad \vee \quad x \in \tfrac{1}{2}\mathbb{Z} + \tfrac{1}{4} \, .$$

The case $y = 0$ need not be considered any further, because, since $a > 1$,

$$y = 0 \quad \Rightarrow \quad \operatorname{Re}(a + \sin(2\pi z)) = a + \sin(2\pi x) > 0 \, .$$

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, $\operatorname{Re}(a + \sin(2\pi z)) \leq 0$ is possible only if

$$x = (4k+3)\frac{1}{4} \, , \quad k \in \mathbb{Z} \, , \quad a - \cosh(2\pi y) \leq 0 \quad \Leftrightarrow \quad |y| \geq \frac{1}{2\pi}\cosh^{-1}(a) \, .$$

Summing up we have found

$$D = \mathbb{C} \setminus \{ z \in \mathbb{C} : \ \operatorname{Re}(z) \in \mathbb{Z} + \tfrac{3}{4} \, , \ |\operatorname{Im}(z)| \geq \zeta, \ \zeta > 0, \ \cosh(2\pi\zeta) = a \} \, .$$

The presence of the parameter b has no impact on the domain of analyticity.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \operatorname{Re}(z) \in \mathbb{Z} + \frac{3}{4} \, , \\ \operatorname{Im}(z) \in \{ y \in \mathbb{R} : \ |y| \geq q, \ q > 0, \ \cosh(2\pi q) = a \} \end{array} \right\} \, .$$

▲

(6-9.b) ⚄ (6 min.) We consider the 1-periodic function

$$f(t) = a\log(b - \cos(2\pi t)) \, , \quad t \in \mathbb{R} \, , \quad b > 1, \ a > 1 \, . \tag{6.9.4}$$

Determine the largest possible subset D of $\mathbb{C}$ to which f can be extended analytically.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{ll} \operatorname{Re}(z) \in & \boxed{} \, , \\ \operatorname{Im}(z) \in & \boxed{} \end{array} \right\} \, .$$

SOLUTION of (6-9.b):

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \operatorname{Re}(z) \in \mathbb{Z} \, , \\ \operatorname{Im}(z) \in \{ y \in \mathbb{R} : \ |y| \geq q, \ \cosh(2\pi q) = b \} \end{array} \right\} \, .$$

▲

(6-9.c) ⚄ (6 min.) We consider the 1-periodic function

$$f(t) = a\sqrt{b - \sin(2\pi t)}\,, \quad t \in \mathbb{R}\,, \quad b > 1,\ a > 1\,. \tag{6.9.5}$$

Determine the largest possible subset D of $\mathbb{C}$ to which f can be extended analytically.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \boxed{}\,, \\ \mathrm{Im}(z) \in \boxed{} \end{array} \right\}.$$

SOLUTION of (6-9.c):

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \mathbb{Z} + \frac{1}{4}\,, \\ \mathrm{Im}(z) \in \{ y \in \mathbb{R} : |y| \geq q,\ \cosh(2\pi q) = b \} \end{array} \right\}.$$

▲

End Problem 6-9, 18 min.

Chapter 7

Numerical Quadrature

Problem 7-1: Smooth integrand by transformation

In [Lecture → Rem. 7.4.3.10] we saw how knowledge about the structure of a non-smooth integrand can be used to restore its smoothness by transformation. The very same idea can be put to use in this problem.

This problem practises the use of Gauss-Legendre quadrature and of the techniques introduced in [Lecture → Rem. 7.4.3.10].

Given a smooth, odd function $f : [-1,1] \to \mathbb{R}$, consider the integral

$$I := \int_{-1}^{1} \arcsin(t)\, f(t)\, \mathrm{d}t. \tag{7.1.1}$$

We want to approximate this integral using global Gauss-Legendre quadrature as introduced in [Lecture → Section 7.4.2].

The nodes and the weights of the n-point Gauss-Legendre quadrature rule on $[-1,1]$ can be computed using the provided C++ function

```
QuadRule gaussquad(const unsigned n);
```

which initializes a data structure describing a quadrature rule

```
struct QuadRule {
  Eigen::VectorXd nodes_;
  Eigen::VectorXd weights_;
};
```

The names of the fields tell their function. The function `gaussquad()` can be found in `gaussquad.hpp`.

(7-1.a) ⚀ (30 min.) Based on MATPLOTLIBCPP write a C++ function

```
template <class Function>
double gaussConv(const Function& fh, const double I_ex, const
   unsigned N);
```

that produces an *appropriate* plot of the quadrature error of the Gauss-Legendre quadrature rules versus the number $n = 1,\ldots,50$ of quadrature points. The function should output the best approximation of

the integral, i.e. the approximated value at the maximum number of quadrature points. Here `f` is a functor object [Lecture → Section 0.3.3] providing the function f in (7.1.1). `I_ex` is the exact solution of the integral and `N` is the maximum number of quadrature points.

Save your convergence plot for $f(t) = \sinh(t)$ as `GaussConv.png`.

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()`

```
2  template <class Function>
3  double integrate(const QuadRule& qr, const Function& f) {
4    double I = 0;
5
6    // TODO: (7-1.a) optional:  compute the integral with this helper
       function
7    // START
8
9    // Compute weighted sum of function values according to
10   // the concept of a generic quadrature formula
11   for (unsigned i = 0; i < qr.weights_.size(); ++i) {
12     I += qr.weights_(i) * f(qr.nodes_(i));
13   }
14
15   // END
16   return I;
17 }
```

C++-code 7.1.3: Implementation of the function `gaussConv()`

```
2  template <class Function>
3  double gaussConv(const Function& fh, const double I_ex, const unsigned N) {
4    std::vector<double> evals,  // used to save no. of quad nodes
5        error,                  // used to save the error
6        evals_ref;  // used to plot reference line of order of convergence
7
8    double I_N = 0.;  // best approx.  of integral
9    plt::figure();
10
11   // TODO: (7-1.a) write code that plots the no.  of nodes versus the
       error
12   // using matplotlibcpp and returns the best approximation
13   // START
14
15   // Build integrand
16   auto f = [fh](double x) { return std::asin(x) * fh(x); };
17
18   double I;
19   for (unsigned n = 1; n <= N; ++n) {
20     QuadRule qr = gaussquad(n);  // Create quadrature rule
21
22     I = integrate(qr, f);  // Compute integral
23
24     evals.push_back(n);  // Save no.  of quadrature nodes
```

```
25      evals_ref.push_back(
26          std::pow(n, -3));  // used for plotting to show convergence rate
27      error.push_back(std::abs(I - I_ex));  // Save error
28    }
29    // 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.

▲

(7-1.b) ⚀ (15 min.) Describe qualitatively (type of convergence) and quantitatively (speed of convergence) the asymptotic (for $n \to \infty$) convergence of the quadrature error you observe in (7-1.a).

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):

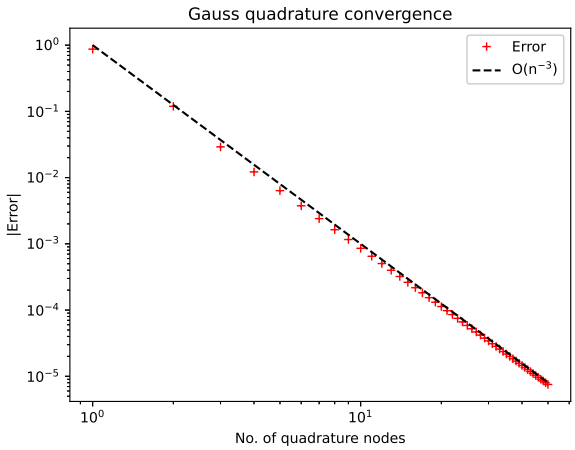


Fig. 59

◁ *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 $n \to \infty$.

▲

(7-1.c) ⚄ (20 min.) With a suitable change of variable transform the integral of (7.1.1) into an equivalent one, so that Gauss-Legendre quadrature applied to the transformed integral can be expected to *converge exponentially* in the number of quadrature nodes, if f possesses an analytic extension beyond $[-1,1]$.

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 → 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 → Rem. 6.2.3.26] and [Lecture → Thm. 7.4.3.3] to conclude exponential convergence of the quadrature errors.

▲

(7-1.d) ⚅ (25 min.) [depends on Sub-problem (7-1.c)]

Now, write a C++ function

```
template <class Function>
double gaussConvCV(const Function& f, const double I_ex, const
   unsigned N);
```

which plots (employing MATPLOTLIBCPP) the quadrature error versus the number $n = 1,\ldots,50$ of quadrature points for the integral obtained in the previous subtask and returns the best approximation of the integral, i.e. the approximated value at the maximum number of quadrature points.

Again, as in Sub-problem (7-1.a), choose $f(t) = \sinh(t)$ and save your convergence plot as `GaussConvCV.png`.

SOLUTION 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 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 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
28     // Evaluate g at quadrature nodes c
29     Eigen::VectorXd gc = c.unaryExpr(g);
30
31     // Same as I = \sum_{i=1}^n w_i g(c_i)
32     I = w.dot(gc);
33
34     evals.push_back(n);                  // Save no. of quadrature nodes
35     error.push_back(std::abs(I - I_ex));  // Save error
36   }
37
38   // save approximation of integral computed with highest-order rule
39   I_N = I;
40
41   // Create convergence plots
42   plt::title("Gauss quadrature convergence");
43   plt::semilogy(
```

```
44        evals, error, " +r",
45        {{"label", "Error"}}); // plot error as lin-log plot to see
              convergence
46    plt::xlabel("No. of quadrature nodes");
47    plt::ylabel("|Error|");
48    plt::legend("best");
49    // END
50
51    plt::savefig("./cx_out/GaussConvCV.png");
52    // return the best approximation
53    return I_N;
54  }
```

Get it on GitLab (gaussConvergence.hpp).

▲

(7-1.e) ⚀ (10 min.) [depends on Sub-problem (7-1.d)]

Analogous question as in Sub-problem (7-1.b): describe qualitatively the asymptotic (for $n \to \infty$) convergence of the quadrature error you observe in Sub-problem (7-1.d).

SOLUTION of (7-1.e):

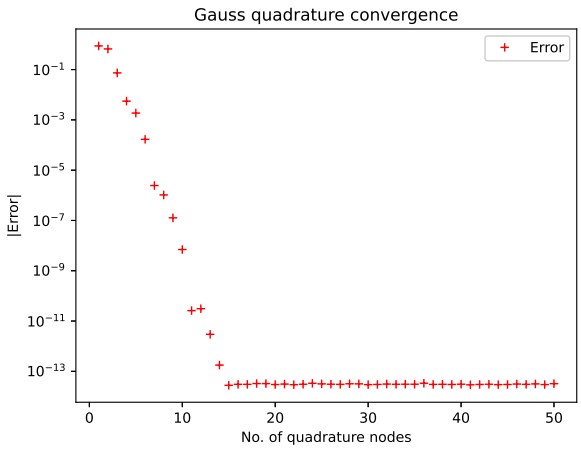


Fig. 60

◁ *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).

▲

(7-1.f) ⚂ [depends on Sub-problem (7-1.b) and Sub-problem (7-1.d)]

Explain the difference between observations made in Sub-problem (7-1.b) and Sub-problem (7-1.d).

SOLUTION 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.

▲

End Problem 7-1 , 100 min.

Problem 7-2: Generalized "Hermite-type" quadrature formula

In this exercise we will meet a new class of quadrature formulas and then use one of them in an application.

This is a purely theoretical exercise. You may want to have a look at the methods used in [Lecture → Ex. 7.4.2.2], as they will come handy.

(7-2.a) ⚄ Determine $A, B, C, x_1 \in \mathbb{R}$ such that the quadrature formula

$$\int_0^1 f(x)dx \approx Af(0) + Bf'(0) + Cf(x_1) \tag{7.2.1}$$

is exact for polynomials of highest possible degree.

SOLUTION 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.

▲

(7-2.b) ⚄ Compute an approximation of $z(2)$ using the quadrature rule of Eq. (7.2.1), where the function z is defined as the solution of the initial value problem

$$z'(t) = \frac{t}{1+t^2} \quad , \quad z(1) = 1\,. \tag{7.2.2}$$

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':

$$z(2) = \int_1^2 z'(x)dx + z(1) = \frac{1}{2}\log(x^2+1)|_1^2 + 1 = 1.45815\ldots$$

▲

End Problem 7-2

Problem 7-3: Numerical Quadrature of Improper Integrals

In analysis you have seen so-called improper integrals over unbounded intervals, which, due to a fast decay of the integrand have a finite value. Occasionally such integrals have to evaluated numerically and this problem will be concerned with corresponding quadrature rules.

This problem makes extensive use of substitution rules for integrals, implementation in C++ is requested.

We want to devise a numerical method for the computation of improper integrals of the form $\int_{-\infty}^{\infty} f(t)dt$ for continuous functions $f : \mathbb{R} \to \mathbb{R}$ that decay sufficiently fast for $|t| \to \infty$ (such that they are integrable on $\mathbb{R}$).

A first option is the truncation of the domain to a bounded interval $[-b, b]$, $b \leq \infty$, that is, we approximate:

$$\int_{-\infty}^{\infty} f(t)dt \approx \int_{-b}^{b} f(t)dt$$

and then use a standard quadrature rule like Gauss-Legendre quadrature [Lecture → Def. 7.4.2.18] on $[-b, b]$.

(7-3.a) ⚀ (10 min.) For the integrand $g(t) := 1/(1+t^2)$ determine b such that the truncation error E_T satisfies:

$$E_T := \left| \int_{-\infty}^{\infty} g(t)dt - \int_{-b}^{b} g(t)dt \right| \leq 10^{-6} . \tag{7.3.1}$$

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

$$E_T = 2\int_{b}^{\infty} g(t)dt = \lim_{x\to\infty} 2\mathrm{atan}(x) - 2\mathrm{atan}(b) = \pi - 2\mathrm{atan}(b) \overset{!}{<} 10^{-6} , \tag{7.3.2}$$

i.e. $b > \tan\big((\pi - 10^{-6})/2\big) = \cot(10^{-6}/2) \approx 2 \cdot 10^6$.

▲

(7-3.b) ⚀ (5 min.) What is the algorithmic difficulty faced in the implementation of the truncation approach for a generic integrand?

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.

▲

A second option is the transformation of the improper integral to a bounded domain by substitution. For instance, we may use the map $t = \cot(s) := \frac{\cos s}{\sin s}$, $\cot :]0,\pi[\to]-\infty,\infty[$.

(7-3.c) ⚀ (15 min.) Into which integral does the substitution $t = \cot(s)$ convert $\int_{-\infty}^{\infty} f(t)dt$?

SOLUTION 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 $\sin^2(\theta) = \frac{1}{1+\cot^2(\theta)}$.

▲

(7-3.d) ⚀ (10 min.) Write down the transformed integral explicitly for the integrand $g(t) := \frac{1}{1+t^2}$. Simplify the integrand.

HINT 1 for (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:

$$\int_0^{\pi} \frac{1}{1+\cot^2(s)}\frac{1}{\sin^2(s)}ds = \int_0^{\pi} \frac{1}{\sin^2(s)+\cos^2(s)}ds = \int_0^{\pi} ds = \pi$$

▲

(7-3.e) ⚀ (20 min.) [depends on Sub-problem (7-3.c)]

Write a C++ function that uses the previous transformation together with the n-point Gauss-Legendre quadrature rule to evaluate $\int_{-\infty}^{\infty} f(t)dt$:

```
template <typename Function>
double quadinf(int n, Function && f);
```

f passes an object that provides an evaluation operator of the form:

```
double operator() (double x) const;
```

You can use the function `golubwelsh()` implemented in the file `golubwelsh.hpp` that computes nodes and weights for Gauss quadrature rules using the Golub-Welsch algorithm discussed in [Lecture → Rem. 7.4.2.23], see also [Lecture → Code 7.4.2.24].

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).

▲

(7-3.f) ⚀ (15 min.) Implement a C++ function

```
void cvgQuadInf(void);
```

in order to study the convergence for $n \to \infty$ of the quadrature method implemented in the previous subproblem for the integrand $h(t) := \exp\left(-(t-1)^2\right)$ (shifted Gaussian). To compute the error you can use that $\int_{-\infty}^{\infty} h(t)dt = \sqrt{\pi}$.

Use MATPLOTLIBCPP to create a suitable plot of the quadrature error versus the number of quadrature nodes. What kind of convergence do you observe?

SOLUTION of (7-3.f):

C++-code 7.3.6: Implementation of `cvgQuadInf()`

```
2  void cvgQuadInf(void) {
3    // Number of max Gauss pts.
4    constexpr int N = 100;
5    plt::figure();
6    // TODO: (7-3.f) plot convergence errors against number of
       integration nodes
7    // START
8    // variables for the plots
9    std::vector<int> num_nodes;
10   std::vector<double> err;
11   std::vector<double> cvg_rate;
12
13   // Integrand and exact integral
14   auto f = [](double t) { return std::exp(-std::pow((t - 1), 2)); };
15   const double I_ex = std::sqrt(PI);
16
17   for (unsigned int n = 5; n <= N; n += 5) {
18     // Approximated value of integral
19     const double QS = quadinf(n, f);
20     // plot data
21     num_nodes.push_back(n);
22     err.push_back(std::abs(QS - I_ex));
23     //'straight' line in plot
24     cvg_rate.push_back(std::pow(0.73, n));
25   }
26   plt::semilogy(num_nodes, err, "+", {{"label", "Error"}});
27   plt::semilogy(num_nodes, cvg_rate, "--", {{"label", "O(0.73^n)"}});
28   plt::legend("best");
29   plt::xlabel("No. of quadrature nodes");
30   plt::ylabel("|Error|");
31
32   plt::grid("True");
33   // END
34   plt::savefig("./cx_out/convergence.png");
35 }
```

Get it on GitLab (quadinf.hpp).

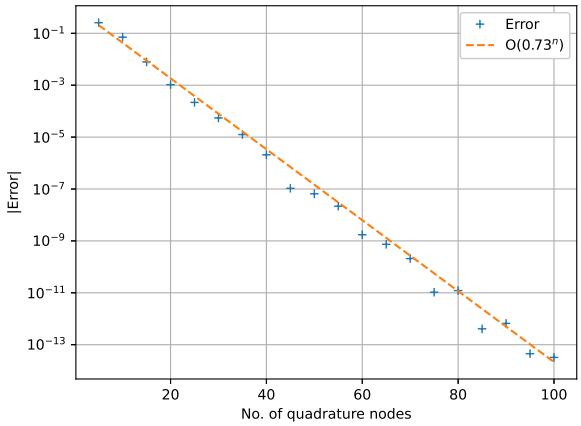


Fig. 61

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.

▲

End Problem 7-3 , 75 min.

Problem 7-4: Nested numerical quadrature

This problem addresses numerical quadrature over a two-dimensional domain. It turns out that this problem can be reduced to two one-dimensional integrals, which are amenable to the techniques covered in [Lecture → Chapter 7].

Implementation in C++ requested connected with [Lecture → Chapter 7].

A laser beam has intensity

$$I(x,y) := \exp(-\alpha((x-p)^2+(y-q)^2)) \,, \quad x,y \in \mathbb{R} \,, \quad \alpha > 0 \,, \tag{7.4.1}$$

on the plane orthogonal to the direction of the beam. The beam is directed orthogonal to the $x-y$-plane.

Note that the radiant power absorbed by a surface is the integral of the intensity over the surface.

(7-4.a) ⚀ (10 min.) Write down a formula involving only 1D integrals and giving the radiant power absorbed by the triangle

$$\Delta := \{(x,y)^T \in \mathbb{R}^2 \mid x \geq 0, y \geq 0, x+y \leq 1\} \,.$$

as a double integral.

SOLUTION 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.

▲

(7-4.b) ⚀ (15 min.) Write a C++ function

```
template <class Function>
double evalquad(const double a, const double b,
                Function &&f, const QuadRule & Q);
```

that evaluates an n-point quadrature formula as introduced in [Lecture → Section 7.2] for an integrand passed in `f` on domain $[a,b]$. It should rely on the quadrature rule on the reference interval $[-1,1]$ that is supplied through an object of type **QuadRule**:

```
struct QuadRule { Eigen::VectorXd nodes, weights; };
```

(The vectors `weights` and `nodes` denote the weights and nodes, respectively, of a quadrature rule on the reference interval $[-1,1]$, see [Lecture → Def. 7.2.0.1].)

SOLUTION 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()`.

```
template <class Function>
double evalquad(const double a, const double b, Function &&f,
                const QuadRule &Q) {
  double I = 0.;
  // TODO: (7-4.b) Use Q to approximate the integral of f over [a,b]
  // START
  // Loop over all nodes/weights pairs
  for (unsigned int i = 0; i < Q.weights.size(); ++i) {
    I += f((Q.nodes(i) + 1.) * (b - a) / 2. + a) * Q.weights(i);
  }
  I = I * (b - a) / 2.;
  // END
  return I;
}
```

▲

(7-4.c) ⚄ (20 min.) [depends on Sub-problem (7-4.a)]

Write a C++ function

```
template <class Function>
double gaussquadtriangle(const Function &f, const unsigned N)
```

for the computation of the integral

$$\int_{\Delta} f(x,y)\,dx\,dy \tag{7.4.3}$$

using nested n-point, 1D **Gauss quadrature** [Lecture → Section 7.4] in combination with the function `evalquad()` of Sub-problem (7-4.b). Use the function `gaussquad()` from `gaussquad.hpp` to obtain weights and nodes of Gauss quadrature formulas on $[-1,1]$.

HINT 1 for (7-4.c): Write (7.4.3) explicitly as a double integral as you were asked to do in Sub-problem (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()`.

```
template <class Function>
double gaussquadtriangle(const Function &f, const unsigned N) {
  double I = 0.;
  // TODO: (7-4.c) Use N-node gaussquad() to integrate f over
  // the triangle 0<=x,y, x+y<=1.
  // START
  // Get nodes/weights for integral over dx and dy
  QuadRule Q;
  gaussquad(N, Q);

  // We define the function g as the function of y
  // g(y) := ∫_0^{1-y} f(x,y)dx.
  auto g = [&f, &Q](double y) {
    return evalquad(
        0, 1 - y, [&f, &y](double x) { return f(x, y); }, Q);
  };
  // We integrate the function g over y from 0 to 1 to get
  // ∫_0^1 g(y)dy := ∫_0^1 ∫_0^{1-y} f(x,y)dxdy.
  I = evalquad(0, 1, g, Q);
  // END
  return I;
}
```

This problem can also be solved using loops (so no nested lambda function). This more straightforward approach is listed next.

C++-code 7.4.7: Loop-based implementation of `gaussquadtriangle_loop()`.

```
template <class Function>
double gaussquadtriangle_loop(const Function &f, const unsigned N) {
  // Get nodes/weights for integral over dx and dy
  QuadRule Q;
  gaussquad(N, Q);
  // Integration over y from 0 to 1 of g(y) := ∫_0^{1-y} I(x,y)dx
  double I = 0;
  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.;
}
```

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.

▲

(7-4.d) ⊡ Write a C++ function

```
void convtest2DQuad(unsigned int nmax = 20);
```

that applies the function `gaussquadtriangle()` from Sub-problem (7-4.c) to approximate $\int_\Delta I(x,y)\,\mathrm{d}x\mathrm{d}y$ using the parameters $\alpha = 1,\ p = 0,\ q = 0$. Tabulate and plot (using MATPLOTLIBCPP, storing the figure in `convergence.png`) the quadrature error w.r.t. to the number of nodes $n = 1,2,3,\ldots,20$ using the "exact" value of the integral $I \approx 0.366046550000405$. What kind of convergence do you observe and why?

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 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.

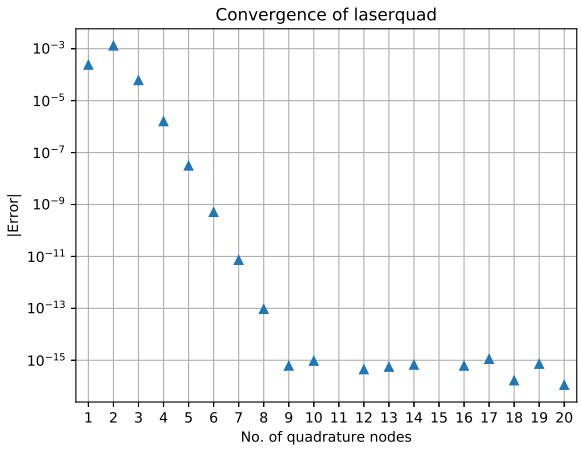


Fig. 62

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
12   auto I = [](double x, double y) {
13     return std::exp(-alpha * ((x - p) * (x - p) + (y - q) * (y - q)));
14   };
15
16   // memory allocation for plot
17   std::vector<double> err(nmax);
18   std::vector<int> num_pts(nmax);
19
20   std::cout << std::setw(3) << "N" << std::setw(15) << "I_approx"
21             << std::setw(15) << "error" << std::endl;
22   for (unsigned int n = 1; n <= nmax; n++) {
23     const double I_approx = gaussquadtriangle(I, n);
24
25     err[n - 1] = std::abs(I_ex - I_approx);
26     num_pts[n - 1] = n;
27
28     std::cout << std::setw(3) << n << std::setw(15) << I_approx << std::setw(15)
29               << err[n - 1] << std::endl;
30   }
31   // Error plot rendered by matplotlibcpp
32   plt::semilogy(num_pts, err, "^", {{"label", "error"}});
33   plt::xlim(0.5, nmax + 0.5);
34   plt::xticks(num_pts);
35   plt::xlabel("No. of quadrature nodes");
36   plt::ylabel("|Error|");
37   plt::title("Convergence of laserquad");
38   plt::grid("True");
39   // END
40   plt::savefig("./cx_out/convergence.png");
41 }
```

▲

End Problem 7-4 , 45 min.

Problem 7-5: Quadrature plots

In this problem, we will try and deduce the type of quadrature and the integrand from an error plot.

This is a purely theoretical problem discussing convergence of quadrature rules, see [Lecture → Lemma 7.4.3.6], [Lecture → Ex. 7.4.3.9], [Lecture → § 7.5.0.9], [Lecture → Exp. 7.5.0.12].

We consider three different functions on the interval $I = [0,1]$, which have the following properties:

$$\begin{aligned} &\text{function A:} \quad f_A \in C^\infty(I)\,, \quad f_A \notin \mathcal{P}_k \;\forall\, k \in \mathbb{N}\,; \\ &\text{function B:} \quad f_B \in C^0(I)\,, \quad f_B \notin C^1(I)\,; \\ &\text{function C:} \quad f_C \in \mathcal{P}_{12}\,, \end{aligned}$$

where $\mathcal{P}_k$ is the space of the polynomials of degree at most k defined on I. The following quadrature rules are applied to these functions:

- quadrature rule A is a global Gauss quadrature;
- quadrature rule B is a composite trapezoidal rule;
- quadrature rule C is a composite 2-point Gauss quadrature.

The corresponding absolute values of the quadrature errors are plotted against the number of function evaluations in the figure below. Notice that only the quadrature errors obtained with an even number of function evaluations are shown.

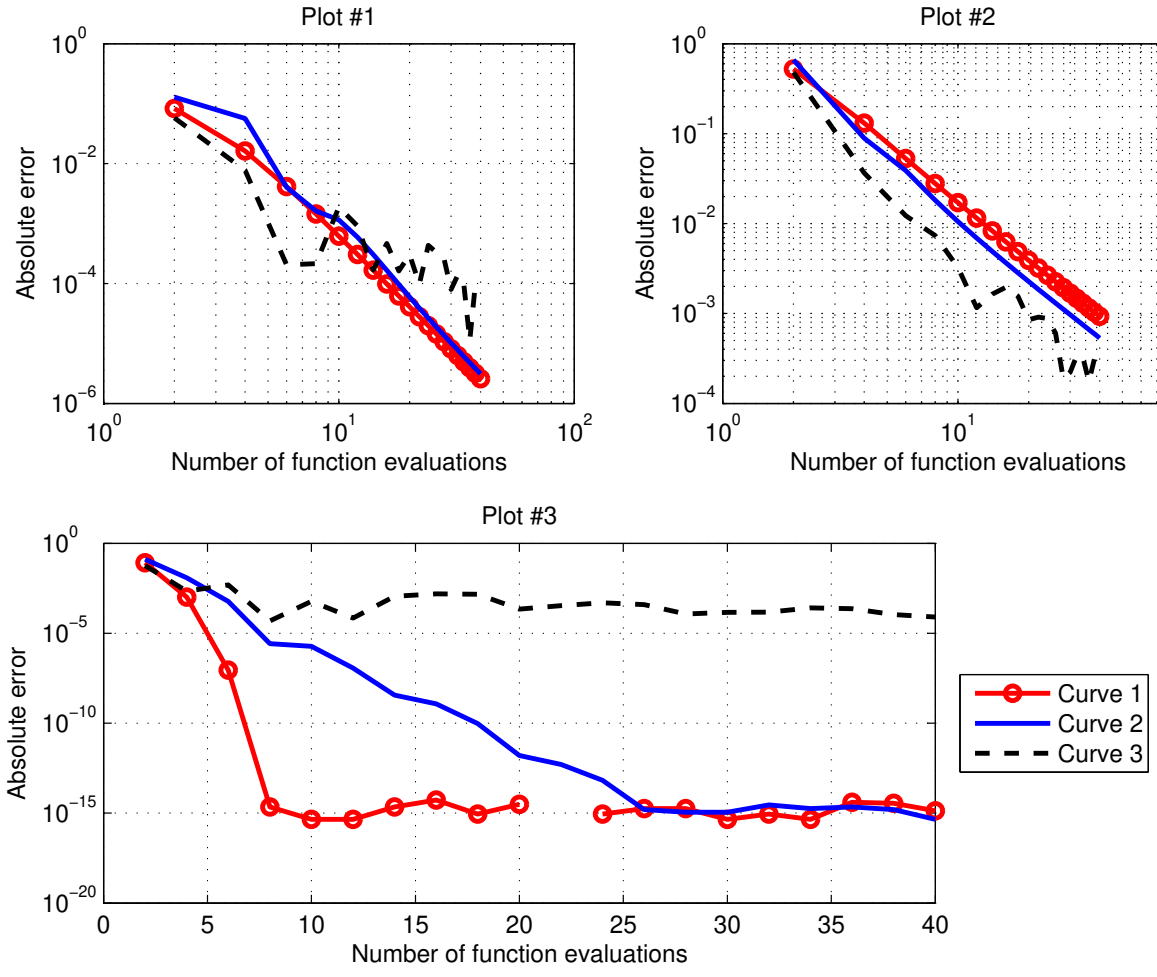


Fig. 63

(7-5.a) Match the three plots (plot #1, #2 and #3) with the three quadrature rules (quadrature rule A, B, and C). Justify your answer.

HINT 1 for (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.

▲

(7-5.b) ⚄ The quadrature error curves for a particular function f_A, f_B and f_C are plotted in the same style (curve 1 as red line with small circles, curve 2 means the blue solid line, curve 3 is the black dashed line). Which curve corresponds to which function (f_A, f_B, f_C)? Justify your answer.

SOLUTION of (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.

▲

End Problem 7-5

Problem 7-6: Weighted Gauss quadrature

In [Lecture → Rem. 7.4.3.10] we saw, how, in special cases, integrals with singular integrands can be transformed into integrals with smooth integrands, which allow much faster convergence of Gauss quadrature. This problem examines an alternative method: If the type of the singular behavior of the integrand is known, special quadrature rules, called weighted Gauss rules, can be designed that enjoy fast convergence.

This problem relies on the material of [Lecture → Section 7.4] and requests a little implementation in C++.

The development of an alternative quadrature formula for a particular singular integral relies on the Chebyshev polynomials of the second kind U_n, defined as

$$U_n(t) = \frac{\sin((n+1)\arccos t)}{\sin(\arccos t)}, \qquad n \in \mathbb{N}.$$

The U_n are **orthogonal polynomials** with respect to a weighted L^2 inner product (see [Lecture → Eq. (6.3.2.3)]) with weight function given by $w(\tau) = \sqrt{1-\tau^2}$.

(7-6.a) ⚀ (10 min.) Recall the role of the orthogonal Legendre polynomials in the derivation and definition of Gauss-Legendre quadrature rules (see [Lecture → § 7.4.2.15]). ▲

(7-6.b) ⚄ (15 min.) Show that the U_n satisfy the 3-term recursion

$$U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t), \qquad U_0(t) = 1, \qquad U_1(t) = 2t,$$

for every $n \geq 1$.

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.

▲

(7-6.c) ⚀ (5 min.) Show that $U_n \in \mathcal{P}_n$ and has leading coefficient 2^n.

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. □

▲

(7-6.d) ⚂ (15 min.) Show that for every $m, n \in \mathbb{N}_0$, we have

$$\int_{-1}^{1} \sqrt{1-t^2}\, U_m(t) U_n(t)\, dt = \frac{\pi}{2}\delta_{mn}.$$

SOLUTION of (7-6.d):

With the substitution $t = \cos s$ we obtain

$$\begin{aligned}
\int_{-1}^{1} \sqrt{1-t^2} U_m(t) U_n(t)\, dt &= \int_{-1}^{1} \sqrt{1-t^2} \frac{\sin((m+1)\arccos t)\sin((n+1)\arccos t)}{\sin^2(\arccos t)}\, dt \\
&= \int_0^{\pi} \sin s \frac{\sin((m+1)s)\sin((n+1)s)}{\sin^2 s} \sin s\, ds \\
&= \int_0^{\pi} \sin((m+1)s)\sin((n+1)s)\, ds \\
&= \frac{1}{2}\int_0^{\pi} \cos((m-n)s) - \cos((m+n+2)s)\, ds.
\end{aligned}$$

▲

(7-6.e) ⚂ (10 min.) What are the zeros ζ_j^n ($j = 1, \ldots, n$) of U_n, $n \geq 1$? Give an explicit formula similar to the formula for the Chebyshev nodes in $[-1, 1]$.

SOLUTION of (7-6.e):

From the definition of U_n we immediately find that the zeros are given by

$$\zeta_j^n = \cos\left(\frac{j}{n+1}\pi\right), \qquad j = 1, \ldots, n. \tag{7.6.1}$$

▲

(7-6.f) ⚄ (30 min.) [depends on Sub-problem (7-6.d)]

Show that the choice of weights

$$w_j = \frac{\pi}{n+1}\sin^2\left(\frac{j}{n+1}\pi\right), \qquad j = 1,\dots,n,$$

ensures that the quadrature formula

$$Q_n^U(f) = \sum_{j=1}^{n} w_j f(\xi_j^n) \tag{7.6.2}$$

provides the exact value of the integral

$$I := \int_{-1}^{1} \sqrt{1-t^2} f(t)\,\mathrm{d}t$$

for $f \in \mathcal{P}_{n-1}$ (assuming exact arithmetic).

SOLUTION of (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

$$\begin{aligned}
\sum_{j=1}^{n} w_j U_k(\xi_j^n) &= \sum_{j=1}^{n} \frac{\pi}{n+1}\sin^2\left(\frac{j}{n+1}\pi\right)\frac{\sin((k+1)\arccos\xi_j^n)}{\sin(\arccos\xi_j^n)} \\
&= x\sum_{j=1}^{n}\sin(jx)\sin((k+1)jx) \\
&= \frac{x}{2}\sum_{j=1}^{n}(\cos((k+1-1)jx) - \cos((k+1+1)jx)) \\
&= \frac{x}{2}\,\mathrm{Re}\sum_{j=0}^{n}\left(e^{ikxj} - e^{i(k+2)xj}\right) \\
&= \frac{x}{2}\,\mathrm{Re}\left(\sum_{j=0}^{n} e^{ikxj} - \frac{1-e^{i\pi(k+2)}}{1-e^{i(k+2)x}}\right).
\end{aligned}$$

Thus, for $k = 0$ we have

$$\sum_{j=1}^{n} w_j U_0(\xi_j^n) = \frac{x}{2}\,\mathrm{Re}\left(\sum_{j=0}^{n} 1 - \frac{1-e^{2\pi i}}{1-e^{2xi}}\right) = \frac{x}{2}\,\mathrm{Re}((n+1)-0) = \frac{\pi}{2}.$$

On the other hand, if $k = 1,\dots,n-1$ we obtain

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{x}{2}\,\mathrm{Re}\left(\frac{1-e^{i\pi k}}{1-e^{ikx}} - \frac{1-e^{i\pi(k+2)}}{1-e^{i(k+2)x}}\right) = \frac{(1-(-1)^k)x}{2}\,\mathrm{Re}\left(\frac{1}{1-e^{ikx}} - \frac{1}{1-e^{i(k+2)x}}\right).$$

In view of the elementary equality $(a+ib)(a-ib) = a^2+b^2$ we have $\mathrm{Re}(1/(a+ib)) = a/(a^2+b^2)$. Thus

$$\mathrm{Re}\left(\frac{1}{1-e^{ikx}}\right) = \mathrm{Re}\left(\frac{1}{1-\cos(kx)-i\sin(kx)}\right) = \frac{1-\cos(kx)}{(1-\cos(kx))^2+\sin(kx)^2} = \frac{1}{2}$$

Arguing in a similar way we have $\mathrm{Re}\,(1-e^{i(k+2)x})^{-1} = 1/2$. Therefore for $k = 1,\ldots,n-1$ 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

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{\pi}{2}\delta_{k0}, \qquad k = 0,\ldots,n-1.$$

Finally, the claim follows from Sub-problem (7-6.d), since $U_0(t) = 1$.

▲

(7-6.g) ⚀ Show that the quadrature formula (7.6.2) gives the exact value of (**??**) even for every $f \in \mathcal{P}_{2n-1}$.

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 → Thm. 7.4.2.11] with the weighted L^2 scalar product with weight w defined above.

▲

(7-6.h) ⚁ Show that the quadrature error

$$|Q_n^U(f) - W(f)|$$

decays to 0 exponentially as $n \to \infty$ for every $f \in C^\infty([-1,1])$ that admits an analytic extension to an open subset of the complex plane.

HINT 1 for (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 → § 7.4.3.2], we obtain the exponential decay, as desired.

▲

(7-6.i) ⚂ Write a C++ function

```
template<typename Function>
double quadU(const Function & f, const unsigned n);
```

that gives $Q_n^U(f)$ as output, where `f` is an object with an evaluation operator, like a lambda function, representing f, e.g.

```
auto f = [] (double & t) { return 1/(2 + exp(3*t)); };
```

SOLUTION of (7-6.i):

The full code can be found in `quadU.cpp`.

C++ code 7.6.3: Solution for `quadU`.

```
template <typename Function>
double quadU(const Function& f, const unsigned int n) {
  // Value of integral
  double I = 0;

  // TODO: (7-6.i) Integrate f using weighted Gauss quadrature
  // START
  for (unsigned int j = 0; j < n; ++j) {
    // Weight
    const double w =
        M_PI / (n + 1) * std::pow(std::sin((j + 1) * M_PI / (n + 1)), 2);

    // Node
    const double xi = std::cos((j + 1.) / (n + 1) * M_PI);

    I += w * f(xi);
  }
  // END
  return I;
}
```

▲

(7-6.j) ⚀ Devise a C++ function

```
void testQuadU(unsigned int nmax = 20);
```

that, for the function $f(t) = 1/(2+e^{3t})$ and $n = 1,\ldots,n_{\max}$ plots (employing MATPLOTLIBCPP) the quadrature error $E_n(f) = |W(f) - Q_n^U(f)|$ using the "exact" value $W(f) = 0.483296828976607$. Estimate the parameter $0 \leq q < 1$ in the asymptotic decay law $E_n(f) \approx Cq^n$ characterising (sharp) exponential convergence, see [Lecture → Def. 6.2.2.7].

SOLUTION of (7-6.j):

The full code can be found in `quadU.cpp`.

C++ code 7.6.4: Computation of convergence of `quadU`.

```
void testQuadU(unsigned int nmax = 20) {
  // "Exact" value of integral
```

```
4   constexpr double exact = 0.483296828976607;
5   // Integrand
6   auto f = [](double t) { return 1. / (2. + std::exp(3. * t)); };
7
8   plt::figure();
9
10  // TODO: (7-6.j) Tabulate and plot the quadrature error.
11  // START
12  // memory allocation for plot
13  std::vector<double> err(nmax);
14  std::vector<int> num_pts(nmax);
15
16  std::cout << std::setw(5) << "n" << std::setw(20) << "Error" << std::setw(20)
17            << "q" << std::endl;
18
19  for (unsigned int n = 1; n <= nmax; ++n) {
20    // Compute error
21    err[n - 1] = std::abs(quadU(f, n) - exact);
22    num_pts[n - 1] = n;
23
24    if (n > 1) {
25      // Print table
26      std::cout << std::setw(5) << n << std::setw(20) << err[n - 1]
27                << std::setw(20) << err[n - 1] / err[n - 2] << std::endl;
28    }
29  }
30  // Error plot rendered by matplotlibcpp
31  plt::semilogy(num_pts, err, "^", {{"label", "error"}});
32  plt::xlim(0.5, nmax + 0.5);
33  plt::xticks(num_pts);
34  plt::xlabel("No. of quadrature nodes");
35  plt::ylabel("|Error|");
36  plt::title("Convergence of quadU");
37  plt::grid("True");
38  // END
39  plt::savefig("./cx_out/convergence.png");
40 }
```

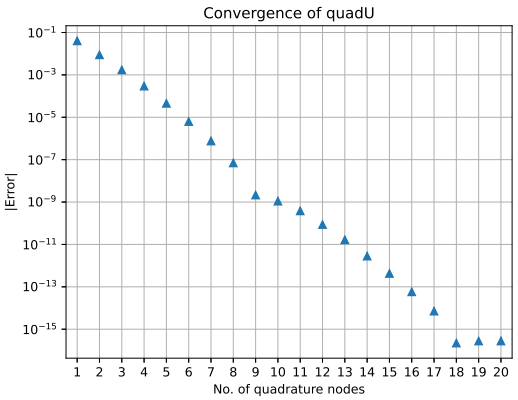


Fig. 64

An approximation of q is given by $E_n(f)/E_{n-1}(f)$:

n	Error	Approximated q
2	0.00171981	0.194545
3	0.000294532	0.171259
4	4.54948e-05	0.154465
5	6.30403e-06	0.138566
6	7.57006e-07	0.120083
7	6.99966e-08	0.0924651
8	2.14329e-09	0.0306199
9	1.09843e-09	0.512495
10	3.82291e-10	0.348035
11	8.71921e-11	0.228078
12	1.66109e-11	0.190509
13	2.80703e-12	0.168988
14	4.28158e-13	0.15253
15	5.80647e-14	0.135615
16	7.10543e-15	0.122371
17	2.77556e-16	0.0390625
18	2.77556e-16	1
19	2.77556e-16	1

▲

End Problem 7-6 , 85 min.

Problem 7-7: Quadrature by transformation

In [Lecture → Rem. 7.4.3.10] a suitable transformation converted a singular integrand into a smooth one. In this problem we see another example. Also Problem 7-9 and Problem 7-1 cover this topic.

This problem does not contain an implementation part. Study [Lecture → Rem. 7.4.3.10] to learn about regularizing transformations.

For $f \in C^0([0,2])$ we consider the definite improper integral with *singular integrand*

$$I(f) := \int_0^2 \sqrt{\frac{2-t}{t}} f(t)\,\mathrm{d}t\,. \tag{7.7.1}$$

(7-7.a) ⚀ Transform $I(f)$ into an integral over $[-1,1]$, whose integrand is $\in C^\infty([-1,1])$ provided that $f \in C^\infty([0,2])$.

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.

▲

(7-7.b) ⚀ (20 min.) For $n \in \mathbb{N}^*$, derive a family of $2n$-point quadrature formulas

$$Q_n(f) = \sum_{j=1}^{2n} w_j^n f(c_j^n), \qquad f \in C([0,2])$$

for which $Q_n(f)$ converges to $I(f)$ exponentially as $n \to \infty$, if $f \in C^\infty([0,2])$.

Remark. A more precise statement of the problem would be ", if f has an analytic extension to a complex neighborhood of $[0,2]$". This is, how the statement $f \in C^\infty([0,2])$ above should be read.

SOLUTION 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

$$w_j^n = \begin{cases} \tilde{w}_j^n \sqrt{2 - c_j^n} & \text{if } j = 1, \dots, n, \\ \tilde{w}_{j-n}^n \frac{2 - c_j^n}{\sqrt{c_j^n}} & \text{if } j = n+1, \dots, 2n. \end{cases}$$

▲

(7-7.c) ⚀ (10 min.) Given $n > 0$, determine the maximal $d \in \mathbb{N}$ such that $I(p) = Q_n(p)$ for every polynomial $p \in \mathcal{P}_{d-1}$.

SOLUTION 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 $\mathcal{P}_{-1} = \emptyset$.

▲

End Problem 7-7, 30 min.

Problem 7-8: Discretization of an integral operator

In this problem we consider a completely new concept, an **integral operator** of the form

$$(\mathsf{T}f)(x) = \int_I k(x,y)f(y)\,\mathrm{d}y\,, \quad x \in I \subset \mathbb{R}\,, \tag{7.8.1}$$

where the kernel k is continuous on $I \times I$, $I \subset \mathbb{R}$ a closed interval. It maps a function on I to another function on I. Such integral operators are very common in models of continuous physics. Of course, their numerical treatment heavily draws on numerical quadrature.

Requires knowledge about Gauss quadrature from [Lecture → Section 7.4] and involves substantial implementation in C++.

Given $f \in C^0([0,1])$, the integral

$$g(x) := \int_0^1 e^{|x-y|} f(y)\,dy$$

defines a function $g \in C^0([0,1])$. We approximate the integral by means of n-point Gauss quadrature, which yields a function $g_n(x)$.

(7-8.a) ⚄ Let $\{\xi_j^n\}_{j=1}^n$ be the nodes for the Gauss quadrature on the interval $[0,1]$. Assume that the nodes are ordered, namely $\xi_j^n < \xi_{j+1}^n$ for every $j = 1,\ldots,n-1$. We can write

$$(g_n(\xi_l^n))_{l=1}^n = M(f(\xi_j^n))_{j=1}^n,$$

for a suitable matrix $M \in \mathbb{R}^{n,n}$. Give a formula for the entries of M.

SOLUTION of (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 $l,j = 1,\ldots,n$.

▲

(7-8.b) ⚄ Implement a function

```
template <typename Function>
Eigen::VectorXd comp_g_gausspts(Function f, unsigned int n)
```

that computes the n-vector $(g_n(\xi_l^n))_{l=1}^n$ with optimal complexity $O(n)$ (excluding the computation of the nodes and weights), where `f` is an object with an evaluation operator, e.g. a lambda function, that represents the function f.

You may use the provided function

```
void gaussrule(int n, Eigen::VectorXd & w, Eigen::VectorXd & xi)
```

that computes the weigths `w` and the ordered nodes `xi` relative to the `n`-point Gauss quadrature on the interval $[-1,1]$.

SOLUTION of (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).

▲

(7-8.c) ⚄ Test your implementation and write a C++ function

```
double testCompGGaussPts(void);
```

that computes $g(\xi_{11}^{21})$ for $f(y) = e^{-|0.5-y|}$. What result do you expect?

SOLUTION of (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).

▲

End Problem 7-8

Problem 7-9: Efficient quadrature of singular integrands

We know that smoothness of the integrand is essential for fast covergence of high-order numerical quadrature like Gauss quadrature rules. In some cases smart transformation can convert the integral into another with a smooth integrand. This problem uses this trick for the sake of efficient numerical quadrature of non-smooth integrands with a special structure.

Before you tackle this problem, read about regularization of integrands by transformation, cf. [Lecture → Rem. 7.4.3.10]. For other problems on the same topic see Problem 7-7 and Problem 7-1.

Our task is to develop quadrature formulas for integrals of the form:

$$W(f) := \int_{-1}^{1} \sqrt{1-t^2}\, f(t)\, dt, \tag{7.9.1}$$

where f possesses an analytic extension to a complex neighbourhood of $[-1,1]$.

(7-9.a) ⚀ (10 min.) Understand how to use the C++ function

```
QuadRule gauleg(unsigned int n);
```

(contained in `gauleg.hpp`), which returns a structure `QuadRule` containing nodes (x_j) and weights (w_j) of a Gauss-Legendre quadrature [Lecture → Def. 7.4.2.18] on $[-1,1]$ with n nodes. ▲

(7-9.b) ⚁ (15 min.) Study [Lecture → § 7.4.3.2] in order to learn about the convergence of Gauss-Legendre quadrature. What can you say about the least expected convergence for an integrand $f \in C^r([a,b])$? What about convergence in the case $f \in C^\infty([a,b])$?

SOLUTION 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.

▲

(7-9.c) ⚁ (40 min.) Based on the function `gauleg()`, implement a C++ function

```
template <class Function>
double quadsingint(const Function& f, const unsigned n);
```

that approximately evaluates $W(f)$ using $2n$ evaluations of f. An object of type `Function` must provide an evaluation operator

```
double operator(double t) const;
```

Ensure that, as $n \to \infty$, the error of your implementation has asymptotic exponential convergence to zero.

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).

▲

(7-9.d) ⚀ Give formulas for the nodes c_j and weights $\tilde{w}_j$ of a $2n$-point quadrature rule on $[-1,1]$, whose application to the integrand f will produce the same results as the function `quadsingint` that you implemented in the previous subproblem.

SOLUTION of (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:

$$\tilde{w}_{2j-i} = 2w_j x_j^2\sqrt{2-x_j^2}, \qquad j = 1,\ldots,n,\ i = 0,1.$$

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:

$$\tilde{w}_j = w_j\cos^2(x_j\pi/2)\pi/2.$$

▲

(7-9.e) 2 Implement a C++ function

```
void tabQuadErr(void)
```

that tabulates the quadrature error:

$$\epsilon_i := |W(f) - \texttt{quadsingint(f,n)}|$$

for $f(t) := \dfrac{1}{2+\exp(3t)}$ and $n = 1,2,...,25$. Then describe the type of convergence observed, see [Lecture → Def. 6.2.2.7].

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.

```
2  void tabAndPlotQuadErr() {
3    // Max num of Gauss points to use (in each direction)
4    constexpr unsigned max_N = 25;
5    // "Exact" integral
6    constexpr double I_ex = 0.483296828976607;
7
8    auto f = [](double t) { return 1. / (2. + std::exp(3. * t)); };
9
10   plt::figure();
11   // TODO: (7-9.e) Tabulate and plot quadrature error for the given function.
12   // START
13   std::vector<int> num_nodes(max_N);  // number of nodes on x axis
14   std::vector<double> err(max_N);     // error on y axis
15
16   // Table header
```

```
17    std::cout << std::setw(3) << "N" << std::setw(15) << "I_approx"
18              << std::setw(15) << "error" << std::endl;
19
20    for (unsigned N = 1; N <= max_N; ++N) {
21      const double I_approx = quadsingint(f, N);
22      num_nodes[N - 1] = N;
23      err[N - 1] = std::abs(I_ex - I_approx);
24
25      std::cout << std::setw(3)
26                << N
27                // Value of integral
28                << std::setw(15)
29                << I_approx
30                // Error
31                << std::setw(15) << err[N - 1] << std::endl;
32    }
33    plt::semilogy(num_nodes, err, "+", {{"label", "Error"}});
34    plt::xlabel("No. of quadrature nodes");
35    plt::ylabel("|Error|");
36    plt::title("Convergence of quadsingint");
37    plt::grid("True");
38    // END
39    plt::savefig("./cx_out/convergence.png");
40  }
```

Get it on GitLab (quadsingint.hpp).

▲

End Problem 7-9 , 65 min.

Problem 7-10: Zeros of orthogonal polynomials

This problem combines elementary methods for finding zeros from [Lecture → Section 8.4] and 3-term recursions satisfied by orthogonal polynomials, *cf.* [Lecture → Thm. 6.3.2.14]. This will permit us to find the zeros of Legendre polynomials, the so-called Gauss nodes (see [Lecture → Def. 7.4.2.18]).

Connected with [Lecture → Section 7.4] and [Lecture → Section 8.4]

The zeros of the Legendre polynomial P_n (see [Lecture → Def. 7.4.2.16]) are the n Gauss points ξ_j^n, $j = 1, \ldots, n$. In this problem we compute the Gauss points by zero-finding methods applied to P_n. The 3-term recursion [Lecture → Eq. (7.4.2.21)] for Legendre polynomials will play an essential role. Moreover, recall that, by definition, the Legendre polynomials are $L^2(]-1,1[)$-orthogonal.

(7-10.a) ⚄ Prove the following interleaving property of the zeros of the Legendre polynomials. For all $n \in \mathbb{N}_0$ we have:

$$-1 < \xi_j^n < \xi_j^{n-1} < \xi_{j+1}^n < 1, \qquad j = 1, \ldots, n-1 \,.$$

HINT 1 for (7-10.a): You may follow these steps:

1. Understand that it is enough to show that every pair of zeros (ξ_l^n, ξ_{l+1}^n) 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 $l = 1, \ldots, n-1$ 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:

$$0 = \int_{-1}^{1} P_{n-1}q\,dt \quad = \sum_{j=1}^{n} w_j P_{n-1}(\xi_j^n)q(\xi_j^n) = w_l P_{n-1}(\xi_l^n)q(\xi_l^n) + w_{l+1}P_{n-1}(\xi_{l+1}^n)q(\xi_{l+1}^n)\,.$$

Hence, both terms of the right hand side must have different sign, namely:

$$\left[w_l P_{n-1}(\xi_l^n)q(\xi_l^n)\right]\left[w_{l+1}P_{n-1}(\xi_{l+1}^n)q(\xi_{l+1}^n)\right] \le 0\,.$$

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 $j = 1,\ldots,n$. 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:

$$(-1)^{n+1-j}P_{n+1}(\xi_j^n) > 0, \qquad j = 1,\ldots,n\,. \tag{7.10.3}$$

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 $n+1$.

▲

(7-10.b) ⊡ By differentiating [Lecture → Eq. (7.4.2.21)] derive a combined 3-term recursion for the sequences $(P_n)_n$ and $(P_n')_n$.

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.

▲

(7-10.c) ⚀ Use the recursions obtained in (7-10.b) to write a C++ function

```
void legvals(const Eigen::VectorXd& x,
Eigen::MatrixXd& Lx, Eigen::MatrixXd& DLx);
```

that fills the matrices `Lx` and `DLx` in $\mathbb{R}^{N\times(n+1)}$ with the values $\{P_k(x_j)\}_{jk}$ and $\{P'_k(x_j)\}_{jk}$, $j = 0,\ldots,N-1$ and $k = 0,\ldots,n$, for an input vector $x \in \mathbb{R}^N$ (passed in `x`):

SOLUTION of (7-10.c):

C++ code 7.10.4: Suggested solution for `legvals()`

```
2  void legvals(const Eigen::VectorXd& x, Eigen::MatrixXd& Lx,
3               Eigen::MatrixXd& DLx) {
4    const unsigned int n = Lx.cols() - 1;
5    const unsigned int N = x.size();
6
7    // TODO: (7-10.c) Fill matrices Lx and DLx with values of the
       legendre
8    // polynomials at x, i.e.  {P_k(x_j)}_jk and {P'_k(x_j)}_jk START
9    assert(n >= 2);
10   for (unsigned int j = 0; j < N; ++j) {
11     Lx(j, 0) = 1.;
12     Lx(j, 1) = x(j);
13     DLx(j, 0) = 0;
14     DLx(j, 1) = 1.;
15     for (unsigned int k = 2; k <= n; ++k) {
16       Lx(j, k) =
17           (2 * k - 1.) / k * x(j) * Lx(j, k - 1) - (k - 1.) / k * Lx(j, k - 2);
18       DLx(j, k) = (2 * k - 1.) / k * Lx(j, k - 1) +
19                   (2 * k - 1.) / k * x(j) * DLx(j, k - 1) -
20                   (k - 1.) / k * DLx(j, k - 2);
21     }
22   }
23   // END
24 }
```

Get it on GitLab (legendre.hpp).

▲

(7-10.d) ⚀ We can compute the zeros of P_k, $k = 1,\ldots,n$, by means of the secant rule (see [Lecture → § 8.4.2.28]) using the endpoints $\{-1,1\}$ of the interval and the zeros of the previous Legendre polynomial as initial guesses; see (7-10.a). We opt for a correction-based termination criterion (see [Lecture → Section 8.2.3]) based on prescribed relative and absolute tolerance (see [Lecture → Code 8.4.2.31]).

Write a C++ function that computes the Gauss points $\zeta_j^k \in [-1,1]$, $j = 1,\ldots,k$ and $k = 1,\ldots,n$, using the zero finding approach outlined above:

```
Eigen::MatrixXd gaussPts(const int n,
                         const double rtol=1e-10, const double
                            atol=1e-12)
```

The Gauss points should be returned in an upper triangular $n \times n$-matrix.

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);
  Px(0) = 1.;
  Px(1) = x;
  for (unsigned int k = 2; k <= n; ++k) {
```

```
    Px(k) = (2 * k - 1.) / k * x * Px(k - 1) - (k - 1.) / k * Px(k - 2);
  }
  // END
  return Px(n);
}
```

Get it on GitLab (legendre.hpp).

▲

(7-10.e) Validate your implementation of the function `gaussPts` with $n = 8$ by computing the values of the Legendre polynomials in the zeros obtained (use the function `legvals`). Explain the failure of the method.

HINT 1 for (7-10.e): Fig. 65 shows P_7 and P_8 on a part of $[-1,1]$. The secant method fails to find the zeros of P_8 (blue curve) when started with the zeros of P_7 (red curve).

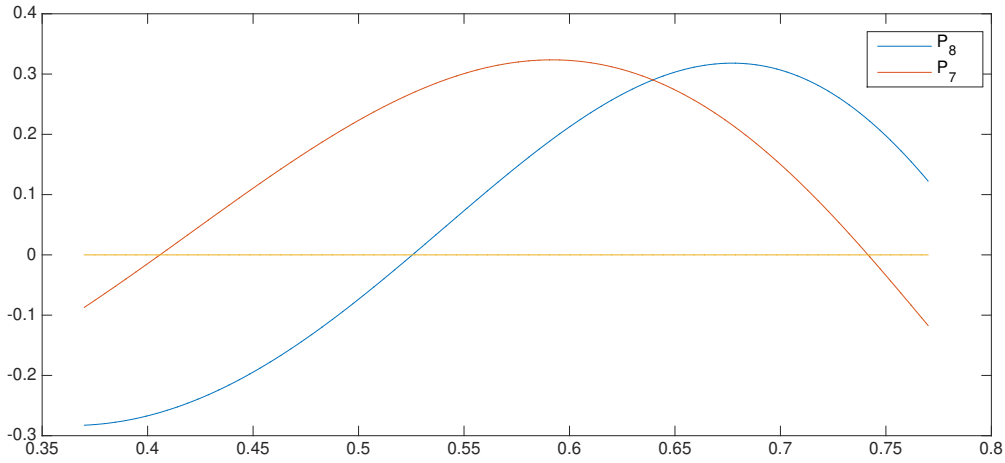


Fig. 65

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. 65, 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.

▲

(7-10.f) Fix your function `gaussPts` taking into account the above considerations. You should use the *regula falsi*, that is a variant of the secant method in which, at each step, we choose the old iterate to keep depending on the signs of the function. More precisely, given two approximations $x^{(k)}$, $x^{(k-1)}$ of a zero in which the function f has different signs, compute another approximation $x^{(k+1)}$ as zero of the secant. Use this as the next iterate, but then chose as $x^{(k)}$ the value $z \in \{x^{(k)}, x^{(k-1)}\}$, for which $\operatorname{sign}\left[f\left(x^{(k+1)}\right)\right] \neq \operatorname{sign}[f(z)]$. This ensures that f has always a different sign in the last two iterates.

The regula falsi variation of the secant method can be easily implemented with a little modification of the code.

SOLUTION 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)

```
2  Eigen::MatrixXd gaussPts_regulaFalsi(const unsigned int n,
3                                       const double rtol = 1e-10,
4                                       const double atol = 1e-12) {
5    Eigen::MatrixXd zeros = Eigen::MatrixXd::Zero(n, n);
6
7    // TODO: (7-10.f) Find the zeros of the legendre polynomials using
         the secant
8    // method with regula falsi.  START
9    double x0, x1, f0, f1, s;
10   for (unsigned int k = 1; k <= n; ++k) {
11     for (unsigned int j = 1; j <= k; ++j) {
12       // Initial guesses
13       if (j == 1)
14         x0 = -1;
15       else
16         x0 = zeros(j - 2, k - 2);
17       if (j == k)
18         x1 = 1;
19       else
20         x1 = zeros(j - 1, k - 2);
21
22       // Secant method
23       f0 = Pnx(x0, k);
24       for (unsigned int i = 0; i < 1e4; ++i) {
25         f1 = Pnx(x1, k);
26         s = f1 * (x1 - x0) / (f1 - f0);
27         if (Pnx(x1 - s, k) * f1 < 0) {  // regula falsi
28           x0 = x1;
29           f0 = f1;
30         }
31         x1 -= s;
32         if ((std::abs(s) <
33              std::max(atol, rtol * std::min(std::abs(x0), std::abs(x1))))) {
34           zeros(j - 1, k - 1) = x1;
35           break;
36         }
37       }
38     }
39   }
40   // END
41   return zeros;
42 }
```

Get it on GitLab (legendre.hpp).

▲

End Problem 7-10

Problem 7-11: Quadrature rules and quadrature errors

When sequences of quadrature rules are applied to evaluate $\int_a^b f(t)\,\mathrm{d}t$, then the asymptotic convergence of quadrature error in terms of $n \to \infty$, $n \mathrel{\hat{=}}$ the number of quadrature nodes, is determined both by

(i) the smoothness of the integrand f,
(ii) family of quadrature rules used.

This problems asks you to guess the family of quadrature rules from error curves.

To prepare for this problem study [Lecture → Section 7.4], in particular [Lecture → Ex. 7.4.3.9], and also [Lecture → Exp. 7.5.0.12], [Lecture → Exp. 7.5.0.16]

In this problem we consider three different families of quadrature rules, for which we use the following abbreviations:

(TR) $\mathrel{\hat{=}}$ Equidistant trapezoidal rule, see [Lecture → Eq. (7.5.0.17)],

(SR) $\mathrel{\hat{=}}$ Equidistant Simpson rule, see [Lecture → Eq. (7.5.0.5)],

(GR) $\mathrel{\hat{=}}$ Gauss-Legendre quadrature rules, as defined in [Lecture → Def. 7.4.2.18].

(7-11.a) ⚄ (15 min.) The following C++ function implements the n-point equidistant trapezoidal rule, $n \in \mathbb{N}$, $n \geq 2$, for the approximate evaluation of $\int_a^b f(t)\,\mathrm{d}t$. The integrand is passed through the functor `f`.

```
template <typename Functor>
double equidTrapezoidalRule(Functor &&f, double a, double b, unsigned int n) {
  assert(n>=2);
  const double h = (b - a)/(n-1);
  double t = a + h, s = 0.0;
  for (unsigned int i = 1; i < n-1; t += h, ++i) { s += f(t); }
  return (          * f(a) +          * f(b) +          * s);
}
```

Supplement the missing C++ code in the boxes.

SOLUTION 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`.

▲

(7-11.b) ⚂ (15 min.) The following plots display the quadrature errors for the approximation of the integral

$$\int_0^1 f_1(t)\,\mathrm{d}t \quad , \quad f_1(t) := |\sin(5t)| \,,$$

by the three families of quadrature rules and numbers $n = 4, \ldots, 100$ of quadrature points.

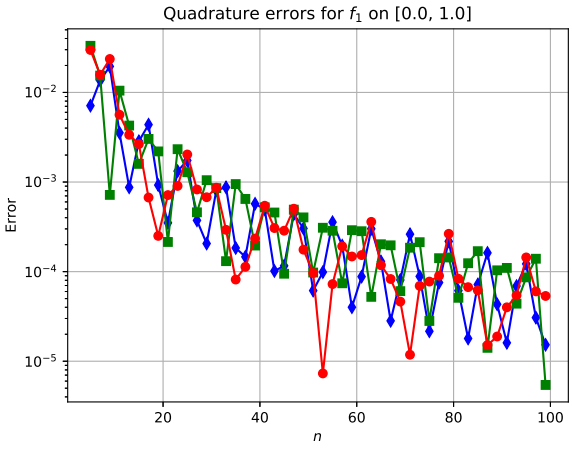


Fig. 66

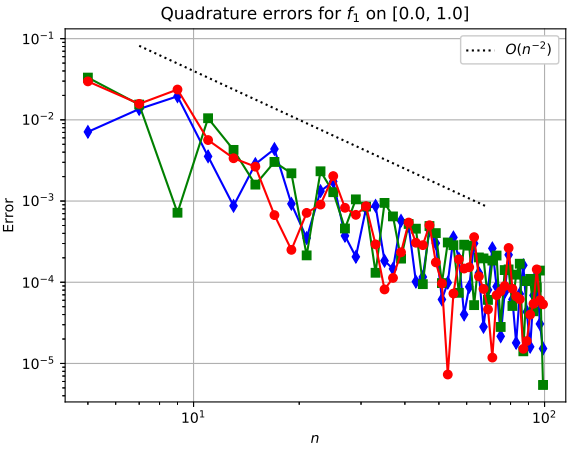
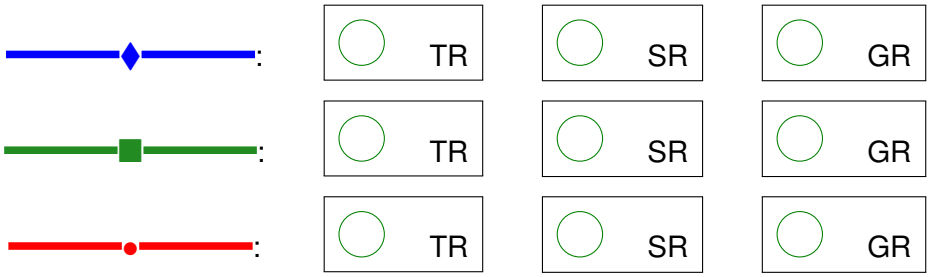


Fig. 67

Tick the families of quadrature rules, if the curve correctly represents the expected behavior of its quadrature error as a function of n.



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.



The actual association of rules and error curves as as follows:

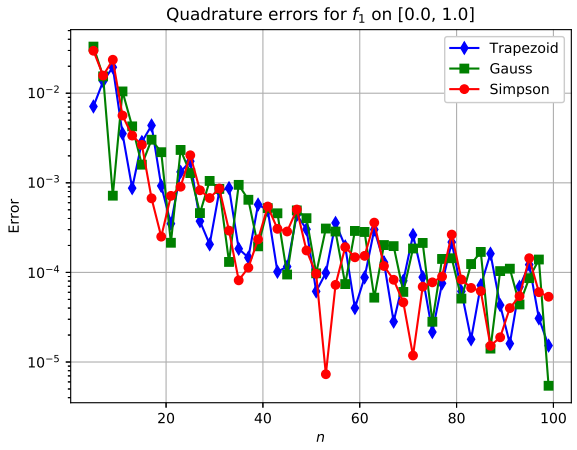


Fig. 68

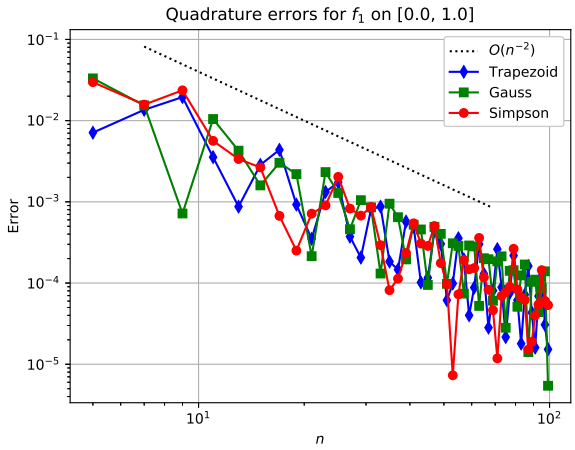


Fig. 69

▲

(7-11.c) ⚂ (15 min.) Below we have plotted the quadrature errors for the approximation of the integral

$$\int_0^1 f_2(t)\,\mathrm{d}t \quad , \quad f_2(t) := |\sin(5t)|^2 \,,$$

for the three families of quadrature rules and numbers $n = 4, \ldots, 100$ of quadrature points.

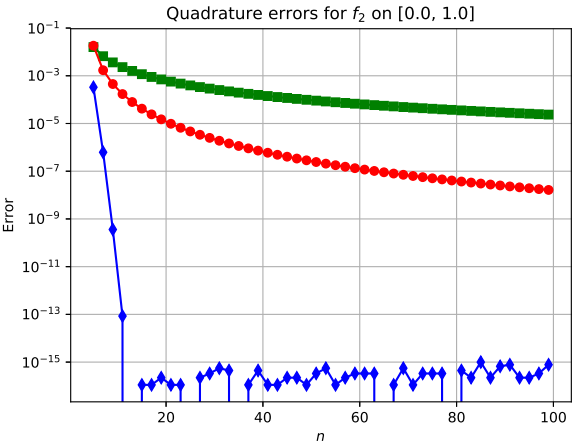


Fig. 70

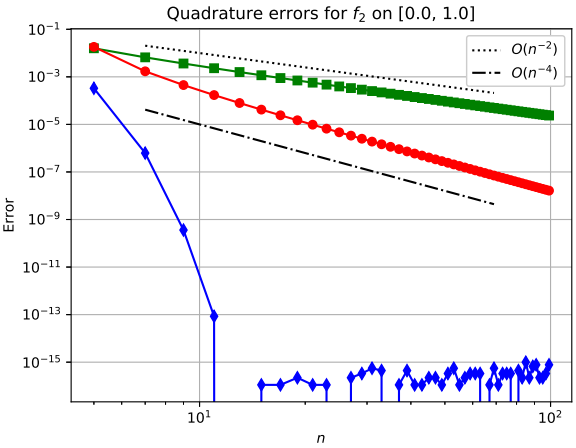


Fig. 71

Select the quadrature rule, if the curve correctly represents the expected behavior of its quadrature error as a function of n.



: ☐ TR ☐ SR ☐ GR



: ☐ TR ☐ SR ☐ GR



: ☐ TR ☐ SR ☐ GR

SOLUTION of (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:

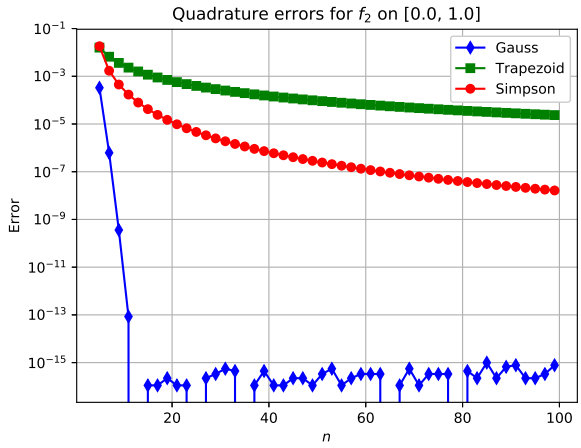


Fig. 72

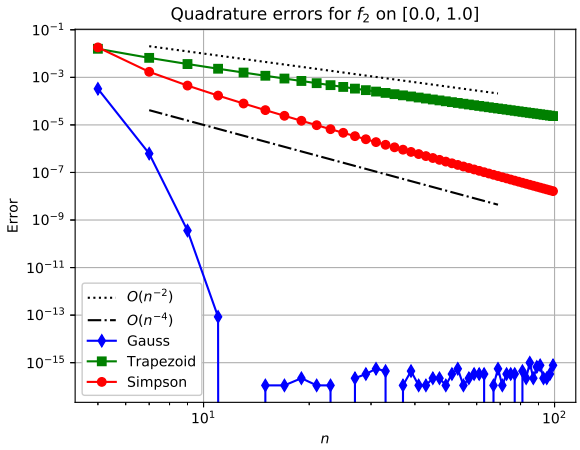


Fig. 73

▲

(7-11.d) ⚂ (15 min.) The plots below display the quadrature errors for

$$\int_0^1 f_3(t)\,\mathrm{d}t\,, \quad f_3(t) := |\sin(5t)|^5\,,$$

and for equidistant trapezoidal rules (TR), equidistant Simpson rules (SR), and Gauss-Legendre rules (GR) as a function of the number n of quadrature points, $n = 4, \ldots, 100$.

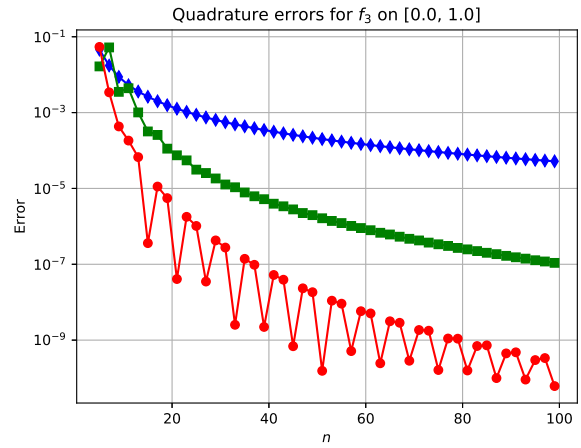


Fig. 74

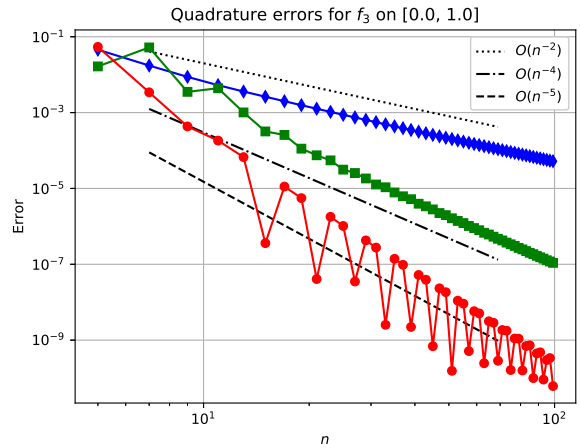
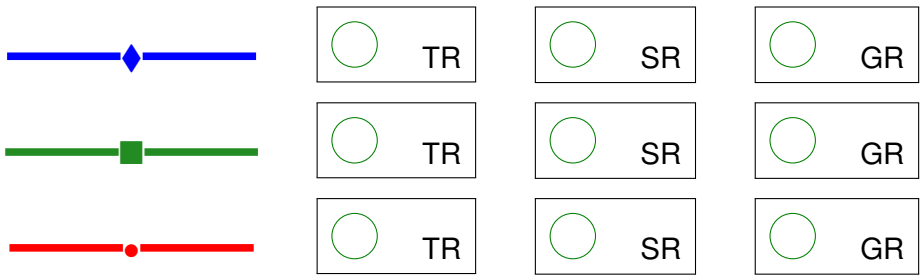


Fig. 75

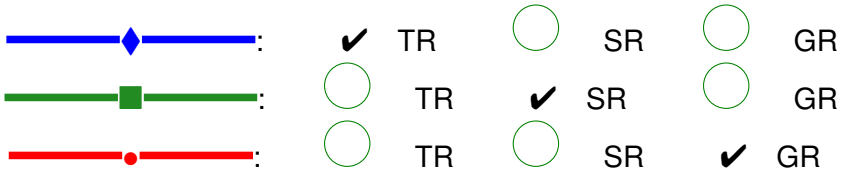
Select those quadrature rules, for which the curve correctly represents the expected behavior of its quadrature error as a function of n.



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:

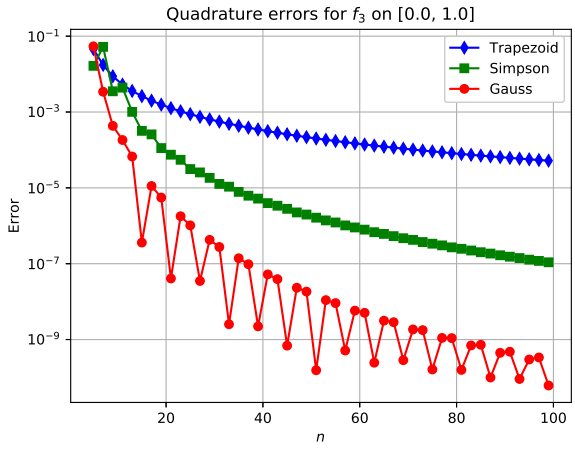


Fig. 76

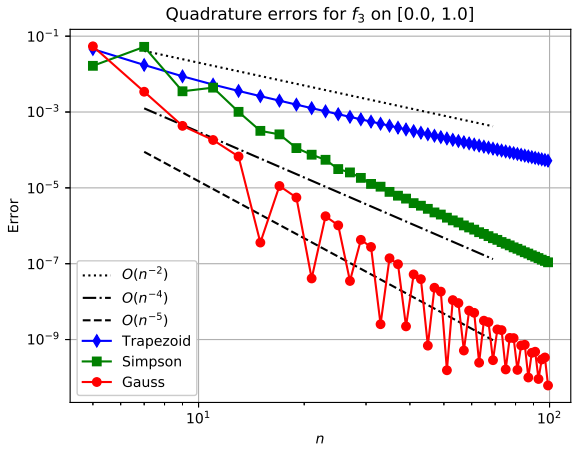


Fig. 77

▲

(7-11.e) ⚂ (20 min.) Decide whether the following assertions about quadrature rules are true or false

(i) For any given set of n quadrature nodes $\in [0,1]$ there is a quadrature rule of order n with those nodes.

○ true ○ false

(ii) Let Q_n designate an n-point quadrature rule of order n on $[0,1]$. Then

$$f \geq 0 \quad \Rightarrow \quad Q_n(f) \geq 0\,.$$

○ true ○ false