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

Main Examination

Prof. R. Hiptmair, SAM, ETH Zurich

January 26, 2017

Duration: 3h 20m (computer-based)

(Examination for Course at ETH Zurich in Autumn Term 2016)

<table>
<thead>
<tr>
<th>Family name</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td></td>
</tr>
<tr>
<td>Study program</td>
<td></td>
</tr>
<tr>
<td>Computer name</td>
<td></td>
</tr>
<tr>
<td>Legi no.</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>26.01.2016</td>
</tr>
</tbody>
</table>

Points:

<table>
<thead>
<tr>
<th>Task</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. pts.</td>
<td>18</td>
<td>14</td>
<td>19</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1st Corr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2nd Corr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See next page for detailed instructions.
Instructions:

- Fill in this cover sheet first.
- Always keep your Legi visible on the table.
- Keep your phones, tablets and computers turned off in your bag.
- Start each handwritten problem on a new sheet.
- Put your name on each sheet.
- Do not write with red/green/pencil.
- Write your solutions clearly and work carefully.
- **Write all your solutions only in the folder questions!**
- Any other location will not be backed-up and will be discarded.
- Files in resources may be overridden at any time.
- Make sure to regularly save your solutions.
- Time spent on restroom breaks is considered examination time.
- **Never turn off or log off from your computer!**

Instructions for coding problems:

- In the folder "~/questions" you will find the template files for the solution of the problems. You can use these templates to write your solution.
- We provide a "CMake" file that automatically compiles all the templates. To generate a "Makefile" for all problems, type "cmake ." in the folder "~/questions". Compile your programs with "make".
- In order to compile and run the C++ code related to a single problem, like Problem 0.3, type "make problem3". Execute the program using "./problem3".
- If you want to manually compile your code without CMake, use:

  ```
g++ -I. -std=c++11 -Wno-deprecated-declarations \  -Wno-ignored-attributes filename.cpp -Wno-misleading-indentation \  -Wno-unused-variable -o program_name
```

  or

  ```
clang++ -I. -std=c++11 -Wno-deprecated-declarations \  -Wno-ignored-attributes filename.cpp -Wno-misleading-indentation \  -Wno-unused-variable -o program_name
```

  We use the flags -Wno-deprecated-declarations, -Wno-ignored-attributes, -Wno-misleading-indentation and -Wno-unused-variable to suppress some unwanted EIGEN warnings.
- For each problem requiring C++ implementation, a template file named problemX.cpp is provided (where X is the problem number). For your own convenience, there is a marker TODO in the places where you are supposed to write your own code. All templates should compile even if left unchanged.
Problem 0.1: Estimating point locations from distances (18 pts)

We consider a linear least squares problem from Chapter 3.

[ This problem involves implementation in C++ ]

Consider \( n > 2 \) points located on the real axis, the leftmost point situated at \( x_1 := 0 \), the other points at unknown locations \( x_i \in \mathbb{R}, i = 2, \ldots, n \) with \( x_i < x_{i+1}, i = 1, \ldots, n-1 \). We measure the \( m := \binom{n}{2} = \frac{n(n-1)}{2} \) distances \( d_{ij} := |x_i - x_j|, i, j \in 1, \ldots, n, i > j \). The distances are arranged in a vector according to

\[
d := [d_{2,1}, d_{3,1}, \ldots, d_{n,1}, d_{3,2}, d_{4,2}, \ldots, d_{n,n-1}]^T \in \mathbb{R}^m. \tag{0.0.1}
\]

In absence of measurement errors, the point positions \( x_i \) and the distances satisfy an overdetermined linear system of equations

\[
Ax = d, \quad x = [x_2, \ldots, x_n]^T \in \mathbb{R}^{n-1}. \tag{0.0.2}
\]

(0.1.a) (2 pts) Show that the coefficient matrix/system matrix \( A \in \mathbb{R}^{m,n-1} \) from (0.0.2) has full rank.

SOLUTION of (0.1.a):

As in Eq. (3.0.11), we find that

\[
x_i - x_j = d_{ij}, \quad 1 \leq j < i \leq n. \quad \leftrightarrow \quad 
\begin{bmatrix}
-1 & 1 & 0 & \ldots & 0 \\
-1 & 0 & 1 & 0 & 0 \\
\vdots & \ddots & \ddots \\
-1 & \ldots & 0 & 1 & 0 \\
0 & -1 & 1 & 0 & \ldots \\
\vdots & & & \ddots & \ddots \\
0 & \ldots & 0 & -1 & 0 \\
0 & \ldots & 0 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n
\end{bmatrix}
= 
\begin{bmatrix}
d_{2,1} \\
d_{3,1} \\
\vdots \\
d_{n,1} \\
d_{3,2} \\
\vdots \\
d_{n,n-1}
\end{bmatrix}. \tag{0.0.3}
\]

[1 pts. for the correct system of eq. (at least the identity part)] Setting \( x_1 := 0 \) amounts to dropping the first column of the system matrix. The remaining matrix is the matrix \( A \) from (0.0.2), which is of the form

\[
A = \begin{bmatrix} I_{n-1} \end{bmatrix} \in \mathbb{R}^{m,n-1}.
\]

[1 pts. for argument] Since the top \( (n-1) \times (n-1) \) block is the identity matrix, \( A \) must have full rank.
Provide an implementation of a function

\[
\text{SparseMatrix<double> buildDistanceLSQMatrix(int n);} 
\]

that initializes the system matrix \(A\) from (0.0.2). The function must be **efficient** for large \(n\).

**HINT 1** for (0.1.b): A template for the function buildDistanceLSQMatrix is provided within the file problem1.cpp. You can compile the file with make problem1. The executable ./problem1 tests the routine buildDistanceLSQMatrix by printing the resulting matrix.

**SOLUTION of (0.1.b):**

The matrix \(A\) is sparse with \(2m - (n - 1) = (n - 1)^2 < \frac{n(n-1)^2}{2}\) non-zero entries. The signature of the function buildDistanceLSQMatrix already imposes the usage of sparse matrix data formats.

There are two alternative methods that guarantee an efficient implementation, see \(\rightarrow\) Section 2.7.3.

- **Matrix assembly via intermediate triplet format:**

  1. **[3 pts. for correct construction of triplet vectors]** A vector of triplets is preallocated.
     
     This is possible, because we know that \(A\) has a total of \(2m - (n - 1) = (n - 1)^2\) non-zero entries. The vector is then filled with triplets.

  2. **[1 pts. for correct construction of triplet vectors]** Initialization via an intermediate triplet (COO) format and EIGEN's method setFromTriplets().

- **[3 pts. for correct matrix construction]** Direct entry specification via \(\text{SparseMatrix<T>::insert}\) (also \(\text{SparseMatrix<T>::coeffRef}\) is accepted). **[1 pts. for meaningful preallocation]**

  To avoid unnecessary memory reallocations, \(\text{SparseMatrix<T>::reserve}\) must be called with an appropriate estimate.

**C++11-code 0.0.4: Solution of Sub-problem (0.1.b).**

```cpp
SparseMatrix<double> buildDistanceLSQMatrix(int n) {
    SparseMatrix<double> A(n*(n-1)/2, n-1);

    // Assembly
    std::vector<Triplet<double>> triplets; // List of non-zeros coefficients
    triplets.reserve(((n-1)*(n-1))); // Two non-zeros per row (at most), first n-1 rows only one entry
    // -> (n-1)^2 total non-zero entries

    // Loops over vertical blocks
    int row = 0; // Current row counter
    for(int i = 0; i < n-1; ++i) { // Block with same "-1" column
        for(int j = i; j < n-1; ++j) { // Loop over block
            triplets.push_back(Triplet<double>(row, j, 1));
            if(i > 0) { // Remove first column
                triplets.push_back(Triplet<double>(row, i-1, -1));
            }
        }
        row++; // Next row
    }
}
```
Give explicit formulas for the entries of the system matrix (coefficient matrix) $M$ of the normal equations corresponding to the overdetermined linear system (0.0.2).

**SOLUTION of (0.1.c):**

**[1 pts. for off-diagonal entries]** The entries of matrix $M = A^\top A$ can be expressed as inner products of two different columns of $A$:

$$
(A^\top A)_{i,j} = (A)_{:,i}^\top (A)_{:,j}.
$$

Two columns of $A$ have both non-zero entries, $\pm 1$ of opposite sign, only in a single position, hence $(M)_{i,j} = -1$ for $i \neq j$. **[1 pts. for diagonal entries]** The diagonal entries of $M$ are the squares of the Euclidean norms of the columns of $A$. Every column of $A$ has exactly $n - 1$ entries with value $\pm 1$, which means $(M)_{i,i} = n - 1$.

**(0.1.d) (3 pts) [depends on (0.1.c)]**

Show that the system matrix $M$ of the normal equations for the overdetermined linear system from (0.0.2); as found in Sub-problem (0.1.c), can be written as a rank-1 perturbation of a diagonal matrix.

**SOLUTION of (0.1.d):**

As

$$
(M)_{i,j} = \begin{cases} 
-1, & \text{if } i \neq j, \\
1, & \text{if } i = j
\end{cases}, \quad 1 \leq i,j \leq n - 1,
$$

(0.0.5)

we have **[1 pts. for correct matrix] [1 pts. for correct vector(s) and for argument that modification has rank 1]** that

$$
M = nI_{n-1} - 1 \cdot 1^\top, \quad 1 = [1, \ldots, 1]^\top \in \mathbb{R}^{n-1}.
$$

(0.0.6)

**[1 pts. for correct form of rank-1 perturbation]** The tensor product matrix $1 \cdot 1^\top$ has rank 1.
(0.1.e) (6 pts) [depends on (0.1.d)]

Implement an efficient C++ function

```cpp
VectorXd estimatePointsPositions(const MatrixXd & D);
```

that computes a least squares estimate for \(x_2, \ldots, x_n\) by solving the normal equations for (0.0.2) and returns the column vector \(x := [x_2, \ldots, x_n]^\top\).

The distances \(d_{ij}\) are passed as entries of an \(n \times n\)-matrix \(D\) according to

\[
(D)_{ij} = \begin{cases} 
  d_{ij} , & \text{if } i > j, \\
  0 , & \text{if } i = j, \\
  -d_{ji} , & \text{if } i < j.
\end{cases}
\]

Use the observation made in Sub-problem (0.1.d).

**HINT 1 for (0.1.e):** A template for the function \texttt{estimatePointsPositions} is provided in the file \texttt{problem1.cpp}. You can compile the file with \texttt{make problem1}. The generated executable \texttt{./problem1} tests the routine \texttt{estimatePointsPositions}. The program prints a test matrix \(D\). Then, the program prints the vector \(x\) obtained using the function \texttt{estimatePointsPositions} on the measured distances given by \(D\).

**Example output:**

The matrix \(D\) is:

```
  0 -2.1 -3 -4.2 -5
2.1  0  0.9 -2.2 -3.3
  3  0.9  0 -1.3 -1.1
4.2  2.2  1.3  0 -1.1
  5  3.3  1.1  1.1  0
```

The positions \([x_2, \ldots, x_n]\) obtained from the LSQ system are:

2.316
4.18
4.96

**SOLUTION of (0.1.e):**

We rely on the techniques introduced in \(\rightarrow\S 2.6.13\) and apply the [1 pts. for stating the correct SMW formula and realize it can be used] Sherman-Morrison-Woodbury formula from \(\rightarrow\text{Lemma 2.6.22}\) to the normal equations

\[
\left( nI_{n-1} - 1 \cdot 1^\top \right) x = A^\top d.
\]

Then \(\rightarrow\text{Eq. (2.6.23)}\) yields

\[
x = \frac{1}{n} b + \frac{1}{n} \cdot 1^\top b = \frac{1}{n} \left( b + 1 \cdot 1^\top b \right), \quad b := A^\top d. \tag{0.0.7}
\]

Note that the entries of the vector \(b \in \mathbb{R}^{n-1}\) can be computed by summing the entries of the last \(n - 1\) rows of \(D\) (the intermediate points of the distances cancel each other out) [2 pts. correct r.h.s., also valid to use matrix-vector multiplication]. [3 pts. for correct application of SMW, including matrix inversion with \(\frac{1}{n}\)].
What is the asymptotic complexity of the function `estimatePointsPositions` implemented in Sub-problem (0.1.e) for \( n \to \infty \)?

**Solution of (0.1.f):**

An implementation of (0.0.7) involves SAXPY operations and inner products for vectors of length \( n - 1 \), all of which can be carried out with asymptotic complexity \( O(n) \).

[1 pts. for noticing that complexity is dominated by r.h.s. and specify it correctly] However, forming the vector \( b \) has to access all distances and involves computational cost \( O(n^2) \), which dominates the total asymptotic complexity.

### Problem 0.2: Zero finding in two dimensions (14 pts)

This problem studies Newton’s method for a \( 2 \times 2 \) non-linear system of equations.

[ This problem involves implementation in C++]

Let \( f \) be a strictly increasing, positive, continuously differentiable function \( f \in C^1(\mathbb{R}), f(t) > 0 \).

We seek two real numbers \( a, b \in \mathbb{R} \) such that

\[
\int_a^b f(t) \, dt = a + b , \quad (0.0.9a)
\]
\[
\int_a^b e^{f(t)} \, dt = 1 + a^2 + b^2 . \quad (0.0.9b)
\]
(0.2.a) (2 pts) Eq. (0.0.9) is a nonlinear system of equations which can be rewritten as

\[ F(x) = 0 \]

Give an explicit formula for \( F(x) \) still involving the generic function \( f : \mathbb{R} \to \mathbb{R} \). What are the components of \( x \)?

Solution of (0.2.a):

[0.5 pts. for formula for \( x \)]

[1.5 pts. for formula for \( F \)]

We have \( x = [a, b]^T \) and

\[
F : \begin{cases} 
\mathbb{R}^2 & \to \mathbb{R}^2 \\
[ x_1 ] \\
[ x_2 ] & \to \begin{bmatrix} 
\int_{x_1}^{x_2} f(t) \, dt - x_1 - x_2 \\
\int_{x_1}^{x_2} e^{f(t)} \, dt - 1 - x_1^2 - x_2^2 
\end{bmatrix}
\end{cases}
\]

(0.2.b) (4 pts) [depends on Sub-problem (0.2.a)]

State the Newton’s iteration for solving Eq. (0.0.9) as explicitly as possible.

Hint 1 for (0.2.b): The explicit formula for the inverse of a \( 2 \times 2 \) matrix is

\[
A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \implies A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}, \quad \text{if} \ ad - bc \neq 0.
\]

Solution of (0.2.b):

[2 pts. for Jacobian of \( F \)]

[2 pts. for inverse of Jacobian]]

Using the fundamental theorem of calculus \( \frac{d}{dx} \int_{a}^{x} g(t) \, dt = g(x) \), we find for the Jacobian of \( F \):

\[
D \, F(x) = \begin{bmatrix} 
-\frac{f(x_1) - 1}{-e^{f(x_1)} - 2x_1} & f(x_2) - 1 \\
-\frac{e^{f(x_2)} - 2x_2}{-e^{f(x_1)} - 2x_1} & e^{f(x_2)} - 2x_2 
\end{bmatrix}.
\]

Using this and the formula for the inverse of a regular \( 2 \times 2 \)-matrix, we can write the Newton iteration
as
\[ x^{(k+1)} = x^{(k)} - \frac{1}{(-f(x_1) - 1)(e^{f(x_2)} - 2x_2) - (f(x_2) - 1)(-e^{f(x_1)} - 2x_1)} \begin{bmatrix} e^{f(x_2)} - 2x_2 & -f(x_2) + 1 \\ e^{f(x_1)} + 2x_1 & -f(x_1) - 1 \end{bmatrix} \cdot \begin{bmatrix} \int_{x_1}^{x_2} f(t) \, dt \\ \int_{x_1}^{x_2} e^{f(t)} \, dt \end{bmatrix} - x_1 - x_2 \] 
\begin{bmatrix} \int_{x_1}^{x_2} f(t) \, dt \\ \int_{x_1}^{x_2} e^{f(t)} \, dt \end{bmatrix} - 1 - x_1^2 - x_2^2.

\[ \text{(0.2.c) (8 pts) \ [depends on Sub-problem (0.2.b)] } \]

Implement a C++ function
\[
\text{template <class Function, class QuadRule>}
\text{std::pair<double, double> getIntv(const Function& } f, \text{ const QuadRule& } qr, \text{ const QuadRule& } qr, \text{ double atol, double rtol, unsigned maxit = 10);}
\]

that solves Eq. (0.0.9) by means of Newton’s method with initial guess \( a^{(0)} = 0, b^{(0)} = 1 \).

The argument \( qr \) provides a quadrature rule on \([0,1]\) in terms of weights and nodes. Use it for the evaluation of all occurring definite integrals.

Use a correction-based termination criterion controlled by relative tolerance \( rtol \) and absolute tolerance \( atol \). The variable \( maxit \) specifies the maximum number of iterations.

**Hint 1** for (0.2.c): Recall the definition of the \textbf{QuadRule} class
\[
\text{struct QuadRule} 
\{ 
\text{VectorXd nodes; }
\text{VectorXd weights; }
\};
\]

For numerical quadrature based on the quadrature rule QuadRule, you may implement an auxiliary function
\[
\text{template <class Function, class QuadRule>}
\text{double integrate(const Function& } f, \text{ const QuadRule& } qr, \text{ const Vector2d & } x);\]

which takes the integration bounds as argument vector \( x \).

**Hint 2** for (0.2.c): A template for the functions \texttt{getIntv} and \texttt{integrate} is provided within the file \texttt{problem2.cpp}. You can compile the file with make \texttt{problem2}. The executable \texttt{./problem2} tests the routine \texttt{getIntv} by printing the approximate \((a,b)\) (for a given function \( f(t) := t \)) and the reference solution.

\[
\text{Solution of (0.2.c):}
\]
[2 pts. for correct function integrate or an equivalent correct integration]
[2 pts. for correct transformation of QuadRule on $[0,1]$]
[4 pts. for correct function getIntv]

For the correction-based a posteriori termination criterion, look at ➔ Section 8.4.1.

```cpp
C++11-code 0.0.10: Function integrate.

```template<class Function, class QuadRule>

double integrate(const Function& f, const QuadRule& qr, const Vector2d& x) {
    double I = 0;
    VectorXd nodes = qr.nodes;
    VectorXd weights = qr.weights;
    assert(nodes.size() == weights.size() &&
           "Nodes and weights of QuadRule have different lengths");
    for(unsigned i = 0; i < nodes.size(); ++i) {
        double t = (x(1) + x(0)) / 2. + (x(1) - x(0)) * (nodes(i) - 0.5);
        // Adjust nodes of $[0,1]$-QuadRule to domain $[x_1,x_2]$.
        I += f(t) * weights(i);
    }
    I *= x(1) - x(0); // Adjust weights of $[0,1]$-QuadRule to domain $[x_1,x_2]$.

    return I;
}
```

```cpp
C++11-code 0.0.11: Function getIntv.

```template<class Function, class QuadRule>

std::pair<double, double> getIntv(const Function& f, const QuadRule& qr,
                                    double atol, double rtol,
                                    unsigned maxit=10) {
    std::pair<double, double> x_end;
    Vector2d x;
    x << 0, 1;
    Vector2d x_new = x;
    auto exp_f = [&] (double t) { return std::exp(f(t));};
    for(unsigned i = 0; i < maxit; ++i) {
        // Compute inverse of Jacobian.
        Matrix2d invDF;
```
\begin{verbatim}
invDF << exp_f(x(1)) - 2*x(1), -f(x(1)) + 1,
      exp_f(x(0)) + 2*x(0), -f(x(0)) - 1;

invDF /= (-f(x(0)) - 1)*(exp_f(x(1)) - 2*x(1)) -
           (f(x(1)) - 1)*(-exp_f(x(0)) - 2*x(0));

// Evaluate F(x(k)).
Vector2d F;
F << integrate(f, qr, x) - (x(0)+x(1)),
    integrate(exp_f, qr, x) - (1+x(0)*x(0)+x(1)*x(1));

// Newton's iteration.
x_new = x - invDF*F;

// Correction-based termination (relative and absolute).
double r = (x_new - x).norm();
if (r < atol || r < rtol * x_new.norm()) {
    break;
}

x = x_new;

x_end = {x_new(0), x_new(1)};
return x_end;
\end{verbatim}

---

**Problem 0.3: Low rank approximation (19 pts)**

This problem discusses a compressed model for a filter.

This problem involves implementation in C++

---

A causal, linear, time-invariant and finite (LT-FIR) channel has the impulse response

\[ (0, \ldots, 0, h_0, \ldots, h_{n-1}, 0, \ldots, 0) \]  \hspace{1cm} (0.0.12)

of duration \((n - 1)\Delta t\). When we feed into it a signal \(x := (0, \ldots, 0, x_0, \ldots, x_{n-1}, 0, \ldots, 0)\) of duration 
\((n - 1)\Delta t\), the filter produces an output signal \(y := (0, \ldots, 0, y_0, \ldots, y_{2n-2}, 0, \ldots, 0)\) of duration 
\((2n - 2)\Delta t\). The linear mapping

\[
I : \begin{cases}
\mathbb{R}^n \rightarrow \mathbb{R}^{2n-1} \\
(x_j)_{j=0}^{n-1} \rightarrow (y_j)_{j=0}^{2n-2}
\end{cases}
\]

can be represented by the matrix-vector product

\[ (y_j)_{j=0}^{2n-2} = C (x_j)_{j=0}^{n-1}, \]  \hspace{1cm} (0.0.13)
which can be expressed as the following matrix \times vector multiplication, see \ref{Rem. 4.1.17}:

\[
\begin{bmatrix}
y_0 \\
y_1 \\
\vdots \\
y_{2n-2}
\end{bmatrix}
= 
\begin{bmatrix}
h_0 & 0 & 0 & \cdots & 0 \\
h_1 & h_0 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & h_{n-2} & \cdots & h_1 & h_0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & \cdots & 0 & h_{n-1}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
\vdots \\
x_{n-1}
\end{bmatrix}.
\]

(0.3.a) (2 pts)

Using \texttt{EIGEN}, implement a C++ function with signature

\[
\text{MatrixXd buildLTFIRMatrix(const VectorXd \&h)};
\]

that initializes the matrix \( C \) from (0.0.13). The vector \( h \) specifies the entries of \( C \).

Hints for (0.3.a): You will find a template for the function buildLTFIRMatrix within the file problem3.cpp. You can compile the file with make problem3. The executable ./problem3 tests the routine buildLTFIRMatrix by printing the resulting matrix. The correct matrix (for \( n = 6 \)) is reported as a comment in the code (within main of problem3.cpp).

Solution of (0.3.a):

[2 pts. Many ways to do this]

\texttt{C++11-code 0.0.14: Function buildDistanceLSQMatrix.}

\[
\text{MatrixXd buildLTFIRMatrix(const VectorXd \& h)}
\{

// Initialization

unsigned int n = h.size();
MatrixXd C(2*n-1, n);
C.setZero();

for(unsigned i=0; i<n;++i) {
    C.col(i).segment(i,n) = h;
}

return C;
\}
\]

Now the goal is to implement a compressed model for the channel. Consider the class...
class LTFIR_lowrank {
public:
    LTFIR_lowrank(const VectorXd & h, unsigned k);
    VectorXd operator()(const VectorXd & x) const;
private:
    // TODO: private members of class LTFIR_lowrank
};

whose evaluation operator realizes \( y = \tilde{C}x \), where \( \tilde{C} \in \mathbb{R}^{2n-1,n} \) is the rank-\( k \) best approximation of \( C \), and \( k \in \{1, \ldots, n\} \) is passed as the second argument of the constructor.

**Problem 3.3(b) (9 pts) [depends on Sub-problem 3.3(a)]**

Implement both member functions of the class `LTFIR_lowrank` such that a call of the evaluation operator involves as little computational effort as possible (asymptotically, for \( n \to \infty \)).

**Hint 1 for 3.3(b):** You may use the function `buildLTFIRMMatrix` from Sub-problem 3.3(a).

**Hint 2 for 3.3(b):** A template for the class `LTFIR_lowrank` is provided within the file `problem3.cpp`. You can compile the file with `make problem3`. The executable `./problem3` tests the routine `operator()` by printing the resulting vector \( y = \tilde{C}x \) for specific inputs \( h \), \( c \) and \( k \). The correct result is reported as a comment in the code.

**Solution of 3.3(b):**

[6 pts. for efficient constructor and private members. Economical SVD should be used.]

[3 pts. for efficient `operator()`]

```cpp
LTFIR_lowrank(const VectorXd & h, unsigned int k) {
    MatrixXd C = buildLTFIRMMatrix(h);
    JacobiSVD<MatrixXd> svd(C, ComputeThinU | ComputeThinV);
    // With Eigen::svd you can ask for thin U or V to be computed.
    // In case of a rectangular \( m \times n \) matrix,
    // with \( j \) the smaller value among \( m \) and \( n \),
    // there can only be at most \( j \) singular values.
    // The remaining columns of U and V do not correspond
    // to actual singular vectors and are not computed in thin format.
    VectorXd s = svd.singularValues();
    s.conservativeResize(k);
    auto S = s.asDiagonal(); // kxk
    MatrixXd U = svd.matrixU();
    MatrixXd V = svd.matrixV();
    U_ = U.leftCols(k) * S; // nxk
    // Already optimised product between dense and diagonal matrix
    Vt_ = V.leftCols(k).transpose(); // kxn
}
```
What is the asymptotic complexity of your implementation of the constructor and the evaluation operator for $n \to \infty$ and $k \to \infty$ (separately, assuming $k \leq n$)?

**Solution of (0.3.c):**

[1 pts. for complexity of SVD]

[1 pts. for complexity of operator()]

The rank-$k$ approximation performed by the constructor involves a singular value decomposition. The complexity of an SVD is $O(n^3)$.

The evaluation operator carries out two matrix-vector multiplications, by $k \times n$ matrix $V^T$ and $(2n - 1) \times k$ matrix $U$. The complexity is therefore $O(kn + nk) = O(nk)$. On the other hand, given the full $(2n - 1) \times n$ approximation matrix $\tilde{C}$, the complexity would have been $O(n^2)$.

(0.3.d) (3 pts)

Decide which of the following properties does the new filter (realized by the evaluation operator of LTFIR_lowrank) still enjoy for any $(h_j)_{j=0}^{n-1}$: linearity, causality, and finiteness.

**Solution of (0.3.d):**
Linearity [1 pts.] Yes: it is ultimately a matrix-vector multiplication.

Causality [1 pts.] Yes: \( y_j = 0 \ \forall j < 0 \).

Finiteness [1 pts.] Yes: the number of nonzero \( y_j \) is up to \( 2n - 1 \).

(0.3.e) (3 pts)

Another way to build a compressed model of the channel is frequency filtering, which is implemented in the following `LTFIR_freq` class.

### C++11-code 0.0.18: Constructor of class `LTFIR_freq`.

```cpp
LTFIR_freq(const VectorXd& h, unsigned k) {
    n_ = h.size();
    k_ = k;

    VectorXd h_ = h;
    h_.conservativeResizeLike(VectorXd::Zero(2*n_-1));

    // Forward DFT
    FFT<double> fft;
    ch_ = fft.fwd(h_);
}
```

### C++11-code 0.0.19: Function `operator()`.

```cpp
VectorXd operator()(const VectorXd& x) const {
    assert(x.size() == n_ && "x must have same length of h");

    VectorXd x_ = x;
    x_.conservativeResizeLike(VectorXd::Zero(2*n_-1));

    // Forward DFT
    FFT<double> fft;
    VectorXcd cx = fft.fwd(x_);
    VectorXcd c = ch_.cwiseProduct(cx);
    // Set high frequency coefficients to zero
    VectorXcd clow = c;
    for(int j=-k_; j<=+k_; ++j) clow(n_+j) = 0;
    // Inverse DFT
    return fft.inv(clow).real();
}
```

### C++11-code 0.0.20: Private members of class `LTFIR_freq`.

```cpp
int n_;
int k_;
VectorXd ch_;`
What is the asymptotic complexity of the evaluation operator \texttt{operator()} for \( n \to \infty \)?

You can find the implementation of the class \texttt{LTFIR_freq} in the file \texttt{problem3.cpp}.

---

**Solution of (0.3.e):**

**[3 pts.]** The most expensive steps of the implemented low-pass filter are the Fourier transforms. For \( n \)-dimensional input vectors \( \mathbf{x} \), the complexity of a fast Fourier transform is \( \mathcal{O}(n \log n) \).

---

**End Problem 0.3**

**Problem 0.4: Single step method (23 pts)**

This problem concerns numerical integration \( \Rightarrow \) Chapter 11 with single step methods.

This problem involves implementation in C++

We consider the initial value problem for \( \mathbf{y}(t) := [y_1(t), y_2(t)]^\top \):

\[
\dot{\mathbf{y}} = \begin{bmatrix} -\theta(y_2) \\ y_1 \end{bmatrix}, \quad \theta \in C^1(\mathbb{R}), \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ y_0 \end{bmatrix}.
\]  

\hfill (0.0.21)

**Solution of (0.4.a):**

\[2 \text{ pts.}\] Consider \( I(\mathbf{y}(t)) = \text{const.} \). We have that \( I(\mathbf{y}) = \text{const} \iff \frac{d}{dt}I(\mathbf{y}(t)) = 0 \). By the scalar chain rule and the product rule we can conclude:

\[
I'(\mathbf{y}) = y_1 \dot{y}_1 + \ddot{\xi}(y_2) \dot{y}_2 = y_2 \dot{y}_1 + \theta(y_2) \dot{y}_2 = \dot{y}_2 y_1 - y_1 \dot{y}_2 = 0.
\]

---

**Solution of (0.4.b):**

\[4 \text{ pts.}\] Give the concrete defining equation for the discrete evolution \( \Psi \) of the implicit midpoint rule \( \Rightarrow \text{Eq. (11.2.18)} \) for (0.0.21).
The discrete evolution operator $\Psi : \mathbb{R} \times \mathbb{R}^2 \to \mathbb{R}^2$ according to § 11.3.1 is defined as the solution operator of the following non-linear system of equations: for a generic autonomous ODE $\dot{y} = f(y)$ it reads

$$\Psi(h, y) := z = y + hf\left(\frac{1}{2}(y + z)\right)$$

and, concretely, for (0.0.21),

$$\Psi(h, y) := z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \text{ such that } z = y + h\begin{bmatrix} -\theta\left(\frac{1}{2}(y_2 + z_2)\right) \\ \frac{1}{2}(y_1 + z_1) \end{bmatrix}, \quad y \in \mathbb{R}^2. \tag{0.0.22}$$

For sufficiently small $h$ there is a unique solution $z = z(h, y)$.

\[\textbf{(0.4.c)} \quad \text{(5 pts) [depends on Sub-problem (0.4.b)]}\]

State the explicit formulas for the Newton’s iteration that can be used to approximately evaluate the discrete evolution of the implicit midpoint rule for (0.0.21). Specify a meaningful initial value in the case of small time steps.

**HINT 1** for (0.4.c): The explicit formula for the inverse of a $2 \times 2$ matrix is

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \implies A^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}, \quad \text{if } ad - bc \neq 0.$$  

**SOLUTION of (0.4.c):**

[1 pts. for functional $F$. Note that there are several ways to recast the implicit midpoint rule as a non-linear equations, among them also the stage form discussed in class.]

[1 pts. for Jacobian of $F$]

[1 pts. for inverse of Jacobian]

[1 pts. for complete Newton’s iteration]

[1 pts. for initial guess]

The non-linear $2 \times 2$ system of equations (0.0.22) can be recast into the standard form

$$F(z) = 0, \quad F(z) := z - y - h\begin{bmatrix} -\theta\left(\frac{1}{2}(y_2 + z_2)\right) \\ \frac{1}{2}(y_1 + z_1) \end{bmatrix}. \tag{0.0.23}$$

For the Jacobian of $F$ we find

$$DF(z) = \begin{bmatrix} 1 & \frac{1}{2}h\theta'\left(\frac{1}{2}(y_2 + z_2)\right) \\ -\frac{1}{2}h & 1 \end{bmatrix}. \tag{0.0.24}$$

Using the formula for the inverse of a $2 \times 2$-matrix, the Newton’s iteration reads

$$z^{(k+1)} = z^{(k)} - \frac{1}{1 + \frac{1}{2}h^2\theta'\left(\frac{1}{2}(y_2 + z^{(k)}_2)\right)}\begin{bmatrix} 1 & -\frac{1}{2}h\theta'\left(\frac{1}{2}(y_2 + z^{(k)}_2)\right) \\ \frac{1}{2}h & 1 \end{bmatrix}\begin{bmatrix} z_1^{(k)} - y_1 + h\theta\left(\frac{1}{2}(y_2 + z^{(k)}_2)\right) \\ z_2^{(k)} - y_2 - \frac{1}{2}h(y_1 + z^{(k)}_1) \end{bmatrix}. \tag{0.0.25}$$
For small $h$ we expect $\Psi(h,y) \approx y$. Hence $z^{(0)} := y$ is the natural initial guess for the Newton's iteration.

Since the implicit midpoint rule is a Runge-Kutta single step method, it can also be cast in stage form, see $\Rightarrow$ Rem. 12.3.21. Then one can use Newton's method to solve for the stages $g_i$, see $\Rightarrow$ Rem. 12.3.24.

(0.4.d) (4 pts)  [ depends on Sub-problem (0.4.c) ]

Implement a function

```
template <class Function, class Jacobian>
Vector2d psi(const Function& theta, const Jacobian& theta_d, double h, const Vector2d& y)
```

that approximately realizes the discrete evolution operator of the implicit midpoint rule for (0.0.21) using, internally, two Newton's steps. The parameter $h$ specifies the step size. The variable $\theta$ resp. $\theta'$ represent the function $\theta$ and its derivative $\theta'$. The vector $y$ passes the value $y$ at the previous step.

HINT 1 for (0.4.d): A template for the function $\Psi$ is provided within the file problem4.cpp. You can compile the file with `make problem4`. The executable `./problem4` tests the routine $\Psi$ by comparing the discrete evolution for $\theta(\zeta) = e^{\zeta}$ with a reference solution. The test performs a single evolution step of size $h = 0.1$ starting from the initial data $y(0)$.

SOLUTION of (0.4.d):

[4 pts.]

C++11-code 0.0.26: Solution of (0.4.d).

```
template<typename Functor, typename Jacobian>
Vector2d psi(Functor& theta, Jacobian& theta_d, double h, const Vector2d& y) {
    Vector2d z;
    z = y;

    for(unsigned i=0; i<2; ++i) {
        Matrix2d invDF;
        invDF << 1., -0.5*h*theta_d(0.5*(y(1)+z(1)))/h, 1.;
        invDF /= 1 + 0.25*h*h*theta_d(0.5*(y(1)+z(1)));

        Vector2d F;
        F << z(0) - y(0) + h*theta(0.5*(y(1)+z(1))), z(1) - y(1) - 0.5*h*(y(0)+z(0));
        z = z - invDF*F;
    }
    return z;
}
```
The following function `lfevl` implements an explicit Runge-Kutta single step method for Eq. (0.0.21) and for some (unknown) smooth function $\theta$ (passed as `theta`). The code applies a Runge-Kutta method on $N$ equidistant steps of size $h$, starting from the initial value $y_0 := y(0)$.

```cpp
C++11-code 0.0.27: Function `lfevl`.

```template<typename Function> Vector2d lfevl(const Function& theta, Vector2d y0, double h, unsigned int N) {
  auto f = [&theta] (const Vector2d& y) -> Vector2d {
    Vector2d y_dot;
    y_dot << -theta(y(1)), y(0);
    return y_dot;
  };

  Vector2d yk = y0;
  for(unsigned k=0; k < N; ++k) {
    Vector2d k1 = f(yk);
    Vector2d k2 = f(yk + h/2.*k1);
    Vector2d k3 = f(yk - h*k1 + 2.*h*k2);
    yk += h/6.*k1 + 2.*h/3.*k2 + h/6.*k3;
  }
  return yk;
}
```

Write down the Butcher scheme for this method.

**Solution of (0.4.e):**

[3 pts.] The code closely follows the structure of the increment equations from Def. 11.4.9 and one can simply read off the coefficients.

\[
\begin{pmatrix}
0 & 0 & 0 \\
\frac{1}{3} & 0 & 0 \\
\frac{1}{3} & -1 & 2 \\
\frac{1}{5} & \frac{2}{3} & \frac{1}{5}
\end{pmatrix}
\]

(0.4.f) (5 pts)  [depends on Sub-problem (0.4.e)]
Consider the C++ function `lfevl` of Sub-problem (0.4.e) and let $\theta(\xi) = e^\xi$ and $y(0) = [0,1]^T$. Empirically determine the order of convergence of the single step method implemented by `lfevl` by studying the errors of the numerical solutions at the final time $T = 10$ and their dependence on the number $N$ of equidistant steps of the single-step method.

**HINT 1** for (0.4.f): Use suitable sequences of numbers of steps $N$ ranging between 50 and $2 \cdot 10^4$.

**HINT 2** for (0.4.f): Implement your code in the main function of the file `problem4.cpp`. You can compile the file with `make problem4`. The executable `.problem4` should print the error and the estimated order of convergence of `lfevl`, for every value of $N$.

**SOLUTION** of (0.4.f):

The empirical order of the method is 3. The smartest choice of a sequence of numbers of steps is geometric progression, doubling $N$ in turn, e.g., $N = 2^6, \ldots, 2^{14}$.

[4 pts. for correct implementation]

[1 pts. for correct order of convergence]

**C++11-code 0.0.28: Solution of (0.4.f).**

```cpp
// Vector of number of steps (each entry is twice the previous entry).
std::vector<unsigned> N = {128, 256, 512, 1024, 2048, 4096, 8192, 16384};
std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::setw(15) << "rate" << std::endl;
double err_old;
for(unsigned int i=0; i < N.size(); ++i) {
    double h = T/N[i];
    auto yk = lfevl(theta, y0, h, N[i]);
    double err = (yk - y_exact).norm();
    std::cout << std::setw(15) << N[i] << std::setw(15) << err;
    if (i > 0) {
        std::cout << std::setw(15) << std::log2(err_old / err);
    }
    err_old = err;
    std::cout << std::endl;
}
```

End Problem 0.4
Problem 0.5: Polar decomposition of a matrix (10 pts)

This problem addresses a special matrix factorization and its numerical realization. [This problem involves implementation in C++]

The following result is obtained in linear algebra:

**Theorem 0.0.29. Polar decomposition**

Given $M \in \mathbb{R}^{n,n}$, there is a symmetric positive semidefinite matrix $A \in \mathbb{R}^{n,n}$ and an orthogonal matrix $Q \in \mathbb{R}^{n,n}$ such that

$$M = AQ.$$  

(0.0.30)

The matrix factorization (0.0.30) is called the polar decomposition of $M$.

**(0.5.a) (4 pts) Give a proof of Thm. 0.0.29.**

**HINT 1 for (0.5.a): Use the singular value decomposition of $M$.**

**SOLUTION of (0.5.a):**

[1 pts. for correct usage of SVD]

[1 pts. for proven symmetry of $A$]

[1 pts. for proven positive definiteness of $A$]

[1 pts. for proven orthogonality of $Q$]

Given $M \in \mathbb{R}^{n,n}$, there always exists a singular value decomposition $M = U \Sigma V^T$, where $U, V \in \mathbb{R}^{n,n}$ are orthogonal and $\Sigma \in \mathbb{R}^{n,n}$ is diagonal. Consider the property of orthogonal matrices $U^T U = I$.

Hence:

$$M = U \Sigma U^T V^T = (U \Sigma U^T)(V^T) = AQ,$$

where we defined $A := U \Sigma U^T$ and $Q := V^T$. We just need to prove that $A$ is symmetric positive semidefinite and $Q$ is orthogonal.

For the first part, we have that $A^T = (U \Sigma U^T)^T = U \Sigma U^T = A$, hence $A$ is symmetric. Furthermore, the singular values of $M$ are the eigenvalues of $A$ (diagonal of $\Sigma$) and singular values are always nonnegative. Hence, the eigenvalues of $A$ are nonnegative and $A$ must be positive semidefinite.

For the second part, we have that $Q^T = (UV^T)^T = (V^T)^T U^T = (V^T)^{-1} U^{-1} = (UV^T)^{-1} = Q^{-1}$, which means that $Q$ is also orthogonal.

**(0.5.b) (5 pts) [depends on (0.5.a)]**

Using Eigen's numerical linear algebra facilities, write a C++ function
that computes the polar decomposition (0.0.30) of $M$, returning the tuple $(A, Q)$.

HINT 1 for (0.5.b): You may use EIGEN's methods for numerical singular value decomposition (SVD).

HINT 2 for (0.5.b): A template for the function polar is provided within the file problem5.cpp. You can compile the file with make problem5. The executable ./problem5 tests the routine polar.

In main(), for the specified matrix

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \\ 6 & 3 & 11 \end{bmatrix},$$

the program computes and prints the matrices $A$ and $Q$.

Example output:

Matrix $A$ is:

$\begin{bmatrix} 2.11118 & 0.847555 & 2.97062 \\ 0.847555 & 1.31722 & 3.39803 \\ 2.97062 & 3.39803 & 12.0677 \end{bmatrix}$

Matrix $Q$ is:

$\begin{bmatrix} -0.352666 & 0.910956 & 0.213977 \\ 0.872437 & 0.402776 & -0.276811 \\ 0.338348 & -0.0890599 & 0.936797 \end{bmatrix}$

The function testPolar is also provided. This function uses an implementation of polar and checks whether it returns a true polar decomposition.

---

SOLUTION of (0.5.b):

[5 pts.]

```cpp
std::pair<MatrixXd, MatrixXd> polar(const MatrixXd& M) {
    assert(M.rows() == M.cols() && "M must be square!");
    unsigned n = M.rows();
    JacobiSVD<MatrixXd> svd(M, ComputeThinU | ComputeThinV);
    VectorXd s = svd.singularValues();
    MatrixXd S; S.setZero(s.size(), s.size());
    S.diagonal() = s;
    MatrixXd U = svd.matrixU();
    MatrixXd V = svd.matrixV();
    return std::make_pair(U * S * U.transpose(), U * V.transpose());
}
```
What is the asymptotic complexity of your implementation of \texttt{polar} for \( n \to \infty \)?

\textbf{Solution of (0.5.c):}

[1 pts.] The most expensive step of a polar decomposition is to compute a singular value decomposition. For \((n \times n)\) square matrices, the complexity of an SVD is \(O(n^3)\).