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)

Family name		Grade
First name		
Study program		
Computer name		
Legi no.		
Date	26.01.2016	

Points:

Task	1	2	3	4	5	Total
Max. pts.	18	14	19	23	10	
1st Corr.						
2nd Corr.						

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.

dist

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 $d_{i,j} := |x_i - x_j|,\ i,j \in 1,\ldots,n,\ i > j$. The distances are arranged in a vector according to

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

{\cp

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

$$\mathbf{A}\mathbf{x} = \mathbf{d}\,, \quad \mathbf{x} = [x_2, \ldots, x_n]^\top \in \mathbb{R}^{n-1} \,. \tag{0.0.2}$$

{\cp

sp:1

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

prb:lsq-dist:lsd:lse

SOLUTION of (0.1.a):

As in ➜Eq. (3.0.11), we find that

$$\left.\begin{array}{l} x_i - x_j = d_{ij}\,, \\ 1 \le j < i \le n\,. \end{array}\right. \quad \leftrightarrow \quad \begin{bmatrix} -1 & 1 & 0 & \ldots & & & 0 \\ -1 & 0 & 1 & 0 & & & \\ \vdots & & \ddots & \ddots & & & \\ & & & & & & \\ -1 & \ldots & & & & 0 & 1 \\ 0 & -1 & 1 & 0 & \ldots & & 0 \\ 0 & -1 & 0 & 1 & 0 & & \vdots \\ \vdots & & & & & & \\ & 0 & -1 & 1 & 0 & & \\ & & & & & & \\ 0 & \ldots & & & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_{2,1} \\ d_{3,1} \\ \vdots \\ d_{n,1} \\ d_{3,2} \\ d_{4,2} \\ \vdots \\ d_{4,3} \\ \vdots \\ d_{n,n-1} \end{bmatrix}. \tag{0.0.3}$$

{\cp

[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 $\mathbf{A}$ from (0.0.2), which is of the form

prb:lsq-dist:lsd:lse

$$\mathbf{A} = \begin{bmatrix} \mathbf{I}_{n-1} \\ * \end{bmatrix} \in \mathbb{R}^{m,n-1}\,.$$

[1 pts. for argument] Since the top $(n-1) \times (n-1)$ block is the identity matrix, $\mathbf{A}$ must have full rank.

▲

sp:a

(0.1.b) (4 pts) [depends on (0.1.a)] prb:lsq-dist:sp:1

Provide an implementation of a function

```
SparseMatrix<double> buildDistanceLSQMatrix(int n);
```

that initializes the system matrix $\mathbf{A}$ from (0.0.2). The function must be *efficient* for large n. prb:lsq-dist:lsd:lse

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 $\mathbf{A}$ 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 ➜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 $\mathbf{A}$ 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 `SparseMatrix<T>::insert` (also `SparseMatrix<T>::coeffRef` is accepted). **[1 pts. for meaningful preallocation]** To avoid unnecessary memory reallocations, `SparseMatrix<T>::reserve` must be called with an appropriate estimate.

ivencode

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

```
2  SparseMatrix<double> buildDistanceLSQMatrix(int n) {
3      SparseMatrix<double> A(n*(n-1)/2, n-1);
4
5      // Assembly
6      std::vector<Triplet<double>> triplets; // List of non-zeros
           coefficients
7      triplets.reserve((n-1)*(n-1)); // Two non-zeros per row (at
           most), first n-1 rows only one entry
8      // -> (n-1)^2 total non-zero entries
9
10     // Loops over vertical blocks
11     int row = 0; // Current row counter
12     for(int i = 0; i < n-1; ++i) { // Block with same "-1" column
13         for(int j = i; j < n-1; ++j) { // Loop over block
14             triplets.push_back(Triplet<double>(row, j, 1));
15             if(i > 0) { // Remove first column
16                 triplets.push_back(Triplet<double>(row, i-1, -1));
17             }
18             row++; // Next row
19         }
```

```
20      }
21
22      // Build matrix
23      A.setFromTriplets(triplets.begin(), triplets.end());
24
25      A.makeCompressed();
26      return A;
27  }
```

▲

sp:b **(0.1.c)** (2 pts) [depends on (0.1.a)]

Give explicit formulas for the entries of the system matrix (coefficient matrix) $\mathbf{M}$ 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 $\mathbf{M} = \mathbf{A}^\top \mathbf{A}$ can be expressed as inner products of two different columns of $\mathbf{A}$:

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

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

▲

sp:2 **(0.1.d)** (3 pts) [depends on (0.1.c)]

Show that the system matrix $\mathbf{M}$ 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

$$(\mathbf{M})_{i,j} = \begin{cases} -1 & \text{, if } i \neq j \,, \\ n-1 & \text{, if } i = j \end{cases} \quad , \quad 1 \leq i,j \leq n-1 \,, \tag{0.0.5}$$

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

$$\mathbf{M} = n\mathbf{I}_{n-1} - \mathbf{1} \cdot \mathbf{1}^\top \,, \quad \mathbf{1} = [1, \ldots, 1]^\top \in \mathbb{R}^{n-1} \,. \tag{0.0.6}$$

[1 pts. for correct form of rank-1 perturbation] The tensor product matrix $\mathbf{1} \cdot \mathbf{1}^\top$ has rank 1.

▲

impl **(0.1.e)** (6 pts) [depends on (0.1.d)] prb:lsq-dist:sp:2

Implement an efficient C++ function

```
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 $\mathbf{x} := [x_2, \ldots, x_n]^\top$. prb:lsq-dist:lsq

The distances $d_{i,j}$ are passed as entries of an $n \times n$-matrix $\mathbf{D}$ according to

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

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

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

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, ..., x_n] obtained from the LSQ system are:
2
3.16
4.18
4.96
```

SOLUTION of (0.1.e):

We rely on the techniques introduced in ➜§ 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 ➜Lemma 2.6.22 to the normal equations

$$\left(n\mathbf{I}_{n-1} - \mathbf{1}\cdot\mathbf{1}^\top\right)\mathbf{x} = \mathbf{A}^\top\mathbf{d} \,.$$

Then ➜Eq. (2.6.23) yields

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

{\cp

Note that the entries of the vector $\mathbf{b} \in \mathbb{R}^{n-1}$ can be computed by summing the entries of the last $n-1$ rows of $\mathbf{D}$ (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 $\frac{1}{n}$]**

C++11-code 0.0.8: Solution of Sub-problem (0.1.e).

```
VectorXd estimatePointsPositions(const MatrixXd& D) {

    VectorXd x;

    // Vector of sum of columns of A
    ArrayXd b = D.rowwise().sum().tail(D.cols()-1);
    // Vector 1
    ArrayXd one = ArrayXd::Constant(D.cols()-1, 1);
    // Apply SMW formula
    x =  (b + one * b.sum()) / D.cols();

    return x;
}
```

▲

(0.1.f) (1 pts) [depends on (0.1.e)]

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 $\mathcal{O}(n)$.

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

▲

End Problem 0.1

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

This problem studies Newton's method for a 2×2 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

$$\int_a^b f(t)\,\mathrm{d}t = a + b\,, \tag{0.0.9a}$$

$$\int_a^b \mathrm{e}^{f(t)}\,\mathrm{d}t = 1 + a^2 + b^2\,. \tag{0.0.9b}$$

sp:1 **(0.2.a)** (2 pts) Eq. (0.0.9) is a nonlinear system of equations which can be rewritten as

$$F(\mathbf{x}) = \mathbf{0}$$

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

SOLUTION of (0.2.a):

[0.5 pts. for formula for $\mathbf{x}$]]

[1.5 pts. for formula for F]]

We have $\mathbf{x} = [a, b]^\top$ and

$$F : \begin{cases} \mathbb{R}^2 & \to \mathbb{R}^2 \\ \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} & \to \begin{bmatrix} \left(\int_{x_1}^{x_2} f(t)\,\mathrm{d}t \right) - x_1 - x_2 \\ \left(\int_{x_1}^{x_2} \mathrm{e}^{f(t)}\,\mathrm{d}t \right) - 1 - x_1^2 - x_2^2 \end{bmatrix} \end{cases}$$

▲

sp:2 **(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 2×2 matrix is

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

⌟

SOLUTION of (0.2.b):

[2 pts. for Jacobian of F]]

[2 pts. for inverse of Jacobian]]

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

$$\mathrm{D}F(\mathbf{x}) = \begin{bmatrix} -f(x_1) - 1 & f(x_2) - 1 \\ -e^{f(x_1)} - 2x_1 & e^{f(x_2)} - 2x_2 \end{bmatrix} .$$

Using this and the formula for the inverse of a regular 2×2-matrix, we can write the Newton iteration

as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \frac{1}{(-f(x_1)-1)\left(\mathrm{e}^{f(x_2)}-2x_2\right)-(f(x_2)-1)\left(-\mathrm{e}^{f(x_1)}-2x_1\right)} \begin{bmatrix} \mathrm{e}^{f(x_2)}-2x_2 & -f(x_2)+1 \\ \mathrm{e}^{f(x_1)}+2x_1 & -f(x_1)-1 \end{bmatrix} \cdot \begin{bmatrix} \left(\int_{x_1}^{x_2} f(t)\,\mathrm{d}t\right) - x_1 - x_2 \\ \left(\int_{x_1}^{x_2} \mathrm{e}^{f(t)}\,\mathrm{d}t\right) - 1 - x_1^2 - x_2^2 \end{bmatrix}.$$

▲

sp:3 **(0.2.c)** (8 pts) [depends on Sub-problem (0.2.b)]

Implement a C++ function

```
template<class Function, class QuadRule>
std::pair<double, double> getIntv(const Function& f,
                                  const QuadRule& qr,
                                  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 **QuadRule** class

```
struct QuadRule {
    VectorXd nodes;
    VectorXd weights;
};
```

For numerical quadrature based on the quadrature rule `QuadRule`, you may implement an auxiliary function

```
template<class Function, class QuadRule>
double integrate(const Function& f, const QuadRule& qr,
                 const Vector2d & x);
```

which takes the integration bounds as argument vector `x`. ⌟

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

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.

ector-31

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;
}
```

ector-32

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;
```

```
        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 Problem 0.2

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

$$(0,\ldots,0,h_0,\ldots,h_{n-1},0,\ldots,0) \tag{0.0.12}$$

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

$$l : \begin{cases} \mathbb{R}^n & \to \mathbb{R}^{2n-1} \\ (x_j)_{j=0}^{n-1} & \to (y_j)_{j=0}^{2n-2} \end{cases}$$

can be represented by the matrix-vector product

$$(y_j)_{j=0}^{2n-2} = \mathbf{C}\,(x_j)_{j=0}^{n-1}\,, \tag{0.0.13}$$

which can be expressed as the following matrix×vector multiplication, see ➜Rem. 4.1.17:

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

sp:0

(0.3.a) (2 pts)

Using EIGEN, implement a C++ function with signature

```
MatrixXd buildLTFIRMatrix(const VectorXd &h);
```

prb:Rankk:fc:1

that initializes the matrix $\mathbf{C}$ from (0.0.13). The vector `h` specifies the entries of $\mathbf{C}$.

HINT 1 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]

Rankk-10

C++11-code 0.0.14: Function `buildDistanceLSQMatrix`.

```
2  MatrixXd buildLTFIRMatrix(const VectorXd & h)
3  {
4      // Initialization
5      unsigned int n = h.size();
6      MatrixXd C(2*n-1, n);
7
8      C.setZero();
9
10     for(unsigned i=0; i<n; ++i) {
11         C.col(i).segment(i,n) = h;
12     }
13
14     return C;
15 }
```

▲

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

sp:1

(0.3.b) (9 pts) [depends on Sub-problem (0.3.a)]

Implement both member funcions 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 (0.3.b): You may use the function `buildLTFIRMatrix` from Sub-problem (0.3.a). ⌟

HINT 2 for (0.3.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 $\mathbf{y} = \tilde{\mathbf{C}}\mathbf{x}$ for specific inputs $\mathbf{h}$, $\mathbf{c}$ and k. The correct result is reported as a comment in the code. ⌟

SOLUTION of (0.3.b):

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

[3 pts. for efficient operator()]

Rankk-11

C++11-code 0.0.15: Constructor of class LTFIR_lowrank.

```
2   LTFIR_lowrank(const VectorXd& h, unsigned int k) {
3       MatrixXd C = buildLTFIRMatrix(h);
4
5       JacobiSVD<MatrixXd> svd(C, ComputeThinU | ComputeThinV);
6       // With Eigen::svd you can ask for thin U or V to be
            computed.
7       // In case of a rectangular m × n matrix,
8       // with j the smaller value among m and n,
9       // there can only be at most j singular values.
10      // The remaining columns of U and V do not correspond
11      // to actual singular vectors and are not computed in thin
            format.
12
13      VectorXd s = svd.singularValues();
14      s.conservativeResize(k);
15      auto S = s.asDiagonal(); // kxk
16
17      MatrixXd U = svd.matrixU();
18      MatrixXd V = svd.matrixV();
19      U_ = U.leftCols(k) * S; // nxk
20      // Already optimised product between dense and diagonal
            matrix
21      Vt_ = V.leftCols(k).transpose(); // kxn
22  }
```

Rankk-12

C++11-code 0.0.16: Function operator().

```
2   VectorXd operator()(const VectorXd& x) const {
3       VectorXd y;
4
5       assert(x.size() == Vt_.cols() &&
6              "x must have same length of h");
7
8       VectorXd tmp = Vt_ * x;
9       y = U_ * tmp;
10      // Complexity is O(kn + nk) = O(nk).
11      // Given precomputed Ck = U · Vt_,
12      // complexity would have been O(nn).
13      return y;
14  }
```

Rankk-13

C++11-code 0.0.17: Private members of class LTFIR_lowrank.

```
2   MatrixXd U_;   // nxk
3   MatrixXd Vt_; // kxn
```

▲

sp:2 **(0.3.c)** (2 pts) [depends on Sub-problem (0.3.b)]

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 $\mathcal{O}(n^3)$.

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

▲

sp:3 **(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 $2n - 1$.

▲

sp:4 **(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.

Rankk-41 **C++11-code 0.0.18: Constructor of class LTFIR_freq.**

```
2     LTFIR_freq(const VectorXd& h, unsigned k) {
3         n_ = h.size();
4         k_ = k;
5
6         VectorXd h_ = h;
              h_.conservativeResizeLike(VectorXd::Zero(2*n_-1));
7
8         // Forward DFT
9         FFT<double> fft;
10        ch_ = fft.fwd(h_);
11    }
```

Rankk-42 **C++11-code 0.0.19: Function operator().**

```
2     VectorXd operator()(const VectorXd& x) const {
3         assert(x.size() == n_ && "x must have same length of h");
4
5         VectorXd x_ = x;
              x_.conservativeResizeLike(VectorXd::Zero(2*n_-1));
6         // Forward DFT
7         FFT<double> fft;
8         VectorXcd cx = fft.fwd(x_);
9         VectorXcd c  = ch_.cwiseProduct(cx);
10        // Set high frequency coefficients to zero
11        VectorXcd clow = c;
12        for(int j=-k_; j<=+k_; ++j) clow(n_+j) = 0;
13        // Inverse DFT
14        return fft.inv(clow).real();
15    }
```

Rankk-43 **C++11-code 0.0.20: Private members of class LTFIR_freq.**

```
2     int n_;
3     int k_;
4     VectorXcd ch_;
```

What is the asymptotic complexity of the evaluation operator **operator** `()` for $n \to \infty$?

You can find the implementation of the class **LTFIR_freq** in the file `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

:ODE

Problem 0.4: Single step method (23 pts)

This problem concerns numerical integration ➜Chapter 11 with single step methods.

[This problem involves implementation in C++]

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

$$\dot{\mathbf{y}} = \begin{bmatrix} -\theta(y_2) \\ y_1 \end{bmatrix}, \quad \theta \in C^1(\mathbb{R}), \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ y_0 \end{bmatrix}. \tag{0.0.21}$$

{\cp

sp:1

(0.4.a) (2 pts)

Denote by $\xi \in C^2(\mathbb{R})$ the principal of θ, that is $\xi' = \theta$.

Show that $I(\mathbf{y}(t)) = \text{const.}$ for $I(\mathbf{z}) = \frac{1}{2}z_1^2 + \xi(z_2)$, $\mathbf{z} = [z_1, z_2]^\top$ and any solution $t \mapsto \mathbf{y}(t)$ of (0.0.21).

prb:ODE:eq:IVP

HINT 1 for (0.4.a): What is an equivalent condition for $I(\mathbf{y}(t)) = \text{const.}$?

SOLUTION of (0.4.a):

[2 pts.] Consider $I(\mathbf{y}) = \frac{1}{2}y_1^2 + \xi(y_2)$. 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:

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

▲

sp:2

(0.4.b) (4 pts)

Give the concrete defining equation for the discrete evolution Ψ of the implicit midpoint rule ➜Eq. (11.2.18) for (0.0.21).

prb:ODE:eq:IVP

SOLUTION of (0.4.b):

[4 pts.] 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 $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ it reads

$$\Psi(h,\mathbf{y}) := \mathbf{z}: \quad \mathbf{z} = \mathbf{y} + h\mathbf{f}\left(\tfrac{1}{2}(\mathbf{y}+\mathbf{z})\right)$$

and, concretely, for (0.0.21),

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

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

▲

(0.4.c) (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 2×2 matrix is

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

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 2×2 system of equations (0.0.22) can be recast into the standard form

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

For the Jacobian of F we find

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

Using the formula for the inverse of a 2×2-matrix, the Newton's iteration reads

$$\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} - \frac{1}{1+\frac{1}{4}h^2\theta'\left(\frac{1}{2}\left(y_2+z_2^{(k)}\right)\right)}\begin{bmatrix} 1 & -\frac{1}{2}h\theta'\left(\frac{1}{2}\left(y_2+z_2^{(k)}\right)\right) \\ \frac{1}{2}h & 1 \end{bmatrix}\begin{bmatrix} z_1^{(k)} - y_1 + h\theta\left(\frac{1}{2}\left(y_2+z_2^{(k)}\right)\right) \\ z_2^{(k)} - y_2 - \frac{1}{2}h\left(y_1+z_1^{(k)}\right) \end{bmatrix} \, . \tag{0.0.25}$$

For small h we expect $\boldsymbol{\Psi}(h,\mathbf{y}) \approx \mathbf{y}$. Hence $\mathbf{z}^{(0)} := \mathbf{y}$ 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 ➜Rem. 12.3.21. Then one can use Newton's method to solve for the stages $\mathbf{g}_i$, see ➜Rem. 12.3.24.

▲

sp:3 **(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)
```

prb:ODE:eg:IVP

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_d` represent the function θ and its derivative θ'. The vector `y` passes the value $\mathbf{y}$ 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 $\theta(\xi) = e^{\xi}$ with a reference solution. The test performs a single evolution step of size $h = 0.1$ starting from the initial data $\mathbf{y}(0)$. ⌟

SOLUTION of (0.4.d):

[4 pts.]

leStep-2 prb:ODE:sp:3

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

```
2  template<typename Functor, typename Jacobian>
3  Vector2d psi(Functor& theta, Jacobian& theta_d,
4               double h, const Vector2d& y) {
5      Vector2d z;
6      z = y;
7
8      for(unsigned i=0; i<2; ++i) {
9
10         Matrix2d invDF;
11         invDF << 1., -0.5*h*theta_d(0.5*(y(1)+z(1))),
12                  0.5*h, 1.;
13         invDF /= 1 + 0.25*h*h*theta_d(0.5*(y(1)+z(1)));
14
15         Vector2d F;
16         F << z(0) - y(0) + h*theta(0.5*(y(1)+z(1))),
17              z(1) - y(1) - 0.5*h*(y(0)+z(0));
18
19         z = z - invDF*F;
```

```
20      }
21      return z;
22  }
```

▲

sp:4 **(0.4.e)** (3 pts) The following function `lfevl` implements an explicit Runge-Kutta single step method for Eq. (0.0.21) and for some (unknown) smooth function θ (passed as `theta`). The code applies a Runge-Kutta method on N equidistant steps of size h, starting from the initial value `y0`$:= \mathbf{y}(0)$.

leStep-5

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

```
2  template<typename Function>
3  Vector2d lfevl(const Function& theta, Vector2d y0,
4                 double h, unsigned int N) {
5      auto f = [&theta] (const Vector2d& y) -> Vector2d {
6          Vector2d y_dot;
7          y_dot << -theta(y(1)), y(0);
8          return y_dot;
9      };
10     Vector2d yk = y0;
11     for(unsigned k=0; k < N; ++k) {
12         Vector2d k1 = f(yk);
13         Vector2d k2 = f(yk + h/2.*k1);
14         Vector2d k3 = f(yk - h*k1+ 2.*h*k2);
15
16         yk += h/6.*k1 + 2.*h/3.*k2 + h/6.*k3;
17     }
18     return yk;
19 }
```

Write down the Butcher scheme for this method.

SOLUTION of (0.4.e):

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

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

▲

sp:5 **(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 $\mathbf{y}(0) = [0,1]^\top$. 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 empiric 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]

leStep-4 prb:ODE:sp:5

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

```
2     // Vector of number of steps (each entry is twice the previous
        entry).
3     std::vector<unsigned> N = {128, 256, 512, 1024, 2048, 4096,
        8192, 16384};
4     std::cout << std::setw(15) << "N"
5               << std::setw(15) << "error"
6               << std::setw(15) << "rate"
7               << std::endl;
8     double err_old;
9     for(unsigned int i=0; i < N.size(); ++i) {
10        double h = T/N[i];
11        auto yk = lfevl(theta, y0, h, N[i]);
12        double err = (yk - y_exact).norm();
13        std::cout << std::setw(15) << N[i]
14                  << std::setw(15) << err;
15        if(i > 0) {
16            std::cout << std::setw(15) << std::log2(err_old / err);
17        }
18        err_old = err;
19        std::cout << std::setw(15) << std::endl;
20    }
```

▲

End Problem 0.4

olar

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

$$\mathbf{M} = \mathbf{A}\mathbf{Q} \,. \tag{0.0.30}$$

The matrix factorization (0.0.30) is called the polar decomposition of $\mathbf{M}$.

(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 $\mathbf{M}$.

SOLUTION of (0.5.a):

[1 pts. for correct usage of SVD]

[1 pts. for proven symmetry of $\mathbf{A}$]

[1 pts. for proven positive definiteness of $\mathbf{A}$]

[1 pts. for proven orthogonality of $\mathbf{Q}$]

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

$$\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}^\top\mathbf{U}\mathbf{V}^\top = \left(\mathbf{U}\mathbf{\Sigma}\mathbf{U}^\top\right)\left(\mathbf{U}\mathbf{V}^\top\right) \equiv \mathbf{A}\mathbf{Q}\,,$$

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

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

For the second part, we have that $\mathbf{Q}^\top = \left(\mathbf{U}\mathbf{V}^\top\right)^\top = \left(\mathbf{V}^\top\right)^\top\mathbf{U}^\top = \left(\mathbf{V}^\top\right)^{-1}\mathbf{U}^{-1} = \left(\mathbf{U}\mathbf{V}^\top\right)^{-1} = \mathbf{Q}^{-1}$, which means that $\mathbf{Q}$ is also orthogonal.

▲

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

Using EIGEN's numerical linear algebra facilities, write a C++ function

```
std::pair<MatrixXd, MatrixXd> polar(const MatrixXd& M);
```

prb:Polar:pol:1

that computes the polar decomposition (0.0.30) of $\mathbf{M}$, returning the tuple $(\mathbf{A}, \mathbf{Q})$.

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

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

the program computes and prints the matrices $\mathbf{A}$ and $\mathbf{Q}$.

Example output:

```
Matrix A is:
  2.11118  0.847555  2.97062
 0.847555   1.31722  3.39803
  2.97062   3.39803  12.0677
Matrix Q is:
 -0.352666    0.910956   0.213977
  0.872437    0.402776  -0.276811
  0.338348  -0.0890599   0.936797
```

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.]

:Polar-2

C++11-code 0.0.31: Solution of Sub-problem (0.5.b).

```
2  std::pair<MatrixXd,MatrixXd> polar(const MatrixXd& M) {
3      assert(M.rows() == M.cols() && "M must be square!");
4      unsigned n = M.rows();
5      JacobiSVD<MatrixXd> svd(M, ComputeThinU | ComputeThinV);
6
7      VectorXd s = svd.singularValues();
8      MatrixXd S; S.setZero(s.size(),s.size());
9      S.diagonal() = s;
10     MatrixXd U = svd.matrixU();
11     MatrixXd V = svd.matrixV();
12
13     return std::make_pair(U * S * U.transpose(), U * V.transpose());
14 }
```

▲

sp:3

(0.5.c) (1 pts) [depends on prb:Polar:sp:2 (0.5.b)]

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

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 $\mathcal{O}(n^3)$.

▲

End Problem 0.5