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.

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

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

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

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

(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.

▲

(0.1.b) (4 pts) [depends on (0.1.a)]

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.

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. ⌟

▲

(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).

▲

(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.

▲

(0.1.e) (6 pts) [depends on (0.1.d)]

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 $\mathbf{x} := [x_2, \ldots, x_n]^\top$.

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

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

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

HINT 1 for (0.1.e): A template for the function `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
```

▲

(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 $n \to \infty$?

▲

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

$$\int_a^b f(t)\,\mathrm{d}t = a + b\,, \tag{0.0.3a}$$

$$\int_a^b \mathrm{e}^{f(t)}\,\mathrm{d}t = 1 + a^2 + b^2\,. \tag{0.0.3b}$$

(0.2.a) (2 pts) Eq. (0.0.3) 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 $\mathbf{x}$?

▲

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

State the Newton's iteration for solving Eq. (0.0.3) as explicitly as possible.

HINT 1 for (0.2.b): The explicit formula for the inverse of a 2×2 matrix is

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

┘

▲

(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. (0.0.3) 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. ┘

▲

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

$$(0,\dots,0,h_0,\dots,h_{n-1},0,\dots,0) \tag{0.0.4}$$

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

$$(y_j)_{j=0}^{2n-2} = \mathbf{C}\,(x_j)_{j=0}^{n-1}\,, \tag{0.0.5}$$

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

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

(0.3.a) (2 pts)

Using EIGEN, implement a C++ function with signature

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

that initializes the matrix $\mathbf{C}$ from (0.0.5). 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`). ┘

▲

Now the goal is to implement a compressed model for the channel. Consider the class

```cpp
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 $k \in \{1, \dots, n\}$ is passed as the second argument of the constructor.

(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. ⌟

▲

(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 $k \leq n$)?

▲

(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.

▲

(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 0.0.6: 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 0.0.7: 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 0.0.8: Private members of class LTFIR_freq.

```cpp
    int n_;
    int k_;
    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`.

▲

End Problem 0.3

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 $\mathbf{y}(t) := [y_1(t), y_2(t)]^\top$:

$$\dot{\mathbf{y}} = \begin{bmatrix} -\theta(y_2) \\ y_1 \end{bmatrix}, \quad \theta \in C^1(\mathbb{R}), \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ y_0 \end{bmatrix}. \tag{0.0.9}$$

(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 (0.0.9).

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

▲

(0.4.b) (4 pts)

Give the concrete defining equation for the discrete evolution $\boldsymbol{\Psi}$ of the implicit midpoint rule ➜Eq. (11.2.18) for (0.0.9).

(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 (0.0.9). 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

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

(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 (0.0.9) 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 $\mathbf{y}(0)$.

(0.4.e) (3 pts) The following function `lfevl` implements an explicit Runge-Kutta single step method for Eq. (0.0.9) 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`$:= \mathbf{y}(0)$.

C++11-code 0.0.10: 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.

(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.

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 0.0.11. 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*

$$\mathbf{M} = \mathbf{A}\mathbf{Q}\,. \tag{0.0.12}$$

The matrix factorization (0.0.12) is called the polar decomposition of $\mathbf{M}$.

(0.5.a) (4 pts) Give a proof of Thm. 0.0.11.

HINT 1 for (0.5.a): Use the singular value decomposition of $\mathbf{M}$.

(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);
```

that computes the polar decomposition (0.0.12) 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. ⌟

▲

(0.5.c) (1 pts) [depends on (0.5.b)]

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

▲

End Problem 0.5