Homework Problems for Course Numerical Methods for CSE

R. Hiptmair, G. Alberti, F. Leonardi

Version of February 14, 2016

A steady and persistent effort spent on homework problems is essential for success in the course.

You should expect to spend 4-6 hours per week on trying to solve the homework problems. Since many involve small coding projects, the time it will take an individual student to arrive at a solution is hard to predict.

- The assignment sheets will be uploaded on the course webpage on Thursday every week.
- Some or all of the problems of an assignment sheet will be discussed in the tutorial classes on Monday $1\frac{1}{2}$ weeks after the problem sheet has been published.
- A few problems on each sheet will be marked as **core problems**. Every participant of the course is strongly advised to try and solve *at least* the core problems.
- If you want your tutor to examine your solution of the current problem sheet, please put it into the plexiglass trays in front of HG G 53/54 by the Thursday after the publication. You should submit your codes using the online submission interface. This is voluntary, but feedback on your performance on homework problems can be important.
- You are encouraged to hand-in incomplete and wrong solutions, since you can receive valuable feedback even on incomplete attempts.
- Please clearly mark the homework problems that you want your tutor to examine.

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 0

These problems are meant as an introduction to EIGEN in the first tutorial classes of the new semester.

Problem 1 Gram-Schmidt orthogonalization with EIGEN

[1, Code 1.5.4] presents a MATLAB code that effects the Gram-Schmidt orthogonalization of the columns of an argument matrix.

(1a) ⚀ Based on the C++ linear algebra library EIGEN implement a function

```
template <class Matrix>
Matrix gramschmidt(const Matrix &A);
```

that performs the same computations as [1, Code 1.5.4].

(1b) ⚀ Test your implementation by applying it to a small random matrix and checking the orthonormality of the columns of the output matrix.

Problem 2 Fast matrix multiplication

[1, Rem. 1.4.9] presents Strassen's algorithm that can achieve the multiplication of two dense square matrices of size $n = 2^k$, $k \in \mathbb{N}$, with an asymptotic complexity better than $O(n^3)$.

(2a) ⚀ Using EIGEN implement a function

```
MatrixXd strassenMatMult(const MatrixXd & A, const
   MatrixXd & B)
```

that uses Strassen's algorithm to multiply the two matrices $\mathbf{A}$ and $\mathbf{B}$ and return the result as output.

(2b) ⚀ Validate the correctness of your code by comparing the result with EIGEN's built-in matrix multiplication.

(2c) ⚁ Measure the runtime of your function `strassenMatMult` for random matrices of sizes 2^k, $k = 4, \ldots, 10$, and compare with the matrix multiplication offered by the $*$-operator of EIGEN.

Problem 3 Householder reflections

This problem is a supplement to [1, Section 1.5.1] and related to Gram-Schmidt orthogonalization, see [1, Code 1.5.4]. Before you tackle this problem, please make sure that you remember and understand the notion of a QR-decomposition of a matrix, see [1, Thm. 1.5.8]. This problem will put to the test your advanced linear algebra skills.

(3a) ⚁ Listing 1 implements a particular MATLAB function.

Listing 1: MATLAB implementation for Problem 3 in file `houserefl.m`

```
function Z = houserefl(v)
% Porting of houserefl.cpp to Matlab code
% v is a column vector
    % Size of v
    n = size(v,1);

    w = v/norm(v);
    u = w + [1;zeros(n-1,1)];
    q = u/norm(u);
    X = eye(n) - 2*q*q';

    % Remove first column X(:,1) \in span(v)
    Z = X(:,2:end);
end
```

Write a C++ function with declaration:

```
void houserefl(const VectorXd &v, MatrixXd &Z);
```

that is equivalent to the MATLAB function `houserefl()`. Use data types from EIGEN.

(3b) ⚂ Show that the matrix $\mathbf{X}$, defined at line 10 in Listing 1, satisfies:

$$\mathbf{X}^\top \mathbf{X} = \mathbf{I}_n$$

HINT: $\|\mathbf{q}\|^2 = 1$.

(3c) ⚃ Show that the first column of $\mathbf{X}$, after line 9 of the function `houserefl`, is a multiple of the vector $\mathbf{v}$.

HINT: Use the previous hint, and the facts that $\mathbf{u} = \mathbf{w} + \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ and $\|\mathbf{w}\| = 1$.

(3d) ⚂ What property does the set of columns of the matrix $\mathbf{Z}$ have? What is the purpose of the function `houserefl`?

HINT: Use (3b) and (3c).

(3e) ⚀ What is the asymptotic complexity of the function `houserefl` as the length n of the input vector $\mathbf{v}$ goes to ∞?

(3f) ⚂ Rewrite the function as MATLAB function and use a *standard function* of MATLAB to achieve the same result of lines 5-9 with a single call to this function.

HINT: It is worth reading [1, Rem. 1.5.11] before mulling over this problem.

Issue date: 21.0.9.2015
Hand-in: — (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 1

You should try to your best to do the core problems. If time permits, please try to do the rest as well.

Problem 1 Arrow matrix-vector multiplication (core problem)

Consider the multiplication of the two "arrow matrices" $\mathbf{A}$ with a vector $\mathbf{x}$, implemented as a function `arrowmatvec(d,a,x)` in the following MATLAB script

Listing 2: multiplying a vector with the product of two "arrow matrices"

```
1 function y = arrowmatvec(d,a,x)
2 % Multiplying a vector with the product of two ``arrow
    matrices''
3 % Arrow matrix is specified by passing two column
    vectors a and d
4 if (length(d) ~= length(a)), error ('size mismatch'); end
5 % Build arrow matrix using the MATLAB function diag()
6 A = [diag(d(1:end-1)),a(1:end-1);(a(1:end-1))',d(end)];
7 y = A*A*x;
```

(1a) ⚀ For general vectors $d = (d_1, \ldots, d_n)^\top$ and $a = (a_1, \ldots, a_n)^\top$, sketch the matrix $\mathbf{A}$ created in line 6 of Listing 2.

HINT: This MATLAB script is provided as file `arrowmatvec.m`.

(1b) ⚁ The `tic-toc` timing results for `arrowmatvec.m` are available in Figure 1. Give a detailed explanation of the results.

HINT: This MATLAB created figure is provided as file `arrowmatvectiming.{eps,jpg}`.

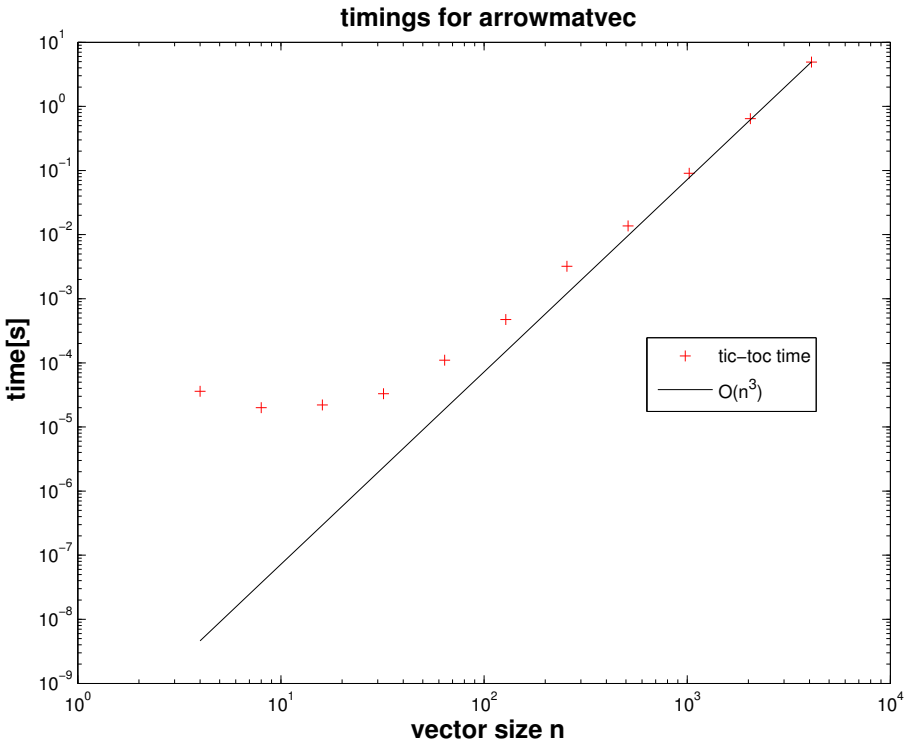


Figure 1: timings for `arrowmatvec(d,a,x)`

(1c) ⚂ Write an *efficient* MATLAB function

```
function y = arrowmatvec2(d,a,x)
```

that computes the same multiplication as in code 2 but with optimal asymptotic complexity with respect to n. Here `d` passes the vector $(d_1, \ldots, d_n)^T$ and `a` passes the vector $(a_1, \ldots, a_n)^T$.

(1d) ⚀ What is the complexity of your algorithm from sub-problem (1c) (with respect to problem size n)?

(1e) ⚂ Compare the runtime of your implementation and the implementation given in code 2 for $n = 2^{5,6,\ldots,12}$. Use the routines `tic` and `toc` as explained in example [1, Ex. 1.4.10] of the Lecture Slides.

(1f) ⚀ Write the EIGEN codes corresponding to the functions `arrowmatvec` and `arrowmatvec2`.

Problem 2 Avoiding cancellation (core problem)

In [1, Section 1.5.4] we saw that the so-called *cancellation phenomenon* is a major cause of numerical instability, *cf.* [1, § 1.5.41]. Cancellation is the massive amplification of *relative errors* when subtracting two real numbers of about the same value.

Fortunately, expressions vulnerable to cancellation can often be recast in a mathematically equivalent form that is no longer affected by cancellation, see [1, § 1.5.50]. There we studied several examples, and this problem gives some more.

(2a) ⚀ We consider the function

$$f_1(x_0, h) := \sin(x_0 + h) - \sin(x_0) \,. \tag{1}$$

It can the transformed into another form, $f_2(x_0, h)$, using the trigonometric identity

$$\sin(\varphi) - \sin(\psi) = 2\cos\left(\frac{\varphi + \psi}{2}\right)\sin\left(\frac{\varphi - \psi}{2}\right).$$

Thus, f_1 and f_2 give the same values, in exact arithmetic, for any given argument values x_0 and h.

1. Derive $f_2(x_0, h)$, which does no longer involve the difference of return values of trigonometric functions.
2. Suggest a formula that avoids cancellation errors for computing the approximation $(f(x_0 + h) - f(x_0))/h)$ of the derivative of $f(x) := \sin(x)$ at $x = x_0$. Write a MATLAB program that implements your formula and computes an approximation of $f'(1.2)$, for $h = 1 \cdot 10^{-20}, 1 \cdot 10^{-19}, \cdots, 1$.

   HINT: For background information refer to [1, Ex. 1.5.43].
3. Plot the error (in doubly logarithmic scale using MATLAB's `loglog` plotting function) of the derivative computed with the suggested formula and with the naive implementation using f_1.
4. Explain the observed behaviour of the error.

(2b) ⚁ Using a trick applied in [1, Ex. 1.5.55] show that

$$\ln(x - \sqrt{x^2 - 1}) = -\ln(x + \sqrt{x^2 - 1}) \,.$$

Which of the two formulas is more suitable for numerical computation? Explain why, and provide a numerical example in which the difference in accuracy is evident.

(2c) ⚀ For the following expressions, state the numerical difficulties that may occur, and rewrite the formulas in a way that is more suitable for numerical computation.

1. $\sqrt{x+\frac{1}{x}} - \sqrt{x-\frac{1}{x}}$, where $x \gg 1$.
2. $\sqrt{\frac{1}{a^2}+\frac{1}{b^2}}$, where $a \approx 0, b \approx 1$.

Problem 3 Kronecker product

In [1, Def. 1.4.16] we learned about the so-called Kronecker product, available in MATLAB through the command `kron`. In this problem we revisit the discussion of [1, Ex. 1.4.17]. Please refresh yourself on this example and study [1, Code 1.4.18] again.

As in [1, Ex. 1.4.17], the starting point is the line of MATLAB code

$$\mathbf{y} = \texttt{kron(A,B)} * \texttt{x}, \tag{2}$$

where the arguments are $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n}, \mathbf{x} \in \mathbb{R}^{n \cdot n}$.

(3a) ⚀ Obtain further information about the `kron` command from MATLAB help issuing `doc kron` in the MATLAB command window.

(3b) ⚀ Explicitly write Eq. (2) in the form $\mathbf{y} = \mathbf{M}\mathbf{x}$ (i.e. write down $\mathbf{M}$), for $\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $\mathbf{B} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$.

(3c) ⚀ What is the asymptotic complexity (→ [1, Def. 1.4.3]) of the MATLAB code (2)? Use the Landau symbol from [1, Def. 1.4.4] to state your answer.

(3d) ⚀ Measure the runtime of (2) for $n = 2^{3,4,5,6}$ and random matrices. Use the MATLAB functions `tic` and `toc` as explained in example [1, Ex. 1.4.10] of the Lecture Slides.

(3e) ⚀ Explain in detail, why (2) can be replaced with the single line of MATLAB code

```
y = reshape( B * reshape(x,n,n) * A', n*n, 1);
```
(3)

and compare the execution times of (2) and (3) for random matrices of size $n = 2^{3,4,5,6}$.

(3f) ⚀ Based on the EIGEN numerical library (→ [1, Section 1.2.3]) implement a C++ function

```
template <class Matrix>
void kron(const Matrix & A, const Matrix & B, Matrix &
   C) {
  // Your code here
}
```

returns the Kronecker product of the argument matrices `A` and `B` in the matrix `C`.

HINT: Feel free (but not forced) to use the partial codes provided in `kron.cpp` as well as the CMake file `CMakeLists.txt` (including `cmake-modules`) and the timing header file `timer.h`.

(3g) ⚂ Devise an implementation of the MATLAB code (2) in C++according to the function definition

```
template <class Matrix, class Vector>
void kron_mv(const Matrix & A, const Matrix & B, const
   Vector & x, Vector & y);
```

The meaning of the arguments should be self-explanatory.

(3h) ⚂ Now, using a function definition similar to that of the previous sub-problem, implement the C++ equivalent of (3) in the function `kron_mv_fast`.

HINT: Study [1, Rem. 1.2.23] about "reshaping" matrices in EIGEN.

(3i) ⚂ Compare the runtimes of your two implementations as you did for the MATLAB implementations in sub-problem (3e).

Problem 4 Structured matrix–vector product

In [1, Ex. 1.4.14] we saw how the particular structure of a matrix can be exploited to compute a matrix-vector product with substantially reduced computational effort. This problem presents a similar case.

Consider the real $n \times n$ matrix $\mathbf{A}$ defined by $(\mathbf{A})_{i,j} = a_{i,j} = \min\{i, j\}$, for $i, j = 1, \ldots, n$. The matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ can be implemented in MATLAB as

$$\texttt{y = min(ones(n,1) * (1:n), (1:n)' * ones(1,n)) * x;} \tag{4}$$

(4a) ⚀ What is the asymptotic complexity (for $n \to \infty$) of the evaluation of the MATLAB command displayed above, with respect to the problem size parameter n?

(4b) ⚂ Write an *efficient* MATLAB function

```
function y = multAmin(x)
```

that computes the same multiplication as (4) but with a better asymptotic complexity with respect to n.

HINT: you can test your implementation by comparing the returned values with the ones obtained with code (4).

(4c) ⚀ What is the asymptotic complexity (in terms of problem size parameter n) of your function `multAmin`?

(4d) ⚂ Compare the runtime of your implementation and the implementation given in (4) for $n = 2^{5,6,\ldots,12}$. Use the routines `tic` and `toc` as explained in example [1, Ex. 1.4.10] of the Lecture Slides.

(4e) ⚂ Can you solve task (4b) without using any `for`- or `while`-loop? Implement it in the function

```
function y = multAmin2(x)
```

HINT: you may use the MATLABbuilt-in command `cumsum`.

(4f) ⚀ Consider the following MATLABscript `multAB.m`:

Listing 3: MATLABscript calling `multAmin`

```
1 n = 10;
2 B = diag(-ones(n-1,1),-1)+diag([2*ones(n-1,1);1],0)...
3     + diag(-ones(n-1,1),1);
4 x = rand(n,1);
5 fprintf('|x-y| = %d\n',norm(multAmin(B*x)-x));
```

Sketch the matrix $\mathbf{B}$ created in line 3 of `multAB.m`.

HINT: this MATLABscript is provided as file `multAB.m`.

(4g) ⚂ Run the code of Listing 3 several times and conjecture a relationship between the matrices $\mathbf{A}$ and $\mathbf{B}$ from the output. Prove your conjecture.

HINT: You must take into account that computers inevitably commit round-off errors, see [1, Section 1.5].

(4h) ⚀ Implement a C++ function with declaration

```
template <class Vector>
void minmatmv(const Vector &x,Vector &y);
```

that realizes the efficient version of the MATLAB line of code (4). Test your function by comparing with output from the equivalent MATLAB functions.

Problem 5 Matrix powers

(5a) ⚀ Implement a MATLAB function

$$\texttt{Pow(A,k)}$$

that, using only basic linear algebra operations (including matrix-vector or matrix-matrix multiplications), computes efficiently the k^{th} power of the $n \times n$ matrix $\mathbf{A}$.

HINT: use the MATLAB operator ^ to test your implementation on random matrices $\mathbf{A}$.

HINT: use the MATLAB functions `de2bi` to extract the "binary digits" of an integer.

(5b) ⚀ Find the asymptotic complexity in k (and n) taking into account that in MATLAB a matrix-matrix multiplication requires a $O(n^3)$ effort.

(5c) ⚀ Plot the runtime of the built-in MATLAB power (^) function and find out the complexity. Compare it with the function `Pow` from (5a).
Use the matrix

$$A_{j,k} = \frac{1}{\sqrt{n}} \exp\left(\frac{2\pi i\, jk}{n}\right)$$

to test the two functions.

(5d) ⚀ Using EIGEN, devise a C++ function with the calling sequence

```
template <class Matrix>
void matPow(const Matrix &A,unsigned int k);
```

that computes the $k^{\texttt{th}}$ power of the square matrix $\mathbf{A}$ (passed in the argument `A`). Of course, your implementation should be as efficient as the MATLAB version from sub-problem (5a).

HINT: matrix multiplication suffers no aliasing issues (you can safely write `A = A*A`).

HINT: feel free to use the provided `matPow.cpp`.

HINT: you may want to use `log` and `ceil`.

HINT: EIGEN implementation of power (`A.pow(k)`) can be found in:

```
#include <unsupported/Eigen/MatrixFunctions>
```

Problem 6 Complexity of a MATLAB function

In this problem we recall a concept from linear algebra, the diagonalization of a square matrix. Unless you can still define what this means, please look up the chapter on "eigenvalues" in your linear algebra lecture notes. This problem also has a subtle relationship with Problem 5

We consider the MATLAB function defined in `getit.m` (cf. Listing 4)

Listing 4: MATLABimplementation of `getit` for Problem 6.

```
function y = getit(A, x, k)
    [S,D] = eig(A);
    y = S*diag(diag(D).^k)* (S\x);
end
```

HINT: Give the command `doc eig` in MATLAB to understand what `eig` does.

HINT: You may use that `eig` applied to an $n \times n$-matrix requires an asymptotic computational effort of $O(n^3)$ for $n \to \infty$.

HINT: in MATLAB, the function `diag(x)` for $\mathbf{x} \in \mathbb{R}^n$, builds a diagonal, $n \times n$ matrix with $\mathbf{x}$ as diagonal. If $\mathbf{M}$ is a $n \times n$ matrix, `diag(M)` returns (extracts) the diagonal of $\mathbf{M}$ as a vector in $\mathbb{R}^n$.

HINT: the operator `v.^k` for $v \in \mathbb{R}^n$ and $k \in \mathbb{N} \setminus \{0\}$ returns the vector with components v_i^k (i.e. component-wise exponent)

(6a) ⚀ What is the output of `getit`, when A is a diagonalizable $n \times n$ matrix, $x \in \mathbb{R}^n$ and $k \in \mathbb{N}$?

(6b) ⚁ Fix $k \in \mathbb{N}$. Discuss (in detail) the asymptotic complexity of `getit` $n \to \infty$.

Issue date: 17.09.2015
Hand-in: 24.09.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

ETH Zürich
D-MATH

Numerical Methods for CSE

Problem Sheet 2

You should try to your best to do the core problems. If time permits, please try to do the rest as well.

Problem 1 Lyapunov Equation (core problem)

Any linear system of equations with a finite number of unknowns can be written in the "canonical form" $\mathbf{Ax} = \mathbf{b}$ with a system matrix $\mathbf{A}$ and a right hand side vector $\mathbf{b}$. However, the LSE may be given in a different form and it may not be obvious how to extract the system matrix. This task gives an intriguing example and also presents an important *matrix equation*, the so-called Lyapunov Equation.

Given $\mathbf{A} \in \mathbb{R}^{n\times n}$, consider the equation

$$\mathbf{AX} + \mathbf{XA}^T = \mathbf{I} \tag{5}$$

with unknown $\mathbf{X} \in \mathbb{R}^{n\times n}$.

(1a) ⊡ Show that for a fixed matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ the mapping

$$L : \begin{cases} \mathbb{R}^{n,n} & \to & \mathbb{R}^{n,n} \\ \mathbf{X} & \mapsto & \mathbf{AX} + \mathbf{XA}^T \end{cases}$$

is linear.

HINT: Recall from linear algebra the definition of a linear mapping between two vector spaces.

In the sequel let $\mathrm{vec}(\mathbf{M}) \in \mathbb{R}^{n^2}$ denote the column vector obtained by reinterpreting the internal coefficient array of a matrix $M \in \mathbb{R}^{n,n}$ stored in column major format as the data array of a vector with n^2 components. In MATLAB, $\mathrm{vec}(\mathbf{M})$ would be the column vector obtained by `reshape(M,n*n,1)` or by `M(:)`. See [1, Rem. 1.2.23] for the implementation with Eigen.

Problem (5) is equivalent to a linear system of equations

$$\mathbf{C}\mathrm{vec}(\mathbf{X}) = \mathbf{b} \tag{6}$$

with system matrix $\mathbf{C} \in \mathbb{R}^{n^2,n^2}$ and right hand side vector $\mathbf{b} \in \mathbb{R}^{n^2}$.

(1b) ⊡ Refresh yourself on the notion of "sparse matrix", see [1, Section 1.7] and, in particular, [1, Notion 1.7.1], [1, Def. 1.7.3].

(1c) ⊡ Determine $\mathbf{C}$ and $\mathbf{b}$ from (6) for $n = 2$ and

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ -1 & 3 \end{bmatrix}.$$

(1d) ⚂ Use the Kronecker product to find a general expression for $\mathbf{C}$ in terms of a general $\mathbf{A}$.

(1e) ⚂ Write a MATLAB function

```
function C = buildC (A)
```

that returns the matrix $\mathbf{C}$ from (6) when given a square matrix $\mathbf{A}$. (The function `kron` may be used.)

(1f) ⚃ Give an upper bound (as sharp as possible) for $\mathrm{nnz}(\mathbf{C})$ in terms of $\mathrm{nnz}(\mathbf{A})$. Can $\mathbf{C}$ be legitimately regarded as a sparse matrix for large n even if $\mathbf{A}$ is dense?

HINT: Run the following MATLAB code:

```
n=4;
A=sym('A',[n,n]);
I=eye(n);
C=buildC(A)
```

(1g) ⚂ Implement a C++ function

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

that builds the Eigen matrix $\mathbf{C}$ from $\mathbf{A}$. Make sure that initialization is done efficiently using an intermediate triplet format. Read [1, Section 1.7.3] very carefully before starting.

(1h) ⚀ Validate the correctness of your C++ implementation of `buildC` by comparing with the equivalent Matlab function for $n = 5$ and

$$A = \begin{bmatrix} 10 & 2 & 3 & 4 & 5 \\ 6 & 20 & 8 & 9 & 1 \\ 1 & 2 & 30 & 4 & 5 \\ 6 & 7 & 8 & 20 & 0 \\ 1 & 2 & 3 & 4 & 10 \end{bmatrix}.$$

(1i) ⚁ Write a C++ function

```
void solveLyapunov(const MatrixXd & A, MatrixXd & X)
```

that returns the solution of (5) in the $n \times n$-matrix $\mathbf{X}$, if $A \in \mathbb{R}^{n,n}$.

(1j) ⚁ Test your C++ implementation of `solveLyapunov` by comparing with Matlab for the test case proposed in (1h).

Problem 2 Partitioned Matrix (core problem)

Based on the block view of matrix multiplication presented in [1, § 1.3.15], we looked a *block elimination* for the solution of block partitioned linear systems of equations in [1, § 1.6.93]. Also of interest are [1, Rem. 1.6.46] and [1, Rem. 1.6.44] where LU-factorization is viewed from a block perspective. Closely related to this problem is [1, Ex. 1.6.96], which you should study again as warm-up to this problem.

Let the matrix $\mathbf{A} \in \mathbb{R}^{n+1,n+1}$ be partitioned according to

$$\mathbf{A} = \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix}, \tag{7}$$

where $\mathbf{v} \in \mathbb{R}^n$, $\mathbf{u} \in \mathbb{R}^n$, and $\mathbf{R} \in \mathbb{R}^{n \times n}$ is upper triangular and regular.

(2a) ⚀ Give a necessary and sufficient condition for the triangular matrix $\mathbf{R}$ to be invertible.

(2b) ⚁ Determine expressions for the subvectors $\mathbf{z} \in \mathbb{R}^n, \xi \in \mathbb{R}$ of the solution vector of the linear system of equations

$$\begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \beta \end{bmatrix}$$

for arbitrary $\mathbf{b} \in \mathbb{R}^n, \beta \in \mathbb{R}$.

HINT: Use blockwise Gaussian elimination as presented in [1, § 1.6.93].

(2c) ⚀ Show that $\mathbf{A}$ is regular if and only if $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$.

(2d) ⚁ Implement the C++ function

```
template <class Matrix, class Vector>
void solvelse(const Matrix & R, const Vector & v, const
   Vector & u, const Vector & b, Vector & x);
```

for computing the solution of $\mathbf{Ax} = \mathbf{b}$ (with $\mathbf{A}$ as in (7)) efficiently. Perform size check on input matrices and vectors.

HINT: Use the decomposition from (2b).

HINT: you can rely on the `triangularView()` function to instruct EIGEN of the triangular structure of $\mathbf{R}$, see [1, Code 1.2.14].

HINT: using the construct:

```
typedef typename Matrix::Scalar Scalar;
```

you can obtain the scalar type of the `Matrix` type (e.g. `double` for `MatrixXd`). This can then be used as:

```
Scalar a = 5;
```

HINT: using `triangularView` and templates you may incur in weird compiling errors. If this happens to you, check http://eigen.tuxfamily.org/dox/TopicTemplateKeyword.html

HINT: sometimes the C++ keyword `auto` (only in std. C++11) can be used if you do not want to explicitly write the return type of a function, as in:

```
MatrixXd a;
auto b = 5*a;
```

(2e) ⚂ Test your implementation by comparing with a standard LU-solver provided by EIGEN.

HINT: Check the page http://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html.

(2f) ⚀ What is the asymptotic complexity of your implementation of `solvelse()` in terms of problem size parameter $n \to \infty$?

Problem 3 Banded matrix

For $n \in \mathbb{N}$ we consider the matrix

$$\mathbf{A} := \begin{bmatrix} 2 & a_1 & 0 & \dots & \dots & \dots & 0 \\ 0 & 2 & a_2 & 0 & \dots & \dots & 0 \\ b_1 & 0 & \ddots & \ddots & \ddots & & \vdots \\ 0 & b_2 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & a_{n-1} \\ 0 & 0 & \dots & 0 & b_{n-2} & 0 & 2 \end{bmatrix} \in \mathbb{R}^{n,n}$$

with $a_i, b_i \in \mathbb{R}$.

Remark. The matrix $\mathbf{A}$ is an instance of a banded matrix, see [1, Section 1.7.6] and, in particular, the examples after [1, Def. 1.7.53]. However, you need not know any of the content of this section for solving this problem.

(3a) ⚀ Implement an *efficient* C++ function:

```
template <class Vector>
void multAx(const Vector & a, const Vector & b, const
   Vector & x, Vector & y);
```

for the computation of $\mathbf{y} = \mathbf{A}\mathbf{x}$.

(3b) ⚁ Show that $\mathbf{A}$ is invertible if $a_i, b_i \in [0, 1]$.

HINT: Give an indirect proof that $\ker \mathbf{A}$ is trivial, by looking at the largest (in modulus) component of an $\mathbf{x} \in \ker \mathbf{A}$.

Remark. That $\mathbf{A}$ is invertible can immediately be concluded from the general fact that kernel vectors of irreducible, diagonally dominant matrices (→ [1, Def. 1.8.8]) must be multiples of $[1, 1, \ldots, 1]^\top$. Actually, the proof recommended in the hint shows this fact first before bumping into a contradiction.

(3c) ⚄ Fix $b_i = 0, \forall i = 1, \ldots, n-2$. Implement an efficient C++ function

```
template <class Vector>
void solvelseAupper(const Vector & a, const Vector &
   r, Vector & x);
```

solving $\mathbf{Ax} = \mathbf{r}$.

(3d) ⚄ For general $a_i, b_i \in [0, 1]$ devise an efficient C++ function:

```
template <class Vector>
void solvelseA(const Vector & a, const Vector & b,
   const Vector & r, Vector & x);
```

that computes the solution of $\mathbf{Ax} = \mathbf{r}$ by means of Gaussian elimination. You cannot use any high level solver routines of EIGEN.

HINT: Thanks to the constraint $a_i, b_i \in [0, 1]$, pivoting is not required in order to ensure stability of Gaussian elimination. This is asserted in [1, Lemma 1.8.9], but you may just use this fact here. Thus, you can perform a straightforward Gaussian elimination from top to bottom as you have learned it in your linear algebra course.

(3e) ⚀ What is the asymptotic complexity of your implementation of `solvelseA` for $n \to \infty$.

(3f) ⚄ Implement `solvelseAEigen` as in (3d), this time using EIGEN's sparse elimination solver.

HINT: The standard way of initializing a sparse EIGEN-matrix efficiently, is via the triplet format as discussed in [1, Section 1.7.3]. You may also use direct initialization of a sparse matrix, provided that you `reserve()` enough space for the non-zero entries of each column, see documentation.

Problem 4 Sequential linear systems

This problem is about a sequence of linear systems, please see [1, Rem. 1.6.87]. The idea is that if we solve several linear systems with the same matrix $\mathbf{A}$, the computational cost may be reduced by performing the LU decomposition only once.

Consider the following MATLAB function with input data $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{b} \in \mathbb{R}^n$.

```
function X = solvepermb(A,b)
[n,m] = size(A);
if ((n ~= numel(b)) || (m ~= numel(b))), error('Size
   mismatch'); end
X = [];
for l=1:n
   X = [X,A\b];
   b = [b(end);b(1:end-1)];
end
```

(4a) ⚀ What is the asymptotic complexity of this function as $n \to \infty$?

(4b) ⚁ Port the MATLAB function `solvepermb` to C++ using EIGEN. (This means that the C++ code should perform exactly the same computations in exactly the same order.)

(4c) ⚂ Design an efficient implementation of this function with asymptotic complexity $O(n^3)$ in Eigen.

Issue date: 24.09.2015

Hand-in: 01.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 3

Problem 1 Rank-one perturbations (core problem)

This problem is another application of the Sherman-Morrison-Woodbury formula, see [1, Lemma 1.6.113]: please revise [1, § 1.6.104] of the lecture carefully.

Consider the MATLAB code given in Listing 5.

Listing 5: Matlab function `rankoneinvit`

```
function lmin = rankoneinvit(d,tol)
if (nargin < 2), tol = 1E-6; end
ev = d;
lmin = 0.0;
lnew = min(abs(d));

while (abs(lnew-lmin) > tol*lmin)
   lmin = lnew;
   M = diag(d) + ev*ev';
   ev = M\ev;
   ev = ev/norm(ev);
   lnew = ev'*M*ev;
end
lmin = lnew;
```

(1a) ⚀ Write an equivalent implementation in EIGEN of the Matlab function `rankoneinvit`. The C++ code should use exactly the same operations. Do not expect top understand what is the purpose of the function.

(1b) ⚀ What is the asymptotic complexity of the loop body of the function `rankoneinvit`? More precisely, you should look at the asymptotic complexity of the code in the lines 8-12 of Listing 5.

(1c) ⚂ Write an efficient implementation in EIGEN of the loop body, possibly with optimal asymptotic complexity. Validate it by comparing the result with the other implementation in EIGEN.

HINT: Take the clue from [1, Code 1.6.114].

(1d) ⚀ What is the asymptotic complexity of the new version of the loop body?

(1e) ⚂ Tabulate the runtimes of the two *inner loops* of the C++ implementations with different vector sizes $n = 2^k$, $k = 1, 2, 3, \ldots, 9$. Use, as test vector

```
Eigen::VectorXd::LinSpaced(n,1,2)
```

How can you read off the asymptotic complexity from these data?

HINT: Whenever you provide figure from runtime measurements, you have to specify the operating system and compiler (options) used.

Problem 2 Approximating the Hyperbolic Sine

In this problem we study how Taylor expansions can be used to avoid cancellations errors in the approximation of the hyperbolic sine, *cf.* the discussion in [1, Ex. 1.5.60] carefully.

Consider the Matlab code given in Listing 6.

Listing 6: Matlab function `sinh_unstable`

```
1 function y = sinh_unstable(x)
2 t = exp(x);
3 y = 0.5*(t-1/t);
4 end
```

(2a) ⚀ Explain why the function given in Listing 6 may not give a good approximation of the hyperbolic sine for small values of x, and compute the relative error

$$\frac{|\texttt{sinh\_unstable}(x) - \sinh(x)|}{|\sinh(x)|}$$

with Matlab for $x = 10^{-k}$, $k = 1, 2, \ldots, 10$ using as "exact value" the result of the MATLAB built-in function `sinh`.

(2b) ⚀ Write the Taylor expansion of length m around $x = 0$ of the function e^x and also specify the remainder.

(2c) ⚁ Prove that for every $x \geq 0$ the following inequality holds true:

$$\sinh x \geq x. \tag{8}$$

(2d) ⚁ Based on the Taylor expansion, find an approximation for $\sinh(x)$, with $0 \leq x \leq 10^{-3}$, so that the relative approximation error is smaller than 10^{-15}.

Problem 3 C++ project: triplet format to CRS format (core problem)

This exercise deals with sparse matrices and their storage in memory. Before beginning, make sure you are prepared on the subject by reading section [1, Section 1.7] of the lecture notes. In particular, refresh yourself in the various *sparse storage formats* discussed in class (cf. [1, Section 1.7.1]). This problem will test your knowledge of algorithms and of advanced C++ features (i.e. structures and classes[1]). You do not need to use EIGEN to solve this problem.

The ultimate goal of this exercise is to devise a function that allows the conversion of a matrix given in *triplet list format* (COO, → [1, § 1.7.6]) to a matrix in *compressed row storage* (CRSm → [1, Ex. 1.7.9]) format. You do not have to follow the subproblems, provided you can devise a suitable conversion function and suitable data structures for you matrices.

(3a) ⚁ In section [1, § 1.7.6] you saw how a matrix can be stored in triplet (or coordinate) list format. This format stores a collection of triplets (i, j, v) with $i, j \in \mathbb{N}, i, j \geq 0$ (the indices) and $v \in \mathbb{K}$ (the value at (i, j)). Repetitions of i, j are allowed, meaning that the values at the same indices i, j must be *summed* together in the final matrix.

Define a suitable structure:

```
template <class scalar>
struct TripletMatrix;
```

[1] You may have a look at http://www.cplusplus.com/doc/tutorial/classes/.

that stores a matrix of type `scalar` in COO format. You can store sizes and indices in `std::size_t` predefined type.

HINT: Store the rows and columns of the matrix inside the structure.

HINT: You can use a `std::vector<your_type>` to store the collection of triplets.

HINT: You can define an auxiliary structure `Triplet` containing the values i, j, v (with the appropriate types), but you may also use the type `Eigen::Triplet<double>`.

(3b) ⚀ Another format for storing a sparse matrix is the compressed row storage (CRS) format (have a look at [1, Ex. 1.7.9]).

Remark. Here, we are not particularly strict about the "compressed" attribute, meaning that you can store your data in `std::vector`. This may "waste" some memory, because the `std::vector` container adds a padding at the end of is data that allows for `push_back` with amortized $O(1)$ complexity.

Devise a suitable structure:

```
template <class scalar>
struct CRSMatrix;
```

holding the data of the matrix in CRS format.

HINT: Store the rows and columns of the matrix inside the structure. To store the data, you can use `std::vector`.

HINT: You can pair the column indices and value in a single structure `ColValPair`. This will become handy later.

HINT: Equivalently to store the array of pointers to the column indices you can use a nested `std::vector< std::vector<your_type> >`.

(3c) ⚁ Optional: write member functions

```
Eigen::Matrix<scalar, Eigen::Dynamic, Eigen::Dynamic>
    TripletMatrix<scalar>::densify();
Eigen::Matrix<scalar, Eigen::Dynamic, Eigen::Dynamic>
    CRSMatrix<scalar>::densify();
```

for the structure `TripletMatrix` and `CRSMatrix` that convert your matrices structures to EIGEN dense matrices types. This can be helpful in debugging your code.

(3d) ⚄ Write a function:

```
template <class scalar>
void tripletToCRS(const TripletMatrix<scalar> & T,
  CRSMatrix<scalar> & C);
```

that converts a matrix **T** in COO format to a matrix **C** in CRS format. Try to be as efficient as possible.

HINT: The parts of the column indices vector in CRS format that correspond to indicdual rows of the matrix must be ordered and without repetitions, whilst the triplets in the input may be in arbitrary ordering and with repetitions. Take care of those aspects in you function definition.

HINT: If you use a `std::vector` container, have a look at the function `std::sort` or at the functions `std::lower_bound` and `std::insert` (both lead to a valid function with different complexities). Look up their precise definition and specification in a C++11 reference.

HINT: You may want to sort a vector containing a structure with multiple values using a particular ordering (i.e. define a custom ordering on your structure and sort according to this ordering). In C++, the standard function `std::sort` provides a way to sort a vector of type `std::vector<your_type>` by defining a `your_type` member operator:

```
bool your_type::operator<(const your_type & other) const;
```

that returns true if `*this` is less than `other` in your particular ordering. Sorting is then performed according to this ordering.

(3e) ⚅ What is the worse case complexity of your function (in the number of triplets)?

HINT: Make appropriate assumptions.

HINT: If you use the C++ standard library functions, look at the documentation: there you can find the complexity of basic container operations.

(3f) ⚀ Test the correctness and runtime of your function `tripletToCRS`.

Problem 4 MATLAB project: mesh smoothing

[1, Ex. 1.7.21] introduced you to the concept of a (planar) triangulation (→ [1, Def. 1.7.22]) and demonstrated how the node positions for smoothed triangulations (→ [1, Def. 1.7.26]) can be computed by solving sparse linear systems of equations with system matrices describing the combinatorial graph Laplacian of the triangulation.

In this problem we will develop a MATLAB code for mesh smoothing and refinement (→ [1, Def. 1.7.33]) of planar triangulations. Before you start make sure that you understand the definitions of planar triangulation ([1, Def. 1.7.22]), the terminology associated with it, and the notion of a smoothed triangulation ([1, Def. 1.7.26]).

(4a) ⚀ Learn about MATLAB's way of describing triangulations by two vectors and one so-called triangle-node incidence matrix from [1, Ex. 1.7.21] or the documentation of the MATLAB function `triplot`.

(4b) ⚂ (This problem is inspired by the dreary reality of software development, where one is regularly confronted with undocumented code written by somebody else who is no longer around.)

Listing 7 lists an uncommented MATLAB code, which takes the triangle-node incidence matrix of a planar triangulation as input.

Describe in detail what is the purpose of the function `processmesh` defined in the file and how exactly this is achieved. Comment the code accordingly.

Listing 7: An undocumented MATLAB function extracting some information from a triangulation given in MATLAB format

```
function [E,Eb] = processmesh(T)
N = max(max(T)); M = size(T,1);
T = sort(T')';
C = [T(:,1) T(:,2); T(:,2) T(:,3); T(:,1) T(:,3)];
% Wow! A creative way to use 'sparse'
A = sparse(C(:,1),C(:,2),ones(3*M,1),N,N);
[I,J] = find(A > 0); E = [I,J];
[I,J] = find(A == 1); Eb = [I,J];
```

HINT: Understand what `find` and `sort` do using MATLAB `doc`.

HINT: The MATLAB file `meshtest.m` demonstrates how to use the function `processmesh`.

(4c) ⚂ Listing 8 displays another undocumented function `getinfo`. As arguments it expects the triangle-node incidence matrix of a planar triangulation (according to MATLAB's conventions) and the output `E` of `processmesh`. Describe in detail how the function works.

Listing 8: Another undocumented function for extracting specific information from a planar triangulation

```
function ET = getinfo(T,E)
% Another creative us of 'sparse'
L = size(E,1); A = sparse(E(:,1),E(:,2),(1:L)',L,L);
ET = [];
for tri=T'
  Eloc = full(A(tri,tri)); Eloc = Eloc + Eloc';
  ET = [ET; Eloc([8 7 4])];
end
```

(4d) ⚄ In [1, Def. 1.7.33] you saw the definition of a regular refinement of a triangular mesh. Write a MATLAB-function:

```
function [x_ref, y_ref, T_ref] = refinemesh(x,y,T)
```

that takes as argument the data of a triangulation in MATLAB format and returns the corresponding data for the new, refined mesh.

(4e) ⚄ [1, Eq. (1.7.29)]–[1, Eq. (1.7.30)] describe the sparse linear system of equations satisfied by the coordinates of the interior nodes of a smoothed triangulation. Justify rigorously, why the linear system of equations [1, Eq. (1.7.32)] always has a unique solution. In other words, show that that the part $\mathbf{A}_{\mathrm{int}}$ of the matrix of the combinatorial graph Laplacian associated with the interior nodes is invertible for any planar triangulation.

HINT: Notice that $\mathbf{A}_{\mathrm{int}}$ is diagonally dominant (→ [1, Def. 1.8.8]).

This observation paves the way for using the same arguments as for sub-problem (3b) of Problem 3 You may also appeal to [1, Lemma 1.8.12].

(4f) ⚂ A planar triangulation can always be transformed uniquely to a smooth triangulation under translation of the interior nodes (maintaining the same connectivity) (see the lecture notes and the previous subproblem).

Write a MATLAB-function

```
function [xs, ys] = smoothmesh(x,y,T);
```

that performs this transformation to the mesh defined by x, y, T. Return the column vectors xs, ys with the new position of the nodes.

HINT: Use the system of equations in [1, (1.7.32)].

Problem 5 Resistance to impedance map

In [1, § 1.6.104], we learned about the Sherman-Morrison-Woodbury update formula [1, Lemma 1.6.113], which allows the efficient solution of a linear system of equations after a low-rank update according to [1, Eq. (1.6.108)], provided that the setup phase of an elimination (→ [1, § 1.6.42]) solver has already been done for the system matrix.

In this problem, we examine the concrete application from [1, Ex. 1.6.115], where the update formula is key to efficient implementation. This application is the computation of the impedance of the circuit drawn in Figure 2 as a function of a variable resistance of a *single* circuit element.

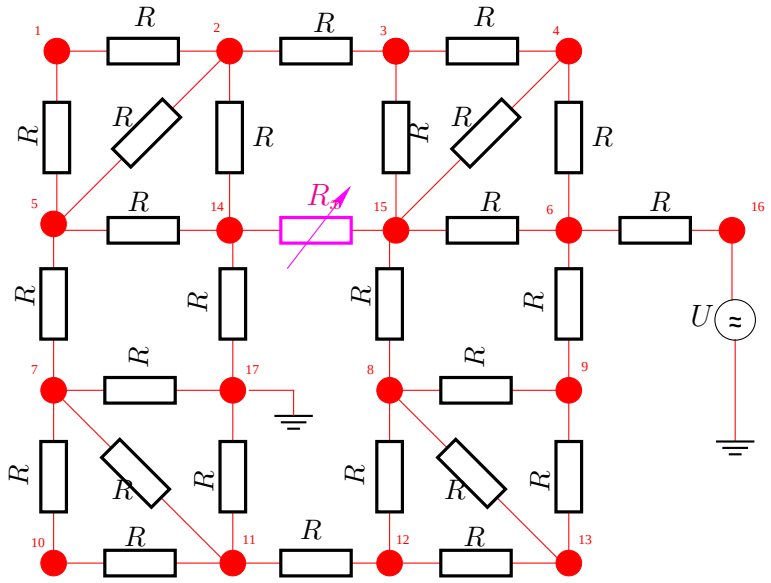


Figure 2: Resistor circuit with a single controlled resistance

(5a) ⚂ Study [1, Ex. 1.6.3] that explains how to compute voltages and currents in a linear circuit by means of nodal analysis. Understand how this leads to a linear system of equations for the unknown nodal potentials. The fundamental laws of circuit analysis should be known from physics as well as the principles of nodal analysis.

(5b) ⚂ Use nodal analysis to derive the linear system of equations satisfied by the nodal potentials of the circuit from Figure 2. The voltage W is applied to node #16 and node #17 is grounded. All resistors except for the controlled one (colored magenta) have the same resistance R. Use the numbering of nodes indicated in Figure 2.

HINT: Optionally, you can make the computer work for you and find a fast way to build a matrix providing only the essential data. This is less tedious, less error prone and more flexible than specifying each entry individually. For this you can use auxiliary data structures.

(5c) ⚂ Characterize the change in the circuit matrix derived in sub-problem (5b) induced by a change in the value of R_x as a low-rank modification of the circuit matrix. Use as a base state $R = R_x$.

HINT: Four entries of the circuit matrix will change. This amounts to a rank-2-modification in the sense of [1, Eq. (1.6.108)] with suitable matrices $\mathbf{u}$ and $\mathbf{v}$.

(5d) ⚂ Based on the EIGEN library, implement a C++ class

```
class ImpedanceMap {
public:
  ImpedanceMap(double R_, double W_) : R(R_), W(W_) {
      // TODO: build A0 = A(1), the rhs and factorize A_0
         with lu = A0.lu()
  };
  double operator() (double Rx) const {
      // TODO: compute the perturbation matrix U and
         solve (A+UU^T) x = rhs, from x, U and R compute
         the impedance
  };
private:
  Eigen::PartialPivLU<Eigen::MatrixXd>> lu;
  Eigen::VectorXd rhs;
  double R, W;
```

```
};
```

whose `()`-operator returns the impedance of the circuit from Figure 2 when supplied with a concrete value for R_x. Of course, this function should be implemented efficiently using [1, Lemma 1.6.113]. The setup phase of Gaussian elimination should be carried out in the constructor performing the LU-factorization of the circuit matrix.

Test your class using $R = 1, W = 1$ and $R_x = 1, 2, 4, \cdots, 1024$.

HINT: See the file `impedancemap.cpp`.

HINT: The impedance of the circuit is the quotient of the voltage at the input node #16 and the current through the voltage source.

Issue date: 01.10.2015

Hand-in: 08.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 4

Problem 1 Order of convergence from error recursion (core problem)

In [1, Exp. 2.3.26] we have observed *fractional* orders of convergence (→ [1, Def. 2.1.17]) for both the secant method, see [1, Code 2.3.25], and the quadratic inverse interpolation method. This is fairly typical for 2-point methods in 1D and arises from the underlying recursions for error bounds. The analysis is elaborated for the secant method in [1, Rem. 2.3.27], where a linearized error recursion is given in [1, Eq. (2.3.31)].

Now we suppose the recursive bound for the norms of the iteration errors

$$\|e^{(n+1)}\| \leq \|e^{(n)}\| \sqrt{\|e^{(n-1)}\|}\,, \tag{9}$$

where $e^{(n)} = x^{(n)} - x^*$ is the error of n-th iterate.

(1a) ⊡ Guess the maximal order of convergence of the method from a numerical experiment conducted in MATLAB.

HINT:[1, Rem. 2.1.19]

(1b) ⚄ Find the maximal guaranteed order of convergence of this method through analytical considerations.

HINT: First of all note that we may assume equality in both the error recursion (9) and the bound $\|e^{(n+1)}\| \leq C\|e^{(n)}\|^p$ that defines convergence of order $p > 1$, because in both cases equality corresponds to a worst case scenario. Then plug the two equations into each other and obtain an equation of the type $\ldots = 1$, where the left hand side involves an error norm that can become arbitrarily small. This implies a condition on p and allows to determine $C > 0$. A formal proof by induction (not required) can finally establish that these values provide a correct choice.

Problem 2 Convergent Newton iteration (core problem)

As explained in [1, Section 2.3.2.1], the convergence of Newton's method in 1D may only be local. This problem investigates a particular setting, in which global convergence can be expected.

We recall the notion of a *convex function* and its geometric definition. A differentiable function $f : [a, b] \mapsto \mathbb{R}$ is convex, if and only if its graph lies on or above its tangent at any point. Equivalently, differentiable function $f : [a, b] \mapsto \mathbb{R}$ is convex, if and only if its derivative is non-decreasing.

Give a "graphical proof" of the following statement:

If $F(x)$ belongs to $C^2(\mathbb{R})$, is strictly increasing, is convex, and has a unique zero, then the Newton iteration [1, (2.3.4)] for $F(x) = 0$ is well defined and will converge to the zero of $F(x)$ for any initial guess $x^{(0)} \in \mathbb{R}$.

Problem 3 The order of convergence of an iterative scheme (core problem)

[1, Rem. 2.1.19] shows how to detect the order of convergence of an iterative method from a numerical experiment. In this problem we study the so-called Steffensen's method, which is a derivative-free iterative method for finding zeros of functions in 1D.

Let $f : [a, b] \mapsto \mathbb{R}$ be twice continuously differentiable with $f(x^*) = 0$ and $f'(x^*) \neq 0$. Consider the iteration defined by

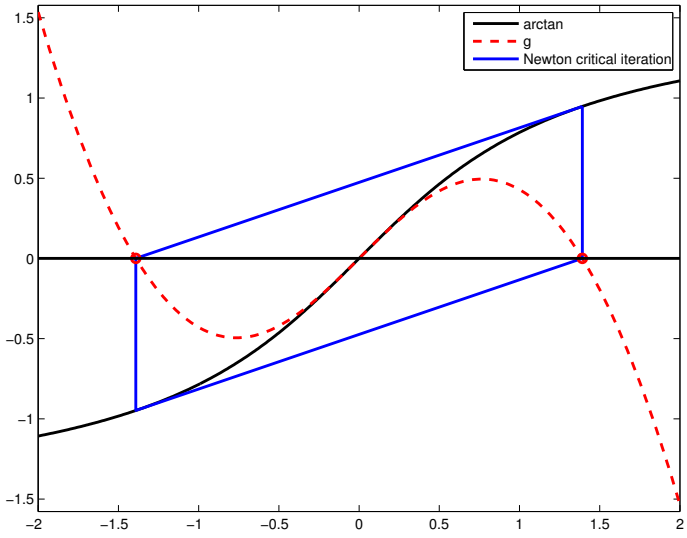
$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{g(x^{(n)})}, \quad \text{where} \quad g(x) = \frac{f(x + f(x)) - f(x)}{f(x)}.$$

(3a) ⚀ Write a MATLAB script that computes the order of convergence to the point x^* of this iteration for the function $f(x) = xe^x - 1$ (see [1, Exp. 2.2.3]). Use $x^{(0)} = 1$.

(3b) ⚀ The function $g(x)$ contains a term like e^{xe^x}, thus it grows very fast in x and the method can not be started for a large $x^{(0)}$. How can you modify the function f (keeping the same zero) in order to allow the choice of a larger initial guess?

HINT: If f is a function and $h : [a, b] \to \mathbb{R}$ with $h(x) \neq 0, \forall x \in [a, b]$, then $(fh)(x) = 0 \Leftrightarrow f(x) = 0$.

Figure 3: Newton iterations with $F(x) = \arctan(x)$ for the critical initial value $x^{(0)}$



Problem 4 Newton's method for $F(x) := \arctan x$

The merely local convergence of Newton's method is notorious, see[1, Section 2.4.2] and [1, Ex. 2.4.46]. The failure of the convergence is often caused by the overshooting of Newton correction. In this problem we try to understand the observations made in [1, Ex. 2.4.46].

(4a) ⚄ Find an equation satisfied by the smallest positive initial guess $x^{(0)}$ for which Newton's method does not converge when it is applied to $F(x) = \arctan x$.

HINT: Find out when the Newton method oscillates between two values.

HINT: Graphical considerations may help you to find the solutions. See Figure 3: you should find an expression for the function g.

(4b) ⚂ Use Newton's method to find an approximation of such $x^{(0)}$, and implement it with Matlab.

Problem 5 Order-p convergent iterations

In [1, Section 2.1.1] we investigated the speed of convergence of iterative methods for the solution of a general non-linear problem $F(\mathbf{x}) = 0$ and introduced the notion of conver-

gence of order $p \geq 1$, see [1, Def. 2.1.17]. This problem highlights the fact that for $p > 1$ convergence may not be guaranteed, even if the error norm estimate of [1, Def. 2.1.17] may hold for some $\mathbf{x}^* \in \mathbb{R}^n$ and all iterates $\mathbf{x}^{(k)} \in \mathbb{R}^n$.

Suppose that, given $\mathbf{x}^* \in \mathbb{R}^n$, a sequence $\mathbf{x}^{(k)}$ satisfies

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq C\|\mathbf{x}^{(k)} - \mathbf{x}^*\|^p \qquad \forall k \quad \text{and some} \quad p > 1 \,.$$

(5a) ⚂ Determine $\epsilon_0 > 0$ such that

$$\|\mathbf{x}^{(0)} - \mathbf{x}^*\| \leq \epsilon_0 \qquad \Longrightarrow \qquad \lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}^*.$$

In other words, ϵ_0 tells us which distance of the initial guess from $\mathbf{x}^*$ still guarantees local convergence.

(5b) ⚂ Provided that $\|\mathbf{x}^{(0)} - \mathbf{x}^*\| < \epsilon_0$ is satisfied, determine the minimal $k_{\min} = k_{\min}(\epsilon_0, C, p, \tau)$ such that

$$\|\mathbf{x}^{(k)} - \mathbf{x}^*\| < \tau.$$

(5c) ⚀ Write a MATLAB function

```
k_min = @(epsilon,C,p,tau) ...
```

and plot $k_{\min} = k_{\min}(\epsilon_0, \tau)$ for the values $p = 1.5, C = 2$. Test you implementation for every $(\epsilon_0, \tau) \in \texttt{linspace}(0, C^{\frac{1}{1-p}})^2 \cap (0,1)^2 \cap \{(i,j) \mid i \geq j\}$

HINT: Use a MATLAB `pcolor` plot and the commands `linspace` and `meshgrid`.

Problem 6 Code quiz

A frequently encountered drudgery in scientific computing is the use and modification of poorly documented code. This makes it necessary to understand the ideas behind the code first. Now we practice this in the case of a simple iterative method.

(6a) ⚄ What is the purpose of the following MATLAB code?

```
1  function y = myfn(x)
2  log2 = 0.693147180559945;
3
```

```
4       y = 0;
5       while (x >   sqrt(2)), x = x/2; y = y + log2; end
6       while (x < 1/sqrt(2)), x = x*2; y = y - log2; end
7       z = x-1;
8       dz = x*exp(-z)-1;
9       while (abs(dz/z) > eps)
10          z = z+dz;
11         dz = x*exp(-z)-1;
12      end
13      y = y+z+dz;
```

(6b) ⚀ Explain the rationale behind the two `while` loops in lines #5, 6.

(6c) ⚁ Explain the loop body of lines #10, 11.

(6d) ⚂ Explain the conditional expression in line #9.

(6e) ⚃ Replace the `while`-loop of lines #9 through #12 with a fixed number of iterations that, nevertheless, guarantee that the result has a relative accuracy `eps`.

Issue date: 08.10.2015

Hand-in: 15.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 5

Problem 1 Modified Newton method (core problem)

The following problem consists in EIGEN implementation of a modified version of the Newton method (in one dimension [1, Section 2.3.2.1] and many dimensions [1, Section 2.4]) for the solution of a nonlinear system. Refresh yourself on stopping criteria for iterative methods [1, Section 2.1.2].

For the solution of the non-linear system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ (with $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$), the following iterative method can be used:

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\, \mathbf{F}(\mathbf{x}^{(k)}) \,, \\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\, \mathbf{F}(\mathbf{y}^{(k)}) \,, \end{aligned} \tag{10}$$

where $D\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{n,n}$ is the Jacobian matrix of $\mathbf{F}$ evaluated in the point $\mathbf{x}$.

(1a) ⚀ Show that the iteration (10) is consistent with $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ in the sense of [1, Def. 2.2.1], that is, show that $\mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every $k \in \mathbb{N}$, if and only if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$ and $D\mathbf{F}(\mathbf{x}^{(0)})$ is regular.

(1b) ⚁ Implement a C++ function

```
template <typename arg, class func, class jac>
void mod_newt_step(const arg & x, arg & x_next,
                   func&& f, jac&& df);
```

that computes a step of the method (10) for a *scalar* function $\mathbf{F}$, that is, for the case $n = 1$.

Here, `f` is a lambda function for the function $F : \mathbb{R} \mapsto \mathbb{R}$ and `df` a lambda to its derivative $F' : \mathbb{R} \mapsto \mathbb{R}$.

HINT: Your lambda functions will likely be:

```
auto f = [ /* captured vars. */ ] (double x)
       { /* fun. body */ };
auto df = [ /* captured vars. */ ] (double x)
        { /* fun. body */ };
```

Notice that here `auto` is `std::function<double(double)>`.

HINT: Have a look at:

- http://en.cppreference.com/w/cpp/language/lambda
- https://msdn.microsoft.com/en-us/library/dd293608.aspx

for more information on lambda functions.

(1c) ⚀ What is the order of convergence of the method?
To investigate it, write a C++function `void mod_newt_ord()` that:

- uses the function `mod_newt_step` from subtask (1b) in order to apply (10) to the following scalar equation

$$\arctan(x) - 0.123 = 0 \, ;$$

- determines empirically the order of convergence, in the sense of [1, Rem. 2.1.19] of the course slides;
- implements meaningful stopping criteria ([1, Section 2.1.2]).

Use $x_0 = 5$ as initial guess.

HINT: the exact solution is $x = \tan(a) = 0.123624065869274\ldots$

HINT: remember that $\arctan'(x) = \frac{1}{1+x^2}$.

(1d) ⚄ Write a C++ function `void mod_newt_sys()` that provides an efficient implementation of the method (10) for the non-linear system

$$\mathbf{F}(\mathbf{x}) := \mathbf{A}\mathbf{x} + \begin{pmatrix} c_1 e^{x_1} \\ \vdots \\ c_n e^{x_n} \end{pmatrix} = \mathbf{0} \, ,$$

where $\mathbf{A} \in \mathbb{R}^{n,n}$ is symmetric positive definite, and $c_i \geq 0$, $i = 1, \ldots, n$. Stop the iteration when the Euclidean norm of the increment $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ relative to the norm of $\mathbf{x}^{(k+1)}$ is smaller than the tolerance passed in `tol`. Use the zero vector as initial guess.

HINT: You can proceed as in the previous task: define function handles as lambda functions and create a "stepping" function, which you the iterate inside a for loop.

HINT: Try and be as efficient as possible. Reuse the matrix factorization when possible.

Problem 2 Solving a quasi-linear system (core problem)

In [1, § 2.4.14] we studied Newton's method for a so-called quasi-linear system of equations, see [1, Eq. (2.4.15)]. In [1, Ex. 2.4.19] we then dealt with concrete quasi-linear system of equations and in this problem we will supplement the theoretical considerations from class by implementation in EIGEN. We will also learn about a simple fixed point iteration for that system, see [1, Section 2.2]. Refresh yourself about the relevant parts of the lecture. You should also try to recall the Sherman-Morrison-Woodbury formula [1, Lemma 1.6.113].

Consider the *nonlinear* (quasi-linear) system:

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b} \,,$$

as in [1, Ex. 2.4.19]. Here, $\mathbf{A} : \mathbb{R}^n \to \mathbb{R}^{n,n}$ is a matrix-valued function:

$$\mathbf{A}(\mathbf{x}) := \begin{bmatrix} \gamma(\mathbf{x}) & 1 & & & & & \\ 1 & \gamma(\mathbf{x}) & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & \gamma(\mathbf{x}) & 1 & \\ & & & & 1 & \gamma(\mathbf{x}) & \end{bmatrix}, \quad \gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2$$

where $\|\cdot\|_2$ is the Euclidean norm.

(2a) ⊡ A fixed point iteration fro (Problem 2) can be obtained by the "frozen argument technique"; in a step we take the argument to the matrix valued function from the previous step and just solve a linear system for the next iterate. State the defining recursion and iteration function for the resulting fixed point iteration.

(2b) ⚀ We consider the fixed point iteration derived in sub-problem (2a). Implement a function computing the iterate $\mathbf{x}^{(k+1)}$ from $\mathbf{x}^{(k)}$ in EIGEN.

HINT: (Optional) This is classical example where *lambda* C++11 functions may become handy.

Write the iteration function as:

```
template <class func, class Vector>
void fixed_point_step(func&& A, const Vector & b, const
    Vector & x, Vector & x_new);
```

where `func` type will be that of a *lambda* function implementing $\mathbf{A}$. The vector `b` will be an input random r.h.s. vector. The vector `x` will be the input $\mathbf{x}^{(k)}$ and `x_new` the output $\mathbf{x}^{(k+1)}$.

Then define a *lambda* function:

```
auto A = [ /* TODO */ ] (const Eigen::VectorXd & x) ->
    Eigen::SparseMatrix<double> & { /* TODO */ };
```

returning $\mathbf{A}(\mathbf{x})$ for an input $\mathbf{x}$ (*capture* the appropriate variables). You can then call your stepping function with:

```
fixed_point_step(A, b, x, x_new);
```

Include `#include <functional>` to use `std::function`.

(2c) ⚀ Write a routine that finds the solution $\mathbf{x}^*$ with the fixed point method applied to the previous quasi-linear system. Use $\mathbf{x}^{(0)} = \mathbf{b}$ as initial guess. Supply it with a suitable correction based stopping criterion as discussed in [1, Section 2.1.2] and pass absolute and relative tolerance as arguments.

(2d) ⚀ Let $\mathbf{b} \in \mathbb{R}^n$ be given. Write the recursion formula for the solution of

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}$$

with the Newton method.

(2e) ⚂ The matrix $\mathbf{A}(\mathbf{x})$, being symmetric and tri-diagonal, is cheap to invert. Rewrite the previous iteration efficiently, exploiting, the Sherman-Morrison-Woodbury inversion formula for rank-one modifications [1, Lemma 1.6.113].

(2f) ⚃ Implement the above step of Newton method in EIGEN.

HINT: If you didn't manage to solve subproblem (2e) use directly the formula from (2d).

HINT: (Optional) Feel free to exploit lambda functions (as above), writing a function:

```
template <class func, class Vector>
void newton_step(func&& A, const Vector & b, const
    Vector & x, Vector & x_new);
```

(2g) ⚀ Repeat subproblem (2c) for the Newton method. As initial guess use $\mathbf{x}^{(0)} = \mathbf{b}$.

Problem 3 Nonlinear electric circuit

In previous exercises we have discussed electric circuits with elements that give rise to linear voltage–current dependence, see [1, Ex. 1.6.3] and [1, Ex. 1.8.1]. The principles of nodal analysis were explained in these cases.

However, the electrical circuits encountered in practise usually feature elements with a *non-linear* current-voltage characteristic. Then nodal analysis leads to non-linear systems of equations as was elaborated in [1, Ex. 2.0.1]. Please note that transformation to frequency domain is not possible for non-linear circuits so that we will always study the direct current (DC) situation.

In this problem we deal with a very simple non-linear circuit element, a diode. The current through a diode as a function of the applied voltage can be modelled by the relationship

$$I_{kj} = \alpha\Big(e^{\beta\frac{U_k-U_j}{U_T}} - 1\Big),$$

with suitable parameters α, β and the thermal voltage U_T.

Now we consider the circuit depicted in Fig. 4 and assume that all resistors have resistance $R = 1$.

(3a) ⚁ Carry out the nodal analysis of the electric circuit and derive the corresponding non-linear system of equations $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ for the voltages in nodes 1,2, and 3, *cf.* [1, Eq. (2.0.2)]. Note that the voltages in nodes 4 and 5 are known (input voltage and ground voltage 0).

(3b) ⚂ Write an EIGEN function

```
    void circuit(const double & alpha, const double &
       beta, const VectorXd & Uin, VectorXd & Uout)
```

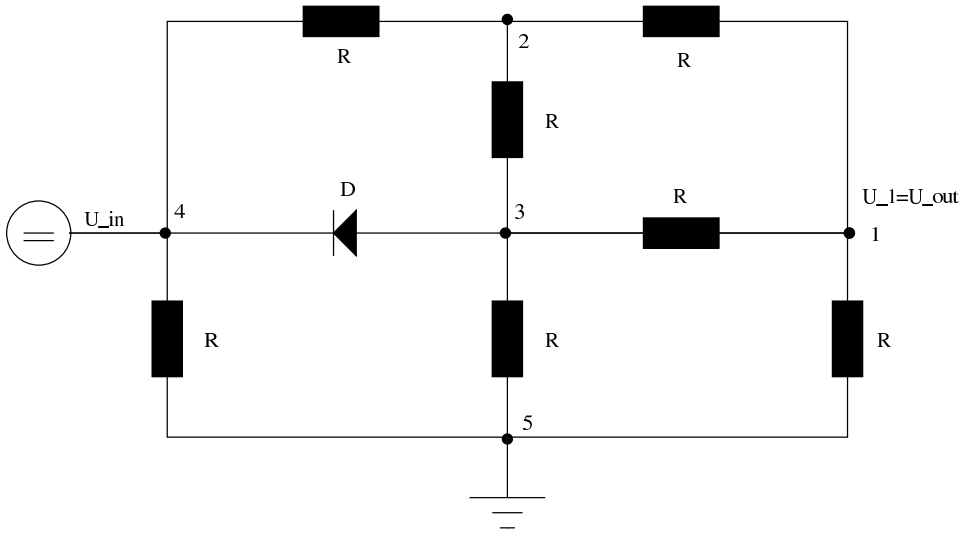


Figure 4: non-linear circuit for Problem 1

that computes the output voltages `Uout` (at node 1 in Fig. 4) for a *sorted* vector of input voltages `Uin` (at node 4) for a thermal voltage $U_T = 0.5$. The parameters `alpha`, `beta` pass the (non-dimensional) diode parameters.

Use Newton's method to solve $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ with a tolerance of $\tau = 10^{-6}$.

(3c) ⚀ We are interested in the nonlinear effects introduced by the diode. Calculate $U_{\text{out}} = U_{\text{out}}(U_{\text{in}})$ as a function of the variable input voltage $U_{\text{in}} \in [0, 20]$ (for non-dimensional parameters $\alpha = 8$, $\beta = 1$ and for a thermal voltage $U_T = 0.5$) and infer the nonlinear effects from the results.

Problem 4 Julia set

Julia sets are famous fractal shapes in the complex plane. They are constructed from the basins of attraction of zeros of complex functions when the Newton method is applied to find them.

In the space $\mathbb{C}$ of complex numbers the equation

$$z^3 = 1 \tag{11}$$

has three solutions: $z_1 = 1$, $z_2 = -\frac{1}{2} + \frac{1}{2}\sqrt{3}i$, $z_3 = -\frac{1}{2} - \frac{1}{2}\sqrt{3}i$ (the cubic roots of unity).

(4a) ⚀ As you know from the analysis course, the complex plane $\mathbb{C}$ can be identified with $\mathbb{R}^2$ via $(x, y) \mapsto z = x + iy$. Using this identification, convert equation (11) into a system of equations $\mathbf{F}(x, y) = \mathbf{0}$ for a suitable function $\mathbf{F} : \mathbb{R}^2 \mapsto \mathbb{R}^2$.

(4b) ⚁ Formulate the Newton iteration [1, Eq. (2.4.1)] for the non-linear equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ with $\mathbf{x} = (x, y)^T$ and $\mathbf{F}$ from the previous sub-problem.

(4c) ⚃ Denote by $\mathbf{x}^{(k)}$ the iterates produced by the Newton method from the previous sub-problem with some initial vector $\mathbf{x}^{(0)} \in \mathbb{R}^2$. Depending on $\mathbf{x}^{(0)}$, the sequence $\mathbf{x}^{(k)}$ will either diverge or converge to one of the three cubic roots of unity.

Analyze the behavior of the Newton iterates using the following procedure:

- use equally spaced points on the domain $[-2, 2]^2 \subset \mathbb{R}^2$ as starting points of the Newton iterations,
- color the starting points differently depending on which of the three roots is the limit of the sequence $\mathbf{x}^{(k)}$.

HINT:: useful MATLABcommands: `pcolor`, `colormap`, `shading`, `caxis`. You may stop the iteration once you are closer in distance to one of the third roots of unity than 10^{-4}.

The three (non connected) sets of points whose iterations are converging to the different z_i are called Fatou domains, their boundaries are the Julia sets.

Issue date: 15.10.2015

Hand-in: 22.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

ETH Zürich
D-MATH

Numerical Methods for CSE

Problem Sheet 6

Problem 1 Evaluating the derivatives of interpolating polynomials (core problem)

In [1, Section 3.2.3.2] we learned about an efficient and "update-friendly" scheme for evaluating Lagrange interpolants at a single or a few points. This so-called Aitken-Neville algorithm, see [1, Code 3.2.31], can be extended to return the derivative value of the polynomial interpolant as well. This will be explored in this problem.

(1a) ⊡ Study the Aitken-Neville scheme introduced in [1, § 3.2.29].

(1b) ⚂ Write an efficient MATLAB function

```
dp = dipoleval(t,y,x)
```

that returns the row vector $(p'(x_1), \ldots, p'(x_m))$, when the argument `x` passes $(x_1, \ldots, x_m)$, $m \in \mathbb{N}$ small. Here, p' denotes the *derivative* of the polynomial $p \in \mathcal{P}_n$ interpolating the data points (t_i, y_i), $i = 0, \ldots, n$, for pairwise different $t_i \in \mathbb{R}$ and data values $y_i \in \mathbb{R}$.

HINT: Differentiate the recursion formula [1, Eq. (3.2.30)] and devise an algorithm in the spirit of the Aitken-Neville algorithm implemented in [1, Code 3.2.31].

(1c) ⚂ For validation purposes devise an alternative, less efficient, implementation of `dipoleval` (call it `dipoleval_alt`) based on the following steps:

1. Use MATLAB's `polyfit` function to compute the monomial coefficients of the Lagrange interpolant.
2. Compute the monomial coefficients of the derivative.
3. Use `polyval` to evaluate the derivative at a number of points.

Use `dipoleval_alt` to verify the correctness of your implementation of `dipoleval` with `t = linspace(0,1,10),y = rand(1,n)` and `x = linspace(0,1,100)`.

Problem 2 Piecewise linear interpolation

[1, Ex. 3.1.8] introduced piecewise linear interpolation as a simple linear interpolation scheme. It finds an interpolant in the space spanned by the so-called tent functions, which are *cardinal basis functions*. Formulas are given in [1, Eq. (3.1.9)].

(2a) ⚂ Write a C++ class `LinearInterpolant` representing the piecewise linear interpolant. Make sure your class has an efficient internal representation of a basis. Provide a constructor and an evaluation `operator()` as described in the following template:

```
class LinearInterpolant {
    public:
        LinearInterpolant( /*  TODO: pass pairs */) {
         // TODO: construct your data from  (t_i, y_i)'s
        }

        double operator() (double x) {
            // TODO: return I(x)
        }
    private:
        // Your data here
};
```

HINT: Recall that C++ provides containers such as `std::vector` and `std::pair`.

(2b) ⚀ Test the correctness of your code.

Problem 3 Evaluating the derivatives of interpolating polynomials (core problem)

This problem is about the Horner scheme, that is a way to efficiently evaluate a polynomial in a given point, see [1, Rem. 3.2.5].

(3a) ⚂ Using the Horner scheme, write an efficient C++ implementation of a function

```
template <typename CoeffVec>
std::pair<double,double> evaldp ( const CoeffVec & c,
    double x )
```

which returns the pair $(p(x), p'(x))$, where p is the polynomial with coefficients in `c`. The vector `c` contains the coefficient of the polynomial in the monomial basis, using Matlab convention (leading coefficient in `c[0]`).

(3b) ⚀ For the sake of testing, write a naive C++ implementation of the above function

```
template <typename CoeffVec>
std::pair<double,double> evaldp_naive ( const CoeffVec &
    c, double x )
```

which returns the same pair $(p(x), p'(x))$. This time, $p(x)$ and $p'(x)$ should be calculated with the simple sums of the monomials constituting the polynomial.

(3c) ⚁ What are the asymptotic complexities of the two implementations?

(3d) ⚁ Check the validity of the two functions and compare the runtimes for polynomials of degree up to $2^{20} - 1$.

Problem 4 Lagrange interpolant

Given data points $(t_i, y_i)_{i=1}^n$, show that the Lagrange interpolant

$$p(x) = \sum_{i=0}^{n} y_i L_i(x), \quad L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - t_j}{t_i - t_j}$$

is given by:

$$p(x) = \omega(x) \sum_{j=0}^{n} \frac{y_j}{(x - t_j)\omega'(t_j)}$$

with $\omega(x) = \prod_{j=0}^{n}(x - t_j)$.

Issue date: 22.10.2015

Hand-in: 29.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

ETH Zürich
D-MATH

Numerical Methods for CSE

Problem Sheet 7

Problem 1 Cubic Splines (core problem)

Since they mimic the behavior of an elastic rod pinned at fixed points, see [1, § 3.5.13], cubic splines are very popular for creating "aesthetically pleasing" interpolating functions. However, in this problem we look at a cubic spline from the perspective of its defining properties, see [1, Def. 3.5.1], in order to become more familiar with the concept of spline function and the consequences of the smoothness required by the definition.

For parameters $\alpha, \beta \in \mathbb{R}$ we define the function $s_{\alpha,\beta} : [-1,2] \to \mathbb{R}$ by

$$s_{\alpha,\beta}(x) = \begin{cases} (x+1)^4 + \alpha(x-1)^4 + 1 & x \in [-1,0] \\ -x^3 - 8\alpha x + 1 & x \in (0,1] \\ \beta x^3 + 8x^2 + \frac{11}{3} & x \in (1,2] \end{cases} \tag{12}$$

(1a) ⚀ Determine α, β such that $s_{\alpha,\beta}$ is a cubic spline in $\mathcal{S}_{3,M}$ with respect to the node set $M = \{-1,0,1,2\}$. Verify that you actually obtain a cubic spline.

(1b) ⚁ Use MATLAB to create a plot of the function defined in (12) in dependance of α and β.

Problem 2 Quadratic Splines (core problem)

[1, Def. 3.5.1] introduces spline spaces $\mathcal{S}_{d,\mathcal{M}}$ of any degree $d \in \mathbb{N}_0$ on a node set $\mathcal{M} \subset \mathbb{R}$. [1, Section 3.5.1] discusses interpolation by means of cubic splines, which is the most important case. In this problem we practise spline interpolation for quadratic splines in order to understand the general principles.

Consider a 1-periodic function $f : \mathbb{R} \to \mathbb{R}$, that is, $f(t+1) = f(t)$ for all $t \in \mathbb{R}$, and a set of nodes

$$\mathcal{M} := \{0 = t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = 1\} \subset [0,1] .$$

We want to approximate f using a *1-periodic* quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$, which interpolates f *in the midpoints* of the intervals $[t_{j-1}, t_j]$, $j = 0, \ldots, n$.

In analogy to the local representation of a cubic spline function according to [1, Eq. (3.5.5)], we parametrize a quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$ according to

$$s|_{[t_{j-1},t_j]}(t) = d_j\,\tau^2 + c_j\,4\,\tau(1-\tau) + d_{j-1}\,(1-\tau)^2\,, \quad \tau := \frac{t - t_{j-1}}{t_j - t_{j-1}}\,, \quad j = 1, \ldots, n\,, \tag{13}$$

with $c_j, d_k \in \mathbb{R}$, $j = 1, \ldots, n,\ k = 0, \ldots, n$. Notice that the coefficients d_k are associated with the nodes t_k while the c_j are associated with the midpoints of the intervals $[t_{j-1}, t_j]$.

(2a) ⊡ What is the dimension of the subspace of *1-periodic* spline functions in $\mathcal{S}_{2,\mathcal{M}}$?

(2b) ⊡ What kind of continuity is already guaranteed by the use of the representation (13)?

(2c) ⚂ Derive a linear system of equations (system matrix and right hand side) whose solution provides the coefficients c_j and d_j in (13) from the function values $y_j := f(\frac{1}{2}(t_{j-1} + t_j))$, $j = 1, \ldots, n$.

HINT: By [1, Def. 3.5.1] we know $\mathcal{S}_{2,\mathcal{M}} \subset C^1([0,1])$, which provides linear constraints at the nodes, analogous to [1, Eq. (3.5.6)] for cubic splines.

(2d) ⚂ Implement an *efficient* MATLAB routine

```
function s=quadspline(t,y,x)
```

which takes as input a (sorted) node vector `t` (of length $n-1$, because $t_0 = 0$ and $t_n = 1$ will be taken for granted), a n-vector `y` containing the values of a function f at the midpoints $\frac{1}{2}(t_{j-1} + t_j)$, $j = 1, \ldots, n$, and a *sorted* N-vector `x` of evaluation points in $[0,1]$.

The function is to return the values of the interpolating quadratic spline s at the positions `x`.

You can test your code with the one provided by `quadspline_p.p` (available on the lecture website).

(2e) ⊡ Plot f and the interpolating periodic quadratic spline s for $f(t) := \exp(\sin(2\pi t))$, $n = 10$ and $\mathcal{M} = \left\{\frac{j}{n}\right\}_{j=0}^{10}$, that is, the spline is to fulfill $s(t) = f(t)$ for all midpoints t of knot intervals.

(2f) ⚀ What is the complexity of the algorithm in (2d) in dependance of n and N?

Problem 3 Curve Interpolation (core problem)

The focus of [1, Chapter 3] was on the interpolation of data points by means of functions belonging to a certain linear space. A different task is to find a curve containing each point of a set $\{\mathbf{p}_0, ..., \mathbf{p}_n\} \subset \mathbb{R}^2$.

This task can be translated in a standard interpolation problem after recalling that a curve in $\mathbb{R}^2$ can be described by a mapping (*parametrization*) $\boldsymbol{\gamma} : [0,1] \mapsto \mathbb{R}^2$, $\boldsymbol{\gamma}(t) = \binom{s_1(t)}{s_2(t)}$. Hence, given the nodes $0 = t_0 < t_1 < ... < t_{n-1} < t_n = 1$, we aim at finding interpolating functions $s_1, s_2 : [0,1] \to \mathbb{R}$, such that $s_i(t_j) = (\mathbf{p}_j)_i$, $i = 1, 2$, $j = 0, ..., n$. This means that we separately interpolate the x and y coordinates of $\mathbf{p}_j$.

A crucial new aspect is that the nodes are not fixed, i.e., there are infinitely many parameterizations for a given curve: for any strictly monotonous and surjective $h : [0,1] \to [0,1]$ the mappings $\boldsymbol{\gamma}$ and $\widetilde{\boldsymbol{\gamma}} := \boldsymbol{\gamma} \circ h$ describe exactly the same curve. On the other hand, the selection of nodes will affect the interpolants s_1 and s_2 and leads to different interpolating curves.

Concerning the choice of the nodes, we will consider two options:

❶ equidistant parametrization: $t_k = k\Delta t$, $\Delta t = \dfrac{1}{n}$ (14)

❷ segment length parametrization: $t_k = \dfrac{\sum_{l=1}^{k} |\mathbf{p}_l - \mathbf{p}_{l-1}|}{\sum_{l=1}^{n} |\mathbf{p}_l - \mathbf{p}_{l-1}|}$. (15)

Point data will be generated by the MATLAB function `heart` that is available on the course webpage.

(3a) ⚂ Write a MATLAB function

```
function  pol = polycurveintp (xy,t,tt)
```

which uses global polynomial interpolation (using the `intpolyval` function, see [1, Code 3.2.28]) through the $n+1$ points $\mathbf{p}_i \in \mathbb{R}^2$, $i = 0, \ldots, n$, whose coordinates are stored in the $2 \times (n+1)$ matrix `xy` and returns sampled values of the obtained curve in a $2 \times N$ matrix `pol`. Here, `t` passes the node vector $(t_0, t_1, \ldots, t_n) \in \mathbb{R}^{n+1}$ in the parameter domain and N is the number of equidistant sampling points.

HINT: Code for `intpolyval` is available as `intpolyval.m`.

(3b) ⚀ Plot the curves obtained by global polynomial interpolation `polycurveintp` of the `heart` data set. The nodes for polynomial interpolation should be generated according to the two options (14) and (15)

(3c) ⚁ Extend your MATLAB function `pol = curveintp` to

```
function   pch = pchcurveintp (xy,t,tt),
```

which has the same purpose, arguments and return values as `polycurveintp`, but now uses monotonicity preserving cubic Hermite interpolation (available through the MATLAB built-in function `pchip`, see also [1, Section 3.4.2]) instead of global polynomial interpolation.

Plot the obtained curves for the `heart` data set in the figure created in sub-problem (3b). Use both parameterizations (14) and (15).

(3d) ⚁ Finally, write yet another MATLAB function

```
function   spl = splinecurveintp (xy,t,tt),
```

which has the same purpose, arguments and return values as `polycurveintp`, but now uses *complete* cubic spline interpolation.

The required derivatives $s_1'(0)$, $s_2'(0)$, $s_1'(1)$, and $s_2'(1)$ should be computed from the directions of the line segments connecting $\mathbf{p}_0$ and $\mathbf{p}_1$, and $\mathbf{p}_{n-1}$ and $\mathbf{p}_n$, respectively. You can use the MATLAB built-in function `spline`. Plot the obtained curves (`heart` data) in the same figure as before using both parameterizations (14) and (15).

HINT: read the MATLAB help page about the `spline` command and learn how to impose the derivatives at the endpoints.

Problem 4 Approximation of π

In [1, Section 3.2.3.3] we learned about the use of polynomial extrapolation (= interpolation outside the interval covered by the nodes) to compute inaccessible limits $\lim_{h\to 0} \Psi(h)$. In this problem we apply extrapolation to obtain the limit of a sequence $x^{(n)}$ for $n \to \infty$.

We consider a quantity of interest that is defined as a limit

$$x^* = \lim_{n\to\infty} T(n) \,, \tag{16}$$

with a function $T : \{n, n+1, \ldots\} \mapsto \mathbb{R}$. However, computing $T(n)$ for very large arguments k may not yield reliable results.

The idea of *extrapolation* is, firstly, to compute a few values $T(n_0), T(n_1), \ldots, T(n_k)$, $k \in \mathbb{N}$, and to consider them as the values $g(1/n_0), g(1/n_1), \ldots, g(1/n_k)$ of a continuous function $g :]0, 1/n_{\min}] \mapsto \mathbb{R}$, for which, obviously

$$x^* = \lim_{h \to 0} g(h) \; . \tag{17}$$

Thus we recover the usual setting for the application of polynomial extrapolation techniques. Secondly, according to the idea of extrapolation to zero, the function g is approximated by an interpolating polynomial $p \in \mathcal{P}_{k-1}$ with $p_{k-1}(n_j^{-1}) = T(n_j)$, $j = 1, \ldots, k$. In many cases we can expect that $p_{k-1}(0)$ will provide a good approximation for x^*. In this problem we study the algorithmic realization of this extrapolation idea for a simple example.

The unit circle can be approximated by inscribed regular polygons with n edges. The length of half of the circumference of such an n-edged polygon can be calculated by elementary geometry:

n	2	3	4	5	6	8	10
$T(n) := \frac{U_n}{2}$	2	$\frac{3}{2}\sqrt{3}$	$2\sqrt{2}$	$\frac{5}{4}\sqrt{10-2\sqrt{5}}$	3	$4\sqrt{2-\sqrt{2}}$	$\frac{5}{2}\left(\sqrt{5}-1\right)$

Write a C++ function

```
double pi_approx(int k);
```

that uses the *Aitken-Neville scheme*, see [1, Code 3.2.31], to approximate π by extrapolation from the data in the above table, using the first k values, $k = 1, \ldots, 7$.

Issue date: 29.10.2015

Hand-in: 05.11.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 8

Problem 1 Natural cubic Splines (core problem)

In [1, Section 3.5.1] we learned about cubic spline interpolation and its variants, the complete, periodic, and natural cubic spline interpolation schemes.

(1a) ⚀ Given a knot set $\mathcal{T} = \{t_0 < t_1 < \cdots < t_n\}$, which also serves as the set of interpolation nodes, and values y_j, $j = 0, \ldots, n$, write down the linear system of equations that yields the slopes $s'(t_j)$ of the natural cubic spline interpolant s of the data points (t_j, y_j) at the knots.

(1b) ⚁ Argue why the linear system found in subsubsection (1a) has a unique solution.

HINT: Look up [1, Lemma 1.8.12] and apply its assertion.

(1c) ⚁ Based on EIGEN devise an *efficient* implementation of a C++ class for the computation of a natural cubic spline interpolant with the following definition:

```
1 class NatCSI {
2 public:
3     //! \brief Build the cubic spline interpolant with
        natural boundaries
4     //! Setup the data structures you need.
5     //! Pre-compute the coefficients of the spline
        (solve system)
6     //! \param[in] t, nodes of the grid (for pairs (t_i,
        y_i)) (sorted!)
7     //! \param[in] y, values y_i at t_i (for pairs (t_i,
        y_i))
```

```
8       NatCSI(const const std::vector<double> & t, const
            const std::vector<double> & y);
9
10      //! \brief Interpolant evaluation at x
11      //! \param[in] x, value x where to evaluate the
            spline
12      //! \return value of the spline at x
13      double operator() (double x) const;
14
15  private:
16      // TODO: store data for the spline
17  };
```

HINT: Assume that the input array of knots is sorted and perform binary searches for the evaluation of the interpolant.

Problem 2 Monotonicity preserving interpolation (core problem)

This problem is about monotonicity preserving interpolation. Before starting, you should revise [1, Def. 3.1.15], [1, § 3.3.2] and [1, Section 3.4.2] carefully.

(2a) ⚄ Prove [1, Thm. 3.4.17]:

> If, for fixed node set $\{t_j\}_{j=0}^n$, $n \geq 2$, an interpolation scheme $\mathsf{I} : \mathbb{R}^{n+1} \to C^1(I)$ is *linear* as a mapping from data values to continuous functions on the interval covered by the nodes (→ [1, Def. 3.1.15]), and *monotonicity preserving*, then $\mathsf{I}(\mathbf{y})'(t_j) = 0$ for all $\mathbf{y} \in \mathbb{R}^{n+1}$ and $j = 1, \ldots, n-1$.

HINT: Consider a suitable basis $\{\mathbf{s}^{(j)} : j = 0, \ldots, n\}$ of $\mathbb{R}^{n+1}$ that consists of monotonic vectors, namely such that $s_i^{(j)} \leq s_{i+1}^{(j)}$ for every $i = 0, \ldots, n-1$.

HINT: Exploit the phenomenon explained next to [1, Fig. 99].

Problem 3 Local error estimate for cubic Hermite interpolation (core problem)

Consider the cubic Hermite interpolation operator $\mathcal{H}$ of a function defined on an interval $[a, b]$ to the space $\mathcal{P}_3$ polynomials of degree at most 3:

$$\mathcal{H} : C^1([a, b]) \to \mathcal{P}_3$$

defined by:

- $(\mathcal{H}f)(a) = f(a)$;
- $(\mathcal{H}f)(b) = f(b)$;
- $(\mathcal{H}f)'(a) = f'(a)$;
- $(\mathcal{H}f)'(b) = f'(b)$.

Assume $f \in C^4([a,b])$. Show that for every $x \in]a,b[$ there exists $\tau \in [a,b]$ such that

$$(f - \mathcal{H}f)(x) = \frac{1}{24} f^{(4)}(\tau)(x-a)^2(x-b)^2. \tag{18}$$

HINT: Fix $x \in]a,b[$. Use an auxiliary function:

$$\varphi(t) := f(t) - (\mathcal{H}f)(t) - C(t-a)^2(t-b)^2. \tag{19}$$

Find C s.t. $\varphi(x) = 0$.

HINT: Use Rolle's theorem (together with the previous hint and the definition of $\mathcal{H}$) to find a lower bound for the number of zeros of $\varphi^{(k)}$ for $k = 1,2,3,4$.

HINT: Use the fact that $(\mathcal{H}f) \in \mathcal{P}_3$ and for $p(t) := (t-a)^2(t-b)^2, p \in \mathcal{P}_4$.

HINT: Conclude showing that $\exists \tau \in]a,b[, \varphi^{(4)}(\tau) = 0$. Use the definition of φ to find an expression for C.

Problem 4 Adaptive polynomial interpolation

In [1, Section 4.1.3] we have seen that the placement of interpolation nodes is key to a good approximation by a polynomial interpolant. The following *greedy algorithm* attempts to find the location of suitable nodes by its own:

> Given a function $f : [a,b] \mapsto \mathbb{R}$ one starts $\mathcal{T} := \{\frac{1}{2}(b+a)\}$. Based on a fixed finite set $\mathcal{S} \subset [a,b]$ of *sampling points* one augments the set of nodes according to
>
> $$\mathcal{T} = \mathcal{T} \cup \left\{ \operatorname*{argmax}_{t \in \mathcal{S}} |f(t) - \mathsf{I}_{\mathcal{T}}(t)| \right\}, \tag{20}$$
>
> where $\mathsf{I}_{\mathcal{T}}$ is the polynomial interpolation operator for the node set $\mathcal{T}$, until
>
> $$\max_{t \in \mathcal{S}} |f(t) - \mathsf{I}_{\mathcal{T}}(t)| \leq \mathtt{tol} \cdot \max_{t \in \mathcal{S}} |f(t)|. \tag{21}$$

(4a) ⚂ Write a MATLAB function

```
function t = adaptivepolyintp(f,a,b,tol,N)
```

that implements the algorithm described above and takes as arguments the function handle `f`, the interval bounds `a`, `b`, the relative tolerance `tol`, and the number `N` of *equidistant* sampling points (in the interval $[a, b]$), that is,

$$\mathcal{S} := \{a + (b-a)\frac{j}{N},\ j = 0, \ldots, N\} .$$

HINT: The function `intpolyval` from [1, Code 3.2.28] is provided and may be used (though it may not be the most efficient way to implement the function).

(4b) ⚁ Extend the function from the previous sub-problem so that it reports the quantity

$$\max_{t\in\mathcal{S}} |f(t) - \mathsf{T}_{\mathcal{T}}(t)| \tag{22}$$

for each intermediate set $\mathcal{T}$.

(4c) ⚃ For $f_1(t) := \sin(e^{2t})$ and $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$ plot the quantity from (22) versus the number of interpolation nodes. Choose plotting styles that reveal the qualitative decay of this error as the number of interpolation nodes is increased. Use interval $[a, b] = [0, 1]$, `N=1000` sampling points, tolerance `tol = 1e-6`.

Issue date: 05.11.2015

Hand-in: 12.11.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 9

Problem 1 Chebychev interpolation of analytic functions (core problem)

This problem concerns Chebychev interpolation (cf. [1, Section 4.1.3]). Using techniques from complex analysis, notably the residue theorem [1, Thm. 4.1.56], in class we derived an expression for the interpolation error [1, Eq. (4.1.62)] and from it an error bound [1, Eq. (4.1.63)], as much sharper alternative to [1, Thm. 4.1.37] and [1, Lemma 4.1.41] for *analytic* interpolands. The bound tells us that for all $t \in [a,b]$

$$|f(t) - \mathsf{L}_\mathcal{T} f(t)| \leq \left| \frac{w(x)}{2\pi i} \int_\gamma \frac{f(z)}{(z-t)w(z)} dz \right| \leq \frac{|\gamma|}{2\pi} \frac{\max_{a\leq\tau\leq b}|w(\tau)|}{\min_{z\in\gamma}|w(z)|} \frac{\max_{z\in\gamma}|f(z)|}{d([a,b],\gamma)} ,$$

where $d([a,b],\gamma)$ is the geometric distance of the integration contour $\gamma \subset \mathbb{C}$ from the interval $[a,b] \subset \mathbb{C}$ in the complex plane. The contour γ must be contractible in the domain D of analyticity of f and must wind around $[a,b]$ exactly once, see [1, Fig. 124].

Now we consider the interval $[-1,1]$. Following [1, Rem. 4.1.87], our task is to find an upper bound for this expression, in the case where f possesses an analytical extension to a complex neighbourhood of $[-1,1]$.

For the analysis of the Chebychev interpolation of analytic functions we used the elliptical contours, see [1, Fig. 138],

$$\gamma_\rho(\theta) := \cos(\theta - i\log(\rho)) , \quad \forall 0 \leq \theta \leq 2\pi , \quad \rho > 1 . \tag{23}$$

(1a) ⚀ Find an upper bound for the length $|\gamma_\rho|$ of the contour γ_ρ.

HINT: You may use the arc-length formula for a curve $\gamma : I \to \mathbb{R}^2$:

$$|\gamma| = \int_I \|\dot\gamma(\tau)\| d\tau, \tag{24}$$

where $\dot{\gamma}$ is the derivative of γ w.r.t the parameter τ. Recall that the "length" of a complex number z viewed as a vector in $\mathbb{R}^2$ is just its modulus.

Now consider the S-curve function (the logistic function):

$$f(t) := \frac{1}{1+e^{-3t}}\,, \quad t \in \mathbb{R}\,.$$

(1b) ⚀ Determine the maximal domain of analyticity of the extension of f to the complex plane $\mathbb{C}$.

HINT: Consult [1, Rem. 4.1.64].

(1c) ⚁ Write a MATLAB function that computes an approximation M of:

$$\min_{\rho>1} \frac{\max_{z\in\gamma_\rho}|f(z)|}{d([-1,1],\gamma_\rho)}, \tag{25}$$

by sampling, where the distance of $[a,b]$ from γ_ρ is formally defined as

$$d([a,b],\gamma) := \inf\{|z-t| \mid z \in \gamma, t \in [a,b]\}. \tag{26}$$

HINT: The result of (1b), together with the knowledge that γ_ρ describes an ellipsis, tells you the maximal range $(1,\rho_{max})$ of ρ. Sample this interval with 1000 equidistant steps.

HINT: Apply geometric reasoning to establish that the distance of γ_ρ and $[-1,1]$ is $\frac{1}{2}(\rho+\rho^{-1})-1$.

HINT: If you cannot find ρ_{max} use $\rho_{max} = 2.4$.

HINT: You can exploit the properties of $\cos$ and the hyperbolic trigonometric functions $\cosh$ and $\sinh$.

(1d) ⚂ Based on the result of (1c), and [1, Eq. (4.1.89)], give an "optimal" bound for

$$\|f - L_n f\|_{L^\infty([-1,1])},$$

where L_n is the operator of Chebychev interpolation on $[-1,1]$ into the space of polynomials of degree $\leq n$.

(1e) ⚄ Graphically compare your result from (1d) with the measured supremum norm of the approximation error of Chebychev interpolation of f on $[-1,1]$ for polynomial degree $n = 1,\ldots,20$. To that end, write a MATLAB-code and rely on the provided function `intpolyval` (cf. [1, Code 4.4.12]).

HINT: Use semi-logarithmic scale for your plot `semilogy`.

(1f) ⚄ Rely on pullback to $[-1,1]$ to discuss how the error bounds in [1, Eq. (4.1.89)] will change when we consider Chebychev interpolation on $[-a,a], a > 0$, instead of $[-1,1]$, whilst keeping the function f fixed.

Problem 2 Piecewise linear approximation on graded meshes (core problem)

One of the messages given by [1, Section 4.1.3] is that the quality of an interpolant depends heavily on the choice of the interpolation nodes. If the function to be interpolated has a "bad behavior" in a small part of the domain, for instance it has very large derivatives of high order, more interpolation points are required in that area of the domain. Commonly used tools to cope with this task, are *graded meshes*, which will be the topic of this problem.

Given a mesh $\mathcal{T} = \{0 \le t_0 < t_1 < \cdots < t_n \le 1\}$ on the unit interval $I = [0,1]$, $n \in \mathbb{N}$, we define the *piecewise linear* interpolant

$$\mathsf{I}_{\mathcal{T}} : C^0(I) \to \mathcal{P}_{1,\mathcal{T}} = \{s \in C^0(I),\ s_{|[t_{j-1},t_j]} \in \mathcal{P}_1\ \forall\ j\}, \quad \text{s.t.} \quad \big(\mathsf{I}_{\mathcal{T}} f\big)(t_j) = f(t_j), \quad j = 0,\ldots,n;$$

(see also [1, Section 3.3.2]).

(2a) ⚂ If we choose the uniform mesh $\mathcal{T} = \{t_j\}_{j=0}^n$ with $t_j = j/n$, given a function $f \in C^2(I)$, what is the asymptotic behavior of the error

$$\|f - \mathsf{I}_{\mathcal{T}} f\|_{L^\infty(I)},$$

when $n \to \infty$?

HINT: Look for a suitable estimate in [1, Section 4.5.1].

(2b) ⚂ What is the regularity of the function

$$f : I \to \mathbb{R}, \qquad f(t) = t^\alpha, \qquad 0 < \alpha < 2\ ?$$

In other words, for which $k \in \mathbb{N}$ do we have $f \in C^k(I)$?

HINT: Notice that I is a closed interval and check the continuity of the derivatives in the endpoints of I.

(2c) ⚃ Study numerically the h-convergence of the piecewise linear approximation of $f(t) = t^\alpha$ $(0 < \alpha < 2)$ on uniform meshes; determine the order of convergence using linear regression based on MATLAB's `polyfit`, see [1, Section 4.5.1].

HINT: Linear regression and `polyfit` have not been introduced yet. Please give a quick look at the examples in http://ch.mathworks.com/help/matlab/ref/polyfit.html#examples to see `polyfit` in action. For instance, the code to determine the slope of a line approximating a sequence of points $(x_i, y_i)_i$ in doubly logarithmic scale is

```
P = polyfit(log(x),log(y),1);
slope = P(1);
```

(2d) ⚄ In which mesh interval do you expect $|f - I_{\mathcal{T}} f|$ to attain its maximum?

HINT: You may use the code from the previous subtask to get an idea.

HINT: What is the meaning of [1, Thm. 4.1.37] in the piecewise linear setting?

(2e) ⚄ Compute by hand the exact value of $\|f - I_{\mathcal{T}} f\|_{L^\infty(I)}$.

Compare the order of convergence obtained with the one observed numerically in (2b).

HINT: Use the result of (2d) to simplify the problem.

(2f) ⚄ Since the interpolation error is concentrated in the left part of the domain, it seems reasonable to use a finer mesh only in this part. A common choice is an **algebraically graded mesh**, defined as

$$\mathcal{G} = \left\{ t_j = \left(\frac{j}{n}\right)^\beta, \quad j = 0, \dots, n \right\},$$

for a parameter $\beta > 1$. An example is depicted in Figure 5 for $\beta = 2$.

For a fixed parameter α in the definition of f, numerically determine the rate of convergence of the piecewise linear interpolant $I_{\mathcal{G}}$ on the graded mesh $\mathcal{G}$ as a function of the parameter β. Try for instance $\alpha = 1/2$, $\alpha = 3/4$ or $\alpha = 4/3$.

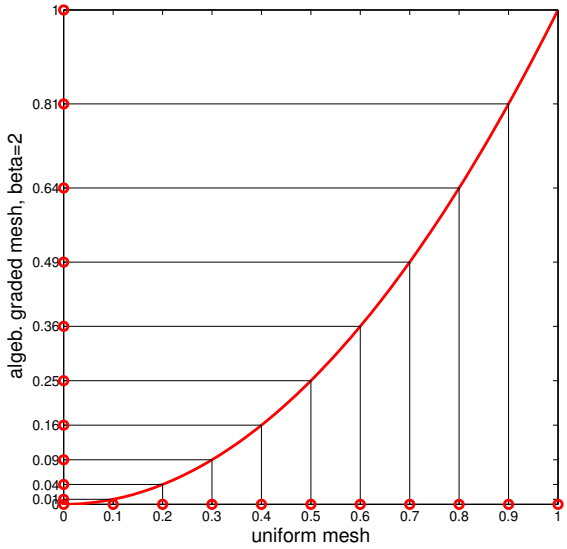
How do you have to choose β in order to recover the optimal rate $\mathcal{O}(n^{-2})$ (if possible)?

Problem 3 Chebyshev polynomials and their properties

Let $T_n \in \mathcal{P}_n$ be the n-th Chebyshev polynomial, as defined in [1, Def. 4.1.67] and $\xi_0^{(n)}, \dots, \xi_{n-1}^{(n)}$ be the n zeros of T_n. According to [1, Eq. (4.1.75)], these are given by

$$\xi_j^{(n)} = \cos\left(\frac{2j+1}{2n}\pi\right), \quad j = 0, \dots, n-1. \tag{27}$$

Figure 5: Graded mesh $x_j = (j/n)^2$, $j = 0, \ldots, 10$.



We define the family of discrete L^2 semi inner products, *cf.* [1, Eq. (4.2.21)],

$$(f,g)_n := \sum_{j=0}^{n-1} f(\xi_j^{(n)}) g(\xi_j^{(n)}), \quad f,g \in C^0([-1,1]) \tag{28}$$

and the special weighted L^2 inner product

$$(f,g)_w := \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} f(t)g(t)\,dt \quad f,g \in C^0([-1,1]) \tag{29}$$

(3a) ⚀ Show that the Chebyshev polynomials are an orthogonal family of polynomials with respect to the inner product defined in (29) according to [1, Def. 4.2.24], namely $(T_k, T_l)_w = 0$ for every $k \neq l$.

HINT: Recall the trigonometric identity $2\cos(x)\cos(y) = \cos(x+y) + \cos(x-y)$.

Consider the following statement.

Theorem. The family of polynomials $\{T_0, \ldots, T_n\}$ is an orthogonal basis (→ [1, Def. 4.2.13]) of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$ defined in (28).

(3b) ⚀ Write a C++ code to test the assertion of the theorem.

HINT: [1, Code 4.1.70] demonstrates the efficient evaluation of Chebychev polynomials based on their 3-term recurrence formula from [1, Thm. 4.1.68].

(3c) ⚁ Prove the theorem.

HINT: Use the relationship of trigonometric functions and the complex exponential together with the summation formula for geometric sums.

(3d) ⚀ Given a function $f \in C^0([-1,1]$, find an expression for the best approximant $q_n \in \mathcal{P}_n$ of f in the discrete L^2-norm:

$$q_n = \operatorname*{argmin}_{p \in \mathcal{P}_n} \|f - p\|_{n+1} ,$$

where $\| \|_{n+1}$ is the norm induced by the scalar product $(\, , \,)_{n+1}$. You should express q_n through an expansion in Chebychev polynomials of the form

$$q_n = \sum_{j=0}^{n} \alpha_j T_j \tag{30}$$

for suitable coefficients $\alpha_j \in \mathbb{R}$.

HINT: The task boils down to determining the coefficients α_j. Use the theorem you have just proven and a slight extension of [1, Cor. 4.2.14].

(3e) ⚀ Write a C++ function

```
template <typename Function>
void bestpolchebnodes(const Function &f, Eigen::VectorXd
    &alpha)
```

that returns the vector of coefficients $(\alpha_j)_j$ in (30) given a function f. Note that the degree of the polynomial is indirectly passed with the length of the output `alpha`. The input `f` is a lambda-function, e.g.

```
auto f = [] (double & x) {return 1/(pow(5*x,2)+1);};
```

(3f) ⚀ Test `bestpolchebnodes` with the function $f(x) = \frac{1}{(5x)^2+1}$ and $n = 20$. Approximate the supremum norm of the approximation error by sampling on an equidistant grid with 10^6 points.

HINT: Again, [1, Code 4.1.70] is useful for evaluating Chebychev polynomials.

(3g) ⚀ Let L_j, $j = 0, \ldots, n$, be the Lagrange polynomials associated with the nodes $t_j = \xi_j^{(n+1)}$ of Chebyshev interpolation with $n+1$ nodes on $[-1,1]$, see [1, Eq. (4.1.75)]. Show that

$$L_j = \frac{1}{n+1} + \frac{2}{n+1} \sum_{l=1}^{n} T_l(\xi_j^{(n+1)}) T_l.$$

HINT: Again use the above theorem to express the coefficients of a Chebychev expansion of L_j.

Problem 4 Piecewise cubic Hermite interpolation

Piecewise cubic Hermite interpolation with exact slopes on a mesh

$$\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_n = b\}$$

was defined in [1, Section 3.4]. For $f \in C^4([a,b])$ it enjoys h-convergence with rate 4 as we have seen in [1, Exp. 4.5.15].

Now we consider cases, where perturbed or reconstructed slopes are used. For instance, this was done in the context of monotonicity preserving piecewise cubic Hermite interpolation as discussed in [1, Section 3.4.2].

(4a) ⚀ Assume that piecewise cubic Hermite interpolation is based on perturbed slopes, that is, the piecewise cubic function s on $\mathcal{M}$ satisfies:

$$s(x_j) = f(x_j) \quad , \quad s'(x_j) = f'(x_j) + \delta_j,$$

where the δ_j may depends on $\mathcal{M}$, too.

Which rate of asymptotic h-convergence of the $\sup$-norm of the approximation error can be expected, if we know that for all j

$$|\delta_j| = O(h^\beta) \ , \quad \beta \in \mathbb{N}_0 \ ,$$

for mesh-width $h \to 0$.

HINT: Use a local generalized cardinal basis functions, cf. [1, § 3.4.3].

(4b) ⚂ Implement a strange piecewise cubic interpolation scheme in C++ that satisfies:

$$s(x_j) = f(x_j) \quad , \quad s'(x_j) = 0$$

and empirically determine its convergence on a sequence of equidistant meshes of $[-5, 5]$ with mesh-widths $h = 2^{-l}, l = 0, \ldots, 8$ and for the interpoland $f(t) := \frac{1}{1+t^2}$.

As a possibly useful guideline, you can use the provided C++ template, see the file `piecewise_hermite_interpolation_template.cpp`.

Compare with the insight gained in (4a).

(4c) ⚀ Assume equidistant meshes and reconstruction of slopes by a particular averaging. More precisely, the $\mathcal{M}$-piecewise cubic function s is to satisfy the generalized interpolation conditions

$$s(x_j) = f(x_j),$$
$$s'(x_j) = \begin{cases} \frac{-f(x_2)+4f(x_1)-3f(x_0)}{2h} & \text{for } j = 0\,, \\ \frac{f(x_{j+1})-f(x_{j-1})}{2h} & \text{for } j = 1, \ldots, n-1\,, \\ \frac{3f(x_n)-4f(x_{n-1})+f(x_{n-2})}{2h} & \text{for } j = n\,. \end{cases}$$

What will be the rate of h-convergence of this scheme (in $\sup$-norm)?

(You can solve this exercise either theoretically or determine an empiric convergence rate in a numerical experiment.)

HINT: If you opt for the theoretical approach, you can use what you have found in subsubsection (4a). To find perturbation bounds, rely on the Taylor expansion formula with remainder, see [1, Ex. 1.5.60].

Issue date: 12.11.2015

Hand-in: 19.11.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 10

Problem 1 Zeros of orthogonal polynomials (core problem)

This problem combines elementary methods for zero finding with 3-term recursions satisfied by orthogonal polynomials.

The zeros of the Legendre polynomial P_n (see [1, Def. 5.3.26]) are the n Gauss points ξ_j^n, $j = 1, \ldots, n$. In this problem we compute the Gauss points by zero finding methods applied to P_n. The 3-term recursion [1, Eq. (5.3.32)] for Legendre polynomials will play an essential role. Moreover, recall that, by definition, the Legendre polynomials are $L^2(]-1,1[)$-orthogonal.

(1a) ⚄ Prove the following interleaving property of the zeros of the Legendre polynomials. For all $n \in \mathbb{N}_0$ we have

$$-1 < \xi_j^n < \xi_j^{n-1} < \xi_{j+1}^n < 1, \qquad j = 1, \ldots, n-1.$$

HINT: You may follow these steps:

1. Understand that it is enough to show that every pair of zeros (ξ_l^n, ξ_{l+1}^n) of P_n is separated by a zero of P_{n-1}.
2. Argue by contradiction.
3. By considering the auxiliary polynomial $\prod_{j \neq l, l+1}(t - \xi_j^n)$ and the fact that the Gauss quadrature is exact on $\mathcal{P}_{2n-1}$ prove that $P_{n-1}(\xi_l^n) = P_{n-1}(\xi_{l+1}^n) = 0$.
4. Choose $s \in \mathcal{P}_{n-2}$ such that $s(\xi_j^n) = P_{n-1}(\xi_j^n)$ for every $j \neq l, l+1$, and using again that Gauss quadrature is exact on $\mathcal{P}_{2n-1}$ obtain a contradiction.

(1b) ⚀ By differentiating [1, Eq. (5.3.32)] derive a combined 3-term recursion for the sequences $(P_n)_n$ and $(P_n')_n$.

(1c) ⚀ Use the recursions obtained in (1b) to write a C++ function

```
void legvals(const Eigen::VectorXd &x, Eigen::MatrixXd
    &Lx, Eigen::MatrixXd &DLx)
```

that fills the matrices `Lx` and `DLx` in $\mathbb{R}^{N\times(n+1)}$ with the values $(P_k(x_j))_{jk}$ and $(P'_k(x_j))_{jk}$, $k = 0,\ldots,n$, $j = 0,\ldots,N-1$, for an input vector $x \in \mathbb{R}^N$ (passed in `x`).

(1d) ⚁ We can compute the zeros of P_k, $k = 1,\ldots,n$, by means of the secant rule (see [1, § 2.3.22]) using the endpoints $\{-1,1\}$ of the interval and the zeros of the previous Legendre polynomial as initial guesses, see (1a). We opt for a correction based termination criterion (see [1, Section 2.1.2]) based on prescribed relative and absolute tolerance (see [1, Code 2.3.25]).

Write a C++ function

```
MatrixXd gaussPts(int n, double rtol=1e-10, double
    atol=1e-12)
```

that computes the Gauss points $\xi_j^k \in [-1,1]$, $j = 1,\ldots,k$, $k = 1,\ldots,n$, using the zero finding approach outlined above. The Gauss points should be returned in an upper triangular $n \times n$-matrix.

HINT: For simplicity, you may want to write a C++ function

```
double Pkx(double x, int k)
```

that computes $P_k(x)$ for a scalar x. Reuse parts of the function `legvals`.

(1e) ⚀ Validate your implementation of the function `gaussPts` with $n = 8$ by computing the values of the Legendre polynomials in the zeros obtained (use the function `legvals`). Explain the failure of the method.

HINT: See Figure 6.

(1f) ⚁ Fix your function `gaussPts` taking into account the above considerations. You should use the *regula falsi*, that is a variant of the secant method in which, at each step, we choose the old iterate to keep depending on the signs of the function. More precisely, given two approximations $x^{(k)}$, $x^{(k-1)}$ of a zero in which the function f has different signs,

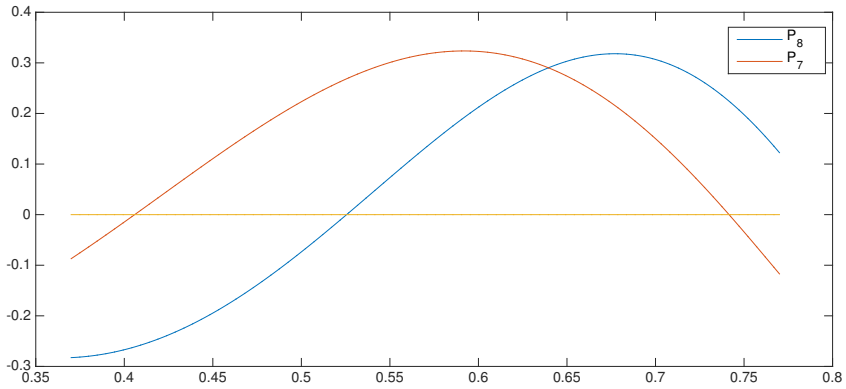


Figure 6: P_7 and P_8 on a part of $[-1,1]$. The secant method fails to find the zeros of P_8 (blue curve) when started with the zeros of P_7 (red curve).

compute another approximation $x^{(k+1)}$ as zero of the secant. Use this as the next iterate, but then chose as $x^{(k)}$ the value $z \in \{x^{(k)}, x^{(k-1)}\}$ for which $\operatorname{sign} f(x^{(k+1)}) \neq \operatorname{sign} f(z)$. This ensures that f has always a different sign in the last two iterates.

HINT: The regula falsi variation of the secant method can be easily implemented with a little modification of [1, Code 2.3.25]:

```
function x = secant_falsi(x0,x1,F,rtol,atol)
fo = F(x0);
for i=1:MAXIT
  fn = F(x1);
  s = fn*(x1-x0)/(fn-fo); % correction
  if (F(x1 - s)*fn < 0)
    x0 = x1;  fo = fn;  end
  x1 = x1 - s;
  if (abs(s) < max(atol,rtol*min(abs([x0;x1]))))
    x = x1; return; end
end
```

Problem 2 Gaussian quadrature (core problem)

Given a smooth, odd function $f : [-1, 1] \to \mathbb{R}$, consider the integral

$$I := \int_{-1}^{1} \arcsin(t)\ f(t)\,\mathrm{d}t. \tag{31}$$

We want to approximate this integral using global Gauss quadrature. The nodes (vector `x`) and the weights (vector `w`) of n-point Gaussian quadrature on $[-1, 1]$ can be computed using the provided MATLAB routine `[x,w]=gaussquad(n)` (in the file `gaussquad.m`).

(2a) ⚂ Write a MATLAB routine

```
function   GaussConv(f_hd)
```

that produces an appropriate convergence plot of the quadrature error versus the number $n = 1, \ldots, 50$ of quadrature points. Here, `f_hd` is a handle to the function f.

Save your convergence plot for $f(t) = \sinh(t)$ as `GaussConv.eps`.

HINT: Use the MATLAB command `quad` with tolerance `eps` to compute a reference value of the integral.

HINT: If you cannot implement the quadrature formula, you can resort to the MATLAB function

```
function   I = GaussArcSin(f_hd,n)
```

provided in implemented `GaussArcSin.p` that computes n-points Gauss quadrature for the integral (31). Again `f_hd` is a function handle to f.

(2b) ⚀ Which kind of convergence do you observe?

(2c) ⚂ Transform the integral (31) into an equivalent one with a suitable change of variable so that Gauss quadrature applied to the transformed integral converges much faster.

(2d) ⚂ Now, write a MATLAB routine

```
function  GaussConvCV(f_hd)
```

which plots the quadrature error versus the number $n = 1, \ldots, 50$ of quadrature points for the integral obtained in the previous subtask.

Again, choose $f(t) = \sinh(t)$ and save your convergence plot as `GaussConvCV.eps`.

HINT: In case you could not find the transformation, you may rely on the function

```
function   I = GaussArcSinCV(f_hd,n)
```

implemented in `GaussArcSinCV.p` that applies n-points Gauss quadrature to the transformed problem.

(2e) ⚃ Explain the difference between the results obtained in subtasks (2a) and (2d).

Problem 3 Numerical integration of improper integrals

We want to devise a numerical method for the computation of improper integrals of the form $\int_{-\infty}^{\infty} f(t)dt$ for continuous functions $f : \mathbb{R} \to \mathbb{R}$ that decay sufficiently fast for $|t| \to \infty$ (such that they are integrable on $\mathbb{R}$).

A first option (T) is the truncation of the domain to a bounded interval $[-b, b], b \leq \infty$, that is, we approximate:

$$\int_{-\infty}^{\infty} f(t)dt \approx \int_{-b}^{b} f(t)dt$$

and then use a standard quadrature rule (like Gauss-Legendre quadrature) on $[-b, b]$.

(3a) ⚀ For the integrand $g(t) := 1/(1 + t^2)$ determine b such that the truncation error E_T satisfies:

$$E_T := \left| \int_{-\infty}^{\infty} g(t)dt - \int_{-b}^{b} g(t)dt \right| \leq 10^{-6} \tag{32}$$

(3b) ⚀ What is the algorithmic difficulty faced in the implementation of the truncation approach for a generic integrand?

A second option (S) is the transformation of the improper integral to a bounded domain by substitution. For instance, we may use the map $t = \cot(s)$.

(3c) ⚃ Into which integral does the substitution $t = \cot(s)$ convert $\int_{-\infty}^{\infty} f(t)dt$?

(3d) ⚀ Write down the transformed integral explicitly for $g(t) := \frac{1}{1+t^2}$. Simplify the integrand.

(3e) ⚀ Write a C++ function:

```
template <typename function>
double quadinf(int n, const function &f);
```

that uses the transformation from (3d) together with n-point Gauss-Legendre quadrature to evaluate $\int_{-\infty}^{\infty} f(t)dt$. f passes an object that provides an evaluation operator of the form:

```
double operator() (double x) const;
```

HINT: See `quadinf_template.cpp`.

HINT: A lambda function with signature

```
(double) -> double
```

automatically satisfies this requirement.

(3f) ⚁ Study the convergence as $n \to \infty$ of the quadrature method implemented in (3e) for the integrand $h(t) := \exp(-(t-1)^2)$ (shifted Gaussian). What kind of convergence do you observe?

HINT:

$$\int_{-\infty}^{\infty} h(t)dt = \sqrt{\pi} \tag{33}$$

Problem 4 Quadrature plots

We consider three different functions on the interval $I = [0, 1]$:

$$\begin{aligned} &\text{function A:} \quad f_A \in \text{analytic}\,, \quad f_A \notin \mathcal{P}_k \;\forall\; k \in \mathbb{N}\,; \\ &\text{function B:} \quad f_B \in C^0(I)\,, \quad f_B \notin C^1(I)\,; \\ &\text{function C:} \quad f_C \in \mathcal{P}_{12}\,, \end{aligned}$$

where $\mathcal{P}_k$ is the space of the polynomials of degree at most k defined on I. The following quadrature rules are applied to these functions:

- quadrature rule A, global Gauss quadrature;

- quadrature rule B, composite trapezoidal rule;
- quadrature rule C, composite 2-point Gauss quadrature.

The corresponding absolute values of the quadrature errors are plotted against the number of function evaluations in Figure 7. Notice that only the quadrature errors obtained with an even number of function evaluations are shown.

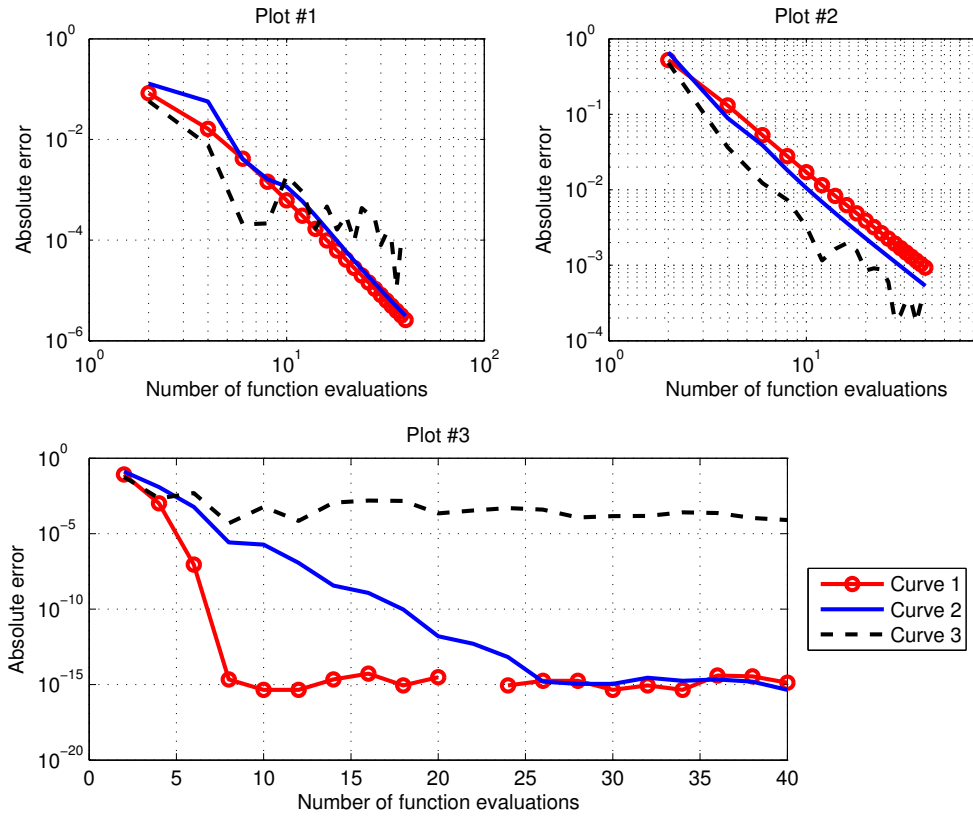


Figure 7: Quadrature convergence plots for different functions and different rules.

(4a) ⚂ Match the three plots (plot #1, #2 and #3) with the three quadrature rules (quadrature rule A, B, and C). Justify your answer.

HINT: Notice the different axis scales in the plots.

(4b) ⚂ The quadrature error curves for a particular function f_A, f_B and f_C are plotted in the same style (curve 1 as red line with small circles, curve 2 means the blue solid line, curve 3 is the black dashed line). Which curve corresponds to which function (f_A, f_B, f_C)? Justify your answer.

Issue date: 19.11.2015

Hand-in: 26.11.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 11

Problem 1 Efficient quadrature of singular integrands (core problem)

This problem deals with efficient numerical quadrature of non-smooth integrands with a special structure. Before you tackle this problem, read about regularization of integrands by transformation [1, Rem. 5.3.45].

Our task is to develop quadrature formulas for integrals of the form:

$$W(f) := \int_{-1}^{1} \sqrt{1-t^2}\, f(t)dt, \tag{34}$$

where f possesses an analytic extension to a complex neighbourhood of $[-1,1]$.

(1a) ⊡ The provided function

```
QuadRule gauleg(unsigned int n);
```

returns a structure `QuadRule` containing nodes (x_j) and weights (w_j) of a Gauss-Legendre quadrature (→ [1, Def. 5.3.28]) on $[-1,1]$ with n nodes. Have a look at the file `gauleg.hpp` and `gauleg.cpp`, and understand how the implementation works and how to use it.

HINT: Learn/remember how linking works in C++. To use the function `gauleg` (declared in `gauleg.hpp` and defined in `gauleg.cpp`) in a file `file.cpp`, first `include` the header file `gauleg.hpp` in the file `file.cpp`, and then compile and link the files `gauleg.cpp` and `file.cpp`. Using `gcc`:

```
g++ [compiler opts.] -c gauleg.cpp
g++ [compiler opts.] -c file.cpp
g++ [compiler opts.]  gauleg.o file.o -o exec_name
```

If you want to use `CMake`, have a look at the file `CMakeLists.txt`.

(1b) ⚀ Study [1, § 5.3.37] in order to learn about the convergence of Gauss-Legendre quadrature.

(1c) ⚂ Based on the function `gauleg`, implement a C++ function

```
template <class func>
double quadsingint(func&& f, unsigned int n);
```

that approximately evaluates (34) using $2n$ evaluations of f. An object of type `func` must provide an evaluation operator

```
double operator(double t) const;
```

For the quadrature error asymptotic exponential convergence to zero for $n \to \infty$ must be ensured by your function.

HINT: A C++ lambda function provides such operator.

HINT: You may use the classical binomial formula $\sqrt{1-t^2} = \sqrt{1-t}\sqrt{1+t}$.

HINT: You can use the template `quadsingint_template.cpp`.

(1d) ⚁ Give formulas for the nodes c_j and weights $\tilde{w}_j$ of a $2n$-point quadrature rule on $[-1,1]$, whose application to the integrand f will produce the same results as the function `quadsingint` that you implemented in (1c).

(1e) ⚀ Tabulate the quadrature error:

$$|W(f) - \texttt{quadsingint(f,n)}|$$

for $f(t) := \frac{1}{2+\exp(3t)}$ and $n = 1, 2, ..., 25$. Estimate the $0 < q < 1$ in the decay law of exponential convergence, see [1, Def. 4.1.31].

Problem 2 Nested numerical quadrature

A laser beam has intensity

$$I(x,y) = \exp(-\alpha((x-p)^2 + (y-q)^2))$$

on the plane orthogonal to the direction of the beam.

(2a) ⚀ Write down the radiant power absorbed by the triangle

$$\triangle := \{(x,y)^T \in \mathbb{R}^2 \mid x \geq 0, y \geq 0, x+y \leq 1\}$$

as a double integral.

HINT: The radiant power absorbed by a surface is the integral of the intensity over the surface.

(2b) ⚁ Write a C++ function

```
template <class func>
double evalgaussquad(double a, double b, func&& f, const
    QuadRule & Q);
```

that evaluates an the N-point quadrature for an integrand passed in `f` in $[a,b]$. It should rely on the quadrature rule on the reference interval $[-1,1]$ that supplied through an object of type `QuadRule`. (The vectors `weights` and `nodes` denote the weights and nodes of the reference quadrature rule respectively.)

HINT: Use the function `gauleg` declared in `gauleg.hpp` and defined in `gauleg.cpp` to compute nodes and weights in $[-1,1]$. See Problem 1 for further explanations.

HINT: You can use the template `laserquad_template.cpp`.

(2c) ⚂ Write a C++ function

```
template <class func>
double gaussquadtriangle(func&& f, int N)
```

for the computation of the integral

$$\int_{\triangle} f(x,y)dxdy, \tag{35}$$

using nested N-point, 1D Gauss quadratures (using the functions `evalgaussquad` of (2b) and `gauleg`).

HINT: Write (35) explicitly as a double integral. Take particular care to correctly find the intervals of integration.

HINT: Lambda functions of C++ are well suited for this kind of implementation.

(2d) ⚀ Apply the function `gaussquadtriangle` of (2c) to the subproblem (2a) using the parameter $\alpha = 1, p = 0, q = 0$. Compute the error w.r.t to the number of nodes N. What kind of convergence do you observe? Explain the result.

HINT: Use the "exact" value of the integral 0.366046550000405.

Problem 3 Weighted Gauss quadrature

The development of an alternative quadrature formula for (34) relies on the Chebyshev polynomials of the second kind U_n, defined as

$$U_n(t) = \frac{\sin((n+1)\arccos t)}{\sin(\arccos t)}, \qquad n \in \mathbb{N}.$$

Recall the role of the orthogonal Legendre polynomials in the derivation and definition of Gauss-Legendre quadrature rules (see [1, § 5.3.25]).

As regards the integral (34), this role is played by the U_n, which are orthogonal polynomials with respect to a weighted L^2 inner product, see [1, Eq. (4.2.20)], with weight given by $w(\tau) = \sqrt{1-\tau^2}$.

(3a) ⚁ Show that the U_n satisfy the 3-term recursion

$$U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t), \qquad U_0(t) = 1, \qquad U_1(t) = 2t,$$

for every $n \geq 1$.

(3b) ⚀ Show that $U_n \in \mathcal{P}_n$ with leading coefficient 2^n.

(3c) ⚁ Show that for every $m, n \in \mathbb{N}_0$ we have

$$\int_{-1}^{1} \sqrt{1-t^2}\, U_m(t) U_n(t)\, dt = \frac{\pi}{2}\delta_{mn}.$$

(3d) ⚀ What are the zeros ξ_j^n $(j = 1, \ldots, n)$ of U_n, $n \geq 1$? Give an explicit formula similar to the formula for the Chebyshev nodes in $[-1, 1]$.

(3e) ⚃ Show that the choice of weights

$$w_j = \frac{\pi}{n+1} \sin^2\left(\frac{j}{n+1}\pi\right), \qquad j = 1, \ldots, n,$$

ensures that the quadrature formula

$$Q_n^U(f) = \sum_{j=1}^{n} w_j f(\xi_j^n) \tag{36}$$

provides the exact value of (34) for $f \in \mathcal{P}_{n-1}$ (assuming exact arithmetic).

HINT: Use all the previous subproblems.

(3f) ⚀ Show that the quadrature formula (36) gives the exact value of (34) even for every $f \in \mathcal{P}_{2n-1}$.

HINT: See [1, Thm. 5.3.21].

(3g) ⚄ Show that the quadrature error

$$|Q_n^U(f) - W(f)|$$

decays to 0 exponentially as $n \to \infty$ for every $f \in C^\infty([-1,1])$ that admits an analytic extension to an open subset of the complex plane.

HINT: See [1, § 5.3.37].

(3h) ⚂ Write a C++ function

```
template<typename Function>
double quadU(const Function &f, unsigned int n)
```

that gives $Q_n^U(f)$ as output, where `f` is an object with an evaluation operator, like a lambda function, representing f, e.g.

```
auto f = [] (double & t) {return 1/(2 + exp(3*t));};
```

(3i) ⚁ Test your implementation with the function $f(t) = 1/(2+e^{3t})$ and $n = 1, \ldots, 25$. Tabulate the quadrature error $E_n(f) = |W(f) - Q_n^U(f)|$ using the "exact" value $W(f) = 0.483296828976607$. Estimate the parameter $0 \leq q < 1$ in the asymptotic decay law $E_n(f) \approx Cq^n$ characterizing (sharp) exponential convergence, see [1, Def. 4.1.31].

Problem 4 Generalize "Hermite-type" quadrature formula

(4a) ⚄ Determine $A, B, C, x_1 \in \mathbb{R}$ such that the quadrature formula:

$$\int_0^1 f(x)dx \approx Af(0) + Bf'(0) + Cf(x_1) \tag{37}$$

is exact for polynomials of highest possible degree.

(4b) ⚄

Compute an approximation of $z(2)$, where the function z is defined as the solution of the initial value problem

$$z'(t) = \frac{t}{1+t^2} \quad , \quad z(1) = 1 \, . \tag{38}$$

Issue date: 26.11.2015
Hand-in: 03.12.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 12

Problem 1 Three-stage Runge-Kutta method (core problem)

The most widely used class of numerical integratos for IVPs is that of *explicit* Runge-Kutta (RK) methods as defined in [1, Def. 11.4.9]. They are usually described by giving their coefficients in the form of a Butcher scheme [1, Eq. (11.4.11)].

(1a) ⚄ Implement a header-only C++ class `RKIntegrator`

```
template <class State>
class RKIntegrator {
public:
  RKIntegrator(const Eigen::MatrixXd & A,
               const Eigen::VectorXd & b) {
    // TODO: given a Butcher scheme in A,b, initialize
       RK method for solution of an IVP
  }

  template <class Function>
  std::vector<State> solve(const Function &f, double T,
                           const State & y0,
                           unsigned int N) const {
    // TODO: computes N uniform time steps for the ODE
       y'(t) = f(y) up to time T of RK method with
       initial value y0 and store all steps (y_k) into
       return vector
  }
private:
  template <class Function>
```

```
17    void step(const Function &f, double h,
18              const State & y0, State & y1) const {
19      // TODO: performs a single step from y0 to y1 with
            step size h of the RK method for the IVP with rhs f
20    }
21
22    // TODO: hold data for RK methods
23  };
```

which implements a generic RK method given by a Butcher scheme to solve the autonomous initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \mathbf{y}(t_0) = \mathbf{y}_0$.

HINT: See `rkintegrator_template.hpp` for more details about the implementation.

(1b) ⚂ Test your implementation of the RK methods with the following data. As autonomous initial value problem, consider the predator/prey model (cf. [1, Ex. 11.1.9]):

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t))y_1(t) \tag{39}$$
$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2)y_2(t) \tag{40}$$
$$\mathbf{y}(0) = [100, 5] \tag{41}$$

with coefficients $\alpha_1 = 3, \alpha_2 = 2, \beta_1 = \beta_2 = 0.1$.

Use a Runge-Kutta single step method described by the following *Butcher scheme* (cf. [1, Def. 11.4.9]):

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{3} & \frac{1}{3} & 0 & \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \tag{42}$$

Compute an approximated solution up to time $T = 10$ for the number of steps $N = 2^j, j = 7, \ldots, 14$.

Use, as reference solution, $\mathbf{y}(10) = [0.319465882659820, 9.730809352326228]$.

Tabulate the error and compute the experimental order of algebraic convergence of the method.

HINT: See `rk3prey_template.cpp` for more details about the implementation.

Problem 2 Order is not everything (core problem)

In [1, Section 11.3.2] we have seen that Runge-Kutta single step methods when applied to initial value problems with sufficiently smooth solutions will converge algebraically (with respect to the maximum error in the mesh points) with a rate given by their intrinsic order, see [1, Def. 11.3.21].

In this problem we perform empiric investigations of orders of convergence of several explicit Runge-Kutta single step methods. We rely on two IVPs, one of which has a perfectly smooth solution, whereas the second has a solution that is merely piecewise smooth. Thus in the second case the smoothness assumptions of the convergence theory for RK-SSMs might be violated and it is interesting to study the consequences.

(2a) ⚄ Consider the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{43}$$

where $\mathbf{f}: \mathbb{R}^n \to \mathbb{R}^n$ and $\mathbf{y}_0 \in \mathbb{R}^n$. Using the class `RKIntegrate` of Problem 1 write a C++ function

```
template <class Function>
void errors(const Function &f, const double &T, const
    Eigen::VectorXd &y0, const Eigen::MatrixXd &A,
const Eigen::VectorXd &b)
```

that computes an approximated solution $\mathbf{y}_N$ of (43) up to time T by means of an explicit Runge-Kutta method with $N = 2^k$, $k = 1, \ldots, 15$, uniform timesteps. The method is defined by the Butcher scheme described by the inputs `A` and `b`. The input `f` is an object with an evaluation operator (e.g. a lambda function) for arguments of type `const VectorXd &` representing $\mathbf{f}$. The input `y0` passes the initial value $\mathbf{y}_0$.

For each k, the function should show the error at the final point $E_N = \|\mathbf{y}_N(T) - \mathbf{y}_{2^{15}}(T)\|$, $N = 2^k$, $k = 1, \ldots, 13$, accepting $\mathbf{y}_{2^{15}}(T)$ as exact value. Assuming algebraic convergence for $E_N \approx CN^{-r}$, at each step show an approximation of the order of convergence r_k (recall that $N = 2^k$). This will be an expression involving E_N and $E_{N/2}$.

Finally, compute and show an approximate order of convergence by averaging the relevant r_Ns (namely, you should take into account the cases before machine precision is reached in the components of $\mathbf{y}_N(T) - \mathbf{y}_{2^{15}}(T)$).

(2b) ⚂ Calculate the analytical solutions of the logistic ODE (see [1, Ex. 11.1.5])

$$\dot{y} = (1 - y)y, \quad y(0) = 1/2, \tag{44}$$

and of the initial value problem

$$\dot{y} = |1.1 - y| + 1, \quad y(0) = 1. \tag{45}$$

(2c) ⚂ Use the function `errors` from (2a) with the ODEs (44) and (45) and the methods:

- the explicit Euler method, a RK single step method of order 1,
- the explicit trapezoidal rule, a RK single step method of order 2,
- an RK method of order 3 given by the Butcher tableau

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

- the classical RK method of order 4.

(See [1, Ex. 11.4.13] for details.) Set $T = 0.1$.

Comment the calculated order of convergence for the different methods and the two different ODEs.

Problem 3 Integrating ODEs using the Taylor expansion method

In [1, Chapter 11] of the course we studied single step methods for the integration of initial value problems for ordinary differential equations $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, [1, Def. 11.3.5]. Explicit single step methods have the advantage that they only rely on point evaluations of the right hand side $\mathbf{f}$.

This problem examines another class of methods that is obtained by the following reasoning: if the right hand side $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ of an autonomous initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) , \qquad \mathbf{y}(0) = \mathbf{y}_0 , \tag{46}$$

with solution $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is smooth, also the solution $\mathbf{y}(t)$ will be regular and it is possible to expand it into a Taylor sum at $t = 0$, see [1, Thm. 2.2.15],

$$\mathbf{y}(t) = \sum_{n=0}^{m} \frac{\mathbf{y}^{(n)}(0)}{n!} t^n + R_m(t) \,, \tag{47}$$

with remainder term $R_m(t) = O(t^{m+1})$ for $t \to 0$.

A single step method for the numerical integration of (46) can be obtained by choosing $m = 3$ in (47), neglecting the remainder term, and taking the remaining sum as an approximation of $\mathbf{y}(h)$, that is,

$$\mathbf{y}(h) \approx \mathbf{y}_1 := \mathbf{y}(0) + \frac{d\mathbf{y}}{dt}(0)h + \frac{1}{2}\frac{d^2\mathbf{y}}{dt^2}(0)h^2 + \frac{1}{6}\frac{d^3\mathbf{y}}{dt^3}(0)h^3 \,.$$

Subsequently, one uses the ODE and the initial condition to replace the temporal derivatives $\frac{d^l\mathbf{y}}{dt^l}$ with expressions in terms of (derivatives of) $\mathbf{f}$. This yields a single step integration method called *Taylor (expansion) method*.

(3a) ⚂ Express $\frac{d\mathbf{y}}{dt}(t)$ and $\frac{d^2\mathbf{y}}{dt^2}(t)$ in terms of $\mathbf{f}$ and its Jacobian $\mathbf{Df}$.

HINT: Apply the chain rule, see [1, § 2.4.5], then use the ODE (46).

(3b) ⚄ Verify the formula

$$\frac{d^3\mathbf{y}}{dt^3}(0) = \mathbf{D}^2\mathbf{f}(\mathbf{y}_0)\big(\mathbf{f}(\mathbf{y}_0), \mathbf{f}(\mathbf{y}_0)\big) + \mathbf{Df}(\mathbf{y}_0)^2\mathbf{f}\big(\mathbf{y}_0\big) \,. \tag{48}$$

HINT: this time we have to apply both the product rule [1, (2.4.9)] and chain rule [1, (2.4.8)] to the expression derived in the previous sub-problem.

To gain confidence, it is advisable to consider the scalar case $d = 1$ first, where $f : \mathbb{R} \to \mathbb{R}$ is a real valued function.

Relevant for the case $d > 1$ is the fact that the first derivative of $\mathbf{f}$ is a linear mapping $\mathbf{Df}(\mathbf{y}_0) : \mathbb{R}^n \to \mathbb{R}^n$. This linear mapping is applied by multiplying the argument with the Jacobian of $\mathbf{f}$. Similarly, the second derivative is a *bilinear* mapping $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$. The i-th component of $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)\big(\mathbf{v}, \mathbf{v}\big)$ is given by

$$\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)\big(\mathbf{v}, \mathbf{v}\big)_i = \mathbf{v}^T\mathbf{H}\mathbf{f}_i(\mathbf{y}_0)\mathbf{v},$$

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

(3c) ⚄ We now apply the Taylor expansion method introduced above to the *predator-prey* model (53) introduced in Problem 1 and [1, Ex. 11.1.9].

To that end write a header-only C++ class `TaylorIntegrator` for the integration of the autonomous ODE of (53) using the Taylor expansion method with uniform time steps on the temporal interval $[0, 10]$.

HINT: You can copy the implementation of Problem 1 and modify only the `step` method to perform a single step of the Taylor expansion method.

HINT: Find a suitable way to pass the data for the derivatives of the r.h.s. function $\mathbf{f}$ to the `solve` function. You may modify the signature of `solve`.

HINT: See `taylorintegrator_template.hpp`.

(3d) ⚀ Experimentally determine the order of convergence of the considered Taylor expansion method when it is applied to solve (53). Study the behaviour of the error at final time $t = 10$ for the initial data $\mathbf{y}(0) = [100, 5]$.

As a reference solution use the same data as Problem 1.

HINT: See `taylorprey_template.cpp`.

(3e) ⚀ What is the disadvantage of the Taylor method compared with a Runge-Kutta method?

Problem 4 System of ODEs

Consider the following initial value problem for a second-order system of ordinary differential equations:

$$\begin{aligned} 2\ddot{u}_1 - \ddot{u}_2 &= u_1(u_2 + u_1)\,, \\ -\ddot{u}_{i-1} + 2\ddot{u}_i - \ddot{u}_{i+1} &= u_i(u_{i-1} + u_{i+1})\,, \qquad i = 2, \ldots, n-1\,, \\ 2\ddot{u}_n - \ddot{u}_{n-1} &= u_n(u_n + u_{n-1})\,, \\ u_i(0) &= u_{0,i} \qquad i = 1, \ldots, n\,, \\ \dot{u}_i(0) &= v_{0,i} \qquad i = 1, \ldots, n\,, \end{aligned} \tag{49}$$

in the time interval $[0, T]$.

(4a) ⚂ Write (49) as a first order IVP of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$ (see [1, Rem. 11.1.23]).

(4b) ⚄ Apply the function `errors` constructed in Problem 2 to the IVP obtained in the previous subproblem. Use

$$n = 5\,, \qquad u_{0,i} = i/n\,, \qquad v_{0,i} = -1\,, \qquad T = 1,$$

and the classical RK method of order 4. Construct any sparse matrix encountered as a sparse matrix in EIGEN. Comment the order of convergence observed.

Issue date: 03.12.2015

Hand-in: 10.12.2015 (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

ETH Zürich
D-MATH

Numerical Methods for CSE

Problem Sheet 13

Problem 1 Matrix-valued Differential Equation (core problem)

First we consider the *linear* matrix differential equation

$$\dot{\mathbf{Y}} = \mathbf{A}\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad \text{with} \quad \mathbf{A} \in \mathbb{R}^{n\times n}. \tag{50}$$

whose solutions are *matrix-valued functions* $\mathbf{Y} : \mathbb{R} \to \mathbb{R}^{n\times n}$.

(1a) ⊡ Show that for *skew-symmetric* $\mathbf{A}$, i.e. $\mathbf{A} = -\mathbf{A}^\top$ we have:

$$\mathbf{Y}(0) \text{ orthogonal} \quad \Longrightarrow \quad \mathbf{Y}(t) \text{ orthogonal} \quad \forall t\,.$$

HINT: Remember what property distinguishes an orthogonal matrix. Thus you see that the assertion we want to verify boils down to showing that the bilinear expression $t \mapsto \mathbf{Y}(t)^\top\mathbf{Y}(t)$ does not vary along trajectories, that is, its time derivative must vanish. This can be established by means of the product rule [1, Eq. (2.4.9)] and using the differential equation.

(1b) ⊡ Implement three C++ functions

(i) a single step of the explicit Euler method:

```
Eigen::MatrixXd eeulstep(const Eigen::MatrixXd & A,
    const Eigen::MatrixXd & Y0, double h);
```

(ii) a single step of the implicit Euler method:

```
Eigen::MatrixXd ieulstep(const Eigen::MatrixXd & A,
    const Eigen::MatrixXd & Y0, double h);
```

(iii) a single step of the implicit mid-point method:

```
Eigen::MatrixXd impstep(const Eigen::MatrixXd & A,
    const Eigen::MatrixXd & Y0, double h);
```

which determine, for a given initial value $\mathbf{Y}(t_0) = \mathbf{Y}_0$ and for given step size h, approximations for $\mathbf{Y}(t_0 + h)$ using one step of the corresponding method for the approximation of the ODE (50)

(1c) ⊡ Investigate numerically, which one of the implemented methods preserves orthogonality in the sense of sub-problem (1a) for the ODE (50) and which one doesn't. To that end, consider the matrix

$$\mathbf{M} := \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 9 & 9 & 2 \end{bmatrix}$$

and use the matrix $\mathbf{Q}$ arising from the QR-decomposition of $\mathbf{M}$ as initial data $\mathbf{Y}_0$. As matrix A, use the skew-symmetric matrix

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

To that end, perform $n = 20$ time steps of size $h = 0.01$ with each method and compute the Frobenius norm of $\mathbf{Y}(T)'\mathbf{Y}(T) - \mathbf{I}$. Use the functions from subproblem (1b).

From now we consider a non-linear ODE that is structurally similar to (50). We study the initial value problem

$$\dot{\mathbf{Y}} = -(\mathbf{Y} - \mathbf{Y}^\top)\mathbf{Y} =: f(\mathbf{Y}) \quad , \quad \mathbf{Y}(0) = \mathbf{Y}_0 \in \mathbb{R}^{n,n}, \tag{51}$$

whose solution is given by a *matrix-valued function* $t \mapsto \mathbf{Y}(t) \in \mathbb{R}^{n\times n}$.

(1d) ⊡ Write a C++ function

```
Eigen::MatrixXd matode(const Eigen::MatrixXd & Y0,
    double T)
```

which solves (51) on $[0, T]$ using the C++ header-only class `ode45` (in the file `ode45.hpp`). The initial value should be given by a $n \times n$ EIGEN matrix `Y0`. Set the absolute tolerance to 10^{-10} and the relative tolerance to 10^{-8}. The output should be an approximation of $\mathbf{Y}(T) \in \mathbb{R}^{n \times n}$.

HINT: The `ode45` class works as follows:

1. Call the constructor, and specify the r.h.s function f and the type for the solution and the initial data in `RhsType`, example:

```
ode45<StateType> O(f);
```

   with, for instance, `Eigen::VectorXd` as `StateType`.

2. (optional) Set custom options, modifying the `struct options` inside `ode45`, for instance:

```
O.options.<option_you_want_to_change> = <value>;
```

3. Solve the IVP and store the solution, e.g.:

```
std::vector<std::pair<Eigen::VectorXd, double>> sol
    = O.solve(y0, T);
```

Relative and absolute tolerances for `ode45` are defined as `rtol` resp. `atol` variables in the `struct options`. The return value is a sequence of states and times computed by the adaptive single step method.

HINT: The type `RhsType` needs a vector space structure implemented with operators $*$, $*$, $*=$, $+=$ and assignment/copy operators. Moreover a norm method must be available. Eigen vector and matrix types, as well as fundamental types are eligible as `RhsType`.

HINT: Have a look at the public interface of `ode45.hpp`. Look at the template file `matrix_ode_template.cpp`.

(1e) ⚀ Show that the function $t \mapsto \mathbf{Y}^{\mathsf{T}}(t)\mathbf{Y}(t)$ is constant for the exact solution $\mathbf{Y}(t)$ of (51).

HINT: Remember the general product rule [1, Eq. (2.4.9)].

(1f) ⊡ Write a C++ function

```
bool checkinvariant(const Eigen::MatrixXd & M, double T);
```

which (numerically) determines if the statement from (1e) is true, for $t = T$ and for the output of `matode` from sub-problem (1d). You must take into account round-off errors. The function's input should be the same as that of `matode`.

HINT: See `matrix_ode_template.cpp`.

(1g) ⊡ Use the function `checkinvariant` to test wether the invariant is preserved by `ode45` or not. Use the matrix $\mathbf{M}$ defined above and and $T = 1$.

Problem 2 Stability of a Runge-Kutta Method (core problem)

We consider a 3-stage Runge–Kutta single step method described by the Butcher-Tableau

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 \\ \hline & 1/6 & 1/6 & 2/3 \end{array} \tag{52}$$

(2a) ⊡ Consider the prey/predator model

$$\dot{y}_1(t) = (1 - y_2(t))y_1(t) \tag{53}$$
$$\dot{y}_2(t) = (y_1(t) - 1)y_2(t) \tag{54}$$
$$\mathbf{y}(0) = [100, 1]\,. \tag{55}$$

Write a C++ code to approximate the solution up to time $T = 1$ of the IVP. Use a RK-SSM defined above. Numerically determine the convergence order of the method for uniform steps of size $2^{-j}, j = 2, \ldots, 13$.

Use, as a reference solution, an approximation with 2^{14} steps.

What do you notice for big step sizes? What is the maximum step size for the solution to be stable?

HINT: You can use the `rkintegrator.hpp` implemented in Problem Sheet 12. See `stabrk_template.cpp`.

(2b) ⊡ Calculate the stability function $S(z)$, $z = h\lambda$, $\lambda \in \mathbb{C}$ of the method given by the table (52).

Problem 3 Initial Condition for Lotka-Volterra ODE

Introduction. In this problem we will face a situation, where we need to compute the derivative of the solution of an IVP with respect to the initial state. This paragraph will show how this derivative can be obtained as the solution of another differential equation. Please read this carefully and try to understand every single argument.

We consider IVPs for the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{56}$$

with smooth right hand side $\mathbf{f}: D \to \mathbb{R}^d$, where $D \subseteq \mathbb{R}^d$ is the state space. We take for granted that for all initial states, solutions exist for all times (global solutions, see [1, Ass. 11.1.38]).

By its very definition given in [1, Def. 11.1.39]), the evolution operator

$$\boldsymbol{\Phi}: \mathbb{R} \times D \to D, \quad (t, \mathbf{y}) \mapsto \boldsymbol{\Phi}(t, \mathbf{y})$$

satisfies

$$\frac{\partial \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y})).$$

Next, we can differentiate this identity with respect to the state variable $\mathbf{y}$. We assume that all derivatives can be interchanged, which can be justified by rigorous arguments (which we won't do here). Thus, by the chain rule, we obtain and after swapping partial derivatives $\frac{\partial}{\partial t}$ and $\mathsf{D}_{\mathbf{y}}$

$$\frac{\partial \mathsf{D}_{\mathbf{y}} \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathsf{D}_{\mathbf{y}} \frac{\partial \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathsf{D}_{\mathbf{y}}(\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))) = \mathsf{D}\,\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))\, \mathsf{D}_{\mathbf{y}}\, \boldsymbol{\Phi}(t, \mathbf{y}).$$

Abbreviating $\mathbf{W}(t, \mathbf{y}) := \mathsf{D}_{\mathbf{y}}\, \boldsymbol{\Phi}(t, \mathbf{y})$ we can rewrite this as the non-autonomous ODE

$$\dot{\mathbf{W}} = \mathsf{D}\,\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))\mathbf{W}. \tag{57}$$

Here, the state $\mathbf{y}$ can be regarded as a parameter. Since $\boldsymbol{\Phi}(0, \mathbf{y}) = \mathbf{y}$, we also know $\mathbf{W}(0, \mathbf{y}) = \mathbf{I}$ (identity matrix), which supplies an initial condition for (57). In fact, we can even merge (56) and (57) into the ODE

$$\frac{d}{dt}\left[\mathbf{y}(\cdot)\,,\ \mathbf{W}(\cdot, \mathbf{y}_0)\right] = \left[\mathbf{f}(\mathbf{y}(t))\,,\ \mathsf{D}\,\mathbf{f}(\mathbf{y}(t))\mathbf{W}(t, \mathbf{y}_0)\right]\,, \tag{58}$$

which is autonomous again.

Now let us apply (57)/(58). As in [1, Ex. 11.1.9], we consider the following autonomous Lotka-Volterra differential equation of a predator-prey model

$$\begin{aligned} \dot{u} &= (2-v)u \\ \dot{v} &= (u-1)v \end{aligned} \tag{59}$$

on the state space $D = \mathbb{R}^2_+$, $\mathbb{R}_+ = \{\xi \in \mathbb{R} : \xi > 0\}$. All the solutions of (59) are periodic and their period depends on the initial state $[u(0), v(0)]^T$. In this exercise we want to develop a numerical method which computes a suitable initial condition for a given period.

(3a) ⚀ For fixed state $\mathbf{y} \in D$, (57) represents an ODE. What is its state space?

(3b) ⚀ What is the right hand side function for the ODE (57), in the case of the $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ given by the Lotka-Volterra ODE (59)? You may write $u(t), v(t)$ for solutions of (59).

(3c) ⚀ From now on we write $\boldsymbol{\Phi}: \mathbb{R} \times \mathbb{R}^2_+ \to \mathbb{R}^2_+$ for the evolution operator associated with (59). Based on $\boldsymbol{\Phi}$ derive a function $\mathbf{F} : \mathbb{R}^2_+ \to \mathbb{R}^2_+$ which evaluates to zero for the input $\mathbf{y}_0$ if the period of the solution of system (59) with initial value

$$\mathbf{y}_0 = \begin{bmatrix} u(0) \\ v(0) \end{bmatrix}$$

is equal to a given value T_P.

(3d) ⚀ We write $\mathbf{W}(T, \mathbf{y}_0)$, $T \geq 0$, $\mathbf{y}_0 \in \mathbb{R}^2_+$ for the solution of (57) for the underlying ODE (59). Express the Jacobian of $\mathbf{F}$ from (3c) by means of $\mathbf{W}$.

(3e) ⚂ Argue, why the solution of $\mathbf{F}(\mathbf{y}) = 0$ will, in gneneral, not be unique. When will it be unique?

HINT: Study [1, § 11.1.21] again. Also look at [1, Fig. 356].

(3f) ⚂ A C++ implementation of an adaptive embedded Runge-Kutta method is available, with a functionality similar to MATLAB's `ode45` (see Problem 1). Relying on this implement a C++ function

```
1 std::pair<Vector2d,Matrix2d> PhiAndW(double u0, double
    v0, double T)
```

that computes $\boldsymbol{\Phi}(T, [u_0, v_0]^T)$ and $\mathbf{W}(T, [u_0, v_0]^T)$. The first component of the output pair should contain $\boldsymbol{\Phi}(T, [u_0, v_0]^T)$ and the second component the matrix $\mathbf{W}(T, [u_0, v_0]^T)$. See `LV_template.cpp`.

HINT: As in (58), both ODEs (for $\mathbf{\Phi}$ and $\mathbf{W}$) must be combined into a single autonomous differential equation on the state space $D \times \mathbb{R}^{d\times d}$.

HINT: The equation for $\mathbf{W}$ is a matrix differential equation. These cannot be solved directly using `ode45`, because the solver expects the right hand side to return a vector. Therefore, transform matrices into vectors (and vice-versa).

(3g) ⚂ Using `PhiAndW`, write a C++ routine that determines initial conditions $u(0)$ and $v(0)$ such that the solution of the system (59) has period $T = 5$. Use the multi-dimensional *Newton method* for $\mathbf{F}(\mathbf{y}) = 0$ with $\mathbf{F}$ from (3c). As your initial approximation, use $[3,2]^T$. Terminate the *Newton method* as soon as $\|\mathbf{F}(\mathbf{y})\| \leq 10^{-5}$. Validate your implementation by comparing the obtained initial data $\mathbf{y}$ with $\mathbf{\Phi}(100,\mathbf{y})$.

HINT: Set relative and absolute tolerances of `ode45` to 10^{-14} and 10^{-12}, respectively. See file `LV_template.cpp`.

HINT: The correct solutions are $u(0) \approx 3.110$ and $v(0) = 2.081$.

Problem 4 Exponential integrator

A modern class of single step methods developed for special initial value problems that can be regarded as perturbed linear ODEs are the exponential integrators, see

> M. HOCHBRUCK AND A. OSTERMANN, *Exponential integrators*, Acta Numerica, 19 (2010), pp. 209–286.

These methods fit the concept of single step methods as introduced in [1, Def. 11.3.5] and, usually, converge algebraically according to [1, (11.3.20)].

A step with size h of the so-called *exponential Euler* single step method for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with continuously differentiable $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ reads

$$\mathbf{y}_1 = \mathbf{y}_0 + h\,\varphi\big(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0), \tag{60}$$

where $\mathsf{D}\,\mathbf{f}(\mathbf{y}) \in \mathbb{R}^{d,d}$ is the Jacobian of $\mathbf{f}$ at $\mathbf{y} \in \mathbb{R}^d$, and the matrix function $\varphi : \mathbb{R}^{d,d} \to \mathbb{R}^{d,d}$ is defined as $\varphi(\mathbf{Z}) = (\exp(\mathbf{Z}) - \mathsf{Id})\,\mathbf{Z}^{-1}$. Here $\exp(\mathbf{Z})$ is the matrix exponential of $\mathbf{Z}$, a special function $\exp : \mathbb{R}^{d,d} \to \mathbb{R}^{d,d}$, see [1, Eq. (12.1.32)].

The function φ is implemented in the provided file `ExpEul_template.cpp`. When plugging in the exponential series, it is clear that the function $z \mapsto \varphi(z) := \frac{\exp(z)-1}{z}$ is analytic on $\mathbb{C}$. Thus, $\varphi(\mathbf{Z})$ is well defined for all matrices $\mathbf{Z} \in \mathbb{R}^{d,d}$.

(4a) ⚀ Is the exponential Euler single step method defined in (60) consistent with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ (see [1, Def. 11.3.10])? Explain your answer.

(4b) ⚀ Show that the exponential Euler single step method defined in (60) solves the linear initial value problem

$$\dot{\mathbf{y}} = \mathbf{A}\,\mathbf{y}\,, \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^d\,, \qquad \mathbf{A} \in \mathbb{R}^{d,d}\,,$$

exactly.

HINT: Recall [1, Eq. (12.1.32)]; the solution of the IVP is $\mathbf{y}(t) = \exp(\mathbf{A}t)\mathbf{y}_0$. To facilitate formal calculations, you may assume that $\mathbf{A}$ is regular.

(4c) ⚁ Determine the region of stability of the exponential Euler single step method defined in (60) (see [1, Def. 12.1.49]).

(4d) ⚁ Write a C++ function

```
template <class Function, class Function2>
Eigen::VectorXd ExpEulStep(Eigen::VectorXd y0, Function
    f, Function2 df, double h)
```

that implements (60). Here `f` and `df` are objects with evaluation operators representing the ODE right-hand side function $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ and its Jacobian, respectively.

HINT: Use the supplied template `ExpEul_template.cpp`.

(4e) ⚂ What is the order of the single step method (60)? To investigate it, write a C++ routine that applies the method to the scalar logistic ODE

$$\dot{y} = y\,(1-y)\,, \quad y(0) = 0.1\,,$$

in the time interval $[0,1]$. Show the error at the final time against the stepsize $h = T/N$, $N = 2^k$ for $k = 1,\ldots,15$. As in Problem 2 in Problem Sheet 12, for each k compute and show an approximate order of convergence.

HINT: The exact solution is

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

Issue date: 10.12.2015

Hand-in: – (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 14

Problem 1 Implicit Runge-Kutta method (core problem)

This problem is the analogon of Problem 1, Problem Sheet 12, for general implicit Runge-Kutta methods [1, Def. 12.3.18]. We will adapt all routines developed for the explicit method to the implicit case. This problem assumes familiarity with [1, Section 12.3], and, especially, [1, Section 12.3.3] and [1, Rem. 12.3.24].

(1a) ⚄ By modifying the class `RKIntegrator`, design a header-only C++ class `implicit_RKIntegrator` which implements a general implicit RK method given through a Butcher scheme [1, Eq. (12.3.20)] to solve the autonomous initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$. The stages $\mathbf{g}_i$ as introduced in [1, Eq. (12.3.22)] are to be computed with the damped Newton method (see [1, Section 2.4.4]) applied to the nonlinear system of equations satisfied by the stages (see [1, Rem. 12.3.21] and [1, Rem. 12.3.24]). Use the provided code `dampnewton.hpp`, that is a simplified version of [1, Code 2.4.50]. Note that we do not use the simplified Newton method as discussed in [1, Rem. 12.3.24].

In the code template `implicit_rkintegrator_template.hpp` you will find all the parts from `rkintegrator_template.hpp` that you should reuse. In fact, you only have to write the method `step` for the implicit RK.

(1b) ⚁ Examine the code in `implicit_rk3prey.cpp`. Write down the complete Butcher scheme according to [1, Eq. (12.3.20)] for the implicit Runge-Kutta method defined there. Which method is it? Is it A-stable [1, Def. 12.3.32], L-stable [1, Def. 12.3.38]?

HINT: Scan the particular implicit Runge-Kutta single step methods presented in [1, Section 12.3].

(1c) ⚉ Test your implementation `implicit_RKIntegrator` of general implicit RK SSMs with the routine provided in the file `implicit_rk3prey.cpp` and comment on the observed order of convergence.

Problem 2 Initial Value Problem With Cross Product

We consider the initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \mathbf{a} \times \mathbf{y} + c\mathbf{y} \times (\mathbf{a} \times \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 = [1,1,1]^\top, \tag{61}$$

where $c > 0$ and $\mathbf{a} \in \mathbb{R}^3$, $\|a\|_2 = 1$.

NOTE: $\mathbf{x} \times \mathbf{y}$ denotes the cross product between the vectors $\mathbf{x}$ and $\mathbf{y}$. It is defined by

$$\mathbf{x} \times \mathbf{y} = [x_2y_3 - x_3y_2, x_3y_1 - x_1y_3, x_1y_2 - x_2y_1]^\top.$$

It satisfies $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$. In Eigen, it is available as `x.cross(y)`.

(2a) ⚀ Show that $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}_0\|_2$ for every solution $\mathbf{y}$ of (61).

HINT: Target the time derivative $\frac{d}{dt}\|\mathbf{y}(t)\|_2^2$ and use the product rule.

(2b) ⚁ Compute the Jacobian $D\mathbf{f}(\mathbf{y})$. Compute also the spectrum $\sigma(D\mathbf{f}(\mathbf{y}))$ in the stationary state $\mathbf{y} = \mathbf{a}$, for which $\mathbf{f}(\mathbf{y}) = 0$. For simplicity, you may consider only the case $\mathbf{a} = [1,0,0]^\top$.

(2c) ⚀ For $\mathbf{a} = [1,0,0]^\top$, (61) was solved with the standard MATLAB integrators `ode45` and `ode23s` up to the point $T = 10$ (default Tolerances). Explain the different dependence of the total number of steps from the parameter c observed in Figure 8.

(2d) ⚀ Formulate the non-linear equation given by the implicit mid-point rule for the initial value problem (61).

(2e) ⚁ Solve (61) with $\mathbf{a} = [1,0,0]^\top$, $c = 1$ up to $T = 10$. Use the implicit mid-point rule and the class developed for Problem 1 with $N = 128$ timesteps (use the template `cross_template.cpp`). Tabulate $\|\mathbf{y}_k\|_2$ for the sequence of approximate states generated by the implicit midpoint method. What do you observe?

(2f) ⚁ The linear-implicit mid-point rule can be obtained by a simple linearization of the incremental equation of the implicit mid-point rule around the current solution value.

Give the defining equation of the linear-implicit mid-point rule for the general autonomous differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$$

with smooth f.

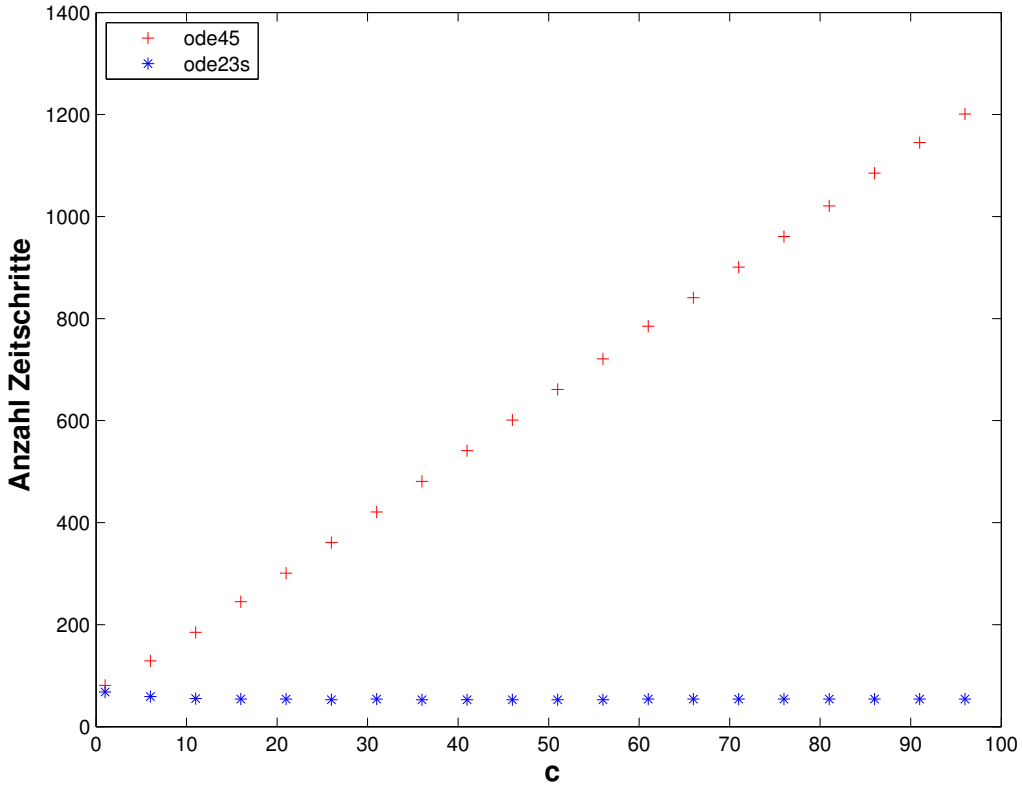


Figure 8: Subproblem (2c): number of steps used by standard MATLAB integrators in relation to the parameter c.

(2g) ⚄ Implement the linear–implicit midpoint rule using the template provided in `cross_template.cpp`. Use this method to solve (61) with $\mathbf{a} = [1,0,0]^\mathsf{T}$, $c = 1$ up to $T = 10$ and $N = 128$. Tabulate $\|\mathbf{y}_k\|_2$ for the sequence of approximate states generated by the linear implicit midpoint method. What do you observe?

Problem 3 Semi-implicit Runge-Kutta SSM (core problem)

General implicit Runge-Kutta methods as introduced in [1, Section 12.3.3] entail solving systems of non-linear equations for the increments, see [1, Rem. 12.3.24]. Semi-implicit Runge-Kutta single step methods, also known as Rosenbrock-Wanner (ROW) methods [1, Eq. (12.4.6)] just require the solution of linear systems of equations. This problem deals with a concrete ROW method, its stability and aspects of implementation.

We consider the following autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{62}$$

and discretize it with a *semi-implicit* Runge-Kutta SSM (*Rosenbrock method*):

$$\begin{aligned} \mathbf{W}\mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_0) \\ \mathbf{W}\mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\mathbf{k}_1) - ah\mathbf{J}\mathbf{k}_1 \\ \mathbf{y}_1 &= \mathbf{y}_0 + h\mathbf{k}_2 \end{aligned} \tag{63}$$

where

$$\begin{aligned} \mathbf{J} &= D\mathbf{f}(\mathbf{y}_0) \\ \mathbf{W} &= \mathbf{I} - ah\mathbf{J} \\ a &= \frac{1}{2+\sqrt{2}}. \end{aligned}$$

(3a) ⚄ Compute the stability function S of the Rosenbrock method (63), that is, compute the (rational) function $S(z)$, such that

$$y_1 = S(z)y_0, \quad z := h\lambda,$$

when we apply the method to perform one step of size h, starting from y_0, of the linear scalar model ODE $\dot{y} = \lambda y, \lambda \in \mathbb{C}$.

(3b) ⚄ Compute the first 4 terms of the Taylor expansion of $S(z)$ around $z = 0$. What is the maximal $q \in \mathbb{N}$ such that

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

for $|z| \to 0$? Deduce the maximal possible order of the method (63).

HINT: The idea behind this sub-problem is elucidated in [1, Rem. 12.1.19]. Apply [1, Lemma 12.1.21].

(3c) ⚄ Implement a C++ function:

```
template <class Func, class DFunc, class StateType>
std::vector<StateType> solveRosenbrock(
                const Func & f, const DFunc & df,
                const StateType & y0,
                unsigned int N, double T)
```

taking as input function handles for $\mathbf{f}$ and $D\mathbf{f}$ (e.g. as lambda functions), an initial data (vector or scalar) `y0` = $\mathbf{y}(0)$, a number of steps N and a final time T. The function returns the sequence of states generated by the single step method up to $t = T$, using N equidistant steps of the Rosenbrock method (63).

HINT: See `rosenbrock_template.cpp`.

(3d) ⚂ Explore the order of the method (63) empirically by applying it to the IVP for the limit cycle [1, Ex. 12.2.5]:

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2)\mathbf{y}, \tag{64}$$

with $\lambda = 1$ and initial state $\mathbf{y_0} = [1, 1]^\top$ on $[0, 10]$. Use fixed timesteps of size $h = 2^{-k}, k = 4, \ldots, 10$ and compute a reference solution with $h = 2^{-12}$ step size. Monitor the maximal mesh error:

$$\max_j \|\mathbf{y}_j - \mathbf{y}(t_j)\|_2.$$

(3e) ⚄ Show that the method (63) is L-stable (cf. [1, § 12.3.37]).

HINT: To investigate the A-stability, calculate the complex norm of $S(z)$ on the imaginary axis $\operatorname{Re} z = 0$ and apply the following maximum principle for holomorphic functions:

Theorem (Maximum principle for holomorphic functions)**.** Let

$$\mathbb{C}^- := \{z \in \mathbb{C} \mid Re(z) < 0\}.$$

Let $f : D \subset \mathbb{C} \to \mathbb{C}$ be non-constant, defined on $\overline{\mathbb{C}^-}$, and analytic in $\mathbb{C}^-$. Furthermore, assume that $w := \lim_{|z|\to\infty} f(z)$ exists and $w \in \mathbb{C}$, then:

$$\forall z \in \mathbb{C}^- |f(z)| < \sup_{\tau \in \mathbb{R}} |f(i\tau)|.$$

Problem 4 Singly Diagonally Implicit Runge-Kutta Method

SDIRK-methods (**S**ingly **D**iagonally **I**mplicit **R**unge-**K**utta methods) are distinguished by Butcher schemes of the particular form

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & \gamma & \cdots & 0 \\ c_2 & a_{21} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & \cdots & a_{s,s-1} \ \gamma \\ \hline & b_1 & \cdots & b_{s-1} \ b_s \end{array}, \tag{65}$$

with $\gamma \neq 0$.

More concretely, in this problem the scalar linear initial value problem of second order

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{66}$$

should be solved numerically using a SDIRK-method (**S**ingly **D**iagonally **I**mplicit **R**unge-**K**utta Method). It is a Runge-Kutta method described by the Butcher scheme

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & 1/2 & 1/2 \end{array}. \tag{67}$$

(4a) ⊡ Explain the benefit of using SDIRK-SSMs compared to using Gauss-Radau RK-SSMs as introduced in [1, Ex. 12.3.44]. In what situations will this benefit matter much?

HINT: Recall that in every step of an implicit RK-SSM we have to solve a non-linear system of equations for the increments, see [1, Rem. 12.3.24].

(4b) ⊡ State the equations for the increments $\mathbf{k}_1$ and $\mathbf{k}_2$ of the Runge-Kutta method (67) applied to the initial value problem corresponding to the differential equation $\dot{\boldsymbol{y}} = \mathbf{f}(t, \boldsymbol{y})$.

(4c) ⊡ Show that, the stability function $S(z)$ of the SDIRK-method (67) is given by

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

and plot the stability domain using the template `stabdomSDIRK.m`.

For $\gamma = 1$ is this method:

- A–stable?
- L–stable?

HINT: Use the same theorem as in the previous exercise.

(4d) ⊡ Formulate (66) as an initial value problem for a linear first order system for the function $\boldsymbol{z}(t) = (y(t), \dot{y}(t))^\intercal$.

(4e) ⊡ Implement a C++-function

```
template <class StateType>
StateType sdirtkStep(const StateType & z0, double h,
    double gamma);
```

that realizes the numerical evolution of one step of the method (67) for the differential equation determined in subsubsection (4d) starting from the value `z0` and returning the value of the next step of size `h`.

HINT: See `sdirk_template.cpp`.

(4f) ⊡ Use your C++ code to conduct a numerical experiment, which gives an indication of the order of the method (with $\gamma = \frac{3+\sqrt{3}}{6}$) for the initial value problem from subsubsection (4d). Choose $\boldsymbol{y}_0 = [1, 0]^\intercal$ as initial value, `T=10` as end time and `N=20,40,80,...,10240` as steps.

Issue date: 17.12.2015

Hand-in: – (in the boxes in front of HG G 53/54).

Version compiled on: February 14, 2016 (v. 1.0).

References

[1] R. Hiptmair. *Lecture slides for course "Numerical Methods for CSE"*. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE/NumCSE15.pdf. 2015.