Homework Problems for Course
Numerical Methods for CSE

R. Hiptmair, G. Alberti, F. Leonardi

Version of July 16, 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].

Solution: See `gramschmidt.cpp`.

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

Solution: See `gramschmidt.cpp`.

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.

Solution: See Listing 1.

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

Solution: See Listing 1.

(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.

Solution: See Listing 1.

Listing 1: EIGEN Implementation of the Strassen's algorithm and runtime comparisons.

```
#include <Eigen/Dense>
#include <iostream>
#include <vector>

#include "timer.h"

using namespace Eigen;
using namespace std;

//! \brief Compute the Matrix product A × B using Strassen's algorithm.
//! \param[in] A Matrix 2^k × 2^k
//! \param[in] B Matrix 2^k × 2^k
//! \param[out] Matrix product of A and B of dim 2^k × 2^k
MatrixXd strassenMatMult(const MatrixXd & A, const MatrixXd & B)
{
    int n=A.rows();
    MatrixXd C(n,n);

    if  (n==2)
    {
        C<< A(0,0)*B(0,0)  + A(0,1)*B(1,0),
            A(0,0)*B(0,1)  + A(0,1)*B(1,1),
            A(1,0)*B(0,0)  + A(1,1)*B(1,0),
            A(1,0)*B(0,1)  + A(1,1)*B(1,1);
        return C;
    }

    else
    {    MatrixXd
       Q0(n/2,n/2),Q1(n/2,n/2),Q2(n/2,n/2),Q3(n/2,n/2),
        Q4(n/2,n/2),Q5(n/2,n/2),Q6(n/2,n/2);

        MatrixXd A11=A.topLeftCorner(n/2,n/2);
        MatrixXd A12=A.topRightCorner(n/2,n/2);
        MatrixXd A21=A.bottomLeftCorner(n/2,n/2);
        MatrixXd A22=A.bottomRightCorner(n/2,n/2);

        MatrixXd B11=B.topLeftCorner(n/2,n/2);
        MatrixXd B12=B.topRightCorner(n/2,n/2);
        MatrixXd B21=B.bottomLeftCorner(n/2,n/2);
        MatrixXd B22=B.bottomRightCorner(n/2,n/2);

        Q0=strassenMatMult(A11+A22,B11+B22);
        Q1=strassenMatMult(A21+A22,B11);
        Q2=strassenMatMult(A11,B12-B22);
        Q3=strassenMatMult(A22,B21-B11);
        Q4=strassenMatMult(A11+A12,B22);
        Q5=strassenMatMult(A21-A11,B11+B12);
        Q6=strassenMatMult(A12-A22,B21+B22);

        C<< Q0+Q3-Q4+Q6  ,
        Q2+Q4,
        Q1+Q3,
        Q0+Q2-Q1+Q5;
        return C;
    }
```

```
}

int main(void)
{
    srand((unsigned int) time(0));

    //check if strassenMatMult works
    int k=2;
    int n=pow(2,k);
    MatrixXd A=MatrixXd::Random(n,n);
    MatrixXd B=MatrixXd::Random(n,n);
    MatrixXd AB(n,n), AxB(n,n);
    AB=strassenMatMult(A,B);
    AxB=A*B;
    cout<<"Using Strassen's method, A*B="<<AB<<endl;
    cout<<"Using standard method, A*B="<<AxB<<endl;
    cout<<"The norm of the error is
       "<<(AB-AxB).norm()<<endl;

    //compare runtimes of strassenMatMult and of direct
       multiplication

    unsigned int repeats = 10;
    timer<> tm_x, tm_strassen;
    std::vector<int> times_x, times_strassen;

    for(unsigned int k = 4; k <= 10; k++) {
        tm_x.reset();
        tm_strassen.reset();
        for(unsigned int r = 0; r < repeats; ++r) {
            unsigned int n = pow(2,k);
            A = MatrixXd::Random(n,n);
            B = MatrixXd::Random(n,n);
            MatrixXd AB(n,n);

            tm_x.start();
            AB=A*B;
            tm_x.stop();

            tm_strassen.start();
            AB=strassenMatMult(A,B);
            tm_strassen.stop();
        }
        std::cout << "The standard matrix multiplication took:       " << tm_x.min().count() / 1000000. << " ms" << std::endl;
        std::cout << "The Strassen's algorithm took:       " << tm_strassen.min().count() / 1000000. << " ms" << std::endl;

        times_x.push_back( tm_x.min().count() );
        times_strassen.push_back( tm_strassen.min().count() );
    }

    for(auto it = times_x.begin(); it != times_x.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    for(auto it = times_strassen.begin(); it != times_strassen.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

}
```

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 2 implements a particular MATLAB function.

Listing 2: 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.

Solution:

Listing 3: C++implementation for Problem 3 in file `houserefl.cpp`

```
void houserefl(const Eigen::VectorXd & v,
   Eigen::MatrixXd & Z)
{
    unsigned int n = v.size();
    Eigen::VectorXd w = v.normalized();
    Eigen::VectorXd u=w;
    u(0) += 1;
    Eigen::VectorXd  q=u.normalized();
    Eigen::MatrixXd X = Eigen::MatrixXd::Identity(n, n)
       - 2*q*q.transpose();
    Z = X.rightCols(n-1);
}
```

(3b) ⚂ Show that the matrix $\mathbf{X}$, defined at line 10 in Listing 2, satisfies:

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

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

Solution:

$$\begin{aligned}\mathbf{X}^\top\mathbf{X} &= (\mathbf{I}_n - 2\mathbf{q}\mathbf{q}^\top)(\mathbf{I}_n - 2\mathbf{q}\mathbf{q}^\top)\\ &= \mathbf{I}_n - 4\mathbf{q}\mathbf{q}^\top + 4\mathbf{q}\ \underbrace{\mathbf{q}^\top\mathbf{q}}_{=\|\mathbf{q}\|=1}\ \mathbf{q}^\top\\ &= \mathbf{I}_n - 4\mathbf{q}\mathbf{q}^\top + 4\mathbf{q}\mathbf{q}^\top\\ &= \mathbf{I}_n\end{aligned}$$

(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 $\|\mathbf{w}\| = 1$.

Solution: Let $\mathbf{X} = [\mathbf{X}_1, \cdots, \mathbf{X}_n]$ be the matrix of line 9 in Listing 2. In view of the identity $\mathbf{X}_1 = \mathbf{e}^{(1)} - 2q_1\mathbf{q}$ we have

$$\mathbf{X}_1 = \begin{bmatrix}1-2q_1^2\\-2q_1q_2\\\vdots\\-2q_1q_n\end{bmatrix} = \begin{bmatrix}1-2\frac{u_1^2}{\sum_{i=1}^n u_i^2}\\-2\frac{u_1u_2}{\sum_{i=1}^n u_i^2}\\\vdots\\-2\frac{u_1u_n}{\sum_{i=1}^n u_i^2}\end{bmatrix} \overset{\text{HINT}}{=} \begin{bmatrix}\frac{(w_1+1)^2+w_2^2+\cdots+w_n^2-2(w_1+1)^2}{(w_1+1)^2+w_2^2+\cdots+w_n^2}\\-\frac{2(w_1+1)w_2}{(w_1+1)^2+w_2^2+\cdots+w_n^2}\\\cdots\\-\frac{2(w_1+1)w_n}{(w_1+1)^2+w_2^2+\cdots+w_n^2}\end{bmatrix} \overset{\|\mathbf{w}\|=1}{=} \begin{bmatrix}\frac{2w_1(w_1+1)}{2(w_1+1)}\\\frac{2(w_1+1)w_2}{2(w_1+1)}\\\cdots\\\frac{2(w_1+1)w_n}{2(w_1+1)}\end{bmatrix} = -\mathbf{w},$$

which is a multiple of $\mathbf{v}$, since $\mathbf{w} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$.

(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).

Solution: The columns of $\mathbf{X} = [\mathbf{X}_1, \cdots, \mathbf{X}_n]$ are an orthonormal basis (ONB) of $\mathbb{R}^n$ (cf. (3b)). Thus, the columns of $\mathbf{Z} = [\mathbf{X}_2, \cdots, \mathbf{X}_n]$ are an ONB of the complement of $\text{Span}(\mathbf{X}_1) \overset{(3c)}{=} \text{Span}(\mathbf{v})$. The function `houserefl` computes an ONB of the complement of $\mathbf{v}$.

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

Solution: $O(n^2)$: this is the asymptotic complexity of the construction of the tensor product at line 9 of Listing 3.

(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.

Solution: Check the code in Listing 2 for the porting to MATLAB code. Using the QR-decomposition `qr` one can rewrite (cf. Listing 4) the C++ code in MATLAB with a few lines.

Listing 4: MATLAB implementation for Problem 3 in file `qr_houserefl.m` using QR decomposition.

```matlab
function Z = qr_houserefl(v)
% Use qr decomposition to find ONB of complement of span(v)
    [X,R] = qr(v);

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

Issue date: 21.0.9.2015

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

Version compiled on: July 16, 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 5: 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 $d = (d_1, \dots, d_n)^\top$ and $a = (a_1, \dots, a_n)^\top$, sketch the matrix $\mathbf{A}$ created in line 6 of Listing 5.

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

Solution: $\mathbf{A} = \begin{pmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{pmatrix}$

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

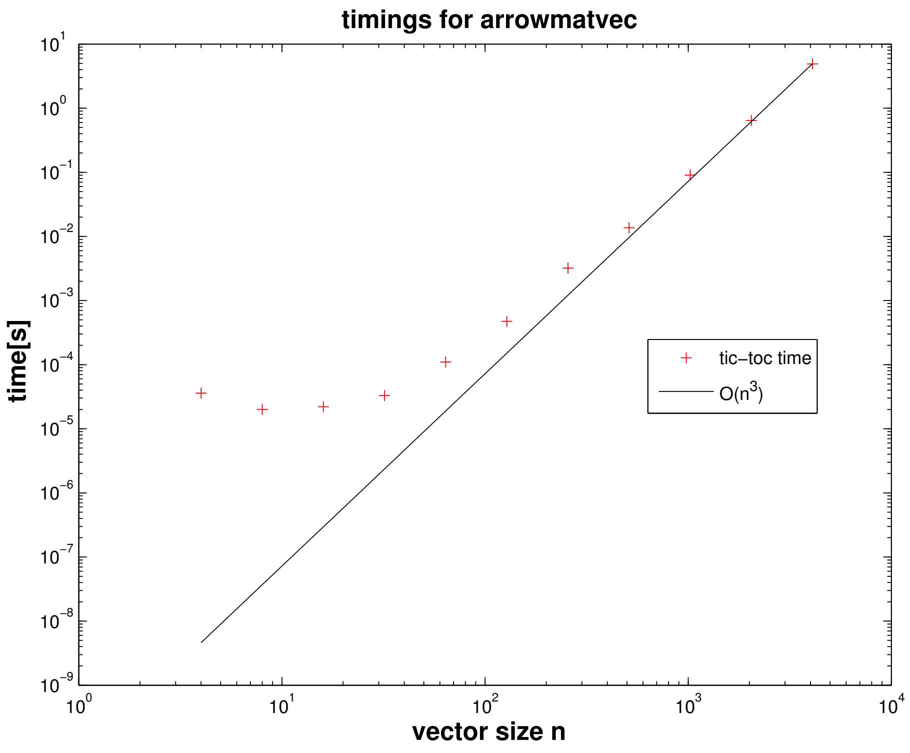


Figure 1: timings for `arrowmatvec(d,a,x)`

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

Solution: The standard matrix-matrix multiplication has runtimes growing with $O(n^3)$ and the standard matrix-vector multiplication has runtimes growing with $O(n^2)$. Hence, the overall computational complexity is dominated by $O(n^3)$.

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

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

that computes the same multiplication as in code 5 but with optimal asymptotic complexity with respect to n. Here `d` passes the vector $(d_1, \dots, d_n)^T$ and `a` passes the vector $(a_1, \dots, a_n)^T$.

Solution: Due to the sparsity and special structure of the matrix, it is possible to write a more efficient implementation than the standard matrix-vector multiplication. See code listing 6

Listing 6: implementation of the function `arrowmatvec2`

```
function y = arrowmatvec2(d,a,x)
if (length(d) ≠ length(a)), error ('size mismatch'); end
% Recursive matrix-vector multiplication to obtain A*A*x = A*(A*x)
y = A(d,a,A(d,a,x));
end

% Efficient multiplication of a vector with the ``arrow matrix''
function Ax = A(d,a,x)
Ax = d.*x;
Ax(1:end-1) = Ax(1:end-1) + a(1:end-1)*x(end);
Ax(end) = Ax(end) + a(1:end-1)'*x(1:end-1);
end
```

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

Solution: The efficient implementation only needs two vector-vector element-wise multiplications and one vector-scalar multiplication. Therefore the complexity is $O(n)$.

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

Solution: The standard matrix multiplication has runtimes growing with $O(n^3)$. The runtimes of the more efficient implementation are growing with $O(n)$. See Listing 7 and Figure 2.

Listing 7: Execution and timings of `arrowmatvec` and `arrowmatvec2`

```
nruns = 3; res = [];
```

```
ns = 2.^(2:12);
for n = ns
  a  = rand(n,1); d = rand(n,1); x = rand(n,1);
  t  = realmax;
  t2 = realmax;
  for k=1:nruns
    tic; y  = arrowmatvec(d,a,x);  t  = min(toc,t);
    tic; y2 = arrowmatvec2(d,a,x); t2 = min(toc,t2);
  end;
  res = [res; t t2];
end
figure('name','timings arrowmatvec and arrowmatvec2');
c1 = sum(res(:,1))/sum(ns.^3);
c2 = sum(res(:,2))/sum(ns);
loglog(ns, res(:,1),'r+', ns, res(:,2),'bo',...
       ns, c1*ns.^3, 'k-', ns, c2*ns, 'g-');
xlabel('{\bf vector size n}','fontsize',14);
ylabel('{\bf time[s]}','fontsize',14);
title('{\bf timings for arrowmatvec and
    arrowmatvec2}','fontsize',14);
legend('arrowmatvec','arrowmatvec2','O(n^3)','O(n)',...
       'location','best');
print -depsc2 '../PICTURES/arrowmatvec2timing.eps';
```

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

Solution: See Listing 8 and Listing 9.

Listing 8: Implementation of `arrowmatvec` in EIGEN

```
#include <Eigen/Dense>
#include <iostream>
#include <ctime>

using namespace Eigen;

template <class Matrix>
```

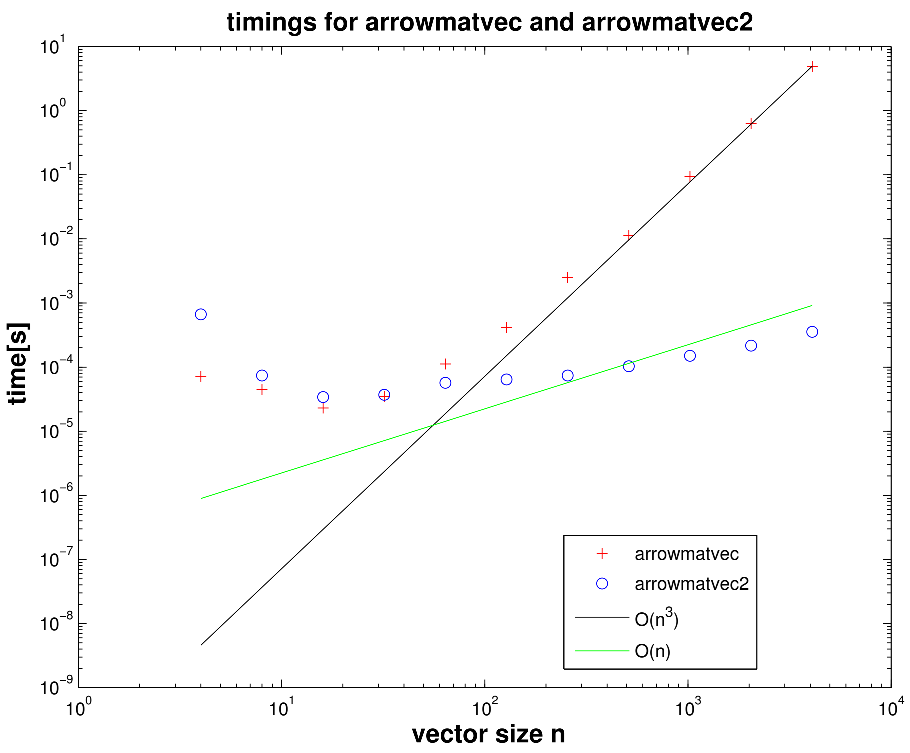


Figure 2: timings for `arrowmatvec2(d,a,x)`

```

// The function arrowmatvec computes the product the
   desired product directly as A*A*x
void arrowmatvec(const Matrix & d, const Matrix & a,
   const Matrix & x, Matrix & y)
{
    // Here, we choose a MATLAB style implementation
       using block construction
    // you can also use loops
    // If you are interested you can compare both
       implementation and see if and how they differ
    int n=d.size();
    VectorXd dcut= d.head(n-1);
    VectorXd acut = a.head(n-1);
    MatrixXd ddiag=dcut.asDiagonal();
    MatrixXd A(n,n);
    MatrixXd D = dcut.asDiagonal();
    // If you do not create the temporary matrix D, you
       will get an error: D must be casted to MatrixXd
    A << D, a.head(n-1), acut.transpose(), d(n-1);

    y=A*A*x;
}

// We test the function arrowmatvec with 5 dimensional
   random vectors.

int main(void)
{
//      srand((unsigned int) time(0));
    VectorXd a=VectorXd::Random(5);
    VectorXd d=VectorXd::Random(5);
    VectorXd x=VectorXd::Random(5);
    VectorXd y;

    arrowmatvec(d,a,x,y);
    std::cout << "A*A*x = " << y << std::endl;
}
```

Listing 9: Implementation of `arrowmatvec2` in EIGEN

```
#include <Eigen/Dense>
#include <iostream>

using namespace Eigen;

template <class Matrix>

// The auxiliary function Atimesx computes the function
   A*x in a smart way, using the particular structure of
   the matrix A.

void Atimesx(const Matrix & d, const Matrix & a, const
    Matrix & x, Matrix & Ax)
{
     int n=d.size();
     Ax=(d.array()*x.array()).matrix();
     VectorXd Axcut=Ax.head(n-1);
     VectorXd acut = a.head(n-1);
     VectorXd xcut = x.head(n-1);

     Ax << Axcut + x(n-1)*acut, Ax(n-1)+
        acut.transpose()*xcut;
}

// We compute A*A*x by using the function Atimesx twice
    with 5 dimensional random vectors.

int main(void)
{
     VectorXd a=VectorXd::Random(5);
     VectorXd d=VectorXd::Random(5);
     VectorXd x=VectorXd::Random(5);
     VectorXd Ax(5);

     Atimesx(d,a,x,Ax);
     VectorXd AAx(5);
     Atimesx(d,a,Ax,AAx);
     std::cout << "A*A*x = " << AAx << std::endl;
}
```

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) $\boxdot$ 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.

Solution: Check the MATLAB implementation in Listing 10 and the plot in Fig. 3. We can clearly observe that the computation using f_1 leads to a big error as $h \to 0$. This is due to the cancellation error given by the subtraction of two number of approximately same magnitude. The second implementation using f_2 is very stable and does not display round-off errors.

Listing 10: MATLAB script for Problem 2

```
%% Cancellation
h = 10.^(-20:0);
x = 1.2;

% Derivative
```

```
g1 = (sin(x+h) - sin(x)) ./ h; % Naive
g2 = 2 .* cos(x + h * 0.5) .* sin(h * 0.5) ./ h; % Better
ex = cos(x); % Exact

% Plot
loglog(h, abs(g1-ex), 'r',h, abs(g2-ex), 'b', h, h,
   'k--');
title('Error of the approximation of f''(x_0)');
legend('g_1','g_2', 'O(h)');
xlabel('h');
ylabel('| f''(x_0) - g_i(x_0,h) |');
grid on
```

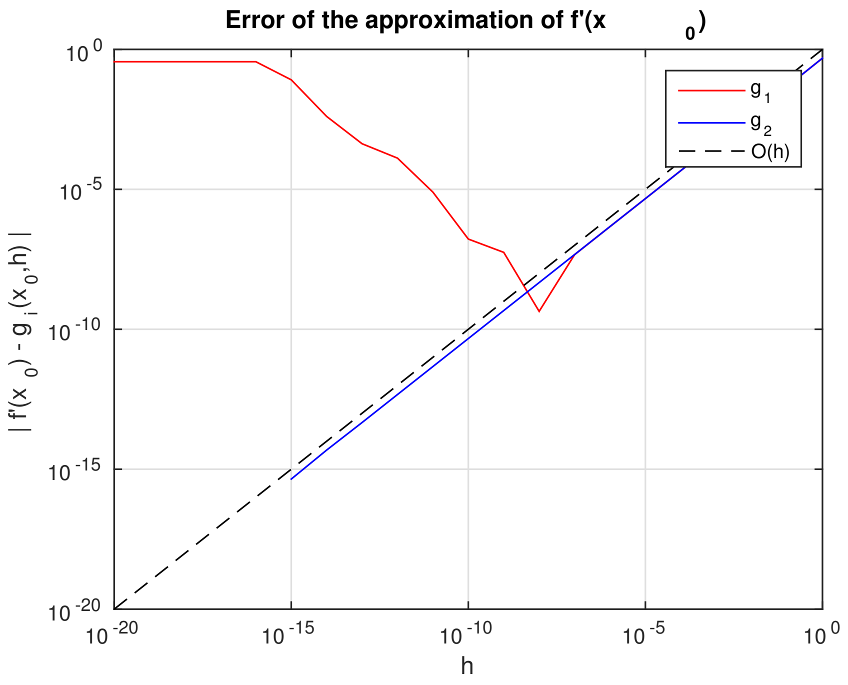


Figure 3: Timings for Problem 2

(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.

Solution: We immediately derive $\ln(x-\sqrt{x^2-1})+\ln(x+\sqrt{x^2-1}) = \log(x^2-(x^2-1)) = 0$. As $x \to \infty$ the left $\log$ consists of subtraction of two numbers of equal magnitude, whilst the right $\log$ consists on the addition of two numbers of approximately the same magnitude. Therefore, in the first case there may be cancellation for large values of x, making it worse for numerical computation. Try, in MATLAB, with $x = 10^8$.

(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 $a \approx 0, b \approx 1$.

Solution:

1. Inside the square roots we have the addition (rest. subtraction) of a small number to a big number. The difference of the square roots incur in cancellation, since they have the same, large magnitude. $A := x+\frac{1}{x}, B := x-\frac{1}{x}$ then $(A-B)(A+B)/(A+B) = \frac{2/x}{\sqrt{x+\frac{1}{x}}+\sqrt{x-\frac{1}{x}}} = \frac{2}{\sqrt{x}(\sqrt{x^2+1}+\sqrt{x^2-1})}$
2. $\frac{1}{a^2}$ becomes very large as a approaches 0, whilst $\frac{1}{b^2} \to 1$ as $b \to 1$. Therefore, the relative size of $\frac{1}{a^2}$ and $\frac{1}{b^2}$ becomes so big, that, in computer arithmetic, $\frac{1}{a^2}+\frac{1}{b^2} = \frac{1}{a^2}$. On the other hand $\frac{1}{a}\sqrt{1+(\frac{a}{b})^2}$ avoids this problem by performing a division between two numbers with very different magnitude, instead of a summation.

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

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

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.

Solution: See MATLAB help.

(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 $\mathbf{B} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$.

Solution: $\mathbf{y} = \begin{pmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{pmatrix} \mathbf{x}$.

(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.

Solution: `kron(A,B)` results in a matrix of size $n^2 \times n^2$ and x has length n^2. So the complexity is the same as a matrix-vector multiplication for the resulting sizes. In total this is $O(n^2 * n^2) = O(n^4)$.

(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.

Solution: Since `kron(A,B)` creates a large matrix consisting of smaller blocks with size n, i.e. B multiplied with $A(i,j)$, we can split the problem up in n matrix-vector multiplications of size n. This results in a routine with complexity $n * O(n^2) = O(n^3)$ The implementation is listed in 11. The runtimes are shown in Figure 4.

Listing 11: An efficient implementation for Problem 3

```matlab
function y = Kron_B(A,B,x)
% Return y = kron(A,B)*x  (smart version)
% Input:   A,B:     2 n x n matrices.
%          x:       Vector of length n*n.
% Output:  y:       Result vector of length n*n.

% check size of A
[n,m] = size(A);
assert(n == m, 'expected quadratic matrix')
```

```matlab

% kron gives a matrix with n x n blocks
% block i,j is A(i,j)*B
% => y = M*x can be done block-wise so that we reuse B*x(...)

% init
y = zeros(n*n,1);
% loop first over columns and then (!) over rows
for j = 1:n
    % reuse B*x(...) part (constant in given column) => O(n^2)
    z = B*x((j-1)*n+1:j*n);
    % add to result vector (need to go through full vector) => O(n^2)
    for i = 1:n
        y((i-1)*n+1:i*n) = y((i-1)*n+1:i*n) + A(i,j)*z;
    end
end
% Note: complexity is O(n^3)
end
```

(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 $n = 2^{3,4,5,6}$.

Solution:

Listing 12: A second efficient implementation for Problem 3 using `reshape`.

```matlab
function y = Kron_C (A,B,x)
% Return y = kron(A,B)*x  (smart version with reshapes)
% Input:   A,B:        2 n x n matrices.
%            x:        Vector of length n*n.
% Output:    y:        Result vector of length n*n.

% check size of A
```

```
[n,m] = size(A);  assert(n == m, 'expected quadratic
   matrix')

% init
yy = zeros(n,n);
xx = reshape(x,n,n);

% precompute the multiplications of B with the parts of
   vector x
Z = B*xx;
for j=1:n
    yy = yy + Z(:,j) * A(:,j)';
end
y = reshape(yy,n^2,1);
end
% Notes: complexity is O(n^3)
```

Listing 13: Main routine for runtime measurements of Problem 3

```
%% Kron (runtimes)
clear all; close all;
nruns = 3;  % we average over a few runs
N = 2.^(2:10);
T = zeros(length(N),5);
for i = 1:length(N)
    n = N(i)    % problem size n
    % % initial matrices A,B and vector x
    % A = reshape(1:n*n, n, n)';
    % B = reshape(n*n+1:2*n*n, n, n)';
    % x = (1:n*n)';
    % alternative choice:
    A=rand(n);    B=rand(n);    x=rand(n^2,1);

    tic;    % smart implementation #1
    for ir = 1:nruns
        yB = Kron_B(A,B,x);
    end
    tb = toc/nruns;

    tic;   % smart implementation #2: with reshapes
    for ir = 1:nruns
        yC = Kron_C(A,B,x);
    end
    tc = toc/nruns;
    fprintf('Error B-vs-C:   %g\n', norm(yB-yC))

    tic;   % implementation with kron and matrix*vector
    if(N(i)<128)
        for ir = 1:nruns
            yA = kron(A,B)*x;
        end
        ta = toc/nruns;
        fprintf('Error A-vs-B:   %g\n', norm(yA-yB))
    else
        ta=0;
    end

    tic;  % implementation with 1-line command!
          % inspired by the solution of Kevin Bocksrocker
    for ir = 1:nruns
        yD = reshape(B*reshape(x,n,n)*A',n^2,1);
    end
    td = toc/nruns;
    fprintf('Error D-vs-B:   %g\n', norm(yD-yB));

    fprintf('Timings: Matrix: %g\n         Smart:
       %g\n',ta,tb)
    fprintf('         Smart+Reshape:  %g\n
       1-line Smart:  %g\n',tc,td)
    T(i,:) = [n, ta, tb, tc, td];
end

% log-scale plot for investigation of asymptotic
    complexity
a2 = sum(T(:,3)) / sum(T(:,1).^2);
a3 = sum(T(:,3)) / sum(T(:,1).^3);
```

```
a4 = sum(T(1:5,2)) / sum(T(1:5,1).^4);
figure('name','kron timings');
loglog(T(:,1),T(:,2),'m+',  T(:,1),T(:,3),'ro',...
       T(:,1),T(:,4),'bd',  T(:,1),T(:,5),'gp',...
       T(:,1),(T(:,1).^2)*a2,'k-',
          T(:,1),(T(:,1).^3)*a3,'k--',...
       T(1:5,1),(T(1:5,1).^4)*a4,'k-.', 'linewidth', 2);
xlabel('{\bf problem size n}','fontsize',14);
ylabel('{\bf average runtime (s)}','fontsize',14);
title(sprintf('tic-toc timing averaged over %d runs',
   nruns),'fontsize',14);
legend('slow evaluation','efficient evaluation',...
    'efficient ev. with reshape','Kevin 1-line',...
    'O(n^2)','O(n^3)','O(n^4)','location','northwest');
print -depsc2 '../PICTURES/kron_timings.eps';
```

(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`.

Solution: See `kron.cpp` or Listing 14.

(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.

Solution: See `kron.cpp` or Listing 14.

(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).

Solution:

Listing 14: Main routine for runtime measurements of Problem 3

```cpp
#include <Eigen/Dense>
#include <iostream>
#include <vector>

#include "timer.h"

//! \brief Compute the Kronecker product $C = A \otimes B$.
//! \param[in] A Matrix $n \times n$
//! \param[in] B Matrix $n \times n$
//! \param[out] C Kronecker product of A and B of dim
    $n^2 \times n^2$
template <class Matrix>
void kron(const Matrix & A, const Matrix & B, Matrix & C)
{
    C = Matrix(A.rows()*B.rows(), A.cols()*B.cols());
    for(unsigned int i = 0; i < A.rows(); ++i) {
        for(unsigned int j = 0; j < A.cols(); ++j) {
            C.block(i*B.rows(),j*B.cols(), B.rows(),
               B.cols()) = A(i,j)*B;
        }
    }
}

```

```
//! \brief Compute the Kronecker product C = $A \otimes B$.
    Exploit matrix-vector product.
//! A,B and x must have dimension n \times n resp. n
//! \param[in] A Matrix $n \times n$
//! \param[in] B Matrix $n \times n$
//! \param[in] x Vector of dim $n$
//! \param[out] y Vector y = kron(A,B)*x
template <class Matrix, class Vector>
void kron_fast(const Matrix & A, const Matrix & B, const
    Vector & x, Vector & y)
{
    y = Vector::Zero(A.rows()*B.rows());

    unsigned int n = A.rows();
    for(unsigned int j = 0; j < A.cols(); ++j) {
        Vector z = B * x.segment(j*n, n);
        for(unsigned int i = 0; i < A.rows(); ++i) {
            y.segment(i*n,n) += A(i,j)*z;
        }
    }
}

//! \brief Compute the Kronecker product $C = A \otimes B$. Uses
    fast remapping tecniques (similar to Matlab reshape)
//! A,B and x must have dimension n \times n resp. n*n
//! Elegant way using reshape
//! WARNING: using Matrix::Map we assume the matrix is
    in ColMajor format, *beware* you may incur in bugs if
    matrix is in RowMajor isntead
//! \param[in] A Matrix $n \times n$
//! \param[in] B Matrix $n \times n$
//! \param[in] x Vector of dim $n$
//! \param[out] y Vector y = kron(A,B)*x
template <class Matrix, class Vector>
void kron_super_fast(const Matrix & A, const Matrix & B,
    const Vector & x, Vector & y)
{
    unsigned int n = A.rows();
    Matrix t = B * Matrix::Map(x.data(),n,n) *
       A.transpose();
    y = Matrix::Map(t.data(), n*n, 1);
}

int main(void) {

    // Check if kron works, cf.
    Eigen::MatrixXd A(2,2);
    A << 1, 2, 3, 4;
    Eigen::MatrixXd B(2,2);
    B << 5, 6, 7, 8;
    Eigen::MatrixXd C;

    Eigen::VectorXd x = Eigen::VectorXd::Random(4);
    Eigen::VectorXd y;
    kron(A,B,C);
    y = C*x;
    std::cout << "kron(A,B)=" << std::endl << C <<
       std::endl;
    std::cout << "Using kron: y=       " << std::endl <<
       y << std::endl;

    kron_fast(A,B,x,y);
    std::cout << "Using kron_fast: y=  " << std::endl <<
       y << std::endl;
    kron_super_fast(A,B,x,y);
    std::cout << "Using kron_super_fast: y=  " <<
       std::endl << y << std::endl;

    // Compute runtime of different implementations of
       kron
    unsigned int repeats = 10;
    timer<> tm_kron, tm_kron_fast, tm_kron_super_fast;
    std::vector<int> times_kron, times_kron_fast,
       times_kron_super_fast;

    for(unsigned int p = 2; p ≤ 9; p++) {
        tm_kron.reset();
        tm_kron_fast.reset();
        tm_kron_super_fast.reset();
        for(unsigned int r = 0; r < repeats; ++r) {
            unsigned int M = pow(2,p);
            A = Eigen::MatrixXd::Random(M,M);
            B = Eigen::MatrixXd::Random(M,M);
            x = Eigen::VectorXd::Random(M*M);

            // May be too slow for large p, comment if so
            tm_kron.start();
//              kron(A,B,C);
//              y = C*x;
            tm_kron.stop();

            tm_kron_fast.start();
            kron_fast(A,B,x,y);
            tm_kron_fast.stop();

            tm_kron_super_fast.start();
            kron_super_fast(A,B,x,y);
            tm_kron_super_fast.stop();
        }

        std::cout << "Lazy Kron took:       " <<
           tm_kron.min().count() / 1000000. << " ms" <<
           std::endl;
        std::cout << "Kron fast took:       " <<
           tm_kron_fast.min().count() / 1000000. << "
           ms" << std::endl;
        std::cout << "Kron super fast took: " <<
           tm_kron_super_fast.min().count() / 1000000.
           << " ms" << std::endl;
        times_kron.push_back( tm_kron.min().count() );
        times_kron_fast.push_back(
           tm_kron_fast.min().count() );
        times_kron_super_fast.push_back(
           tm_kron_super_fast.min().count() );
```

```
    }

    for(auto it = times_kron.begin(); it !=
       times_kron.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    for(auto it = times_kron_fast.begin(); it !=
       times_kron_fast.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    for(auto it = times_kron_super_fast.begin(); it !=
       times_kron_super_fast.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
}
```

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 $i, j = 1, \dots, n$. The matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ can be implemented in MATLAB as

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

(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?

Solution: Matrix–vector multiplication: quadratic dependence $O(n^2)$.

(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.

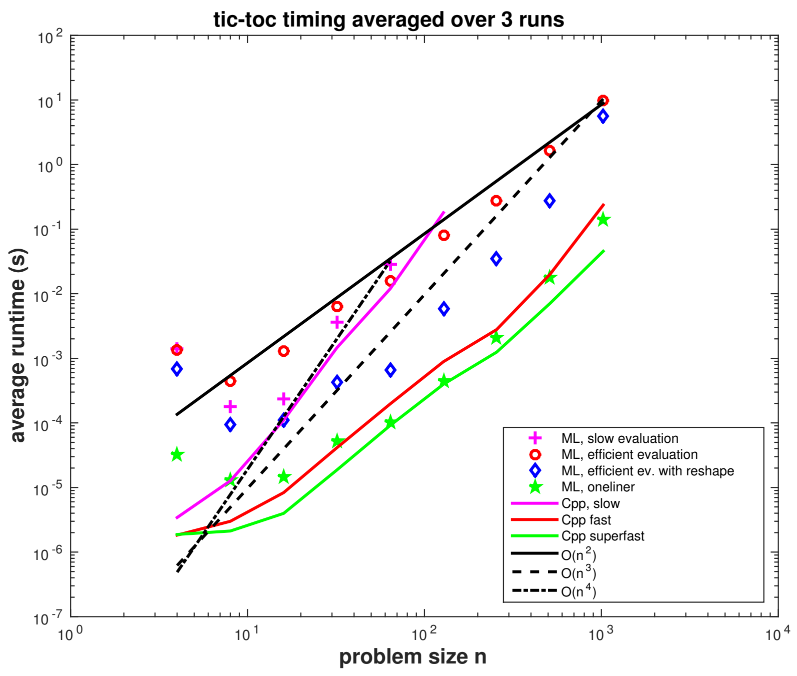


Figure 4: Timings for Problem 3 with MATLABand C++implementations.

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

Solution: For every j we have $y_j = \sum_{k=1}^{j} kx_k + j\sum_{k=j+1}^{n} x_k$, so we pre-compute the two terms for every j only once.

Listing 15: implementation for the function multAmin

```
function y = multAmin(x)
% O(n), slow version
n = length(x);
y = zeros(n,1);
v = zeros(n,1);
w = zeros(n,1);

v(1) = x(n);
w(1) = x(1);
```

```
for j = 2:n
    v(j) = v(j-1)+x(n+1-j);
    w(j) = w(j-1)+j*x(j);
end
for j = 1:n-1
    y(j) = w(j) + v(n-j)*j;
end
y(n) = w(n);

% To check the code, run:
% n=500; x=randn(n,1); y = multAmin(x);
% norm(y - min(ones(n,1)*(1:n), (1:n)'*ones(1,n)) * x)
```

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

Solution: Linear dependence: $O(n)$.

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

Solution:

The matrix multiplication in (4) has runtimes growing with $O(n^2)$. The runtimes of the more efficient implementation with hand-coded loops, or using the MATLABfunction `cumsum` are growing with $O(n)$.

Listing 16: comparison of execution timings

```
ps = 4:12;
ns = 2.^ps;
ts = 1e6*ones(length(ns),3);      % timings
nruns = 10;                       % to average the runtimes

% loop over different Problem Sizes
for j=1:length(ns)
    n = ns(j);
    fprintf('Vector length: %d \n', n);
    x = rand(n,1);

    % timing for naive multiplication
    tic;
    for k=1:nruns
      y = min(ones(n,1)*(1:n), (1:n)'*ones(1,n)) * x;
    end
    ts(j,1) = toc/nruns;

    % timing multAmin
    tic;
    for k=1:nruns
      y = multAmin(x);
    end
    ts(j,2) = toc/nruns;

    % timing multAmin2
    tic;
    for k=1:nruns
      y = multAmin2(x);
    end
    ts(j,3) = toc/nruns;
end

c1 = sum(ts(:,2)) / sum(ns);
c2 = sum(ts(:,1)) / sum(ns.^2);

loglog(ns, ts(:,1), '-k', ns, ts(:,2), '-og', ns,
   ts(:,3), '-xr',...
       ns, c1*ns, '-.b', ns, c2*ns.^2, '--k',
          'linewidth', 2);
legend('naive','multAmin','multAmin2','O(n)','O(n^2)',...
       'Location','NorthWest')
title(sprintf('tic-toc timing averaged over %d runs',
   nruns),'fontsize',14);
xlabel('{\bf problem size n}','fontsize',14);
ylabel('{\bf runtime (s)}','fontsize',14);

print -depsc2 '../PICTURES/multAmin_timings.eps';
```

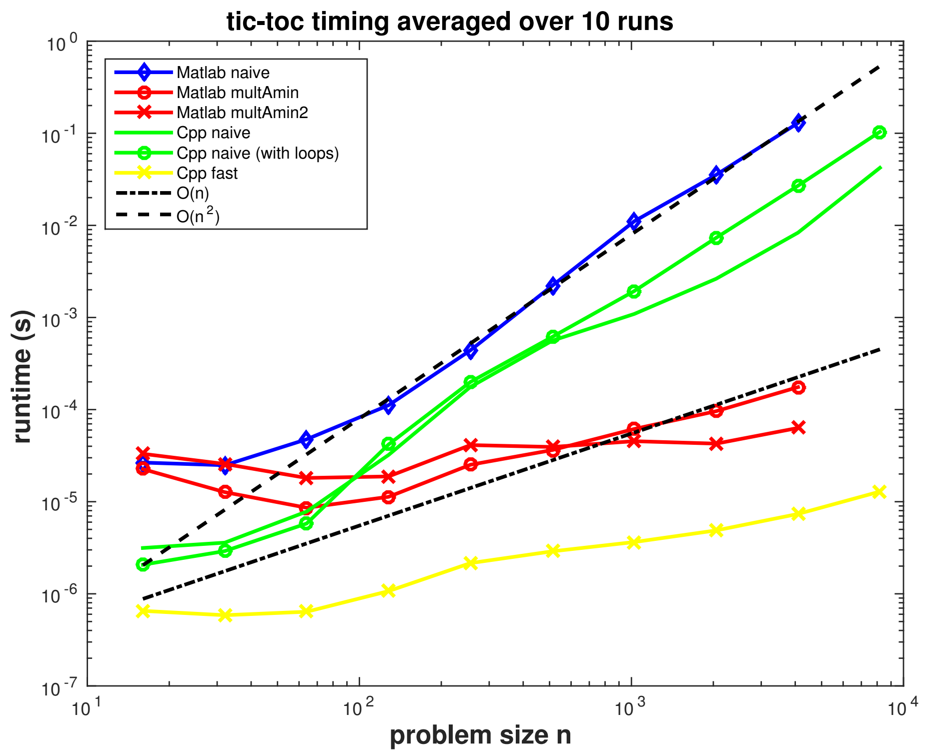


Figure 5: Timings for different implementations of $\mathbf{y} = \mathbf{A}\mathbf{x}$ with both MATLABand C++.

(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`.

Solution: Using `cumsum` to avoid the `for` loops:

Listing 17: implementation for the function multAmin without loops

```
function y = multAmin2(x)
% O(n), no-for version
n = length(x);
v = cumsum(x(end:-1:1));
w = cumsum(x.*(1:n)');
y = w + (1:n)'.*[v(n-1:-1:1);0];
```

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

Listing 18: MATLABscript calling `multAmin`

```
n = 10;
B = diag(-ones(n-1,1),-1)+diag([2*ones(n-1,1);1],0)...
     + diag(-ones(n-1,1),1);
x = rand(n,1);
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`.

Solution:

$$\mathbf{B} := \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 1 \end{pmatrix}$$

Notice the value 1 in the entry (n,n).

(4g) ⚁ Run the code of Listing 18 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].

Solution: It is easy to verify with MATLAB(or to prove) that $\mathbf{B} = \mathbf{A}^{-1}$.
For $2 \le j \le n-1$, we obtain:

$$(\mathbf{AB})_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} = a_{i,j-1} b_{j-1,j} + 2 a_{i,j} b_{j,j} + a_{i,j+1} b_{j+1,j}$$

$$= -\min(i,j-1) + 2\min(i,j) - \min(i,j+1) = \begin{cases} -i + 2i - i = 0 & \text{if } i < j, \\ -(i-1) + 2i - i = 1 & \text{if } i = j, \\ -(j-1) + 2j - (j+1) = 0 & \text{if } i > j. \end{cases}$$

Furthermore, $(\mathbf{AB})_{i,1} = \delta_{i,1}$ and $(\mathbf{AB})_{i,n} = \delta_{i,n}$, hence $\mathbf{AB} = \mathbf{I}$.
The last line of `multAB.m` prints the value of $\|\mathbf{A}\,\mathbf{B}\,\mathbf{x} - \mathbf{x}\| = \|\mathbf{x} - \mathbf{x}\| = 0$.
The returned values are not exactly zero due to round-off errors.

(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.

Solution:

Listing 19: C++script implementing `multAmin`

```
#include <Eigen/Dense>

#include <iostream>
#include <vector>

#include "timer.h"

//! \brief build A*x using array of ranges and of ones.
//! \param[in] x vector x for A*x = y
//! \param[out] y y = A*x
void multAminSlow(const Eigen::VectorXd & x, Eigen::VectorXd & y) {
    unsigned int n = x.size();

    Eigen::VectorXd one = Eigen::VectorXd::Ones(n);
    Eigen::VectorXd linsp = Eigen::VectorXd::LinSpaced(n,1,n);
    y = ( (one * linsp.transpose()).cwiseMin(linsp * one.transpose()) ) *x;
}

//! \brief build A*x using a simple for loop
//! \param[in] x vector x for A*x = y
//! \param[out] y y = A*x
void multAminLoops(const Eigen::VectorXd & x, Eigen::VectorXd & y) {
    unsigned int n = x.size();

    Eigen::MatrixXd A(n,n);

    for(unsigned int i = 0; i < n; ++i) {
        for(unsigned int j = 0; j < n; ++j) {
            A(i,j) = std::min(i+1,j+1);
        }
    }
    y = A * x;
}

//! \brief build A*x using a clever representation
//! \param[in] x vector x for A*x = y
//! \param[out] y y = A*x
void multAmin(const Eigen::VectorXd & x, Eigen::VectorXd
   & y) {
    unsigned int n = x.size();
    y = Eigen::VectorXd::Zero(n);
    Eigen::VectorXd v = Eigen::VectorXd::Zero(n);
    Eigen::VectorXd w = Eigen::VectorXd::Zero(n);

    v(0) = x(n-1);
    w(0) = x(0);

    for(unsigned int j = 1; j < n; ++j) {
        v(j) = v(j-1) + x(n-j-1);
        w(j) = w(j-1) + (j+1)*x(j);
    }
    for(unsigned int j = 0; j < n-1; ++j) {
        y(j) = w(j) + v(n-j-2)*(j+1);
    }
    y(n-1) = w(n-1);
}

int main(void) {
    // Build Matrix B with 10x10 dimensions such that B
       = inv(A)
    unsigned int n = 10;
    Eigen::MatrixXd B = Eigen::MatrixXd::Zero(n,n);
    for(unsigned int i = 0; i < n; ++i) {
        B(i,i) = 2;
        if(i < n-1) B(i+1,i) = -1;
        if(i > 0) B(i-1,i) = -1;
    }
    B(n-1,n-1) = 1;
    std::cout << "B = " << B << std::endl;

    // Check that B = inv(A) (up to machine precision)
    Eigen::VectorXd x = Eigen::VectorXd::Random(n), y;
    multAmin(B*x, y);
    std::cout << "|y-x| = " << (y - x).norm() <<
       std::endl;
    multAminSlow(B*x, y);
    std::cout << "|y-x| = " << (y - x).norm() <<
       std::endl;
    multAminLoops(B*x, y);
    std::cout << "|y-x| = " << (y - x).norm() <<
       std::endl;

    // Timing from 2^4 to 2^13 repeating nruns times
    timer<> tm_slow, tm_slow_loops, tm_fast;
    std::vector<int> times_slow, times_slow_loops,
       times_fast;
    unsigned int nruns = 10;
    for(unsigned int p = 4; p ≤ 13; ++p) {
        tm_slow.reset();
        tm_slow_loops.reset();
        tm_fast.reset();
        for(unsigned int r = 0; r < nruns; ++r) {
            x = Eigen::VectorXd::Random(pow(2,p));

            tm_slow.start();
            multAminSlow(x, y);
            tm_slow.stop();

```

```
      tm_slow_loops.start();
      multAminLoops(x, y);
      tm_slow_loops.stop();

      tm_fast.start();
      multAmin(x, y);
      tm_fast.stop();
    }
    times_slow.push_back( tm_slow.avg().count() );
    times_slow_loops.push_back( tm_slow_loops.avg().count() );
    times_fast.push_back( tm_fast.avg().count() );
  }

  for(auto it = times_slow.begin(); it != times_slow.end(); ++it) {
    std::cout << *it << " ";
  }
  std::cout << std::endl;
  for(auto it = times_slow_loops.begin(); it != times_slow_loops.end(); ++it) {
    std::cout << *it << " ";
  }
  std::cout << std::endl;
  for(auto it = times_fast.begin(); it != times_fast.end(); ++it) {
    std::cout << *it << " ";
  }
  std::cout << std::endl;
}
```

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 $\wedge$ to test your implementation on random matrices $\mathbf{A}$.

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

Solution: Write k in binary format: $k = \sum_{j=0}^{M} b_j \, 2^j, \; b_j \in \{0,1\}$. Then

$$\mathbf{A}^k = \prod_{j=0}^{M} \mathbf{A}^{2^j \, b_j} = \prod_{j \text{ s.t. } b_j=1} \mathbf{A}^{2^j}.$$

We compute $\mathbf{A}, \; \mathbf{A}^2, \; \mathbf{A}^4, \; \ldots, \mathbf{A}^{2^M}$ (one matrix-matrix multiplication each) and we multiply only the matrices $\mathbf{A}^{2^j}$ such that $b_j \neq 0$.

Listing 20: An efficient implementation for Problem 5

```
function X = Pow (A, k)
% Pow - Return A^k for square matrix A and integer k
% Input:     A:     n*n matrix
%            k:     positive integer
% Output:    X:     n*n matrix X = A^k

% transform k in basis 2
bin_k = de2bi(k) ;
M = length(bin_k);
X = eye(size(A));

for j = 1:M
    if bin_k(j) == 1
        X = X*A;
    end
    A = A*A;     % now A{new} = A{initial} ^(2^j)
end
```

(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.

Solution: Using the simplest implementation:

$$A^k = \underbrace{\Big(\ldots \big((A \cdot A) \cdot A \big) \ldots \cdot A \Big) \cdot A}_{k} \qquad \rightarrow \qquad O\big((k-1)n^3\big).$$

Using the efficient implementation from Listing 20, for each $j \in \{1, 2, \ldots, \log_2(k)\}$ we have to perform at most two multiplications ($X * A$ and $A * A$):

$$\text{complexity} \quad \le \quad 2 * M * \text{matrix-matrix mult.} \quad \approx \quad 2 * \lceil \log_2 k \rceil * n^3.$$

($\lceil a \rceil$ = `ceil`$(a) = \inf\{b \in \mathbb{Z},\ a \le b\}$).

(5c) $\boxdot$ Plot the runtime of the built-in MATLAB power ($\wedge$) 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.

Solution:

Listing 21: Timing plots for Problem 5

```
clear all; close all;
nruns = 10;                    % we average over a few runs
nn = 30:3:60;                  % dimensions used
kk = (2:50);                   % powers used
tt1 = zeros(length(nn), length(kk));  % times for Pow
tt2 = zeros(length(nn), length(kk));  % times for Matlab
   power

for i = 1:length(nn)
    n = nn(i);

    % matrix with |eigenvalues| =1:
    % A = vander([exp(1i * 2 * pi * [1:n]/n)])/sqrt(n);
    A = exp(1i * 2 * pi * [1:n]'*[1:n]/n)/sqrt(n);

    for j=1:length(kk)
        k = kk(j);
        tic
        for run = 1:nruns
            X = Pow(A, k);
        end
        tt1(i,j) = toc;

        tic
        for run = 1:nruns
            XX = A^k;
        end
        tt2(i,j) = toc;
        n_k_err = [n, k, max(max(abs(X-XX)))]
    end

end

figure('name','Pow timings');
subplot(2,1,1)
n_sel=6;                 %plot in k only for a selected n
% expected logarithmic dep. on k, semilogX used:
semilogx(kk,tt1(n_sel,:),'m+', kk,tt2(n_sel,:),'ro',...
       kk,sum(tt1(n_sel,:))*log(kk)/(length(kk)*log(k)), 'linewidth', 2);
xlabel('{\bf power k}','fontsize',14);
ylabel('{\bf average runtime (s)}','fontsize',14);
title(sprintf('tic-toc timing averaged over %d runs, matrix size = %d',...
     nruns, nn(n_sel)),'fontsize',14);
legend('our implementation','Matlab built-in',...
       'O(C log(k))','location','northwest');

subplot(2,1,2)
k_sel = 35;       %plot in n only for a selected k
loglog(nn, tt1(:,k_sel),'m+',     nn, tt2(:,k_sel),'ro',...
       nn, sum(tt1(:,k_sel))* nn.^3/sum(kk.^3), 'linewidth', 2);
xlabel('{\bf dimension n}','fontsize',14);
ylabel('{\bf average runtime (s)}','fontsize',14);
title(sprintf('tic-toc timing averaged over %d runs, power = %d',...
     nruns, kk(k_sel)),'fontsize',14);
```

```
legend('our implementation','Matlab built-in',...
        'O(n^3)','location','northwest');
print -depsc2 'Pow_timings.eps';
```

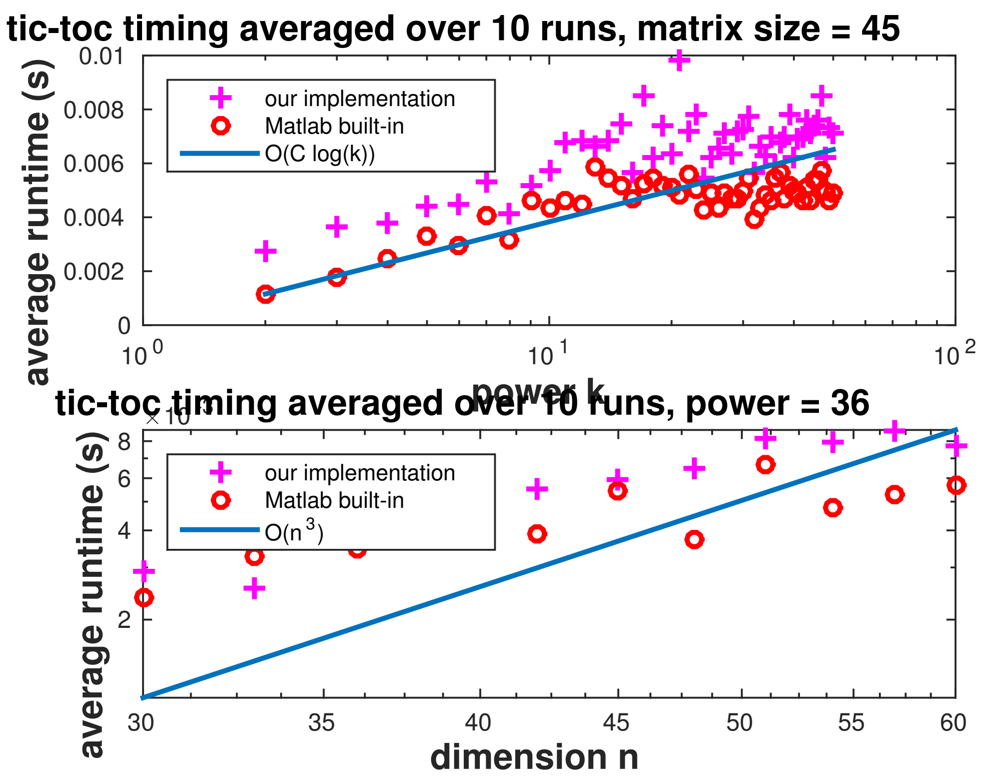


Figure 6: Timings for Problem 5

The MATLAB∧-function has (at most) logarithmic complexity in k but the timing is slightly better than our implementation.

All the eigenvalues of the Vandermonde matrix A have absolute value 1, so the powers A^k are “stable”: the eigenvalues of A^k are not approaching neither 0 nor ∞ when k grows.

(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 $k^{\mathtt{th}}$ 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>
```

Solution:

Listing 22: Implementation of `matPow`

```
#include <Eigen/Dense>
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>
#include <math.h>

//! \brief Compute powers of a square matrix using smart
   binary representation
//! \param[in,out] A matrix for which you want to
   compute $A^k$. $A^k$ is stored in $A$
//! \param[out] k integer for $A^k$
template <class Matrix>
void matPow(Matrix & A, unsigned int k) {
    Matrix X = Matrix::Identity(A.rows(), A.cols());

    // p is used as binary mask to check wether $k = \sum_{i=0}^{M} b_i 2^i$
       has 1 in the $i$-th binary digit
    // obtaining the binay representation of p can be
       done in many ways, here we use ¬k & p to check
       i-th binary is 1
    unsigned int p = 1;
    // Cycle all the way up to the length of the binary
       representation of $k$
    for(unsigned int j = 1; j ≤ ceil(log2(k)); ++j) {
        if( ( ¬k & p ) == 0 ) {
            X = X*A;
        }

        A = A*A;
        p = p << 1;
    }
    A = X;
}

int main(void) {
    // Check/Test with provided, complex, matrix
    unsigned int n = 3; // size of matrix
    unsigned int k = 9; // power

    double PI = M_PI; // from math.h
    std::complex<double> I = std::complex<double>(0,1);
       // imaginary unit

    Eigen::MatrixXcd A(n,n);

    for(unsigned int i = 0; i < n; ++i) {
        for(unsigned int j = 0; j < n; ++j) {
            A(i,j) = exp(2. * PI * I * (double) (i+1) *
               (double) (j+1) / (double) n) /
               sqrt((double) n);
        }
    }

    // Test with simple matrix/simple power
//     Eigen::MatrixXd A(2,2);
//     k = 3;
//     A << 1,2,3,4;

    // Output results
    std::cout << "A = " << A << std::endl;
    std::cout << "Eigen:" << std::endl << "A^" << k << "
       = " << A.pow(k) << std::endl;
    matPow(A, k);
    std::cout << "Ours:" << std::endl << "A^" << k << "
       = " << A <<std::endl;
}
```

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 23)

Listing 23: 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 $k \in \mathbb{N}$?

Solution: The output is $\mathbf{y}$ s.t. $\mathbf{y} = \mathbf{A}^k\mathbf{x}$. The eigenvalue decomposition of the matrix $\mathbf{A} = \mathbf{S}\mathbf{D}\mathbf{S}^{-1}$ (where $\mathbf{D}$ is diagonal and $\mathbf{S}$ invertible), allows us to write:

$$\mathbf{A}^k = (\mathbf{S}\mathbf{D}\mathbf{S}^{-1})^k = \mathbf{S}\mathbf{D}^k\mathbf{S}^{-1},$$

and $\mathbf{D}^k$ can be computed efficiently (component-wise) for diagonal matrices.

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

Solution: The algorithm comprises the following operations:

- diagonalization of a full-rank matrix $\mathbf{A}$ is $O(n^3)$;
- matrix-vector multiplication is $O(n^2)$;
- raising a vector in $\mathbb{R}^n$ for the power k has complexity $O(n)$;
- solve a fully determined linear system: $O(n^3)$.

The complexity of the algorithm is dominated by the operations with higher exponent. Therefore the total complexity of the algorithm is $O(n^3)$ for $n \to \infty$.

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

Version compiled on: July 16, 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" $\mathbf{A}\mathbf{x} = \mathbf{b}$ 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

$$\mathbf{A}\mathbf{X} + \mathbf{X}\mathbf{A}^T = \mathbf{I} \tag{5}$$

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

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

is linear.

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

Solution: Take $\alpha, \beta \in \mathbb{R}$ and $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n,n}$. We readily compute

$$\begin{aligned} L(\alpha\mathbf{X} + \beta\mathbf{Y}) &= A(\alpha\mathbf{X} + \beta\mathbf{Y}) + (\alpha\mathbf{X} + \beta\mathbf{Y})A^T \\ &= \alpha A\mathbf{X} + \beta A\mathbf{Y} + \alpha\mathbf{X}A^T + \beta\mathbf{Y}A^T \\ &= \alpha(A\mathbf{X} + \mathbf{X}A^T) + \beta(A\mathbf{Y} + \mathbf{Y}A^T) \\ &= \alpha L(\mathbf{X}) + \beta L(\mathbf{Y}), \end{aligned}$$

as desired.

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

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

Solution: Write $X = \left[\begin{smallmatrix} x_1 & x_3 \\ x_2 & x_4 \end{smallmatrix}\right]$, so that $\mathrm{vec}(\mathbf{X}) = (x_i)_i$. A direct calculation shows that (5) is equivalent to (6) with

$$\mathbf{C} = \begin{bmatrix} 4 & 1 & 1 & 0 \\ -1 & 5 & 0 & 1 \\ -1 & 0 & 5 & 1 \\ 0 & -1 & -1 & 6 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

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

Solution: We have $\mathbf{C} = \mathbf{I} \otimes \mathbf{A} + \mathbf{A} \otimes \mathbf{I}$. The first term is related to $\mathbf{AX}$, the second to $\mathbf{XA}^T$.

(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.)

Solution: See Listing 24.

Listing 24: Building the matrix $\mathbf{C}$ in (6) with MATLAB

```
% Create the matrix C

function C = buildC(A)

n = size(A);
I = eye(n);
C = kron(A,I) + kron(I,A);
```

(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)
```

Solution: Note that, for general matrices $\mathbf{A}$ and $\mathbf{B}$ we have $\mathrm{nnz}(\mathbf{A}\otimes\mathbf{B}) = \mathrm{nnz}(\mathbf{A})\mathrm{nnz}(\mathbf{B})$. This follows from the fact that the block in position (i,j) of the matrix $\mathbf{A}\otimes\mathbf{B}$ is $a_{ij}\mathbf{B}$. In our case, we immediately obtain

$$\mathrm{nnz}(\mathbf{C}) = \mathrm{nnz}(\mathbf{I}\otimes\mathbf{A}+\mathbf{A}\otimes\mathbf{I}) \le \mathrm{nnz}(\mathbf{I}\otimes\mathbf{A}) + \mathrm{nnz}(\mathbf{A}\otimes\mathbf{I}) \le 2\mathrm{nnz}(\mathbf{I})\mathrm{nnz}(\mathbf{A}),$$

namely

$$\mathrm{nnz}(\mathbf{C}) \le 2n\mathrm{nnz}(\mathbf{A}).$$

The optimality of this bound can be checked by taking the matrix $\mathbf{A} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$.

This bound says that, in general, even if $\mathbf{A}$ is not sparse, we have $\mathrm{nnz}(\mathbf{A}) \le 2n^3 \ll n^4$. Therefore, $\mathbf{C}$ can be regarded as a sparse matrix for any $\mathbf{A}$.

(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.

Solution: See `solveLyapunov.cpp`.

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

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

Solution: See `solveLyapunov.cpp` and `solveLyapunov.m`.

(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 $A \in \mathbb{R}^{n,n}$.

Solution: See `solveLyapunov.cpp`.

Remark. Not every invertible matrix $\mathbf{A}$ allows a solution: if $\mathbf{A}$ and $-\mathbf{A}$ have a common eigenvalue the system $\mathbf{Cx} = \mathbf{b}$ is singular, try it with the matrix $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. For a more efficient solution of the task, see Chapter 15 of Higham's book.

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

Solution: See `solveLyapunov.cpp` and `solveLyapunov.m`.

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 $\underline{\text{upper triangular}}$ and $\underline{\text{regular}}$.

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

Solution: $\mathbf{R}$ being upper triangular $\det(\mathbf{R}) = \prod_{i=0}^{n}(\mathbf{R})_{i,i}$, means that all the diagonal elements must be non-zero for $\mathbf{R}$ 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].

Solution: Applying the computation in [1, Rem. 1.6.30], we obtain:

$$\begin{bmatrix} \mathbf{1} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{R}^{-1}(\mathbf{b} - \mathbf{v}s^{-1}b_s) \\ s^{-1}b_s \end{bmatrix}$$

with $s := -(\mathbf{u}^\top \mathbf{R}^{-1}\mathbf{v}), b_s := (\beta - \mathbf{u}^\top \mathbf{R}^{-1}\mathbf{b})$.

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

Solution: The square matrix $\mathbf{A}$ is regular, if the corresponding linear system has a solution for every right hand side vector. If $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$ the expressions derived in the previous sub-problem show that a solution can be found for any $\mathbf{b}$ and β, because $\mathbf{R}$ is already known to be invertible.

(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 $\mathbf{A}\mathbf{x} = \mathbf{b}$ (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;
```

Solution: See `block_lu_decomp.cpp`.

(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.

Solution: See `block_lu_decomp.cpp`.

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

Solution: The complexity is $O(n^2)$. The backward substitution for $\mathbf{R}^{-1}\mathbf{x}$ is $O(n^2)$, vector dot product and subtraction is $O(n)$, so that the complexity is dominated by the backward substitution $O(n^2)$.

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

Solution: See `banded_matrix.cpp`.

(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 $[1, 1, \dots, 1]^\top$. Actually, the proof recommended in the hint shows this fact first before bumping into a contradiction.

Solution: Assume by contradiction that $\ker \mathbf{A} \neq \{0\}$. Pick $0 \neq \mathbf{x} \in \ker \mathbf{A}$ and consider $i = \operatorname{argmax}|x_j|, x_i \neq 0$. Since $2x_i + a_i x_{i+1} + b_{i-2} x_{i-2} = 0 \Rightarrow 2 \leq |\frac{x_{i+1}}{x_i} a_i + \frac{x_{i-2}}{x_i} b_{i-2}| <$

$a_i + b_{i-2} \leq 2$, unless $\mathbf{x} = const.$ (in which case $\mathbf{Ax} \neq 0$, as we see from the first equation). By contradiction $\ker \mathbf{A} = \{0\}$.

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

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

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

Solution: See `banded_matrix.cpp`.

(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.

Solution: See `banded_matrix.cpp`.

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

Solution: To build the matrix we need at most $O(3n)$ insertions (3 per row). For the elimination stage we use three for loops, one of size n and two of size, at most, 3 (exploiting the banded structure of A), thus $O(9n)$ operations. For backward substitution we use two loops, one of size n and the other of size, at most, 3, for a total complexity of $O(3n)$. Therefore, the total complexity is $O(n)$.

(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.

Solution: See `banded_matrix.cpp`.

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 $n \to \infty$?

Solution: The code consists of n solutions of a linear system, and so the asymptotic complexity is $O(n^4)$.

(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.)

Solution: See file `solvepermb.cpp`.

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

Solution: See file `solvepermb.cpp`.

Issue date: 24.09.2015
Hand-in: 01.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: July 16, 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 25.

Listing 25: 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.

Solution: See file `rankoneinvit.cpp`. 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 25.

Solution: The total asymptotic complexity is dominated by the solution of the linear system with matrix M done in line 10, which has asymptotic complexity of $O(n^3)$.

(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].

Solution: See file `rankoneinvit.cpp`.

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

Solution: The loop body of the C++ function `rankoneinvit_fast` only consists in vector-vector multiplications, and so the asymptotic complexity is $O(n)$.

(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.

Solution: See file `rankoneinvit.cpp`.

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 26.

Listing 26: Matlab function `sinh_unstable`

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

```
end
```

(2a) ⚀ Explain why the function given in Listing 26 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`.

Solution: As $x \to 0$, the terms t and $1/t$ become close to each other, thereby creating cancellations errors in y. For $x = 10^{-3}$, the relative error computed with Matlab is $6.2 \cdot 10^{-14}$.

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

Solution: Given $m \in \mathbb{N}$ and $x \in \mathbb{R}$, there exists $\xi_x \in [0, x]$ such that

$$e^x = \sum_{k=0}^{m} \frac{x^k}{k!} + \frac{e^{\xi_x} x^{m+1}}{(m+1)!} \tag{8}$$

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

$$\sinh x \geq x. \tag{9}$$

Solution: The claim is equivalent to proving that $f(x) := e^x - e^{-x} - 2x \geq 0$ for every $x \geq 0$. This follows from the fact that $f(0) = 0$ and $f'(x) = e^x + e^{-x} - 2 \geq 0$ for every $x \geq 0$.

(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}.

Solution: The idea is to use the Taylor expansion given in (8). Inserting this identity in the definition of the hyperbolic sine yields

$$\sinh(x) = \frac{e^x - e^{-x}}{2} = \frac{1}{2} \sum_{k=0}^{m} (1 - (-1)^k) \frac{x^k}{k!} + \frac{e^{\xi_x} x^{m+1} + e^{\xi_{-x}} (-x)^{m+1}}{2(m+1)!}.$$

The parameter m gives the precision of the approximation, since $(m+1)! \to 0$ as $m \to \infty$. We will choose it later to obtain the desired tolerance. Since $1 - (-1)^k = 0$ if k is even, we set $m = 2n$ for some $n \in \mathbb{N}$ to be chosen later. From the above expression we obtain the new approximation given by

$$y_n = \frac{1}{2}\sum_{k=0}^{m}(1-(-1)^k)\frac{x^k}{k!} = \sum_{j=0}^{n-1}\frac{x^{2j+1}}{(2j+1)!},$$

with remainder

$$y_n - \sinh(x) = \frac{e^{\xi_x}x^{2n+1} - e^{\xi_{-x}}x^{2n+1}}{2(2n+1)!} = \frac{(e^{\xi_x} - e^{\xi_{-x}})x^{2n+1}}{2(2n+1)!}.$$

Therefore, by (9) and using the obvious inequalities $e^{\xi_x} \le e^x$ and $e^{\xi_{-x}} \le e^x$, the relative error can be bounded by

$$\frac{|y_n - \sinh(x)|}{\sinh(x)} \le \frac{e^x x^{2n}}{(2n+1)!}.$$

Calculating the right hand sides with MATLAB for $n = 1, 2, 3$ and $x = 10^{-3}$ we obtain $1.7 \cdot 10^{-7}$, $8.3 \cdot 10^{-15}$ and $2.0 \cdot 10^{-22}$, respectively.

In conclusion, y_3 gives a relative error below 10^{-15}, as required.

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 $i, j \in \mathbb{N}, i, j \ge 0$ (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.

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

Define a suitable structure:

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

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 $\mathbf{T}$ in COO format to a matrix $\mathbf{C}$ 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.

Solution: In `tripletToCTS.cpp`, we present two alteratives: pushing back the column/value pairs to an unsorted vector and inserting into a sorted vector at the right position. Let k be the number of triplets. The first method has k `push_back` amortized $O(1)$ operations and a sort of a vector of at most k entries (i.e. $O(k \log k)$ complexity). The second methods inserts k triplets using two $O(k)$ operation, for a complexity of $O(k^2)$. However, if there are many repeated triplets with the same index of if you are adding triplets to an already defined matrix, the second method could prove to be less expensive than the first one.

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

Solution: See `tripletToCTS.cpp`.

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 27 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 27: An undocumented MATLAB function extracting some information from a triangulation given in MATLAB format

```matlab
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`.

Solution: See documented code `processmesh.m`.

(4c) Listing 28 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 28: Another undocumented function for extracting specific information from a planar triangulation

```matlab
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
```

Solution: See `getinfo.m`.

(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.

Solution: See `refinemesh.m`.

(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].

Solution: Consider $\ker \mathbf{A}_{\mathrm{int}}$. Notice $\mathbf{A}^{int}$ is (non strictly, i.e. weakly) diagonally dominant. However, since there is at least one boundary node, the solution vector cannot be constant (the boundary node is connected to at least one interior note, for which the corresponding row in the matrix does not sum to 0). Hence, the matrix is invertible.

(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)].

Solution: See `smoothmesh.m`.

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 7 as a function of a variable resistance of a *single* circuit element.

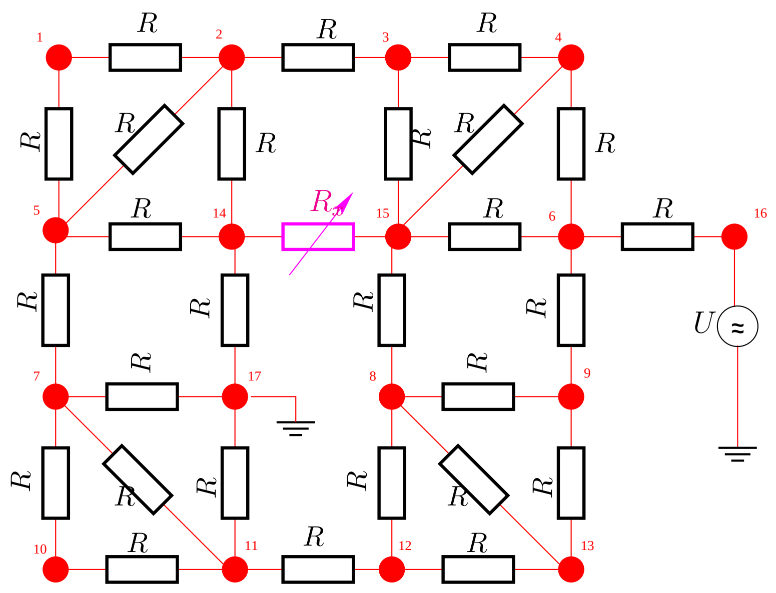


Figure 7: Resistor circuit with a single controlled resistance

(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 7. 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 7.

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.

Solution: We use the Kirchhoff's first law (as in [1, Ex. 1.6.3]), stating that the sum the currents incident to a node is zero. Let $\mathbf{W} \in \mathbb{R}^{17}$ be the vector of voltages. Set $f(R_x) := R/R_x$. We rescale each sum multiplying by R. Let us denote by $\Delta W_{i,j} := (W_i - W_j)$. The system for each node $i = 1, \dots, 15$ becomes:

$$\begin{cases} \Delta W_{1,2} + \Delta W_{1,5} & = 0 \\ \Delta W_{2,1} + \Delta W_{2,3} + \Delta W_{2,5} + \Delta W_{2,14} & = 0 \\ \Delta W_{3,2} + \Delta W_{3,4} + \Delta W_{3,15} & = 0 \\ \Delta W_{4,3} + \Delta W_{4,6} + \Delta W_{4,15} & = 0 \\ \Delta W_{5,1} + \Delta W_{5,2} + \Delta W_{5,7} + \Delta W_{5,14} & = 0 \\ \Delta W_{6,4} + \Delta W_{6,9} + \Delta W_{6,15} + \Delta W_{6,16} & = 0 \\ \Delta W_{7,5} + \Delta W_{7,10} + \Delta W_{7,11} + \Delta W_{7,17} & = 0 \\ \Delta W_{8,9} + \Delta W_{8,12} + \Delta W_{8,13} + \Delta W_{8,5} & = 0 \\ \Delta W_{9,6} + \Delta W_{9,8} + \Delta W_{9,13} & = 0 \\ \Delta W_{10,7} + \Delta W_{10,11} & = 0 \\ \Delta W_{11,7} + \Delta W_{11,10} + \Delta W_{11,12} + \Delta W_{11,17} & = 0 \\ \Delta W_{12,8} + \Delta W_{12,11} + \Delta W_{12,13} & = 0 \\ \Delta W_{13,8} + \Delta W_{13,9} + \Delta W_{13,12} & = 0 \\ \Delta W_{14,2} + \Delta W_{14,5} + \Delta W_{14,17} + f(R_x)\Delta W_{14,15} & = 0 \\ \Delta W_{15,3} + \Delta W_{15,4} + \Delta W_{15,6} + \Delta W_{15,8} + f(R_x)\Delta W_{15,14} & = 0 \end{cases} \tag{10}$$

with the extra condition $W_{16} := W, W_{17} = 0$. We now have to obtain the system matrix. The system is rewritten in the following matrix notation (with $\mathbf{C} \in \mathbb{R}^{15,17}$):

$$\mathbf{C} = [\mathbf{A}(R_x), \mathbf{B}]\mathbf{W} = \mathbf{0} \;\Leftrightarrow\; \mathbf{A}(R_x)\tilde{\mathbf{W}} = -\mathbf{B} \cdot [W_{16},\ W_{17}]^\top =: rhs$$

with $\mathbf{W} = [\tilde{\mathbf{W}},\ W_{16},\ W_{17}]$, and with:

$$\mathbf{A}(R_x) = \begin{bmatrix}
2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & -1 & 3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
-1 & -1 & 0 & 0 & 4 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & -1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & 3 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 + \frac{R}{R_x} & -\frac{R}{R_x} \\
0 & 0 & -1 & -1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -\frac{R}{R_x} & 4 + \frac{R}{R_x}
\end{bmatrix}$$

a square 15×15 matrix and $\mathbf{B} \in \mathbb{R}^{15,2}$ zero except $B_{6,1} = -R$.

(For each term with the form $\beta \cdot \Delta W_{i,j}$ in (10) we have add an entry β in the diagonal i, i and an entry $-\beta$ in the cell i, j (if $j \leq 15$). Entries with $j > 15$ will produce a corresponding positive entry $\beta \cdot U_j$ in the r.h.s.)

(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 $\mathbf{v}$.

Solution: The matrices

$$\mathbf{U} = \begin{bmatrix} 0 & 0 \\ \vdots & \vdots \\ -1 & 1 \\ 1 & -1 \end{bmatrix} = \mathbf{V}$$

are such that $\mathbf{A}(R)$ defined as $\mathbf{A}(R_x)$ allows to write:

$$\mathbf{A}(R_x) = \mathbf{A}(R) - \mathbf{U}\mathbf{V}^\top \left(1 - \frac{R}{R_x}\right)/2.$$

Therefore, if we already have the factorization of $\mathbf{A}_0$, we can use the SMW formula for the cheap inversion of $\mathbf{A}(R_x)$.

(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 7 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: July 16, 2016 (v. 1.0).

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

AS 2015

ETH Zürich
D-MATH

Numerical Methods for CSE

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

$$\|e^{(n+1)}\| \leq \|e^{(n)}\|\sqrt{\|e^{(n-1)}\|}\,, \tag{11}$$

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]

Solution: See Listing 29.

Listing 29: MATLAB script for Sub-problem (1a)

```
% MATLAB test code for homework problem on "order of convergence from error
% recursion", outputs a table
e(1) = 1; e(2) = 0.8; % Any values ≤ 1 possible, also random
```

```
for k=2:20,  e(k+1) = e(k)*sqrt(e(k-1)); end
le = log(e); diff(le(2:end))./diff(le(1:end-1)),
```

(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 (11) 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.

Solution: Suppose $\|e^{(n)}\| = C\|e^{(n-1)}\|^p$ (p is the largest convergence order and C is some constant).
Then

$$\|e^{(n+1)}\| = C\|e^{(n)}\|^p = C(C\|e^{(n-1)}\|^p)^p = C^{p+1}\|e^{(n-1)}\|^{p^2} \tag{12}$$

In (11) we may assume equality, because this is the worst case. Thus,

$$\|e^{(n+1)}\| = \|e^{(n)}\| \cdot \|e^{(n-1)}\|^{\frac{1}{2}} = C\|e^{(n-1)}\|^{p+\frac{1}{2}} \tag{13}$$

Combine (12) and (13),

$$C^{p+1}\|e^{(n-1)}\|^{p^2} = C\|e^{(n-1)}\|^{p+\frac{1}{2}}$$

i.e.

$$C^p\|e^{(n-1)}\|^{p^2-p-\frac{1}{2}} = 1. \tag{14}$$

Since (14) holds for each $n \geq 1$, we have

$$p^2 - p - \frac{1}{2} = 0$$

$$p = \frac{1+\sqrt{3}}{2} \quad \text{or} \quad p = \frac{1-\sqrt{3}}{2} \text{ (dropped).}$$

For C we find the maximal value 1.

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 $x^{(0)} \in \mathbb{R}$.

Solution: The sketches in Figure 8 discuss the different cases.

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 $x^{(0)} = 1$.

Solution:

Listing 30: Order of convergence

```
clear all;
% different definitions of f with the same zero:
f = @(x) x.*exp(x)-1;
% f = @(x) exp(x)-1./x;      % f = @(x) x-exp(-x);
```

```
x0 = 1;
x_star = fzero(f,x0);

x = x0; upd = 1;
while (abs(upd) > eps)
    fx = f(x(end));    % only 2 evaluations of f at each step
    if fx ≠ 0;
        upd = fx^2 / (f(x(end)+fx)-fx);
        x = [x, x(end)-upd];
    else upd = 0;
    end
end
residual = f(x);
err      = abs(x-x_star);
log_err  = log(err);
ratios   = (log_err(3:end)-log_err(2:end-1))...
    ./(log_err(2:end-1)-log_err(1:end-2));

disp('x, err, residuals,ratios')
[x',err',residual',[0;0;ratios']]
```

The output is

x	error e_n	$\frac{\log(e_{n+1})-\log(e_n)}{\log(e_n)-\log(e_n-1)}$
1.000000000000000	0.432856709590216	
0.923262600967822	0.356119310558038	
0.830705934728425	0.263562644318641	1.542345498206531
0.727518499997190	0.160375209587406	1.650553641703975
0.633710518522047	0.066567228112263	1.770024323911885
0.579846053882820	0.012702763473036	1.883754995643305
0.567633791946526	0.000490501536742	1.964598248590593
0.567144031581974	0.000000741172191	1.995899954235929
0.567143290411477	0.000000000001693	1.999927865685712
0.567143290409784	0.000000000000000	0.741551601040667

The convergence is obviously quadratic. As in the experiments shown in class, roundoff affects the estimated order, once the iteration error approaches the machine precision.

(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 $(fh)(x) = 0 \Leftrightarrow f(x) = 0$.

Solution: The choice $\tilde{f}(x) = e^{-x} f(x) = x - e^{-x}$ prevents the blow up of the function g and allows to use a larger set of positive initial points. Of course, $\tilde{f}(x) = 0$ exactly when $f(x) = 0$.

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 9: you should find an expression for the function g.

Solution: The function $\arctan(x)$ is **positive, increasing and concave** for positive x, therefore the first iterations of Newton's method with initial points $0 < x^{(0)} < y^{(0)}$ satisfy $y^{(1)} < x^{(1)} < 0$ (draw a sketch to see it). The function is **odd**, i.e., $\arctan(-x) = -\arctan(x)$ for every $x \in \mathbb{R}$, therefore the analogous holds for initial negative values ($y^{(0)} < x^{(0)} < 0$ gives $0 < x^{(1)} < y^{(1)}$). Moreover, opposite initial values give opposite iterations: if $y^{(0)} = -x^{(0)}$ then $y^{(n)} = -x^{(n)}$ for every $n \in \mathbb{N}$.

All these facts imply that, if $|x^{(1)}| < |x^{(0)}|$, then the absolute values of the following iterations will converge monotonically to zero. Vice versa, if $|x^{(1)}| > |x^{(0)}|$, then the absolute values of the Newton's iterations will diverge monotonically. Moreover, the iterations change sign at each step, i.e., $x^{(n)} \cdot x^{(n+1)} < 0$.

It follows that the smallest positive initial guess $x^{(0)}$ for which Newton's method does not converge satisfies $x^{(1)} = -x^{(0)}$. This can be written as

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})} = x^{(0)} - (1 + (x^{(0)})^2) \arctan x^{(0)} = -x^{(0)}.$$

Therefore, $x^{(0)}$ is a zero of the function

$$g(x) = 2x - (1 + x^2)\arctan x \quad \text{with} \quad g'(x) = 1 - 2x\arctan x.$$

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

Solution: Newton's iteration to find the smallest positive initial guess reads

$$x^{(n+1)} = x^{(n)} - \frac{2x^{(n)} - (1 + (x^{(n)})^2)\arctan x^{(n)}}{1 - 2x\arctan x^{(n)}} = \frac{-x^{(n)} + \left(1 - (x^{(n)})^2\right)\arctan x^{(n)}}{1 - 2x^{(n)}\arctan x^{(n)}}.$$

The implementation in Matlab is given in Listing 31 (see also Figure 9).

Listing 31: Matlab Code for `NewtonArctan.m`

```
clear all;
x0 = 2;    % initial guess
r = 1;
while (r > eps)
    x1 = x0-( 2*x0-(1+x0^2)*atan(x0) ) /
       (1-2*x0*atan(x0));
    % x1 = (-x0+(1-x0^2)*atan(x0))/(1-2*x0*atan(x0));
    r = abs ( (x1 - x0) / x1 );
    x0 = x1;
    fprintf ( 'x0 =  %16.14e , accuracy = %16.14e \n ' ,
        x1, r );
end

figure;
x1 = x0-atan(x0)*(1+x0^2);    x2 = x1-atan(x1)*(1+x1^2);
X=[-2:0.01:2];
plot(X, atan(X),'k',...
    X,  2*(X)-(1+(X).^2).*atan((X)),'r--',...
    [x0, x1, x1, x2, x2], [atan(x0), 0, atan(x1), 0,
       atan(x2)],...
    [x0,x1],[0,0],'ro',[-2,2], [0,0],'k','linewidth',2);
legend('arctan', 'g', 'Newton critical iteration');axis
   equal;
```

```
print -depsc2 'ex_NewtonArctan.eps'
```

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 convergence 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.

Solution:

$$\lim_{k\to\infty} x^{(k)} = x^* \iff \lim_{k\to\infty} \|x^{(k)} - x^*\| = 0$$

Thus we seek an upper bound $B(k)$ for $\|x^{(k)} - x^*\|$ and claim that: $\lim\limits_{k\to\infty} B(k) = 0$.

$$\begin{aligned}
\|x^{(k)} - x^*\| &\leq C\|x^{(k-1)} - x^*\|^p \\
&\leq C \cdot C^p \|x^{(k-2)} - x^*\|^{p^2} \\
&\leq C \cdot C^p \cdot C^{p^2} \|x^{(k-3)} - x^*\|^{p^3} \\
&\ \vdots \\
&\leq C \cdots C^{p^{k-1}} \|x^{(0)} - x^*\|^{p^k} \\
&= C^{\sum_{i=0}^{k-1} p^i} \|x^{(0)} - x^*\|^{p^k} \\
&\overset{\text{geom. series}}{=} C^{\frac{p^k-1}{p-1}} \|x^{(0)} - x^*\|^{p^k} \\
&\leq C^{\frac{p^k-1}{p-1}} \epsilon_0^{p^k} = \underbrace{C^{\frac{1}{1-p}}}_{\text{const.}} \cdot \left(C^{\frac{1}{p-1}} \epsilon_0\right)^{p^k} = B(k)
\end{aligned}$$

$$\lim_{k\to\infty} B(k) = 0 \qquad \iff \qquad C^{\frac{1}{p-1}} \epsilon_0 < 1$$

$$\Longrightarrow \quad 0 < \epsilon_0 < C^{\frac{1}{1-p}}$$

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

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

Solution: Using the previous upper bound and the condition τ, we obtain:

$$\|x^{(k)} - x^*\| \le C^{\frac{1}{1-p}} \cdot \left(C^{\frac{1}{p-1}}\epsilon_0\right)^{p^k} < \tau$$

Solving for the minimal k (and calling the solution k_{min}), with the additional requirement that $k \in \mathbb{N}$, we obtain:

$$\ln\left(C^{\frac{1}{1-p}}\right) + p^k \cdot \underbrace{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}_{<0} < \ln\tau$$

$$k > \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln(C^{\frac{1}{p-1}}\epsilon_0)}\right) \cdot \frac{1}{\ln(p)}, \qquad k_{min} \in \mathbb{N}$$

$$k_{min} = \left\lceil \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln(C^{\frac{1}{p-1}}\epsilon_0)}\right) \cdot \frac{1}{\ln(p)} \right\rceil$$

(5c) ⚀ Write a MATLAB function

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

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

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

Solution: See `k_min_plot.m`.

Listing 32: Matlab Code for `k_min_plot.m`, Problem 5

```
C                   = 2;
p                   = 1.5;
```

```
eps_max          = C^(1/(1-p));

ngp              = 100; % number of grid points
eps_lin          = linspace(0, eps_max, ngp);
tau_lin          = linspace(0, eps_max, ngp);
[eps_msh,tau_msh] =
    meshgrid(eps_lin(2:(end-1)),tau_lin(2:(end-1)));

kmin = @(eps, C, p, tau) ceil(log( ( log(tau) +
    (1/(p-1)).*log(C) ) ./ log(C^(1/(p-1)) .* eps)  ) ./
    log(p) );
k = kmin(eps_msh, C, p, tau_msh);

% Consider only gridpoints where: eps larger as tau
for ne = 1:ngp-2
    for nt = 1:ngp-2
        if (ne > nt)
           k(ne,nt) = 0;
        end
    end
end

% Plotting
pcolor(eps_msh,tau_msh,k)
colorbar()
title('Minimal number of iterations for error < \tau')
xlabel('\epsilon_0')
ylabel('\tau')
xlim([0,eps_max])
ylim([0,eps_max])
shading flat
```

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;
```

Solution: The MATLAB code computes $y = \log(x)$, for a given x. The program can be regarded as Newton iterations for finding the zero of

$$f(z) = e^z - x \tag{15}$$

where z is unknown and x is given.

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

Solution: The purpose of the two while loops is to shift the function values of (15) and modify the initial $z_0 = x - 1$ in such a way that good convergence is reached (according to the function derivative).

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

Solution: The `while`-loop computes the zero of (15) using Newton iterations

$$\begin{cases} z^{(0)} = x - 1, \\ z^{(n+1)} = z^{(n)} - \frac{e^{z^{(n)}} - x}{e^{z^{(n)}}}, \quad n \geq 1 \end{cases}$$

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

Solution: This is a correction based termination criterium, see [1, § 2.1.26].

(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`.

Solution: Denote the zero of $f(z)$ with z^*, and $e^{(n)} = z^{(n)} - z^*$. Use Taylor expansion of $f(z),\ f'(z)$:

$$\begin{aligned}
f(z^{(n)}) &= e^{z^*}e^{(n)} + \frac{1}{2}e^{z^*}(e^{(n)})^2 + O((e^{(n)})^3) \\
&= xe^{(n)} + \frac{1}{2}x(e^{(n)})^2 + O((e^{(n)})^3), \\
f'(z^{(n)}) &= e^{z^*} + e^{z^*}e^{(n)} + O((e^{(n)})^2) \\
&= x + xe^{(n)} + O((e^{(n)})^2)
\end{aligned}$$

Considering Newton's Iteration $z^{(n+1)} = z^{(n)} - \frac{f(z^{(n)})}{f'(z^{(n)})}$,

$$\begin{aligned}
e^{(n+1)} &= e^{(n)} - \frac{f(z^{(n)}}{f'(z^{(n)}} \\
&= e^{(n)} - \frac{xe^{(n)} + \frac{1}{2}x(e^{(n)})^2 + O((e^{(n)})^3)}{x + xe^{(n)} + O((e^{(n)})^2)} \\
&\doteq \frac{1}{2}(e^{(n)})^2 \\
&= \cdots \\
&\doteq (\frac{1}{2})^{1+2+\cdots+2^n}(e^0)^{2^{n+1}} \\
&= 2\cdot(\frac{1}{2})^{2^{n+1}}(e^0)^{2^{n+1}} \\
&= 2\cdot(\frac{1}{2}e^0)^{2^{n+1}},
\end{aligned}$$

where $e^0 = z^0 - z^* = x - 1 - \log(x)$. So it is enough for us to determine the number n of iteration steps by $|\frac{e^{(n+1)}}{\log x}| = \texttt{eps}$. Thus

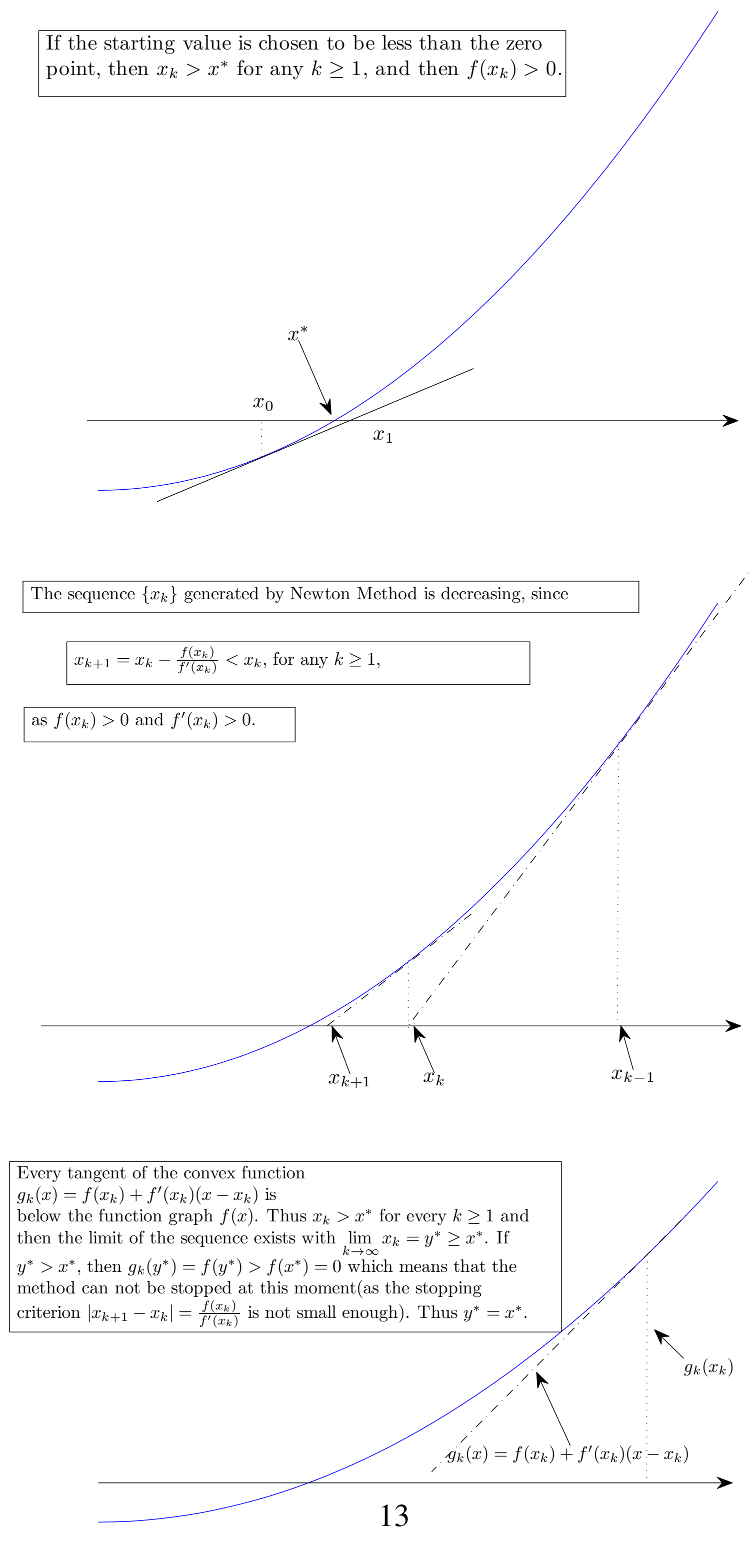
$$\begin{aligned}
n &= \frac{\log(\log(2) - \log(\log(x)) - \log(\texttt{eps})) - \log(\log(2) - \log(|e^0|))}{\log(2)} - 1 \\
&\approx \frac{\log(-\log(\texttt{eps})) - \log(-\log(|e^0|))}{\log(2)} - 1
\end{aligned}$$

The following code is for your reference.

```
function y = myfn(x)
    log2 = 0.693147180559945;
    y = 0;
    while (x > sqrt(2)), x = x/2;  y = y + log2; end
    while (x < 1/sqrt(2)), x = x*2; y = y - log2; end
    z = x-1;
    dz = x*exp(-z)-1;
    e0=z-log(x);
    k=(log(-log(eps))-log(-log(abs(e0))))/log(2);
    for i=1:k
        z = z+dz;
        dz = x*exp(-z)-1;
    end
    y = y+z+dz;
```

Issue date: 08.10.2015
Hand-in: 15.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: July 16, 2016 (v. 1.0).


If the starting value is chosen to be less than the zero point, then $x_k > x^*$ for any $k \geq 1$, and then $f(x_k) > 0$.

The sequence $\{x_k\}$ generated by Newton Method is decreasing, since

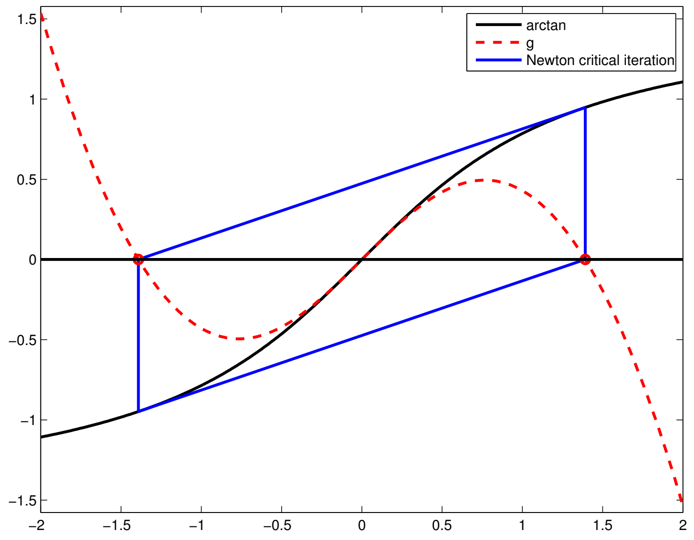
$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} < x_k$, for any $k \geq 1$,

as $f(x_k) > 0$ and $f'(x_k) > 0$.

Every tangent of the convex function
$g_k(x) = f(x_k) + f'(x_k)(x - x_k)$ is
below the function graph $f(x)$. Thus $x_k > x^*$ for every $k \geq 1$ and then the limit of the sequence exists with $\lim\limits_{k\to\infty} x_k = y^* \geq x^*$. If $y^* > x^*$, then $g_k(y^*) = f(y^*) > f(x^*) = 0$ which means that the method can not be stopped at this moment(as the stopping criterion $|x_{k+1} - x_k| = \frac{f(x_k)}{f'(x_k)}$ is not small enough). Thus $y^* = x^*$.

Figure 8: Graphical proof for Problem 2

Figure 9: Newton iterations with $F(x) = \arctan(x)$ for the critical initial value $x^{(0)}$



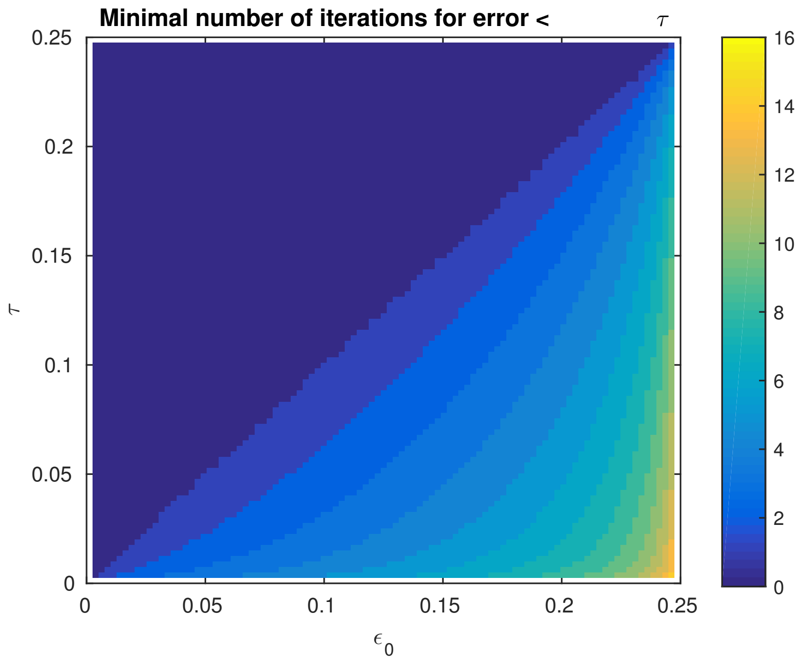


Figure 10: Minimum number of iterations given τ and ϵ_0, Problem 5

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:

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})\,, \\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{y}^{(k)})\,, \end{aligned} \tag{16}$$

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 (16) 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.

Solution: If $\mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{0}$ then $\mathbf{y}^{(k)} = \mathbf{x}^{(k)} + \mathbf{0} = \mathbf{x}^{(k)}$ and $\mathbf{x}^{(k+1)} = \mathbf{y}^{(k)} - \mathbf{0} = \mathbf{x}^{(k)}$.
So, by induction, if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$ then $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every k.

Conversely, if $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$, then, by the recursion of the Newton method:

$$\begin{aligned} \mathbf{F}(x^{(k)}) &= \mathbf{F}(x^{(k)} + D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})) \\ &\Rightarrow D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)}) = 0 \\ &\Rightarrow \mathbf{F}(\mathbf{x}^{(k)}) = 0 \end{aligned}$$

(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 (16) 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.

Solution: See `modnewt.cpp` and `general_nonlinear_solver.h`.

(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 (16) 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 $\arctan'(x) = \frac{1}{1+x^2}$.

Solution: See `modnewt.cpp` and `general_nonlinear_solver.h`. The order of convergence is approximately 3:

sol.	err.	order
5	4.87638	
0.560692	4.87638	
0.155197	0.437068	
0.123644	0.0315725	1.08944
0.123624	2.00333e-05	2.80183
0.123624	5.19029e-15	2.99809

(1d) Write a C++ function `void mod_newt_sys()` that provides an efficient implementation of the method (16) 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.

Solution: See `modnewt.cpp` and `general_nonlinear_solver.h`.

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`.

Solution: See `quasilin.cpp`.

(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.

Solution: See `quasilin.cpp` .

(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.

Solution: The Newton iteration, as provided in the lecture notes, reads:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2}\right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}). \quad (17)$$

(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].

Solution: We replace the inversion with the SMW formula:

$$\begin{aligned}
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2}\right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1}\left(\mathbf{I} - \frac{1}{\|\mathbf{x}^{(k)}\|_2}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}}{1 + \frac{(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|_2}}\right)(\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b} + \mathbf{A}(\mathbf{x}^{(k)})^{-1}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}(\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1}\left(\mathbf{b} + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top(\mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b})}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}\right)
\end{aligned}$$

(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);
```

Solution: See `quasilin.cpp`.

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

Solution: See `quasilin.cpp`.

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.

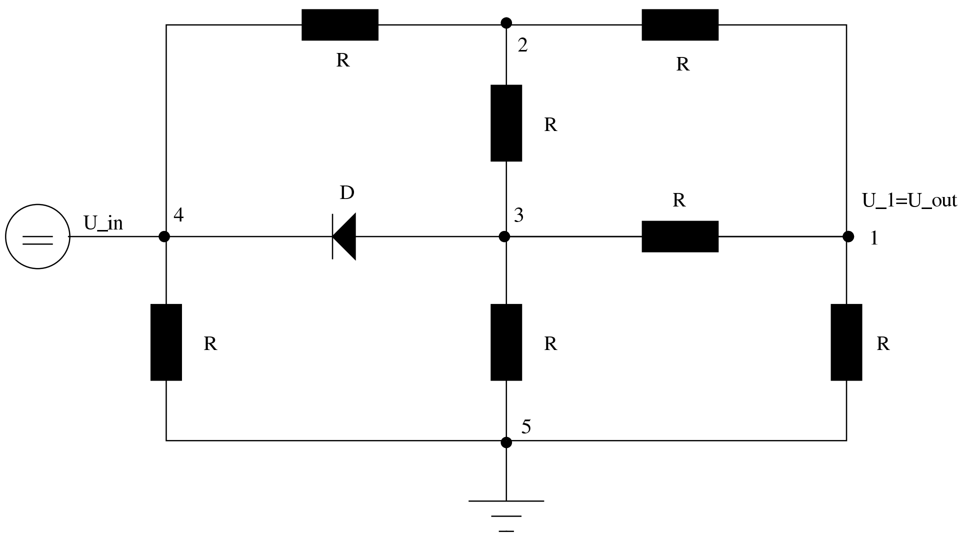


Figure 11: non-linear circuit for Problem 1

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

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

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

Now we consider the circuit depicted in Fig. 11 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).

Solution: We consider Kirchhoff's law $\sum\limits_{k,j} I_{kj} = 0$ for every node k. I_{kj} contributes to the sum if node j is connected to node k trough one element (R,C,L,D).

(1) $3U_1 - 0 - U_2 - U_3 = 0$

(2) $3U_2 - U_1 - U_3 - U = 0$

(3) $3U_3 - 0 - U_1 - U_2 + \alpha\left(e^{\beta\left(\frac{U_3 - U}{U_T}\right)} - 1\right) = 0$

Thus the nonlinear system of equations reads

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 3U_1 - U_2 - U_3 \\ 3U_2 - U_1 - U_3 - U \\ 3U_3 - U_1 - U_2 + \alpha e^{-\frac{\beta}{U_T}U} \cdot e^{\frac{\beta}{U_T}U_3} - \alpha \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad \text{for } \mathbf{u} = \begin{pmatrix} U_1 \\ U_2 \\ U_3 \end{pmatrix}.$$

(3b) Write an EIGEN function

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

that computes the output voltages `Uout` (at node 1 in Fig. 11) 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 $\tau = 10^{-6}$.

Solution: An iteration of the multivariate Newton method is:

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} - \left(D\mathbf{F}(\mathbf{u}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{u}^{(k)})$$

The components of the Jacobian $D\mathbf{F}$ are defined as $D\mathbf{F}(\mathbf{u})_{ij} := \frac{\partial F_i(\mathbf{u})}{\partial u_j}$. For the function obtained in a) we get

$$D\mathbf{F}(\mathbf{u}) = \mathbf{J} = \begin{pmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 + \frac{\alpha\beta}{U_T} e^{\beta\frac{U_3 - U}{U_T}} \end{pmatrix}.$$

For the implementation in Eigen see file `nonlinear_circuit.cpp`.

(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.

Solution: The nonlinear effects can be observed by calculating the differences between the solutions U_{out}, see file `nonlinear_circuit.cpp`.

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

$$z^3 = 1 \tag{18}$$

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 (18) into a system of equations $\mathbf{F}(x, y) = \mathbf{0}$ for a suitable function $\mathbf{F} : \mathbb{R}^2 \mapsto \mathbb{R}^2$.

Solution: We have

$$z^3 - 1 = (x + iy)^3 - 1 = x^3 + 3ix^2y - 3xy^2 - iy^3 - 1 = x^3 - 3xy^2 - 1 + i(3x^2y - y^3) = 0.$$

Thus, equation (18) is equivalent to $\mathbf{F}(x, y) = \begin{pmatrix} x^3 - 3xy^2 - 1 \\ 3x^2y - y^3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

(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.

Solution: The iteration of Newton's method for multiple variables reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(D\mathbf{F}(\mathbf{x}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{x}^{(k)}),$$

where $D\mathbf{F}$ is the Jacobian $D\mathbf{F}(\mathbf{x}) = \begin{pmatrix} 3x^2 - 3y^2 & -6xy \\ 6xy & 3x^2 - 3y^2 \end{pmatrix}$.

(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.

Solution: For each starting point at most `N_it` iterations are accomplished. For a given starting point $\mathbf{x}^{(0)}$, as soon as the condition $|x_1^{(k)} + ix_2^{(k)} - z_i| < 10^{-4}, \ i \in \{1,2,3\}$ is reached, we assign a color depending on the root z_i and on the number of iterations done. Each attractor is associated to red, green or blue; lighter colors correspond to the points with faster convergence. The points that are not converging in `N_it` iterations are white. We set the color scale using the MATLABcommands `colormap` and `caxis`.

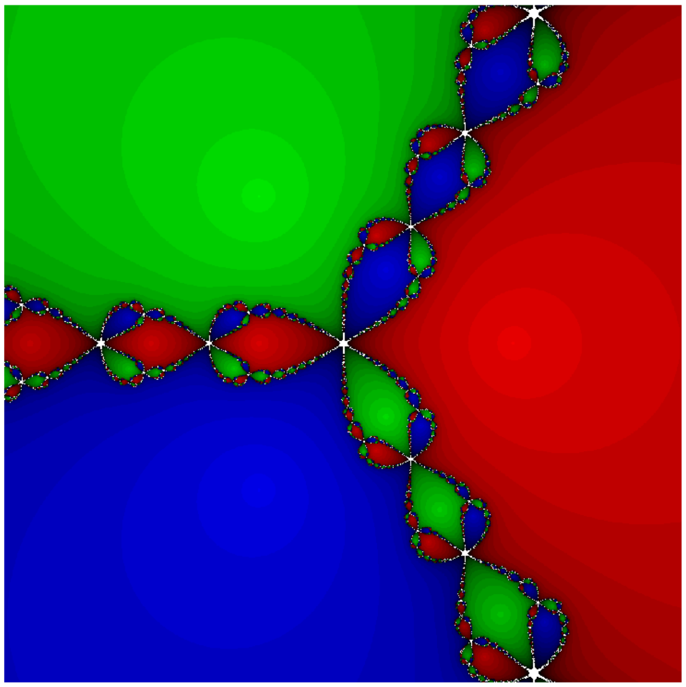


Figure 12: Julia set for $z^3 - 1 = 0$ on a mesh containing 1000×1000 points, `N_it`=20.

Listing 33: `julia_set.m`

```
function julia_set(N_it, num_grid_points)
```

```
close all;
if nargin<2;   N_it = 25;  num_grid_points = 200; end;

% initialize matrix for colors
col = ones(num_grid_points);
a = 2.0;
xx = linspace(-a,+a,num_grid_points);
yy = linspace(-a,+a,num_grid_points);

% roots of unity:
z1 = [ 1;     0];
z2 = [-1/2;  sqrt(3)/2];
z3 = [-1/2; -sqrt(3)/2];

for ny = 1:length(yy)
    for nx = 1:length(xx)
        % for each starting point in the grid
        v = [xx(nx); yy(ny)];

        for k=1:N_it;
            F  = [v(1)^3-3*v(1)*v(2)^2-1;
               3*v(1)^2*v(2)-v(2)^3];
            DF = [3*v(1)^2-3*v(2)^2, -6*v(1)*v(2);
                  6*v(1)*v(2)       , 3*v(1)^2-3*v(2)^2];
            v = v - DF\F;            % Newton update

            if      ((norm(v-z1)) < 1e-4)
                col(ny,nx) = 1 + k;
                   break;
            elseif ((norm(v-z2)) < 1e-4)
                col(ny,nx) = 1 + k + N_it;
                   break;
            elseif ((norm(v-z3)) < 1e-4)
                col(ny,nx) = 1 + k + 2*N_it;
                   break;
            end
        end
    end
```

```
end
st = 1/N_it;          % build a RGB colormap,
% 1s at the beginning to recognize slowly-converging
   points (white)
mycolormap = [ [1,1-st:-st:0, zeros(1,2*N_it)];
               [1,zeros(1,N_it), 1-st:-st:0,
                  zeros(1,N_it)];
               [1,zeros(1,2*N_it), 1-st:-st:0] ]';
% mycolormap = 1 - mycolormap;     % cmy, pyjamas
   version...
% mycolormap = [ [1,1-st:-st:0, st:st:1,    st:st:1,];
%                [1,st:st:1,    1-st:-st:0, st:st:1,];
%                [1,st:st:1,    st:st:1,    1-st:-st:0]
   ]';
colormap(mycolormap);               % built in: colormap
   jet(256);
% this is the command that creates the plot:
pcolor(col);
caxis([1,3*N_it+1]); shading flat;
   axis('square','equal','off');
print -depsc2 'ex_JuliaSet.eps';
```

Issue date: 15.10.2015

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

Version compiled on: July 16, 2016 (v. 1.0).

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

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

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 $(p'(x_1),\dots,p'(x_m))$, when the argument x passes $(x_1,\dots,x_m)$, $m \in \mathbb{N}$ small. Here, p' denotes the *derivative* of the polynomial $p \in \mathcal{P}_n$ interpolating the data points (t_i, y_i), $i = 0,\dots,n$, 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].

Solution: Differentiating the recursion formula [1, (3.2.30)] we obtain

$$
\begin{aligned}
p_i(t) &\equiv y_i, \qquad i = 0,\dots,n,\\
p_i'(t) &\equiv 0, \qquad i = 0,\dots,n,\\
p_{i_0,\dots,i_m}(t) &= \frac{(t-t_{i_0})p_{i_1,\dots,i_m}(t) - (t-t_{i_m})p_{i_0,\dots,i_{m-1}}(t)}{t_{i_m}-t_{i_0}},\\
p'_{i_0,\dots,i_m}(t) &= \frac{p_{i_1,\dots,i_m}(t) + (t-t_{i_0})p'_{i_1,\dots,i_m}(t) - p_{i_0,\dots,i_{m-1}}(t) - (t-t_{i_m})p'_{i_0,\dots,i_{m-1}}(t)}{t_{i_m}-t_{i_0}}.
\end{aligned}
$$

The implementation of the above algorithm is given in file `dipoleval_test.m`.

(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)`.

Solution: See file `dipoleval_test.m`.

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:

```cpp
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`.

Solution: See `linearinterpolant.cpp`.

(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]`).

Solution: See file `horner.cpp`.

(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.

Solution: See file `horner.cpp`.

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

Solution: In both cases, the algorithm requires $\approx n$ multiplications and additions, and so the asymptotic complexity is $O(n)$. The naive implementation also calls the `pow()` function, which may be costly.

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

Solution: See file `horner.cpp`.

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

Solution: Simply exploiting the chain rule of many terms:

$$\omega'(x) = \sum_{i=0}^{n} \prod_{\substack{j=0\\j\neq i}}^{n} (x - t_j)$$

Since $(t_i - t_j) = \delta_{i,j}$, it follows $\omega'(t_i) = \prod_{\substack{j=0\\j\neq i}}^{n}(t_i - t_j)$. Therefore:

$$\begin{aligned} p(x) = \omega(x) \sum_{j=0}^{n} \frac{y_j}{(x - t_j)\omega'(t_j)} &= \sum_{j=0}^{n} \frac{y_j}{(x - t_j)\prod_{\substack{j=0\\j\neq i}}^{n}(t_i - t_j)} \prod_{j=0}^{n}(x - t_j) \\ &= \sum_{j=0}^{n} \frac{y_j}{\prod_{\substack{j=0\\j\neq i}}^{n}(t_i - t_j)} \prod_{\substack{j=0\\j\neq i}}^{n}(x - t_j) = \sum_{j=0}^{n} y_j \prod_{\substack{j=0\\j\neq i}}^{n} \frac{x - t_j}{t_i - t_j}. \end{aligned}$$

Issue date: 22.10.2015

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

Version compiled on: July 16, 2016 (v. 1.0).

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

ETH Zürich
D-MATH

AS 2015

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

$$s_{\alpha,\beta}(x) = \begin{cases} (x+1)^4 + \alpha(x-1)^4 + 1 & x \in [-1,0] \\ -x^3 - 8\alpha x + 1 & x \in (0,1] \\ \beta x^3 + 8x^2 + \frac{11}{3} & x \in (1,2] \end{cases} \tag{19}$$

(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.

Solution: We immediately see that $\alpha = -1$ is necessary to get a polynomial of 3rd degree.

Furthermore, from the condition

$$8 = s_{-1,\beta}(1^-) = s_{-1,\beta}(1^+) = \beta + 8 + \frac{11}{3} \qquad \text{we get} \qquad \beta = -\frac{11}{3}.$$

It remains to check the continuity of s, s', s'' in the nodes 0 and 1. Indeed, we have

$$s'_{-1,\beta}(x) = \begin{cases} 4(x+1)^3 + 4\alpha(x-1)^3 \\ -3x^2 - 8\alpha \\ 3\beta x^2 + 16x \end{cases} \qquad s''_{-1,\beta}(x) \begin{cases} 12(x+1)^2 + 12\alpha(x-1)^2 & -1 \le x \le 0, \\ -6x & 0 < x \le 1, \\ 6\beta x + 16 & 1 < x < 2. \end{cases}$$

Therefore the values in the nodes are

	0^-	0^+	$1-$	1^+
s	$2+\alpha$	1	-8α	$\beta+8+11/3$
s'	$4-4\alpha$	-8α	$-3-8\alpha$	$3\beta+16$
s''	$12+12\alpha$	0	-6	$6\beta+16$

$\alpha=-1,$ $\beta=-11/3\rightarrow$

	0^-	0^+	$1-$	1^+
s	1	1	8	8
s'	8	8	5	5
s''	0	0	-6	-6

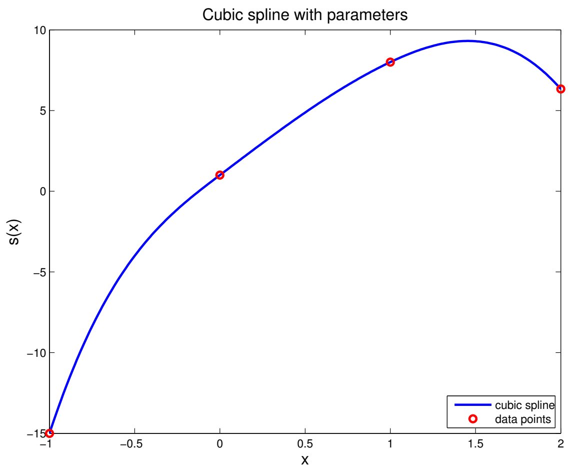
They agree for our choice of the parameters.

(1b) ⊡ Use MATLAB to create a plot of the function defined in (19) in dependance of α and β.

Solution:

Listing 34: Matlab Code for `Cubic Spline`

```matlab
function ex_CubicSpline(alpha,beta)
if nargin==0;  alpha=-1; beta=-11/3;  end;

x1 = -1:0.01:0;
y1 = (x1+1).^4+alpha*(x1-1).^4+1;
x2 = 0:0.01:1;
y2 = -x2.^3-8*alpha*x2+1;
x3 = 1:0.01:2;
y3 = beta*x3.^3 + 8.*x3.^2+11/3;

x = [x1 x2 x3]; y = [y1 y2 y3];
nodes = [-1 0 1 2];
data = [y1(1) y1(end) y2(end) y3(end)];

close all;
plot(x,y,nodes,data,'ro','linewidth',2);
legend('cubic spline', 'data points','Location','SouthEast');
xlabel('x','fontsize',14); ylabel('s(x)','fontsize',14);
title('Cubic spline with parameters','fontsize',14)
print -depsc2 'ex_CubicSpline.eps'
```



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

$$s|_{[t_{j-1},t_j]}(t) = d_j\, \tau^2 + c_j\, 4\, \tau(1-\tau) + d_{j-1}\, (1-\tau)^2 , \quad \tau := \frac{t - t_{j-1}}{t_j - t_{j-1}} , \quad j = 1, \ldots, n , \tag{20}$$

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

Solution: Counting argument, similar to that used to determine the dimensions of the spline spaces. We have $n+1$ unknowns d_k and n unknowns c_j, the constraint $d_0 = s(t_0) =$

$s(t_0+1) = s(t_n) = d_n$ leaves us with a total of $2n$ unknowns. The continuity of the derivatives in the n nodes impose the same number of constraints, therefore the total dimension of the spline space is $2n - n = n$.

(2b) ⚀ What kind of continuity is already guaranteed by the use of the representation (20)?

Solution: We observe that $s(t_j^-) = s|_{[t_{j-1},t_j]}(t_j) = d_j = s|_{[t_j,t_{j+1}]}(t_j) = s(t_j^+)$, thus we get continuity for free. However the derivatives do not necessarily match.

(2c) ⚂ Derive a linear system of equations (system matrix and right hand side) whose solution provides the coefficients c_j and d_j in (20) from the function values $y_j := f(\frac{1}{2}(t_{j-1}+t_j))$, $j = 1, \dots, n$.

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.

Solution: We can plug $t = \frac{1}{2}(t_j + t_{j-1})$ into (20) and set the values equal to y_j. We obtain $\tau = 1/2$ and the following conditions:

$$\frac{1}{4}d_j + c_j + \frac{1}{4}d_{j-1} = y_j, \quad j = 1, ..., n. \tag{21}$$

We obtain conditions on d_j by matching the derivatives at the interfaces. The derivative of the quadratic spline can be computed from (20), after defining $\Delta_j = t_j - t_{j-1}$:

$$s'|_{[t_{j-1},t_j]}(t) = \Delta_j^{-1}\frac{\partial}{\partial\tau}\Big(\tau^2 d_j + 4\tau(1-\tau)c_j + (1-\tau)^2 d_{j-1}\Big) = \Delta_j^{-1}\Big(2\tau d_j + 4(1-2\tau)c_j - 2(1-\tau)d_{j-1}\Big).$$

Setting $\tau = 1$ in $[t_{j-1}, t_j]$ and $\tau = 0$ in $[t_j, t_{j+1}]$, the continuity of the derivative in the node t_j enforces the condition

$$\frac{2d_j - 4c_j}{\Delta_j} = s'|_{[t_{j-1},t_j]}(t_j) = s'(t_j^-) = s'(t_j^+) = s'|_{[t_{j-1},t_j]}(t_j) = \frac{4c_{j+1} - 2d_j}{\Delta_{j+1}};$$

(this formula holds for $j = 1, \dots, n$ if we define $t_{n+1} = t_1 + 1$ and $c_{n+1} = c_1$). Simplifying for d_j we obtain:

$$d_j = \frac{2\frac{c_j}{\Delta_j} + 2\frac{c_{j+1}}{\Delta_{j+1}}}{\frac{1}{\Delta_j} + \frac{1}{\Delta_{j+1}}} = 2\frac{c_j\Delta_{j+1} + c_{j+1}\Delta_j}{\Delta_j + \Delta_{j+1}} = 2\frac{c_j(t_{j+1} - t_j) + c_{j+1}(t_j - t_{j-1})}{t_{j+1} - t_{j-1}}, \qquad j = 1, \dots, n.$$

Plugging this expression into (21), we get the following system of equations:

$$\frac{1}{2}\frac{c_{j+1}(t_j - t_{j-1}) + c_j(t_{j+1} - t_j)}{t_{j+1} - t_{j-1}} + c_j + \frac{1}{2}\frac{c_j(t_{j-1} - t_{j-2}) + c_{j-1}(t_j - t_{j-1})}{t_j - t_{j-2}} = y_j, \qquad j = 1, \dots, n, \tag{22}$$

with the periodic definitions $t_{-2} = t_{n-2} - 1$, $c_0 = c_n$, $c_{-1} = c_{n-1}$. We collect the coefficients and finally we obtain

$$\underbrace{\left\{\frac{1}{2}\frac{t_j - t_{j-1}}{t_j - t_{j-2}}\right\}}_{A_{j-1}} c_{j-1} + \underbrace{\left\{\frac{1}{2}\frac{t_{j+1} - t_j}{t_{j+1} - t_{j-1}} + 1 + \frac{1}{2}\frac{t_{j-1} - t_{j-2}}{t_j - t_{j-2}}\right\}}_{B_j} c_j + \underbrace{\left\{\frac{1}{2}\frac{t_j - t_{j-1}}{t_{j+1} - t_{j-1}}\right\}}_{C_{j+1}} c_{j+1} = y_j, \quad j = 1, \dots, n. \tag{23}$$

If we have an equidistant grid with $t_j - t_{j-1} = 1/n$, this simplifies to

$$\frac{1}{4}c_{j-1} + \frac{3}{2}c_j + \frac{1}{4}c_{j+1} = y_j, \qquad\qquad j = 1, \dots, n.$$

The system of equations in matrix form looks as follows:

$$\begin{pmatrix} B_1 & C_2 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & A_0 \\ A_1 & B_2 & C_3 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_3 & C_4 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & A_{n-2} & B_{n-1} & C_n \\ C_{n+1} & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & A_{n-1} & B_n \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{n-1} \\ c_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix}. \tag{24}$$

(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 $\frac{1}{2}(t_{j-1} + t_j)$, $j = 1, \dots, n$, 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).

Solution: See Listing 35.

Listing 35: Construction of the quadratic spline and evaluation.

```
% part d of quadratic splines problem
% given: t = nodes (n-1 vect.),
%        y = data (n vect.),
%        x = evaluation pts (N vect.)
% create the interpolating quadratic spline and evaluate
   in x
function eval_x = quadspline_better(t,y,x)
  % the number of nodes:
  n = length(y);          % N = length(x);
  % ensure nodes and data are line vectors:
  t = t(:)'; y = y(:)';
  % create (n+3) extended vectors using the periodicity:
  ext_t = [t(end)-1,0,t,1,1+t(1)];
  % increments in t:
  de_t = diff(ext_t);                          % (n+2)
  dde_t = ext_t(3:end) - ext_t(1:end-2);    % (n+1)

  % build the three n-vectors that define the matrix
  A = de_t(2:n+1) ./ (2*dde_t(1:n));
  B = 1 + de_t(3:n+2) ./ (2*dde_t(2:n+1)) + ...
      de_t(1:n) ./ (2*dde_t(1:n));
  C = de_t(2:n+1) ./ (2*dde_t(2:n+1));
  % Assembly of the matrix, be careful with spdiags and
     transpose!
  M = spdiags([C; B; A].', (-1:1), n,n).';
  %M(1,n) = A(1);
  %M(n,1) = C(n);

  %%%% Employ SMW formula for a rank 1 modification,
     which allows to
  %%%% expolit the structure of the matrix
  u = zeros(n,1);
  v = zeros(n,1);
  u(n,1) = 1;
  u(1,1) = 1;
  v(1,1) = C(n);
  v(n,1) = A(1);
```

```matlab

M(1,1) = M(1,1) - C(n);
M(n,n) = M(n,n) - A(1);

% solve the system for the coefficients c:
%c = (M\(y(:))).';
Minvy = M \ y(:);

c = Minvy - (M\(u*(v'*Minvy))) / (1 + v'*(M\u));
c = c';

% compute the coefficients d_1,...,d_n:
ext_c = [c,c(1)];
d = 2*( ext_c(2:end).*de_t(2:n+1)+ c.*de_t(3:n+2) )...
    ./dde_t(2:n+1);
ext_d = [d(n),d];
% evaluate the interpolating spline in x:
eval_x = zeros(1,length(x));

% loop over the n intervals
% loop over the n intervals
ind_end = 0;
for i=1:n
    left_t = ext_t(i+1);
    right_t = ext_t(i+2);
    % find the points in x in the interval [t(i-1),
       t(i)):
    ind_start = ind_end+1;
    if x(ind_start) ≥ left_t
        while ind_end < length(x) && x(ind_end+1) <
           right_t
            ind_end = ind_end + 1;
        end
    end
    ind_eval = ind_start:ind_end;
    tau = (x(ind_eval) - left_t)./( right_t-left_t );
    eval_x(ind_eval) = d(i)*tau.^2 + ...
        c(i)*4*tau.*(1-tau) + ext_d(i)*(1-tau).^2;
    end

end
```

It is important not to get lost in the indexing of the vectors. In this code they can be represented as:
`t=(t1, t2, ..., t{n-1})` length $= n - 1$,
`ext_t=(t{n-1}-1, t0=0, t1, t2,..., t{n-1}, tn=1, 1+t1)` length $= n + 3$,
`de_t=(1-t{n-1}, t1, t2-t1,..., t{n-1}-t{n-2}, 1-t{n-1}, t1),`
`dde_t=(t1+1-t{n-1}, t2, t3-t1,..., 1-t{n-2}, t1+1-t{n-1}).`
The vectors `A,B,C,c,d` have length n and correspond to the definitions given in (2c) (with the indexing from 1 to n).

(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.

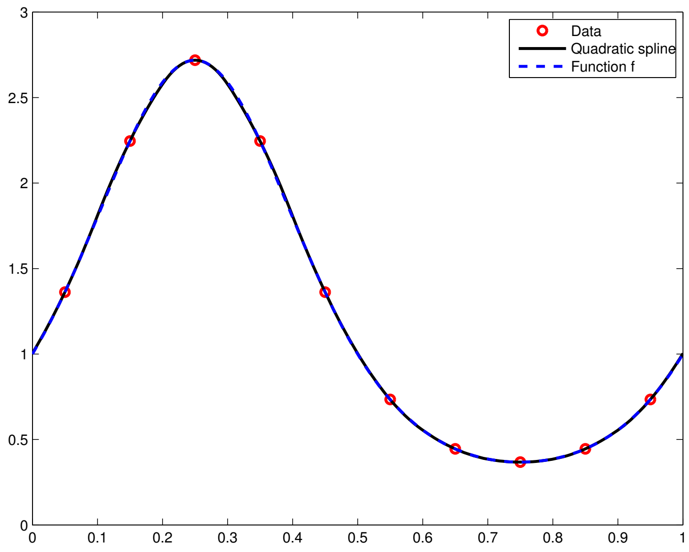
Solution: See code 36 and Figure 13.

Listing 36: Plot of the quadratic splines.

```
function ex_QuadraticSplinesPlot(f,n,N, func)

if nargin<3
    f = @(t) exp(sin(2*pi*t));    % function to be
        interpolated
    n = 10;                       % number of subintervals
    N = 200;                      % number of evaluation
        points
    func = @quadspline_better;
end

t = 1/n:1/n:1-1/n;                % n-1 equispaced nodes
    in (0,1)
x = linspace(0,1,N);              % N evaluation points
x = x(0 ≤ x & x < 1);            % be sure x belongs to
    [0,1)
middle_points = ([t,1]+[0,t])/2; % computes the n middle
```

```
    points [0,t,1]
y = f(middle_points);                 % evaluate f on the
    middle_points

% compute and evaluate the quadratic spline
eval_x = func(t,y,x);
%eval_x_better = quadspline(t,y,x);

close all; figure
plot(middle_points,y,'ro',x,eval_x,'k',x,f(x),'--','linewidth',2);
legend('Data','Quadratic spline','Function f');
print -depsc2 'interpol_quad_T2.eps'

end

% test:
% ex_QuadraticSplinesPlot(@(t)sin(2*pi*t).^3, 10,500)
```

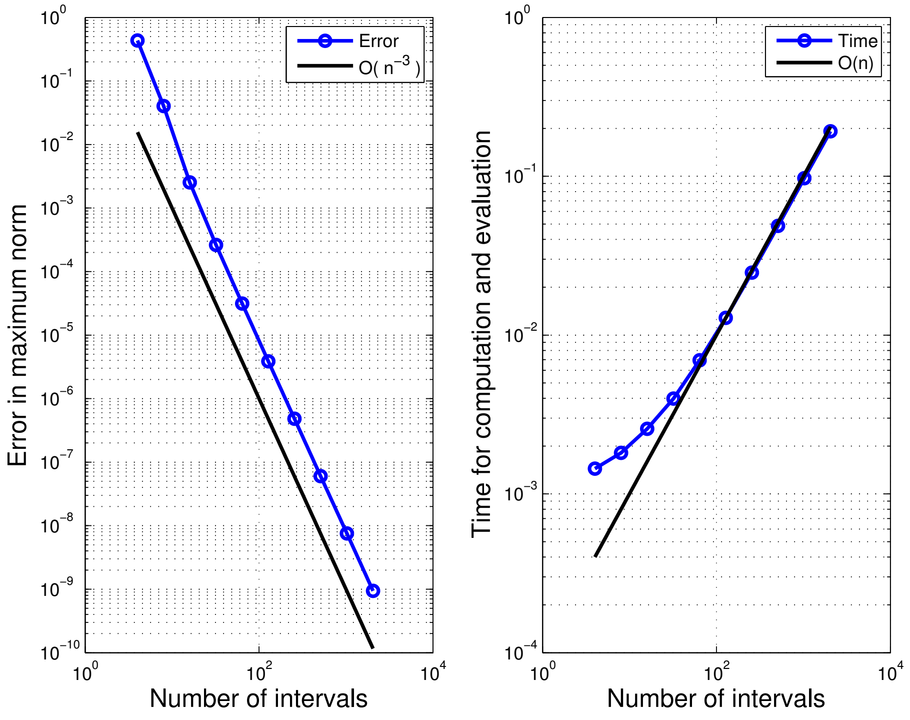
Figure 13: Quadratic spline interpolation of f in 10 equispaced nodes.



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

Solution: The complexity is linear both in N (sequential evaluation in different points) and in n, provided one exploits the sparse structure of the system and uses a rank-one modification to reduce the system to the inversion of a (strictly diagonally dominant) tridiagonal matrix.

Figure 14: Error and timing for quadratic splines in equispaced nodes.



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 γ and $\widetilde{\gamma} := \gamma \circ h$ 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: $t_k = k\Delta t,\ \Delta t = \dfrac{1}{n}$ (25)

❷ segment length parametrization: $t_k = \dfrac{\sum_{l=1}^{k} |\mathbf{p}_l - \mathbf{p}_{l-1}|}{\sum_{l=1}^{n} |\mathbf{p}_l - \mathbf{p}_{l-1}|}$. (26)

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 $\mathbf{p}_i \in \mathbb{R}^2$, $i = 0,\dots,n$, 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 $(t_0, t_1, \dots, t_n) \in \mathbb{R}^{n+1}$ in the parameter domain and N is the number of equidistant sampling points.

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

Solution: See Listing 37.

Listing 37: Matlab Code for `curveintp`

```
function [pol spl pch] = curveintp (xy,t,tt)

x = xy(1,:);
y = xy(2,:);

% compute the slopes at the extremes (necessary for
   complete spline):
x_start = (x(2)-x(1)) ./ (t(2)-t(1));
y_start = (y(2)-y(1)) ./ (t(2)-t(1));
x_end = (x(end)-x(end-1)) ./ (t(end)-t(end-1));
y_end = (y(end)-y(end-1)) ./ (t(end)-t(end-1));
```

```

% compute the interpolants
polx = intpolyval(t,x,tt);
poly = intpolyval(t,y,tt);

% complete splines, using the extended vector
splx = spline(t,[x_start x x_end],tt);
sply = spline(t,[y_start y y_end],tt);
pchx = pchip(t,x,tt);
pchy = pchip(t,y,tt);

pol = [polx; poly];
spl = [splx; sply];
pch = [pchx; pchy];

end
```

(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 (25) and (26)

Solution: See Listing 38 and Figure 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 (25) and (26).

Solution: See Listing 37 and Figure 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 $\mathbf{p}_{n\text{-}1}$ 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 (25) and (26).

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

Solution: The code for the interpolation of the heart:

Listing 38: Matlab Code for `heart_Sol`

```
function heart_Sol ()

xy = heart();
n = size(xy,2) - 1;

%evaluation points:
tt = 0:0.005:1;

figure;
hold on;

% equidistant parametrization of [0,1]:
t_eq = (0:n)/n;
[pol spl pch] = curveintp (xy,t_eq,tt);
subplot(1,2,1); xlabel('Equidistant Parametrization');
plot_interpolations (xy,pol,spl,pch);

% segment length parametrization of [0,1]:
t_seg = segment_param(xy);
[pol spl pch] = curveintp (xy,t_seg,tt);
subplot(1,2,2); xlabel('Segment Length Parametrization');
plot_interpolations (xy,pol,spl,pch);

hold off;
print -depsc2 '../PICTURES/ex_CurveIntp.eps'
```

```

end

% segment length parametrization of [0,1]:
function t_seg = segment_param (xy)
  increments = sqrt(sum(diff(xy,1,2).^2));
  t_seg = cumsum(increments);
  t_seg = [0,t_seg/t_seg(end)];
end

% plotting function
function plot_interpolations (xy,pol,spl,pch)
  plot(xy(1,:),xy(2,:),'o', pol(1,:),pol(2,:),'-.', ...
       spl(1,:),spl(2,:),'-', pch(1,:),pch(2,:),'--',
          'linewidth',2);
  axis equal;
  axis([-105 105 -105 105]);
  legend('data','polynomial','spline','pchip','Location','Southoutside'
end
```

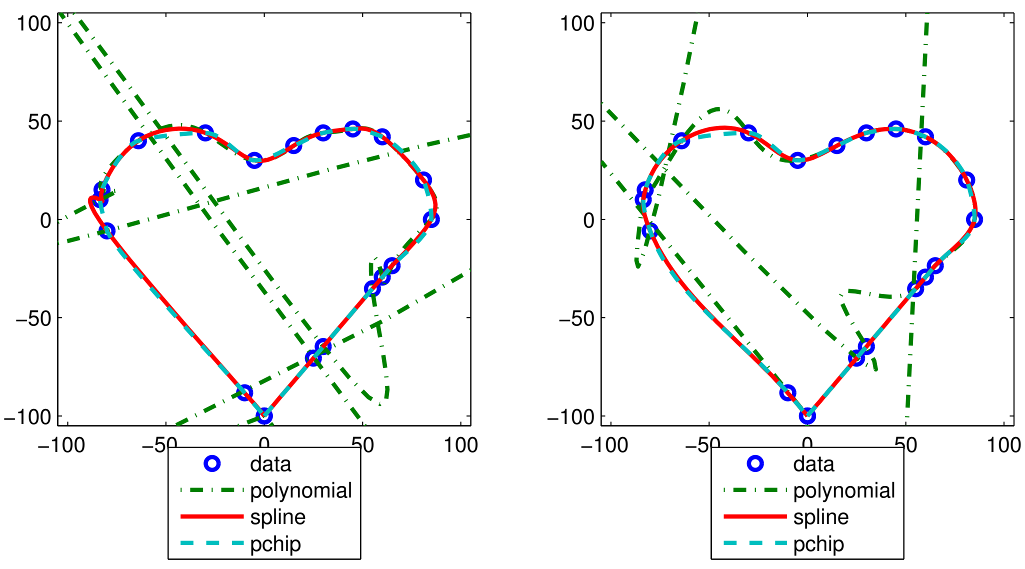


Figure 15: Plots for different parametrizations and interpolation schemes

Remarks:

- Note that `pchip` is a non-linear interpolation, so there is no linear mapping (matrix) from the nodes to the polynomial coefficients.
- Global polynomial interpolation fails. The degree of the polynomial is too high (n), and the typical oscillations can be observed.
- In this case, when two nodes are too close, spline interpolation with equidistant parametrization introduces small spurious oscillations.

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

$$x^* = \lim_{n\to\infty} T(n) \; , \tag{27}$$

with a function $T : \{n, n+1, \dots\} \mapsto \mathbb{R}$. 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 $T(n_0), T(n_1), \dots, T(n_k)$, $k \in \mathbb{N}$, and to consider them as the values $g(1/n_0), g(1/n_1), \dots, g(1/n_k)$ of a continuous function $g :]0, 1/n_{\min}] \mapsto \mathbb{R}$, for which, obviously

$$x^* = \lim_{h\to 0} g(h) \; . \tag{28}$$

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 $p_{k-1}(n_j^{-1}) = T(n_j)$, $j = 1, \dots, k$. 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, $k = 1, \dots, 7$.

Solution: See `pi_approx.cpp`.

Issue date: 29.10.2015

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

Version compiled on: July 16, 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.

Solution: Let $h_i := t_i - t_{i-1}$. Given the natural condition on the spline, one can remove the columns relative to $c_0 := s'(t_0)$ and $c_n := s'(t_n)$ from the system matrix, which becomes:

$$\mathbf{A} := \begin{pmatrix} 2/h_1 & 1/h_1 & 0 & 0 & \cdots & & 0 \\ b_0 & a_1 & b_1 & 0 & \cdots & & 0 \\ 0 & b_1 & a_2 & b_2 & \ddots & & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & b_{n-3} & a_{n-2} & b_{n-2} & 0 \\ 0 & & \cdots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & & \cdots & \cdots & 0 & 1/h_n & 2/h_n \end{pmatrix}, a_i := \frac{2}{h_i} + \frac{2}{h_{i+1}}, b_i := \frac{1}{h_{i+1}} \tag{29}$$

$$\mathbf{c} := [c_0, c_1, \ldots, c_n] \tag{30}$$

$$\mathbf{b} := [r_0, \ldots, r_n] \tag{31}$$

Where $c_i := s'(t_i)$. We define $r_i := 3\left(\frac{y_i - y_{i-1}}{h_i^2} + \frac{y_{i+1} - y_i}{h_{i+1}^2}\right)$, $i = 1, \ldots, n-1$, and

$$r_0 = 3\frac{y_1 - y_0}{h_1^2}, r_n = 3\frac{y_n - y_{n-1}}{h_n^2}$$

The system becomes $\mathbf{Ac} = \mathbf{b}$.

(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.

Solution: Notice that $a_i := \frac{2}{h_i} + \frac{2}{h_{i+1}} > \frac{1}{h_{i+1}} + \frac{1}{h_i} =: b_i + b_{i-1}$. The matrix is (strictly) diagonally dominant and, therefore, invertible.

(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:

```cpp
class NatCSI {
public:
    //! \brief Build the cubic spline interpolant with
       natural boundaries
    //! Setup the data structures you need.
    //! Pre-compute the coefficients of the spline
       (solve system)
    //! \param[in] t, nodes of the grid (for pairs (t_i,
       y_i)) (sorted!)
    //! \param[in] y, values y_i at t_i (for pairs (t_i,
       y_i))
    NatCSI(const const std::vector<double> & t, const
       const std::vector<double> & y);

    //! \brief Interpolant evaluation at x
    //! \param[in] x, value x where to evaluate the
       spline
    //! \return value of the spline at x
    double operator() (double x) const;

private:
    // TODO: store data for the spline
};
```

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

Solution: See `natcsi.cpp`.

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 ($\to$ [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 $j = 1, \dots, n-1$.

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

HINT: Exploit the phenomenon explained next to [1, Fig. 117].

Solution: Without loss of generality assume that $t_0 < t_1 < \dots < t_n$. For $j = 0, \dots, n$ let $\mathbf{s}^{(j)} \in \mathbb{R}^{n+1}$ be defined by

$$s_i^{(j)} = \begin{cases} 0 & i = 0, \dots, j-1 \\ 1 & i = j, \dots, n. \end{cases}$$

Clearly, $(t_i, s_i^{(j)})_i$ are monotonic increasing data, according to [1, Def. 3.3.3].

Take $j = 0, \dots, n$. Note that $\mathbf{s}^{(j)}$ has a local extremum in t_i for every $i = 1, \dots, n-1$. Thus, since I is monotonicity preserving (see [1, § 3.3.6]), $\mathsf{I}(\mathbf{s}^{(j)})$ has to be flat in t_i for every $i = 1, \dots, n-1$ (see [1, Section 3.4.2]). As a consequence,

$$(\mathsf{I}(\mathbf{s}^{(j)}))'(t_i) = 0, \quad i = 1, \dots, n-1. \tag{32}$$

Note now that $\{\mathbf{s}^{(j)} : j = 0, \dots, n\}$ is a basis for $\mathbb{R}^{n+1}$ (indeed, they constitute a linearly independent set with cardinality equal to the dimension of the space). As a consequence, every $\mathbf{y} \in \mathbb{R}^{n+1}$ can be written as a linear combination of the $\mathbf{s}^{(j)}$s, namely

$$\mathbf{y} = \sum_{j=0}^n \alpha_j \mathbf{s}^{(j)}.$$

Therefore, by the linearity of I and using (32), for every $i = 1, \dots, n-1$ we obtain

$$(\mathsf{I}(\mathbf{y}))'(t_i) = \left(\mathsf{I}\left(\sum_{j=0}^n \alpha_j \mathbf{s}^{(j)}\right)\right)'(t_i) = \left(\sum_{j=0}^n \alpha_j \mathsf{I}(\mathbf{s}^{(j)})\right)'(t_i) = \sum_{j=0}^n \alpha_j \mathsf{I}(\mathbf{s}^{(j)})'(t_i) = 0,$$

as desired.

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

$$(f - \mathcal{H}f)(x) = \frac{1}{24} f^{(4)}(\tau)(x-a)^2(x-b)^2. \tag{33}$$

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

$$\varphi(t) := f(t) - (\mathcal{H}f)(t) - C(t-a)^2(t-b)^2. \tag{34}$$

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.

Solution: Fix $x \in]a,b[$. Define:

$$\varphi(t) := f(t) - \mathcal{H}f(t) - C(t-a)^2(t-b)^2 \tag{35}$$

with

$$C := \frac{f(x) - \mathcal{H}f(x)}{(x-a)^2(x-b)^2}.$$

Then $\varphi(a) = \varphi(b) = \varphi'(a) = \varphi'(b) = 0$ (using the definition of $\mathcal{H}$). Moreover $\varphi(x) = 0$ (by construction). Therefore, by Rolle's theorem (φ has at least two local extrema), $\exists \xi_1, \xi_2 \in]a, b[, \xi_1 \neq \xi_2$ such that $\varphi'(\xi_1) = \varphi'(\xi_2) = 0$.
$\Rightarrow \varphi'$ has at least 4 zeros in $[a, b]$ (a, b, ξ_1 and ξ_2 are pairwise distinct).
$\Rightarrow \varphi''$ has at least 3 zeros in $[a, b]$ (φ' has at least 3 local extrema).
$\Rightarrow \varphi^{(3)}$ has at least 2 zeros in $[a, b]$ (φ'' has at least 2 local extrema).
$\Rightarrow \varphi^{(4)}$ has at least 1 zero in $[a, b]$ ($\varphi^{(3)}$ has at least one local extrema), let τ be such zero.
$\Rightarrow 0 = \varphi^{(4)}(\tau) = f^{(4)}(\tau) - 24C$.
$\Rightarrow C = \frac{1}{24} f^{(4)}(\tau)$. $\square$

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
>
> $$\mathcal{T} = \mathcal{T} \cup \left\{ \operatorname*{argmax}_{t \in \mathcal{S}} |f(t) - \mathsf{I}_{\mathcal{T}}(t)| \right\} , \tag{36}$$
>
> where $\mathsf{I}_{\mathcal{T}}$ is the polynomial interpolation operator for the node set $\mathcal{T}$, until
>
> $$\max_{t \in \mathcal{S}} |f(t) - \mathsf{I}_{\mathcal{T}}(t)| \leq \mathtt{tol} \cdot \max_{t \in \mathcal{S}} |f(t)| . \tag{37}$$

(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,

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

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).

Solution: See Listing 39.

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

$$\max_{t \in \mathcal{S}} |f(t) - \mathsf{T}_{\mathcal{T}}(t)| \tag{38}$$

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

Solution: See Listing 39.

Listing 39: Implementation of the function `adaptivepolyintp`

```matlab
function [t,errs] = adaptivepolyintp(f,a,b,tol,N)
% Adaptive polynomial interpolation of function f:[a,b] -> R.
% argument 'f' must be a handle that allows vectorized evaluation,
% argument 'tol' specifies the relative tolerance,
% the optional argument 'N' passes the number of sampling points.
if (nargin < 5), N = 1000; end
sp = (a:(b-a)/N:b);        % N+1 equdistant sampling points
fvals = f(sp);             % evaluate function at sampling points
fmax  = max(abs(fvals)); % Maximum of f in sampling points
t = sp(floor(N/2));        % set of interpolation nodes
y = fvals(floor(N/2));     % function values at interpolation nods
errs = [];
for i=1:N
  err = abs(fvals - intpolyval(t,y,sp)); % modulus of interpolation error
  [mx,idx] = max(err); errs = [errs, mx];
  % finishes, once maximal pointwise interpolation error below threshold
  if (mx < tol*fmax), return; end
  t = [t,sp(idx)];
  y = [y,fvals(idx)];
end
error('Desired accuracy could not be reached');
```

(4c) ⚁ For $f_1(t) := \sin(e^{2t})$ and $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$ plot the quantity from (38) 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`.

Solution: See Listing 40 and Figure 16.

Listing 40: Implementation of the function `plot_adaptivepolyintp`

```
function plot_adaptivepolyintp ()
f1 = (t) sin(exp(2*t));f2 =
    (t) sqrt(t) ./ ( 1 + 16*t.^2 );
a = 0; b = 1; tol = 1e-6;
[¬,err1] = adaptivepolyintp (f1,a,b,tol);
[¬,err2] = adaptivepolyintp (f2,a,b,tol);

semilogy(1:length(err1), err1, 'r-', 1:length(err2),
    err2, 'b-');
legend('f_1', 'f_2');
title('{\bf Error decay for adaptive polynomial
    interpolation}');
xlabel('{\bf number of nodes}');
ylabel('{\bf error}');

print -depsc '../PICTURES/plot_adaptivepolyintp.eps'
```

Issue date: 05.11.2015

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

Version compiled on: July 16, 2016 (v. 1.0).

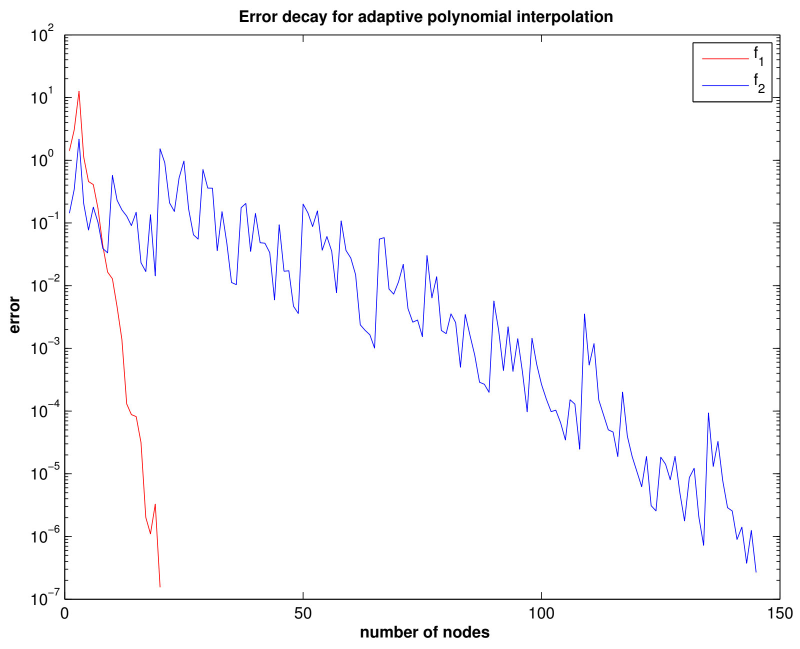


Figure 16: The quantity from (38) versus the number of interpolation nodes.

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. 142].

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. 156],

$$\gamma_\rho(\theta) := \cos(\theta - i\log(\rho)) , \quad \forall 0 \leq \theta \leq 2\pi , \quad \rho > 1 . \tag{39}$$

(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 $\gamma : I \to \mathbb{R}^2$:

$$|\gamma| = \int_I \|\dot{\gamma}(\tau)\| d\tau, \tag{40}$$

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.

Solution: $\frac{\partial \gamma_\rho(\theta)}{\partial \theta} := -\sin(\theta - i\log(\rho))$, therefore:

$$\begin{aligned}
|\gamma_\rho| &= \int_{[0,2\pi]} |\sin(\tau - i\log(\rho))| d\tau && (41)\\
&= \frac{1}{2\rho}\int_{[0,2\pi]} \sqrt{\sin^2(\tau)(1+\rho^2)^2 + \cos^2(\tau)(1-\rho^2)^2} d\tau && (42)\\
&\leq \frac{1}{2\rho}\int_{[0,2\pi]} \sqrt{2+2\rho^4} d\tau \qquad \leq \frac{1}{\rho}\pi\sqrt{2(1+\rho^4)} && (43)
\end{aligned}$$

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].

Solution: f is analytic in $D := \mathbb{C} \setminus \{\frac{2}{3}\pi i c - \frac{1}{3}\pi i \mid c \in \mathbb{Z}\}$. In fact, $g(t) := 1 + exp(-3t)$ is an entire function, whereas $h(x) := \frac{1}{x}$ is analytic in $\mathbb{C}\setminus\{0\}$. Therefore, using [1, Thm. 4.1.65], f is analytic in $\mathbb{C} \setminus \{z \in \mathbb{C} \mid g(z) = 0\} =: \mathbb{C} \setminus S$. Let $z := a + ib$, $a, b \in \mathbb{R}$. Since:

$$\begin{aligned}
-1 = \exp(z) = \exp(a+ib) = \exp(a)(\cos(b) + i\sin(b)) \Leftrightarrow a = 0, b \in 2\pi\mathbb{Z} + \pi && (44)\\
\exp(z) = -1 \Leftrightarrow z \in i(2\pi\mathbb{Z} + \pi) && (45)\\
\exp(-3z) = -1 \Leftrightarrow z \in \frac{i(2\pi\mathbb{Z} - \pi)}{3} && (46)
\end{aligned}$$

Therefore $S = \frac{2}{3}\pi i\mathbb{Z} - \frac{1}{3}\pi i$.

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

$$\min_{\rho>1} \frac{\max_{z\in\gamma_\rho}|f(z)|}{d([-1,1],\gamma_\rho)}, \tag{47}$$

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

$$d([a,b],\gamma) := \inf\{|z-t| \mid z \in \gamma, t \in [a,b]\}. \tag{48}$$

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 $\sinh$.

Solution: The ellipse must be restricted such that the minor axis has length $\leq 2\pi/3$ (2 times the smallest point, in absolute value, where f is not-analytic). Since this corresponds to the imaginary part of $\gamma_\delta(\theta)$, when $\theta = \pi/2$, we find:

$$\cos(\pi/2 - i\log(\rho_{max})) = 1/3\pi i \Leftrightarrow \sinh(\log(\rho_{max})) = \pi/3 \Leftrightarrow \rho_{max} = \exp(\sinh^{-1}(\pi/3)).$$

See `cheby_approx.m` and `cheby_analytic.m` for the MATLAB code.

(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 $\leq n$.

Solution: Let M be the approximation of (1c). Then

$$\|f - L_n f\|_{L^\infty([-1,1])} \lesssim \frac{M\sqrt{2(\rho^{-2} + \rho^2)}}{\rho^{n+1} - 1}.$$

(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`.

Solution: See `cheby_analytic.m`.

(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.

Solution: The rescaled function $\Phi^* f$ will have a different domain of analyticity and a different growth behavior in the complex plane. The larger a, the closer the pole of $\Phi^* f$ will move to $[-1,1]$, the more the choice of the ellipses is restricted (i.e. ρ_{max} becomes smaller). This will result in a larger bound.

Using [1, Eq. (4.1.99)], if follows immediately that the asymptotic behaviour of the interpolation does not change after rescaling of the interval. In fact, if Φ^* is the affine pullback from $[-a,a]$ to $[-1,1]$, then:

$$\|f - (\hat{L_n})f\|_{L^\infty([-a,a])} = \|\Phi^* f - L_n \Phi^* f\|_{L^\infty([-1,1])},$$

where $(\hat{L_n})$ is the interpolation on $[-a,a]$.

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

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

(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].

Solution: Equation [1, (4.5.12)] says

$$\|f - \mathsf{I}_\mathcal{T} f\|_{L^\infty(I)} \leq \frac{1}{2n^2} \left\|f^{(2)}\right\|_{L^\infty(I)},$$

because the meshwidth is $h = 1/n$. So, the convergence is quadratic, i.e., algebraic with order 2.

(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.

Solution: If $\alpha = 1$, $f(t) = t$ clearly belongs to $C^\infty(I)$. If $0 < \alpha < 1$, $f'(t) = \alpha t^{\alpha-1}$ blows up to infinity for t going to 0, therefore $f \in C^0(I) \setminus C^1(I)$. If $1 < \alpha < 2$, f' is continuous but $f''(t) = \alpha(\alpha-1)t^{\alpha-2}$ blows up to infinity for t going to 0, therefore $f \in C^1(I) \setminus C^2(I)$.

More generally, for $\alpha \in \mathbb{N}$ we have $f(t) = t^\alpha \in C^\infty(I)$; on the other hand if $\alpha > 0$ is not an integer, $f \in C^{\lfloor\alpha\rfloor}(I)$, where $\lfloor\alpha\rfloor = \mathrm{floor}(\alpha)$ is the largest integer not larger than α.

(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);
```

Solution: The interpolant is implemented in Listing 41, the convergence for our choice of f is studied in file `PWlineConv.m` and the results are plotted in Figure 17. The convergence is clearly algebraic, the rate is equal to α if it is smaller than 2, and equal to 2 otherwise. In brief, we can say that the order is $\min\{\alpha, 2\}$.

Be careful with the case $\alpha = 1$: here the interpolant gets exactly the solution, with every mesh.

Listing 41: Piecewise linear interpolation.

```
function  y_ev = PWlineIntp (t,y,t_ev)
```

```
% compute and evaluate piecewise linear interpolant
% t and y      data vector of the same size
% t_ev         vector with evaluation points
% --> y_ev     column vector with evaluations in t_ev

t=t(:);  y = y(:);  t_ev = t_ev(:);
n = size(t,1)-1;     % # intervals
if n≠size(y,1)-1; error('t, y must have the same size');
   end;

[t,I] = sort(t);     % sort t and y if not sorted
y = y(I);

y_ev = zeros(size(t_ev));
for k=1:n
    t_left  = t(k);
    t_right = t(k+1);
    ind = find ( t_ev ≥ t_left & t_ev < t_right );
    y_ev(ind) = y(k) +
       (y(k+1)-y(k))/(t_right-t_left)*(t_ev(ind)-t_left);
end
% important! take care of last node:
y_ev(find(t_ev == t(end))) = y(end);
```

(2d) ⚁ In which mesh interval do you expect $|f - \mathsf{I}_{\mathcal{T}} f|$ 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?

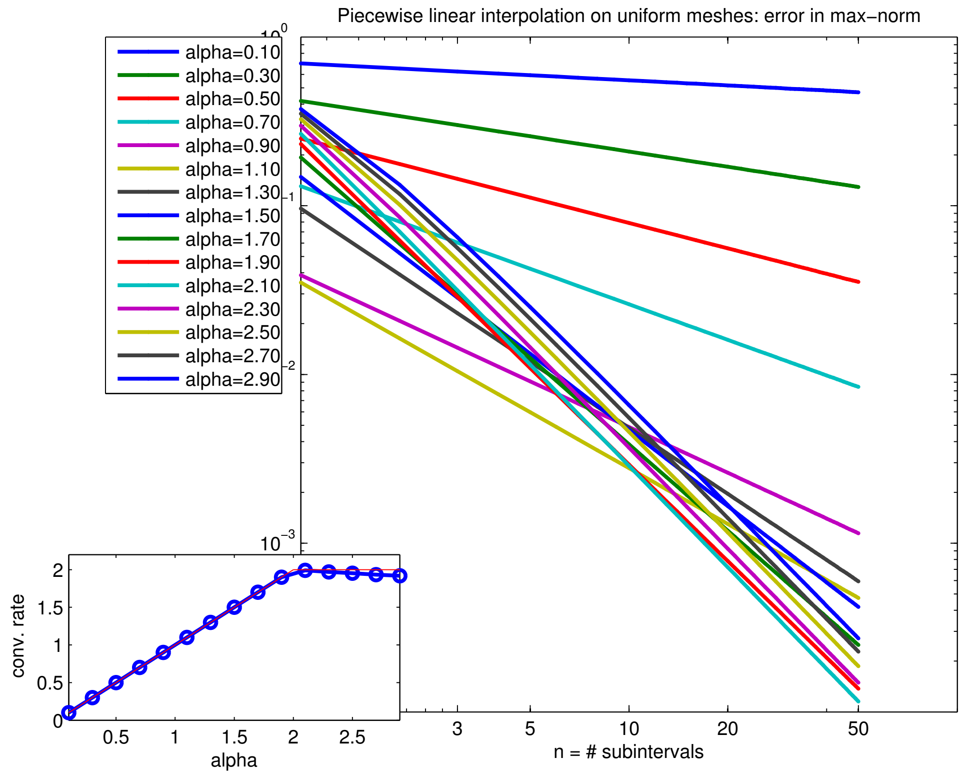
Solution: The error representation [1, (4.1.38)] in the linear case ($n = 1$) reads

$$\forall\, t \in (t_j, t_{j+1}) \qquad \left| f(t) - \left(\mathsf{I}_{\mathcal{T}} f\right)(t) \right| = \frac{1}{2} |f''(\tau_t)\ (t - t_j)(t - t_{j+1})|$$
$$\leq \frac{1}{8} |f''(\tau_t)|\ (t_{j+1} - t_j)^2 = \frac{1}{8n^2} |f''(\tau_t)|,$$

for some $\tau_t \in (t_j, t_{j+1})$. Therefore the error can be large only in the subintervals where the second derivative of f is large. But

$$|f''(t)| = \left|\alpha(\alpha - 1) t^{\alpha - 2}\right|$$

Figure 17: h-convergence of piecewise linear interpolation for $f(t) = t^{\alpha}$, $\alpha = 0.1, 0.3, \ldots, 2.9$. The convergence rates are shown in the small plot.



is monotonically decreasing for $0 < \alpha < 2$, therefore we can expect a large error in the first subinterval, the one that is closer to 0.

In line 23 of the code in `PWlineConv.m`, we check our guess: the maximal error is found in the first interval for every $\alpha \in (0, 2)$ $(\alpha \neq 1)$ and in the last one for $\alpha > 2$.

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

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

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

Solution: From (2d) we expect that the maximum is taken in the first subinterval. For every $t \in (0, 1/n)$ and $0 < \alpha < 2$ ($\alpha \neq 1$) we compute the minimum of the error function φ

$$\begin{aligned}
\varphi(t) &= f(t) - \big(\mathsf{I}_{\mathcal{T}} f\big)(t) = t^\alpha - t\,\frac{1}{n^{\alpha-1}}, & \qquad (\varphi(0) = \varphi(1/n) = 0),\\
\varphi'(t) &= \alpha t^{\alpha-1} - \frac{1}{n^{\alpha-1}},\\
\varphi'(t^*) &= 0 \quad \text{if} \quad t^* = \frac{1}{n}\alpha^{1/(1-\alpha)} \leq \frac{1}{2n},\\
\max_{t\in(0,1/n)} |\varphi(t)| &= |\varphi(t^*)| = \left|\frac{\alpha^{\alpha/(1-\alpha)}}{n^\alpha} - \frac{\alpha^{1/(1-\alpha)}}{n^\alpha}\right| = \frac{1}{n^\alpha}\left|\alpha^{\alpha/(1-\alpha)} - \alpha^{1/(1-\alpha)}\right| = \mathcal{O}(n^{-\alpha}) = \mathcal{O}(h^\alpha).
\end{aligned}$$

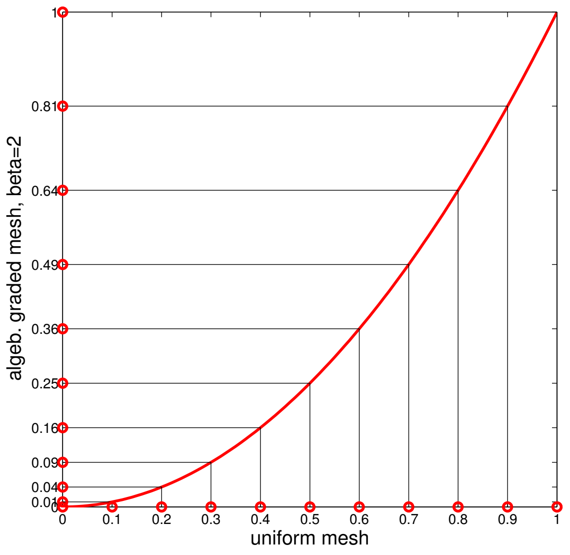
The order of convergence in $h = 1/n$ is equal to the parameter α, as observed in Figure 17.

(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 18 for $\beta = 2$.

Figure 18: Graded mesh $x_j = (j/n)^2$, $j = 0, \dots, 10$.



For a fixed parameter α in the definition of f, numerically determine the rate of convergence of the piecewise linear interpolant $\mathsf{I}_{\mathcal{G}}$ 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)?

Solution: The code in file `PWlineGraded.m` studies the dependence of the convergence rates on β and α. The result for $\alpha = 0.5$ is plotted in Figure 19.

The comparison of this plot with the analogous ones for different values of α suggests that the choice of $\beta = 2/\alpha$ guarantees quadratic convergence, run the code to observe it.

Proceeding as in (2e), we can see that the maximal error in the first subinterval $(0, t_1) = (0, 1/n^\beta)$ is equal to $1/n^{\alpha\beta}\,(\alpha^{\alpha/(1-\alpha)} - \alpha^{1/(1-\alpha)}) = \mathcal{O}(n^{-\alpha\beta})$. This implies that a necessary condition to have quadratic convergence is $\beta \geq 2/\alpha$. In order to prove un upper bound on the optimal β, we should control the error committed in every subinterval, here the exact computation of $\varphi(t^*)$ becomes quite long and complicate.

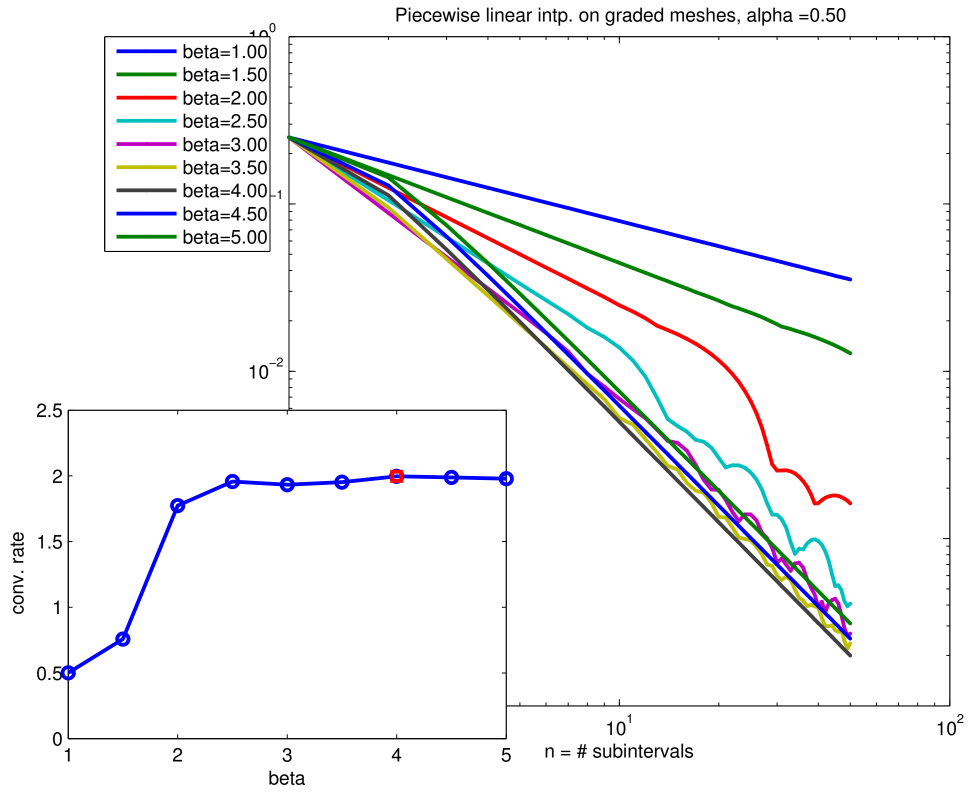
For larger values of the grading parameter, the error in last few subintervals begins to increase. The variable `LocErr` contains the index of the interval where the maximal error is attained (take a look at its values). It confirms that the largest error appears in the first subinterval if $\alpha\beta \ll 2$ and in the last one if $\alpha\beta \gg 2$, the intermediate cases are not completely clear.

Figure 18 has been created with the code in Listing 42.

Listing 42: Plot of algebraically graded mesh.

```
n = 10;                    b = 2;
t_eq = (0:n)/n;            t_gr =t_eq.^b;
close all;
   plot([0:0.01:1],(0:0.01:1).^b,'r','linewidth',2); hold
   on;
for j=1:n+1;    plot([t_eq(j),t_eq(j),0],
   [0,t_gr(j),t_gr(j)],'k');
     plot([t_eq(j),0], [0,t_gr(j)],'ro','linewidth',2);
        end;
axis square;  xlabel('uniform mesh','fontsize',14);
ylabel('algeb. graded mesh, beta=2','fontsize',14);
set(gca,'xtick',t_eq,'ytick',t_gr);print -depsc2
   'GradedMesh.eps';
```

Figure 19: h-convergence of piecewise linear interpolation for $f(t) = t^\alpha$, $\alpha = 0.5$, on algebraically graded meshes with parameters $\beta \in [1,5]$. The convergence rates in dependence on β are shown in the small plot.



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

$$\xi_j^{(n)} = \cos\left(\frac{2j+1}{2n}\pi\right), \quad j = 0, \dots, n-1. \tag{49}$$

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

$$(f,g)_n := \sum_{j=0}^{n-1} f(\xi_j^{(n)}) g(\xi_j^{(n)}), \quad f, g \in C^0([-1,1]) \tag{50}$$

and the special weighted L^2 inner product

$$(f,g)_w := \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} f(t)g(t)\,dt \quad f,g \in C^0([-1,1]) \tag{51}$$

(3a) Show that the Chebyshev polynomials are an orthogonal family of polynomials with respect to the inner product defined in (51) according to [1, Def. 4.2.24], namely $(T_k, T_l)_w = 0$ for every $k \neq l$.

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

Solution: For $k,l = 0,\dots,n$ with $k \neq l$, by using the substitution $s = \arccos t$ ($ds = -\frac{1}{\sqrt{1-t^2}}\,dt$) and simple trigonometric identities we readily compute

$$\begin{aligned}
(T_k, T_l)_w &= \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} \cos(k \arccos t)\cos(l \arccos t)\,dt \\
&= \int_0^{\pi} \cos(ks)\cos(ls)\,ds \\
&= \frac{1}{2}\int_0^{\pi} \cos((k+l)s) + \cos((k-l)s)\,ds \\
&= \frac{1}{2}([\sin((k+l)s)/(k+l)]_0^{\pi} + [\sin((k-l)s)/(k-l)]_0^{\pi}) \\
&= 0,
\end{aligned}$$

since $k+l \neq 0$ and $k-l \neq 0$.

Consider the following statement.

Theorem. The family of polynomials $\{T_0,\dots,T_n\}$ is an orthogonal basis ($\rightarrow$ [1, Def. 4.2.13]) of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$ defined in (50).

(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].

Solution: As a consequence of [1, Thm. 4.1.68], we already know that $\{T_0,\dots,T_n\}$ is a basis for $\mathcal{P}_n$. We check the orthogonality in the C++ code given in Listing 43.

Listing 43: Orthogonality of Chebyshev polynomials

```
#include <iostream>
```

```
#include <math.h>
#include <vector>
#include <Eigen/Dense>

using namespace std;

//Evaluate the Chebyshev polynomials up to order n in x.
vector<double> chebpolmult(const int &n,const double &x)
{
    vector<double> V={1,x};
    for (int k=1; k<n; k++)
        V.push_back(2*x*V[k]-V[k-1]);
    return V;
}

//Check orthogonality of Chebyshev polynomials.
int main(){
    int n=10;
    vector<double> V;
    Eigen::MatrixXd scal(n+1,n+1);
    for (int j=0; j<n+1; j++) {
        V=chebpolmult(n,cos(M_PI*(2*j+1)/2/(n+1)));
        for (int k=0; k<n+1; k++) scal(j,k)=V[k];
    }
    cout<<"Scalar products: "<<endl;
    for (int k=0; k<n+1; k++)
        for (int l=k+1; l<n+1; l++)
            cout<<scal.col(k).dot(scal.col(l))<<endl;
}
```

(3c) ⚃ Prove the theorem.

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

Solution: It remains to check the orthogonality condition, namely that $(T_k, T_l)_n = 0$ for

$k \neq l$. For $k, l = 0, \dots, n+1$ with $k \neq l$ by (70) we have

$$\begin{aligned}(T_k, T_l)_{n+1} &= \sum_{j=0}^{n} T_k(\xi_j^{(n)}) T_l(\xi_j^{(n)}) \\ &= \sum_{j=0}^{n} \cos\left(k \frac{2j+1}{2(n+1)}\pi\right) \cos\left(l \frac{2j+1}{2(n+1)}\pi\right) \\ &= \frac{1}{2}\sum_{j=0}^{n} (\cos\left((k+l)\frac{2j+1}{2(n+1)}\pi\right) + \cos\left((k-l)\frac{2j+1}{2(n+1)}\pi\right)).\end{aligned} \tag{52}$$

It is now enough to show that

$$\sum_{j=0}^{n} \cos\left(m \frac{2j+1}{2(n+1)}\pi\right) = 0, \qquad m \in \mathbb{Z}^*. \tag{53}$$

In order to verify this, observe that

$$\sum_{j=0}^{n} \cos\left(m \frac{2j+1}{2(n+1)}\pi\right) = \mathrm{Re}(\sum_{j=0}^{n} e^{im\frac{2j+1}{2(n+1)}\pi}) = \mathrm{Re}(e^{im\frac{1}{2(n+1)}\pi} \sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi}).$$

Finally, by the standard formula for the geometric sum we have

$$\begin{aligned} e^{im\frac{1}{2(n+1)}\pi} \sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi} &= e^{im\frac{1}{2(n+1)}\pi} \frac{1 - e^{im\frac{n+1}{n+1}\pi}}{1 - e^{im\frac{1}{n+1}\pi}} \\ = \frac{1 - e^{im\pi}}{e^{-im\frac{1}{2(n+1)}\pi} - e^{im\frac{1}{2(n+1)}\pi}} &= -\frac{1 - e^{im\pi}}{2} \frac{1}{i \sin m \frac{1}{2(n+1)}\pi} \in i\mathbb{R}, \end{aligned}$$

implying $\mathrm{Re}(e^{im\frac{1}{2(n+1)}\pi} \sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi}) = 0$ as desired.

(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:

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

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

$$q_n = \sum_{j=0}^{n} \alpha_j T_j \tag{54}$$

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].

Solution: In view of the theorem, the family $\{T_0, \dots, T_n\}$ is an orthogonal basis of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$. By (52) and (53) we have

$$\lambda_k^2 := \|T_k\|_{n+1}^2 = (T_k, T_k)_{n+1} = \begin{cases} \frac{1}{2}\sum_{j=0}^{n}(\cos(0)+\cos(0)) = n+1 & \text{if } k=0, \\ \frac{1}{2}\sum_{j=0}^{n}\cos(0) = (n+1)/2 & \text{otherwise.} \end{cases} \tag{55}$$

The family $\{T_k/\lambda_k : k = 0, \dots, n\}$ is an ONB of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$. Hence, by [1, Cor. 4.2.14] we have that

$$q_n = \sum_{j=0}^{n} (f, T_j/\lambda_j)_{n+1} \frac{T_j}{\lambda_j} = \sum_{j=0}^{n} \alpha_j T_j, \qquad \alpha_j = \frac{1}{n+1}\begin{cases} (f, T_j)_{n+1} & \text{if } k=0, \\ 2(f, T_j)_{n+1} & \text{otherwise.} \end{cases}$$

(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 (54) 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);};
```

Solution: See file `ChebBest.cpp`.

(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.

Solution: See file `ChebBest.cpp`. The output (plotted with Matlab) is shown in Figure 20.

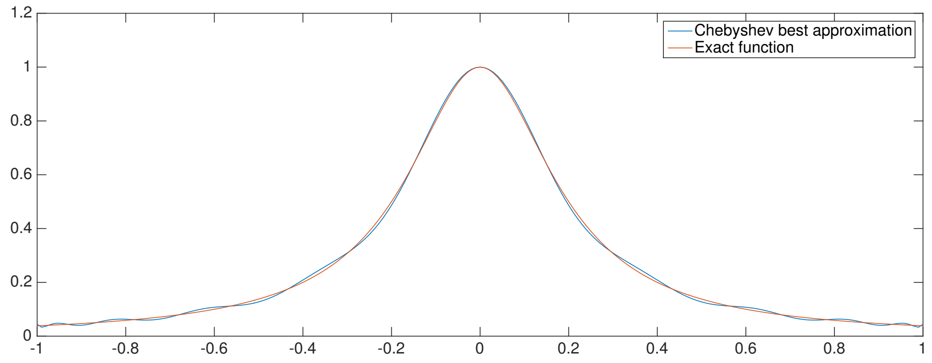


Figure 20: The result of code `ChebBest.m`.

(3g) ⚁ Let L_j, $j = 0, \dots, n$, 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.

Solution: We have already seen that $\{T_l/\lambda_l : l = 0, \dots, n\}$ is an ONB of $\mathcal{P}_n$. Thus we can write

$$L_j = \sum_{l=0}^{n} \frac{(L_j, T_l)_{n+1}}{\lambda_l^2} T_l = \sum_{l=0}^{n}\sum_{k=0}^{n} L_j(\xi_k^{(n+1)})T_l(\xi_k^{(n+1)})\frac{T_l}{\lambda_l^2}$$

By definition of Lagrange polynomials we have $L_j(\xi_k^{(n+1)}) = \delta_{jk}$, whence

$$L_j = \sum_{l=0}^{n} T_l(\xi_l^{(n+1)})\frac{T_l}{\lambda_l^2}.$$

Finally, the conclusion immediately follows from (55).

Problem 4 Piecewise cubic Hermite interpolation

Piecewise cubic Hermite interpolation with exact slopes on a mesh

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

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].

Solution: Let s be the piecewise cubic polynomial interpolant of f. We can rewrite s using the local representation with cardinal basis:

$$\begin{aligned} s(t) &= y_{i-1}H_1(t) + y_iH_2(t) + c_{i-1}H_3(t) + c_iH_4(t) \\ &= y_{i-1}H_1(t) + y_iH_2(t) + (f'(t_{i-1}) + \delta_{i-1})H_3(t) + (f'(t_i) + \delta_i)H_4(t) \\ &= y_{i-1}H_1(t) + y_iH_2(t) + f'(t_{i-1})H_3(t) + f'(t_i)H_4(t) + \delta_{i-1}H_3(t) + \delta_iH_4(t) \end{aligned}$$

Hence, if we denote by $\tilde{s}$ the Hermite interpolant with exact slopes:

$$\begin{aligned} \|f - s\|_{L^\infty([a,b])} \leq \|f - \tilde{s} + \tilde{s} - s\|_{L^\infty([a,b])} &\leq \|f - \tilde{s}\|_{L^\infty([a,b])} + \|\tilde{s} - s\|_{L^\infty([a,b])} \\ &\leq O(h^4) + \max_i \|\delta_{i-1}H_3(t) + \delta_iH_4(t)\|_{L^\infty([t_{i-1},t_i])} \\ &= O(h^4) + O(h^{b+1}) = O(\min(4, \beta + 1)) \end{aligned}$$

since $\|H_3(t)\|_{L^\infty([t_{i-1},t_i])} = \|H_4(t)\|_{L^\infty([t_{i-1},t_i])} = O(h)$ (attain maximum at $t = \frac{1}{3h}(t_i - t)$ resp. minimum at $t = \frac{2}{3h}(t_i - t)$, with value $h((\frac{2}{3})^3 - (\frac{2}{3})^2)$).

(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 $h = 2^{-l}, l = 0, \dots, 8$ 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).

Solution: According to the previous subproblem, since $s'(x_j) = f'(x_j) - f'(x_j)$, i.e. $|\delta_j| = O(1)$, $\beta = 0$, the convergence order is limited to $O(h)$.

For the C++ solution, cf. `piecewise_hermite_interpolation.cpp`.

(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

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

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 sub-subsection (4a). To find perturbation bounds, rely on the Taylor expansion formula with remainder, see [1, Ex. 1.5.60].

Solution: First, we show that the approximation $s'(x_j) = f'(x_j) + O(h^2)$. This follows from Taylor expansion:

$$f(x) = f(x_j) + f'(x_j)(x - x_j) + f''(x_j)(x - x_j)^2/2 + O(h^3)$$

Using $x = x_{j-1}$ and $x = x_{j+1}$ (and $h = x_{j+1} - x_j$):

$$\frac{f(x_{j+1}) - f(x_j)}{h} = f'(x_j) + f''(x_j)h/2 + O(h^2)$$
$$\frac{f(x_{j-1}) - f(x_j)}{h} = -f'(x_j) + f''(x_j)h/2 + O(h^2)$$

Subtracting the second equation to the first equation:

$$\frac{f(x_{j+1}) - f(x_{j-1})}{h} = 2f'(x_j) + O(h^2)$$

For the one-sided difference we expand at $x = x_{j+2}$ and $x = x_{j+1}$:

$$\begin{aligned}\frac{f(x_{j+1}) - f(x_j)}{h} &= f'(x_j) + f''(x_j)h/2 + O(h^2)\\ \frac{f(x_{j+2}) - f(x_j)}{2h} &= f'(x_j) + f''(x_j)h + O(h^2)\end{aligned}$$

Subtracting the first equation to half of the second equation:

$$\begin{aligned}\frac{f(x_{j+2}) - f(x_j)}{4h} - \frac{f(x_{j+1}) - f(x_j)}{h} &= f'(x_j)(1/2 - 1) + O(h^2)\\ \frac{f(x_{j+1}) - f(x_j) - 4f(x_{j+1}) + 4f(x_j)}{4h} &= f'(x_j)(1/2 - 1) + O(h^2)\\ \frac{-f(x_{j+2}) + 4f(x_{j+1}) - 3f(x_j)}{2h} &= f'(x_j) + O(h^2)\end{aligned}$$

The other side is analogous.

According to the previous subproblem, since $s'(x_j) = f'(x_j) + O(h^2)$ and $\beta = 2$, the convergence order is limited to $O(h^3)$.

For the C++ solution, cf. `piecewise_hermite_interpolation.cpp`.

Issue date: 12.11.2015

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

Version compiled on: July 16, 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, $j = 1, \dots, n$. 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

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

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.

Solution: By [1, Lemma 5.3.27], P_n has exactly n distinct zeros in $]-1,1[$. Therefore, it is enough to prove that every pair of zeros of P_n is separated by one zero of P_{n-1}. By contradiction, assume that there exists $l = 1, \dots, n-1$ such that $]\xi_l^n, \xi_{l+1}^n[$ does not contain any zeros of P_{n-1}. As a consequence, $P_{n-1}(\xi_l^n)$ and $P_{n-1}(\xi_{l+1}^n)$ have the same sign, namely

$$P_{n-1}(\xi_l^n)P_{n-1}(\xi_{l+1}^n) \geq 0. \tag{56}$$

Recall that, by construction, we have the following orthogonality property

$$\int_{-1}^{1} P_{n-1} q \, dt = 0, \qquad q \in \mathcal{P}_{n-2}. \tag{57}$$

Consider the auxiliary polynomial

$$q(t) = \prod_{j \neq l, l+1} (t - \xi_j^n).$$

By construction, $q \in \mathcal{P}_{n-2}$, whence $P_{n-1}q \in \mathcal{P}_{2n-3}$. Thus, using (57) and the fact that Gaussian quadrature is exact on P_{2n-1} we obtain

$$0 = \int_{-1}^{1} P_{n-1} q \, dt = \sum_{j=1}^{n} w_j P_{n-1}(\xi_j^n) q(\xi_j^n) = w_l P_{n-1}(\xi_l^n) q(\xi_l^n) + w_{l+1} P_{n-1}(\xi_{l+1}^n) q(\xi_{l+1}^n).$$

Hence

$$[w_l P_{n-1}(\xi_l^n) q(\xi_l^n)] \, [w_{l+1} P_{n-1}(\xi_{l+1}^n) q(\xi_{l+1}^n)] \leq 0.$$

On the other hand, by [1, Lemma 5.3.29] we have $w_l, w_{l+1} > 0$. Moreover, by construction of q we have that $q(\xi_l^n) q(\xi_{l+1}^n) > 0$. Combining these properties with (56) we obtain that

$$P_{n-1}(\xi_l^n) = P_{n-1}(\xi_{l+1}^n) = 0.$$

Choose now $s \in \mathcal{P}_{n-2}$ such that $s(\xi_j^n) = P_{n-1}(\xi_j^n)$ for every $j \neq l, l+1$. Using again (57) and the fact that Gaussian quadrature is exact on P_{2n-1} we obtain

$$0 = \int_{-1}^{1} P_{n-1} s \, dt = \sum_{j=1}^{n} w_j P_{n-1}(\xi_j^n) s(\xi_j^n) = \sum_{j \neq l, l+1} w_j (P_{n-1}(\xi_j^n))^2.$$

Since P_{n-1} has only $n-1$ zeros and the weights w_j are all positive, the right hand side of this equality is strictly positive, thereby contradicting the equality itself.

Solution 2: There is a shorter proof of this fact based on the recursion formula [1, Eq. (5.3.32)]. We sketch the main steps.

We will prove the statement by induction. If $n = 1$ there is nothing to prove. Suppose now that the statement is true for n. By [1, Eq. (5.3.32)] we have $P_{n+1}(\xi_j^n) = -\frac{n}{n+1} P_{n-1}(\xi_j^n)$ for every $j = 1, \dots, n$. Further, since the statement is true for n we have $(-1)^{n-j} P_{n-1}(\xi_j^n) > 0$. Therefore

$$(-1)^{n+1-j} P_{n+1}(\xi_j^n) > 0, \qquad j = 1, \dots, n. \tag{58}$$

Since the leading coefficient of P_{n+1} is positive, we have $P_{n+1}(x) > 0$ for all $x > \xi_{n+1}^{n+1}$ and $(-1)^{n+1} P_{n+1}(x) > 0$ for all $x < \xi_1^{n+1}$. Combining these two inequalities with (58) yields the result for $n + 1$.

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

Solution: Differentiating [1, Eq. (5.3.32)] immediately gives

$$P_{n+1}'(t) = \frac{2n+1}{n+1} P_n(t) + \frac{2n+1}{n+1} t P_n'(t) - \frac{n}{n+1} P_{n-1}'(t), \quad P_0' = 0, \quad P_1' = 1,$$

which combined with [1, Eq. (5.3.32)] gives the desired recursions.

(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 $(P_k'(x_j))_{jk}$, $k = 0, \dots, n$, $j = 0, \dots, N-1$, for an input vector $x \in \mathbb{R}^N$ (passed in `x`).

Solution: See file `legendre.cpp`.

(1d) ⚂ We can compute the zeros of P_k, $k = 1, \dots, n$, 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 $\xi_j^k \in [-1, 1]$, $j = 1, \dots, k$, $k = 1, \dots, n$, 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`.

Solution: See file `legendre.cpp`.

(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 21.

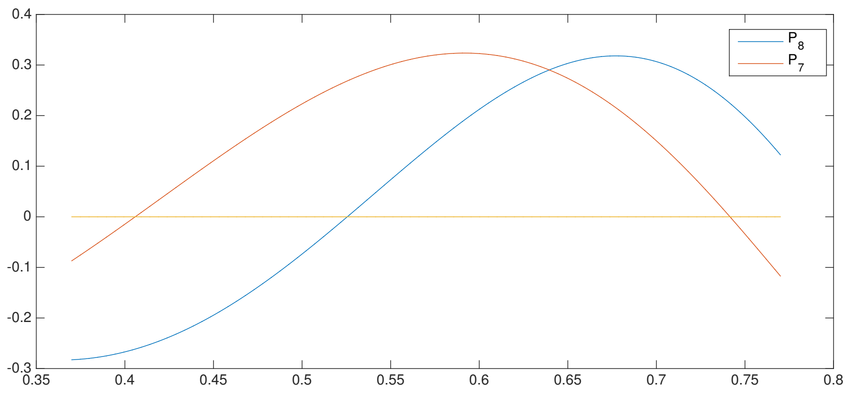


Figure 21: P_7 and P_8 on a part of $[-1,1]$. The secant method fails to find the zeros of P_8 (blue curve) when started with the zeros of P_7 (red curve).

Solution: See file `legendre.cpp` for the implementation. The zeros of P_k, $k = 1,\dots,7$ are all correctly computed, but the zero $\xi_6^8 \approx 0.525532$ of P_8 is not correct. This is due to the fact that in this case the initial guesses for the secant method are not close enough to the zero (remember, only local convergence is guaranteed). Indeed, taking $x^{(0)}$ and $x^{(1)}$ as the two consecutive zeros of P_7 in Figure 21, the third iterate $x^{(2)}$ will satisfy $P_8(x^{(1)})P_8(x^{(2)}) > 0$ and the forth iterate will be very far from the desired zero.

(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, 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
```

Solution: See file `legendre.cpp`.

Problem 2 Gaussian quadrature (core problem)

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

$$I := \int_{-1}^{1} \arcsin(t)\, f(t)\, \mathrm{d}t. \tag{59}$$

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 (59). Again `f_hd` is a function handle to f.

Solution: See Listing 44 and Figure 22:

Listing 44: implementation for the function `GaussConv`

```
function GaussConv(f_hd)
if nargin<1;    f_hd = @(t) sinh(t);    end;
I_exact = quad(@(t) asin(t).*f_hd(t),-1,1,eps)

n_max = 50;  nn = 1:n_max;
err = zeros(size(nn));
for j = 1:n_max
    [x,w] = gaussquad(nn(j));
    I = dot(w, asin(x).*f_hd(x));
    % I = GaussArcSin(f_hd,nn(j));   % using pcode
    err(j) = abs(I - I_exact);
end

close all; figure;
loglog(nn,err,[1,n_max],[1,n_max].^(-3),'--','linewidth',2);
title('{\bf Convergence of Gauss
   quadrature}','fontsize',14);
xlabel('{\bf n = # of evaluation points}','fontsize',14);
ylabel('{\bf error}','fontsize',14);
legend('Gauss quad.','O(n^{-3})');
print -depsc2 '../PICTURES/GaussConv.eps';
```

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

Solution: Algebraic convergence, approximately $O(n^{-2.7})$.

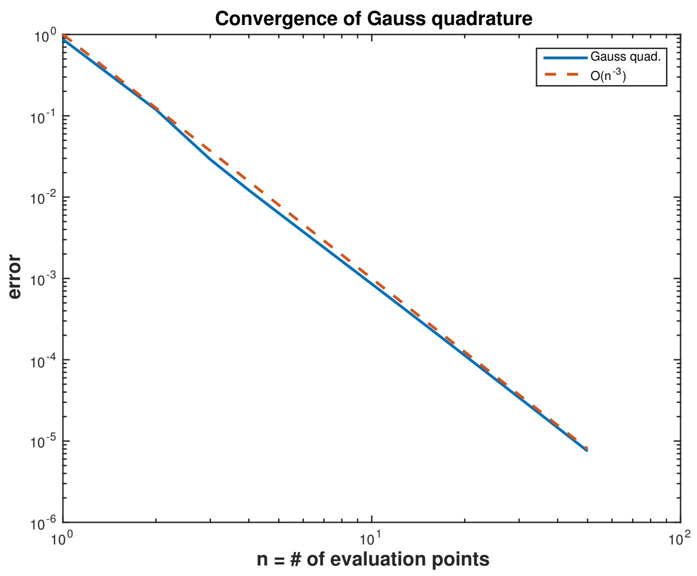


Figure 22: Convergence of `GaussConv.m`.

(2c) ⚂ Transform the integral (59) into an equivalent one with a suitable change of variable so that Gauss quadrature applied to the transformed integral converges much faster.

Solution: With the change of variable $t = \sin(x)$, $\mathrm{d}t = \cos x \mathrm{d}x$

$$I = \int_{-1}^{1} \arcsin(t)\ f(t)\,\mathrm{d}t = \int_{-\pi/2}^{\pi/2} x\ f(\sin(x))\cos(x)\,\mathrm{d}x.$$

(the change of variable has to provide a smooth integrand on the integration interval)

(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.

Solution: See Listing 45 and Figure 23:

Listing 45: implementation for the function `GaussConvCV`

```
function GaussConvCV(f_hd)
if nargin<1;    f_hd = @(t) sinh(t);    end;
g = @(t) t.*f_hd(sin(t)).*cos(t);
I_exact = quad(@(t) asin(t).*f_hd(t),-1,1,eps)
%I_exact = quad(@(t) g(t),-pi/2,pi/2,eps)

n_max = 50;  nn = 1:n_max;
err = zeros(size(nn));
for j = 1:n_max
    [x,w] = gaussquad(nn(j));
    I = pi/2*dot(w, g(x*pi/2));
    % I = GaussArcSinCV(f_hd,nn(j));   % using pcode
    err(j) = abs(I - I_exact);
end

close all; figure;
semilogy(nn,err,[1,n_max],[eps,eps],'--','linewidth',2);
title('{\bf Convergence of Gauss quadrature for
   rephrased problem}','fontsize',14);
xlabel('{\bf n = # of evaluation points}','fontsize',14);
ylabel('{\bf error}','fontsize',14);
legend('Gauss quad.','eps');
print -depsc2 '../PICTURES/GaussConvCV.eps';
```

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

Solution: The convergence is now exponential. The integrand of the original integral belongs to $C^0([-1,1])$ but not to $C^1([-1,1])$ because the derivative of the $\arcsin$ function blows up in ± 1. The change of variable provides an analytic integrand: $x\cos(x)\sinh(\sin x)$. Gauss quadrature ensures exponential convergence only if the integrand is analytic. This explains the algebraic and the exponential convergence.

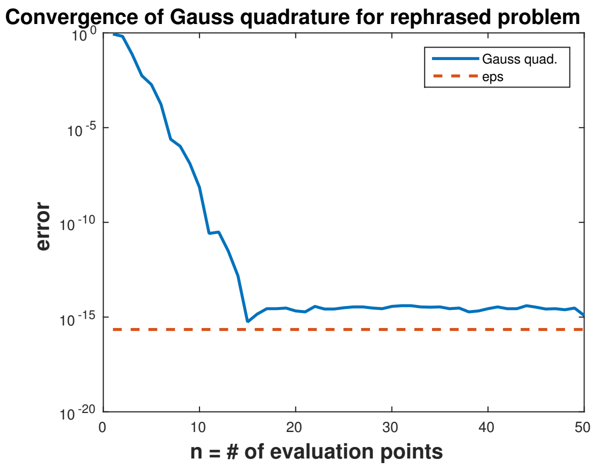


Figure 23: Convergence of `GaussConvCV.m`.

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) $\boxed{\cdot}$ For the integrand $g(t) := 1/(1+t^2)$ determine b such that the truncation error E_T satisfies:

$$E_T := \left| \int_{-\infty}^{\infty} g(t)dt - \int_{-b}^{b} g(t)dt \right| \leq 10^{-6} \tag{60}$$

Solution: An antiderivative of g is atan. The function g is even.

$$E_T = 2\int_{b}^{\infty} g(t)dt = \lim_{x\to\infty} 2\mathrm{atan}(x) - 2\mathrm{atan}(b) = \pi - 2\mathrm{atan}(b) \overset{!}{<} 10^{-6} \tag{61}$$

i.e. $b > \tan((\pi - 10^{-6})/2) = \cot(10^{-6}/2)$.

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

Solution: A good choice of b requires a detailed knowledge about the decay of f, which may not be available for f defined implicitly.

A second option (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 $\int_{-\infty}^{\infty} f(t)dt$?

Solution:

$$\frac{dt}{ds} = -(1+\cot^2(s)) = -(1+t^2) \tag{62}$$

$$\int_{-\infty}^{\infty} f(t)dt = -\int_{\pi}^{0} f(\cot(s))(1+\cot^2(s))ds = \int_0^{\pi} \frac{f(\cot(s))}{\sin^2(s)}ds, \tag{63}$$

because $\sin^2(\theta) = \frac{1}{1+\cot^2(\theta)}$.

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

Solution:

$$\int_0^{\pi} \frac{1}{1+\cot^2(s)} \frac{1}{\sin^2(s)} ds = \int_0^{\pi} \frac{1}{\sin^2(s)+\cos^2(s)} ds = \int_0^{\pi} ds = \pi \tag{64}$$

(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.

Solution: See `quadinf.cpp`.

(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:

$$\int_{-\infty}^{\infty} h(t)dt = \sqrt{\pi} \tag{65}$$

Solution: We observe exponential convergence. See `quadinf.cpp`.

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 24. Notice that only the quadrature errors obtained with an even number of function evaluations are shown.

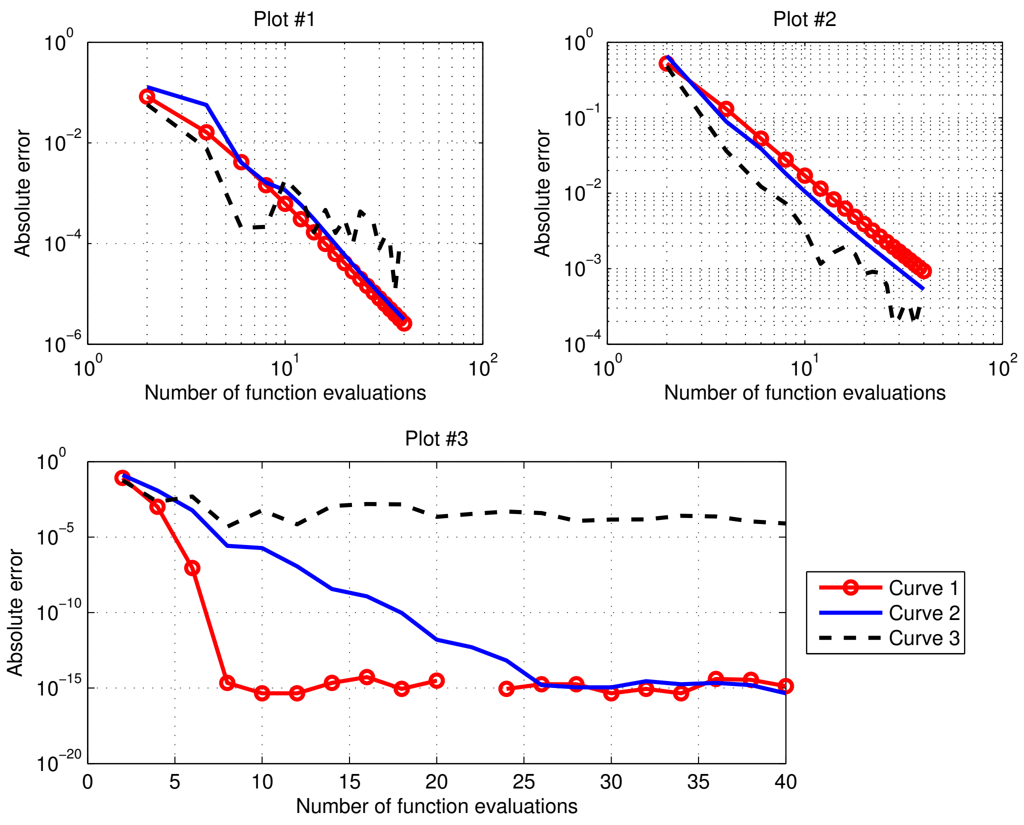


Figure 24: Quadrature convergence plots for different functions and different rules.

(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.

Solution: Plot #1 — Quadrature rule C, Composite 2-point Gauss:
algebraic convergence for every function, about 4^{th} order for two functions.

Plot #2 — Quadrature rule B, Composite trapezoidal:
algebraic convergence for every function, about 2^{nd} order.

Plot #3 — Quadrature rule A, Global Gauss:
algebraic convergence for one function, exponential for another one, exact integration with 8 evaluations for the third one.

(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.

Solution: Curve 1 red line and small circles — f_C polynomial of degree 12:
integrated exactly with 8 evaluations with global Gauss quadrature.

Curve 2 blue continuous line only — f_A analytic function:
exponential convergence with global Gauss quadrature.

Curve 3 black dashed line — f_B non smooth function:
algebraic convergence with global Gauss quadrature.

Issue date: 19.11.2015
Hand-in: 26.11.2015 (in the boxes in front of HG G 53/54).

Version compiled on: July 16, 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:

$$W(f) := \int_{-1}^{1} \sqrt{1-t^2}\, f(t)dt, \tag{66}$$

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`.

Solution: See documentation in `gauleg.hpp` and `gauleg.cpp`.

(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 (66) 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`.

Solution: Exploiting the hint, we see that the integrand is non-smooth in ± 1.

The first possible solution is the following (I): we split the integration domain $[-1,1]$ in $[0,1]$ and $[-1,0]$. Applying the substitution $s = \sqrt{1 \pm t}$ (sign depending on which part of the integrals considered), $t = \pm(s^2 - 1)$:

$$\frac{dt}{ds} = \pm 2s$$

$$\begin{aligned} W(f) := \int_{-1}^{1} \sqrt{1-t^2} f(t) dt &= \int_{-1}^{0} \sqrt{1-t^2} f(t) dt + \int_{0}^{1} \sqrt{1-t^2} f(t) dt \\ &= \int_{0}^{1} 2 \cdot s^2 \sqrt{2-s^2} f(-s^2+1) ds + \int_{0}^{1} 2 \cdot s^2 \sqrt{2-s^2} f(s^2-1) ds. \end{aligned}$$

Notice how the resulting integrand is analytic in a neighbourhood of the domain of integration because, for instant, $t \mapsto \sqrt{1+t}$ is analytic in a neighborhood of $[0,1]$.

Alternatively (II), one may use the trigonometric substitution $t = \sin s$, with $\frac{dt}{ds} = \cos s$

obtaining

$$\begin{aligned} W(f) &:= \int_{-1}^{1} \sqrt{1-t^2} f(t) dt \\ &= \int_{-\pi/2}^{\pi/2} \sqrt{1-\sin^2 s}\, f(\sin s) \cos s\, ds = \int_{-\pi/2}^{\pi/2} \cos^2 s\, f(\sin s) ds. \end{aligned}$$

This integrand is also analytic. The C++ implementation is in `quadsingint.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).

Solution: Using substitution (I). Let (x_j, w_j), $j = 1, \dots, n$ be the Gauss nodes and weights relative to the Gauss quadrature of order n in the interval $[0,1]$. The nodes are mapped from x_j in $[0,1]$ to c_l for $l \in 1, \dots, 2n$ in $[-1,1]$ as follows:

$$c_{2j-i} = (-1)^i (1 - x_j^2), \qquad j = 1, \dots, n, \ i = 0, 1.$$

The weights $\tilde{w}_l, l = 1, \dots, 2n$, become:

$$\tilde{w}_{2j-i} = 2 w_j x_j^2 \sqrt{2 - x_j^2}, \qquad j = 1, \dots, n, \ i = 0, 1.$$

Using substitution (II). Let (x_j, w_j), $j = 1, \dots, n$ be the Gauss nodes and weights relative to the Gauss quadrature of order n in the interval $[-1,1]$. The nodes are mapped from x_j to c_j as follows:

$$c_j = \sin(x_j \pi/2), \qquad j = 1, \dots, n$$

The weights $\tilde{w}_j, j = 1, \dots, n$, become:

$$\tilde{w}_j = w_j \cos^2(x_j \pi/2) \pi/2.$$

(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].

Solution: The convergence is exponential with both methods. The C++ implementation is in `quadsingint.cpp`.

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.

Solution: The radiant power absorbed by $\triangle$ can be written as:

$$\int_{\triangle} I(x,y)dxdy = \int_0^1 \int_0^{1-y} I(x,y)dxdy.$$

(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`.

Solution: See `laserquad.cpp` and `CMakeLists.txt`.

(2c) ⚃ Write a C++ function

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

for the computation of the integral

$$\int_{\triangle} f(x,y)dxdy, \tag{67}$$

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

HINT: Write (67) 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.

Solution: The integral can be written as

$$\int_{\triangle} f(x,y)dxdy = \int_0^1 \int_0^{1-y} f(x,y)dxdy.$$

In the C++ implementation, we define the auxiliary (lambda) function f_y:

$$\forall y \in [0,1], f_y : [1, 1-y] \to \mathbb{R}, x \mapsto f_y(x) := f(x,y)$$

We also define the (lambda) approximated integrand:

$$g(y) := \int_0^{1-y} f_y(x)dx \approx \frac{1}{1-y}\sum_{i=0}^{N} w_i f_y\left(\frac{x_i+1}{2}(1-y)\right) =: \mathcal{I}(y), \tag{68}$$

the integral of which can be approximated, using a nested Gauss quadrature:

$$\int_{\triangle} f(x,y)dxdy = \int_0^1 \int_0^{1-y} f_y(x)dxdx = \int_0^1 g(y)dy \approx \frac{1}{2}\sum_{j=1}^{N} w_j \mathcal{I}\left(\frac{y_j+1}{2}\right). \tag{69}$$

The implementation can be found in `laserquad.cpp`.

(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.

Solution: As one expects from theoretical considerations, the convergence is exponential. The implementation can be found in `laserquad.cpp`.

Problem 3 Weighted Gauss quadrature

The development of an alternative quadrature formula for (66) 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 (66), 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 $n \geq 1$.

Solution: The case $n = 0$ is trivial, since $U_0(t) = \frac{\sin(\arccos t)}{\sin(\arccos t)} = 1$, as desired. Using the trigonometric identity $\sin 2x = \sin x \cos x$, we have $U_1(t) = \frac{2\sin(\arccos t)}{\sin(\arccos t)} = 2\cos\arccos t = 2t$, as desired. Finally, using the identity $\sin(x+y) = \sin x \cos y + \sin y \cos x$, we obtain for $n \geq 2$

$$\begin{aligned} U_{n+1}(t) &= \frac{\sin((n+1)\arccos t)t + \cos((n+1)\arccos t)\sin(\arccos t)}{\sin(\arccos t)} \\ &= U_n(t)t + \cos((n+1)\arccos t). \end{aligned}$$

Similarly, we have

$$\begin{aligned} U_{n-1}(t) &= \frac{\sin((n+1-1)\arccos t)}{\sin(\arccos t)} \\ &= \frac{\sin((n+1)\arccos t)t - \cos((n+1)\arccos t)\sin(\arccos t)}{\sin(\arccos t)} \\ &= U_n(t)t - \cos((n+1)\arccos t). \end{aligned}$$

Combining the last two equalities we obtain the desired 3-term recursion.

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

Solution: Let us prove the claim by induction. The case $n = 0$ is trivial, since $U_0(t) = 1$. Let us now assume that the statement is true for every $k = 0, \dots, n$ and let us prove it for $n + 1$. In view of $U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t)$, since by inductive hypothesis $U_n \in \mathcal{P}_n$ and $U_{n-1} \in \mathcal{P}_{n-1}$, we have that $U_{n+1} \in \mathcal{P}_{n+1}$. Moreover, the leading coefficient will be 2 times the leading order coefficient of U_n, namely 2^{n+1}, as desired.

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

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

Solution: With the substitution $t = \cos s$ we obtain

$$\begin{aligned}
\int_{-1}^{1} \sqrt{1-t^2} U_m(t)U_n(t)\, dt &= \int_{-1}^{1} \sqrt{1-t^2} \frac{\sin((m+1)\arccos t)\sin((n+1)\arccos t)}{\sin^2(\arccos t)}\, dt \\
&= \int_0^{\pi} \sin s \frac{\sin((m+1)s)\sin((n+1)s)}{\sin^2 s} \sin s\, ds \\
&= \int_0^{\pi} \sin((m+1)s)\sin((n+1)s)\, ds \\
&= \frac{1}{2}\int_0^{\pi} \cos((m-n)s) - \cos((m+n+2)s)\, ds.
\end{aligned}$$

The claim immediately follows, as it was done in Problem Sheet 9, Problem 3.

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

Solution: From the definition of U_n we immediately find that the zeros are given by

$$\xi_j^n = \cos\left(\frac{j}{n+1}\pi\right), \qquad j = 1, \dots, n. \tag{70}$$

(3e) ⚃ Show that the choice of weights

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

ensures that the quadrature formula

$$Q_n^U(f) = \sum_{j=1}^{n} w_j f(\xi_j^n) \tag{71}$$

provides the exact value of (66) for $f \in \mathcal{P}_{n-1}$ (assuming exact arithmetic).

HINT: Use all the previous subproblems.

Solution: Since U_k is a polynomial of degree exactly k, the set $\{U_k : k = 0, \ldots, n-1\}$ is a basis of $\mathcal{P}_{n-1}$. Therefore, by linearity it suffices to prove the above identity for $f = U_k$ for every k. Fix $k = 0, \ldots, n-1$. Setting $x = \pi/(n+1)$, from (70) we readily derive

$$\begin{aligned}
\sum_{j=1}^{n} w_j U_k(\xi_j^n) &= \sum_{j=1}^{n} \frac{\pi}{n+1} \sin^2\left(\frac{j}{n+1}\pi\right) \frac{\sin((k+1)\arccos \xi_j^n)}{\sin(\arccos \xi_j^n)} \\
&= x \sum_{j=1}^{n} \sin(jx)\sin((k+1)jx) \\
&= \frac{x}{2} \sum_{j=1}^{n} (\cos((k+1-1)jx) - \cos((k+1+1)jx)) \\
&= \frac{x}{2} \operatorname{Re} \sum_{j=0}^{n} \left(e^{ikxj} - e^{i(k+2)xj}\right) \\
&= \frac{x}{2} \operatorname{Re} \left(\sum_{j=0}^{n} e^{ikxj} - \frac{1 - e^{i\pi(k+2)}}{1 - e^{i(k+2)x}}\right).
\end{aligned}$$

Thus, for $k = 0$ we have

$$\sum_{j=1}^{n} w_j U_0(\xi_j^n) = \frac{x}{2} \operatorname{Re} \left(\sum_{j=0}^{n} 1 - \frac{1 - e^{2\pi i}}{1 - e^{2xi}}\right) = \frac{x}{2} \operatorname{Re}\left((n+1) - 0\right) = \frac{\pi}{2}.$$

On the other hand, if $k = 1, \ldots, n-1$ we obtain

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{x}{2} \operatorname{Re} \left(\frac{1 - e^{i\pi k}}{1 - e^{ikx}} - \frac{1 - e^{i\pi(k+2)}}{1 - e^{i(k+2)x}}\right) = \frac{(1 - (-1)^k)x}{2} \operatorname{Re} \left(\frac{1}{1 - e^{ikx}} - \frac{1}{1 - e^{i(k+2)x}}\right).$$

In view of the elementary equality $(a + ib)(a - ib) = a^2 + b^2$ we have $\operatorname{Re}(1/(a + ib)) = a/(a^2 + b^2)$. Thus

$$\operatorname{Re}\left(\frac{1}{1 - e^{ikx}}\right) = \operatorname{Re}\left(\frac{1}{1 - \cos(kx) - i\sin(kx)}\right) = \frac{1 - \cos(kx)}{(1 - \cos(kx))^2 + \sin(kx)^2} = \frac{1}{2}.$$

Arguing in a similar way we have $\operatorname{Re}\,(1 - e^{i(k+2)x})^{-1} = 1/2$. Therefore for $k = 1, \ldots, n-1$ we have

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{(1 - (-1)^k)x}{2} \left(\frac{1}{2} - \frac{1}{2}\right) = 0.$$

To summarise, we have proved that

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{\pi}{2}\delta_{k0}, \qquad k = 0, \dots, n-1.$$

Finally, the claim follows from (3c), since $U_0(t) = 1$ and so the integral in (66) is nothing else than the weighted scalar product between U_k and U_0.

(3f) ⚁ Show that the quadrature formula (71) gives the exact value of (66) even for every $f \in \mathcal{P}_{2n-1}$.

HINT: See [1, Thm. 5.3.21].

Solution: The conclusion follows by applying the same argument given in [1, Thm. 5.3.21] with the weighted L^2 scalar product with weight w defined above.

(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].

Solution: By definition, the weights defined above are positive, and the quadrature rule is exact for polynomials up to order $2n-1$. Therefore, arguing as in [1, § 5.3.37], we obtain the exponential decay, as desired.

(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));};
```

Solution: See file `quadU.cpp`.

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

Solution: See file `quadU.cpp`. An approximation of q is given by $E_n(f)/E_{n-1}(f)$.

Problem 4 Generalize "Hermite-type" quadrature formula

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

$$\int_0^1 f(x)dx \approx Af(0) + Bf'(0) + Cf(x_1) \tag{72}$$

is exact for polynomials of highest possible degree.

Solution: The quadrature is exact for every polynomial $p(x) \in \mathcal{P}^n$, if and only if it is exact for $1, x, x^2, \dots, x^n$. If we apply the quadrature to the first monomials:

$$1 = \int_0^1 1dx = A\cdot 1 + B\cdot 0 + C\cdot 1 = A + C \tag{73}$$

$$\frac{1}{2} = \int_0^1 xdx = A\cdot 0 + B\cdot 1 + C\cdot x_1 = B + Cx_1 \tag{74}$$

$$\frac{1}{3} = \int_0^1 x^2dx = A\cdot 0 + B\cdot 0 + C\cdot x_1^2 = Cx_1^2 \tag{75}$$

$$\frac{1}{4} = \int_0^1 x^3dx = A\cdot 0 + B\cdot 0 + C\cdot x_1^3 = Cx_1^3 \tag{76}$$

$\Rightarrow B = \frac{1}{2} - Cx_1, C = \frac{1}{3x_1^2} \Rightarrow \frac{1}{4} = \frac{1}{3x_1^2}x_1^3 = \frac{1}{3}x_1, A = \frac{11}{27}$, i.e.

$$x_1 = \frac{3}{4}, C = \frac{16}{27}, B = \frac{1}{18}, A = \frac{11}{27}. \tag{77}$$

Then

$$\frac{1}{5} = \int_0^1 x^4dx \neq A\cdot 0 + B\cdot 0 + C\cdot x_1^4 = C\cdot x_1^4 = \frac{16}{27}\frac{81}{256}. \tag{78}$$

Hence, the quadrature is exact for polynomials up to degree 3.

(4b) ⚃

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

$$z'(t) = \frac{t}{1+t^2} \quad , \quad z(1) = 1\ . \tag{79}$$

Solution: We know that

$$z(2) - z(1) = \int_1^2 z'(x)dx, \tag{80}$$

hence, applying (72) and the transformation $x \mapsto x + 1$, we obtain:

$$z(2) = \int_0^1 z'(x+1)dx + z(1) \approx \frac{11}{27} \cdot z'(1) + \frac{1}{18} \cdot z''(1) + \frac{16}{27} \cdot z'\left(\frac{7}{4}\right) + z(1). \tag{81}$$

With $z''(x) = -\frac{2 \cdot x}{(1+x^2)^2}$ and:

$$\begin{aligned} z(1) &= 1, \\ z'(1) &= \frac{1}{1+1^2} = \frac{1}{2}, \\ z''(1) &= -\frac{2 \cdot 1}{(1+1^2)^2} = -\frac{1}{2}, \\ z'\left(\frac{7}{4}\right) &= \frac{(\frac{7}{4})}{1+(\frac{7}{4})^2} = \frac{28}{65}, \end{aligned}$$

we obtain

$$z(2) = \int_0^1 z'(x+1)dx + z(1) \approx \frac{11}{27} \cdot \frac{1}{2} - \frac{1}{18} \cdot \frac{1}{2} + \frac{16}{27} \cdot \frac{28}{65} + 1 = 1.43\ldots$$

For sake of completeness, using the antiderivative of z':

$$z(2) = \int_1^2 z'(x)dx + z(1) = \frac{1}{2} \log(x^2+1)|_1^2 + 1 = 1.45\ldots$$

Issue date: 26.11.2015

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

Version compiled on: July 16, 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>
  void step(const Function &f, double h,
            const State & y0, State & y1) const {
    // TODO: performs a single step from y0 to y1 with step size h of the RK method for the IVP with rhs f
  }

  // TODO: hold data for RK methods
};
```

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.

Solution: See `rkintegrator.hpp`.

(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]):

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t))y_1(t) \tag{82}$$

$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2)y_2(t) \tag{83}$$

$$\mathbf{y}(0) = [100, 5] \tag{84}$$

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]):

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{3} & \frac{1}{3} & 0 & \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \tag{85}$$

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

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.

Solution: See `rk3prey.cpp`.

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

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{86}$$

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 (86) up to time T by means of an explicit Runge-Kutta method with $N = 2^k$, $k = 1, \dots, 15$, 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 $E_N = \|\mathbf{y}_N(T) - \mathbf{y}_{2^{15}}(T)\|$, $N = 2^k$, $k = 1, \dots, 13$, 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 $\mathbf{y}_N(T) - \mathbf{y}_{2^{15}}(T)$).

Solution: Let us find an expression for the order of convergence. Set $N_k = 2^k$. We readily derive

$$\frac{E_{N_{k-1}}}{E_{N_k}} \approx \frac{C2^{-(k-1)r_k}}{C2^{-kr_k}} = 2^{r_k}, \qquad r_k \approx \log\left(\frac{E_{N_{k-1}}}{E_{N_k}}\right)/\log(2).$$

A reasonable approximation of the order of convergence is given by

$$r \approx \frac{1}{\#K}\sum_{k\in K} r_k, \qquad K = \{k = 1,\dots,15 : E_{N_k} > 5n\cdot 10^{-14}\}. \tag{87}$$

See file `errors.hpp` for the implementation.

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

$$\dot{y} = (1-y)y, \quad y(0) = 1/2, \tag{88}$$

and of the initial value problem

$$\dot{y} = |1.1 - y| + 1, \quad y(0) = 1. \tag{89}$$

Solution: As far as (88) is concerned, the solution is $y(t) = (1+e^{-t})^{-1}$ (see [1, Eq. (11.1.7)]).

Let us now consider (89). Because of the absolute value on the right hand side of the differential equation, we have to distinguish two cases $y(t) < 1.1$ and $y(t) > 1.1$. Since the initial condition is given by $y(0) = 1 < 1.1$, we start with the case $y(t) < 1.1$. For $y(t) < 1.1$, the differential equation is $\dot{y} = 2.1 - y$. Separation of variables

$$\int_1^{y(t)} \frac{1}{2.1-\tilde{y}}\,d\tilde{y} = \int_0^t d\tilde{t}$$

yields the solution

$$y(t) = 2.1 - 1.1e^{-t}, \quad \text{for } y(t) < 1.1.$$

For $y(t) > 1.1$, the differential equation is given by $\dot{y} = y - 0.1$ with initial condition $y(\ln(\frac{11}{10})) = 1.1$, where the initial time t^* was derived from the condition $y(t^*) = 2.1 - 1.1e^{-t^*} \overset{!}{=} 1.1$. Separation of variables yields the solution for this IVP

$$y(t) = \tfrac{10}{11}e^t + 0.1.$$

Together, the solution of the initial IVP is given by

$$y(t) = \begin{cases} 2.1 - 1.1e^{-t}, & \text{for } t \le \ln(1.1) \\ \tfrac{10}{11}e^t + 0.1, & \text{for } t > \ln(1.1). \end{cases}$$

(2c) ⚁ Use the function `errors` from (2a) with the ODEs (88) and (89) 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.

Solution: Using the expression for the order of convergence given in (87) we find for (88):

- Eul: 1.06
- RK2: 2.00
- RK3: 2.84
- RK4: 4.01

This corresponds to the expected orders. However, in the case of the ODE (89) we obtain

- Eul: 1.09
- RK2: 1.93
- RK3: 1.94
- RK4: 1.99

The convergence orders of the explicit Euler and Runge–Kutta 2 methods are as expected, but we do not see any relevant improvement in the convergence orders of RK3 and RK4. This is due to the fact that the right hand side of the IVP is not continuously differentiable: the convergence theory breaks down.

See file `order_not_all.cpp` for the implementation.

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

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \,, \qquad \mathbf{y}(0) = \mathbf{y}_0 \,, \tag{90}$$

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],

$$\mathbf{y}(t) = \sum_{n=0}^{m} \frac{\mathbf{y}^{(n)}(0)}{n!} t^n + R_m(t) \,, \tag{91}$$

with remainder term $R_m(t) = O(t^{m+1})$ for $t \to 0$.

A single step method for the numerical integration of (90) can be obtained by choosing $m = 3$ in (91), 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 (90).

Solution: For the first time derivative of $\mathbf{y}$, we just use the differential equation:

$$\frac{d\mathbf{y}}{dt}(t) = \mathbf{y}'(t) = \mathbf{f}(\mathbf{y}(t)).$$

For the second derivative, we use the previous equation and apply chain rule and then once again insert the ODE:

$$\frac{d^2\mathbf{y}}{dt^2}(t) = \frac{d}{dt}\mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}'(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)).$$

Here $\mathbf{Df}(\mathbf{y}(t))$ is the Jacobian of $\mathbf{f}$ evaluated at $\mathbf{y}(t)$.

(3b) ⚄ Verify the formula

$$\frac{d^3\mathbf{y}}{dt^3}(0) = \mathbf{D}^2\mathbf{f}(\mathbf{y}_0)\big(\mathbf{f}(\mathbf{y}_0), \mathbf{f}(\mathbf{y}_0)\big) + \mathbf{D}\mathbf{f}(\mathbf{y}_0)^2\mathbf{f}\big(\mathbf{y}_0\big)\,. \tag{92}$$

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 $\mathbf{D}\mathbf{f}(\mathbf{y}_0) : \mathbb{R}^n \to \mathbb{R}^n$. 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 $\mathbf{y}_0$.

Solution: We follow the hint and first have a look at the scalar case. Here, the Jacobian reduces to $f'(y(t))$. Thus, we have to calculate

$$\frac{d^3y}{dt^3}(t) = \frac{d}{dt}\big(f'(y(t))f(y(t))\big)\,.$$

Product rule and chain rule give

$$\frac{d}{dt}\big(f'(y(t))f(y(t))\big) = f''(y(t))y'(t)f(y(t)) + f'(y(t))f'(y(t))y'(t).$$

Inserting the ODE $y'(t) = f(y(t))$ once again yields

$$\frac{d^3y}{dt^3}(t) = f''(y(t))f(y(t))^2 + f'(y(t))^2f(y(t)).$$

This already resembles Formula 92. The first term is quadratic in $f(y(t))$ and involves the second derivative of f, whereas the second term involves the first derivative of f in quadratic form.

To understand the formula for higher dimensions, we verify it componentwise. For each component $y_i(t)$ we have a function $f_i : \mathbb{R}^n \to \mathbb{R}$.

For the first derivative of $y_i(t)$, this is straightforward:

$$y_i'(t) = f_i(\mathbf{y}(t)).$$

Thus, the second derivative is

$$\begin{aligned} y_i''(t) &= \frac{d}{dt}\big(f_i(\mathbf{y}(t))\big) = \sum_{j=1}^{n} \partial_{y_j} f_i(\mathbf{y}(t))\, y_j'(t) \\ &= \big(\mathbf{grad}\, f_i(\mathbf{y}(t))\big)^T \cdot \mathbf{y}'(t) = \big(\mathbf{grad}\, f_i(\mathbf{y}(t))\big)^T \cdot \mathbf{f}(\mathbf{y}(t)) \ . \end{aligned}$$

Building up all components gives us what we have already obtained in (3a):

$$\mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)),$$

with the Jacobian $\mathbf{Df}(\mathbf{y}(t))$, which contains the gradients of the components of f row-wise.

Now, we apply product rule to $y_i''(t)$ to obtain

$$y_i'''(t) = \left(\frac{d}{dt}\big(\mathbf{grad}\, f_i(\mathbf{y}(t))\big)^T\right) \cdot \mathbf{y}'(t) + \big(\mathbf{grad}\, f_i(\mathbf{y}(t))\big)^T \cdot \mathbf{y}''(t).$$

The second term of the sum again builds up to

$$\mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t))^2 \cdot \mathbf{f}(\mathbf{y}(t)) \ .$$

For the first term, we first write the scalar product of the two vectors as a sum and then interchange the order of derivatives. This is possible as long as functions are sufficiently differentiable.

$$\left(\frac{d}{dt}\big(\mathbf{grad}\, f_i(\mathbf{y}(t))\big)^T\right) \cdot \mathbf{y}'(t) = \sum_{j=1}^{n} \left(\partial_{y_j} \frac{d}{dt} f_i(\mathbf{y}(t))\right) f_j(\mathbf{y}(t)) \ .$$

Now we apply the chain rule:

$$\begin{aligned} \sum_{j=1}^{n} \left(\partial_{y_j} \frac{d}{dt} f_i(\mathbf{y}(t))\right) f_j(\mathbf{y}(t)) &= \sum_{j=1}^{n} \left(\partial_{y_j} \sum_{l=1}^{n} \partial_{y_l} f_i(\mathbf{y}(t)) y_l'(t)\right) f_j(\mathbf{y}(t)) \\ &= \sum_{j,l=1}^{n} \big(\partial_{y_j} \partial_{y_l} f_i(\mathbf{y}(t))\big) f_l(\mathbf{y}(t)) f_j(\mathbf{y}(t)) \\ &= \mathbf{f}(\mathbf{y}(t))^T \cdot \mathbf{H} f_i(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)). \end{aligned}$$

This is the desired result.

(3c) ⚃ We now apply the Taylor expansion method introduced above to the *predator-prey* model (97) 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 (97) 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`.

Solution: For our particular example, we have

$$\mathbf{y} = \begin{pmatrix} u \\ v \end{pmatrix}, \qquad \mathbf{f}(\mathbf{y}) = \begin{pmatrix} (\alpha_1 - \beta_1 y_2) y_1 \\ -(\alpha_2 - \beta_2 y_1) y_2 \end{pmatrix},$$

$$\mathbf{Df}(\mathbf{y}) = \begin{pmatrix} \alpha_1 - \beta_1 y_2 & -\beta_1 y_1 \\ \beta_2 y_2 & -(\alpha_2 - \beta_2 y_1) \end{pmatrix},$$

$$\mathbf{H}f_1(\mathbf{y}) = \begin{pmatrix} 0 & -\beta_1 \\ -\beta_1 & 0 \end{pmatrix}, \qquad \mathbf{H}f_2(\mathbf{y}) = \begin{pmatrix} 0 & \beta_2 \\ \beta_2 & 0 \end{pmatrix}.$$

See `taylorintegrator.hpp`.

(3d) ⚀ Experimentally determine the order of convergence of the considered Taylor expansion method when it is applied to solve (97). 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`.

Solution: From `taylorprey.cpp`, we see cubic algebraic convergence $O(h^3)$.

(3e) ⚀ What is the disadvantage of the Taylor method compared with a Runge-Kutta method?

Solution: As we can see in the error table, the error of the studied Runge-Kutta method and Taylor's method are practically identical. The obvious disadvantage of Taylor's method in comparison with Runge-Kutta methods is that the former involves rather complicated higher derivatives of f. If we want higher order, those formulas get even more complicated, whereas explicit Runge-Kutta methods work with only a few evaluations of f itself, yielding results which are comparable. Moreover, the Taylor expansion method cannot be applied for f, when this is given in procedural form.

Problem 4 System of ODEs

Consider the following initial value problem for a second-order system of ordinary differential equations:

$$\begin{aligned}
2\ddot{u}_1 - \ddot{u}_2 &= u_1(u_2 + u_1)\ , \\
-\ddot{u}_{i-1} + 2\ddot{u}_i - \ddot{u}_{i+1} &= u_i(u_{i-1} + u_{i+1})\ , \qquad i = 2,\dots,n-1\ , \\
2\ddot{u}_n - \ddot{u}_{n-1} &= u_n(u_n + u_{n-1})\ , \\
u_i(0) &= u_{0,i} \qquad i = 1,\dots,n\ , \\
\dot{u}_i(0) &= v_{0,i} \qquad i = 1,\dots,n\ ,
\end{aligned} \tag{93}$$

in the time interval $[0,T]$.

(4a) ⚁ Write (93) 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]).

Solution: The second order IVP can be rewritten as a first order one by introducing $\mathbf{v}$:

$$\begin{aligned}
\dot{u}_i &= v_i \qquad i = 1,\dots,n\ , \\
2\dot{v}_1 - \dot{v}_2 &= u_1(u_2 + u_1)\ , \\
-\dot{v}_{i-1} + 2\dot{v}_i - \dot{v}_{i+1} &= u_i(u_{i-1} + u_{i+1}) \qquad i = 2,\dots,n-1\ , \\
2\dot{v}_n - \dot{v}_{n-1} &= u_n(u_n + u_{n-1})\ , \\
u_i(0) &= u_{0,i} \qquad i = 1,\dots,n\ , \\
v_i(0) &= v_{0,i} \qquad i = 1,\dots,n\ .
\end{aligned}$$

The ODE system can be written in vector form

$$\dot{\mathbf{u}} = \mathbf{v}\ , \qquad \mathbf{C}\dot{\mathbf{v}} = \mathbf{g}(\mathbf{u}) := \begin{bmatrix} u_1(u_2+u_1) \\ u_i(u_{i-1}+u_{i+1}) \\ u_n(u_n+u_{n-1}) \end{bmatrix}\ ,$$

where $\mathbf{C} \in \mathbb{R}^{n,n}$ is

$$\mathbf{C} = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}.$$

In order to use standard interfaces to RK-SSM for first order ODEs, collect $\mathbf{u}$ and $\mathbf{v}$ in a $(2n)$-dimensional vector $\mathbf{y} = [\mathbf{u}; \mathbf{v}]$. Then the system reads

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_{n+1} \\ \vdots \\ y_{2n} \\ \mathbf{C}^{-1}\mathbf{g}(y_1, \dots, y_l) \end{bmatrix} .$$

(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.

Solution: See file `system.cpp` for the implementation. We observe convergence of order 4.00: this is expected since the function $\mathbf{f}$ is smooth.

Issue date: 03.12.2015

Hand-in: 10.12.2015 (in the boxes in front of HG G 53/54).

Version compiled on: July 16, 2016 (v. 1.0).

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

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 13

Problem 1 Matrix-valued Differential Equation (core problem)

First we consider the *linear* matrix differential equation

$$\dot{\mathbf{Y}} = \mathbf{A}\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad \text{with} \quad \mathbf{A} \in \mathbb{R}^{n\times n}. \tag{94}$$

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.

Solution: Let us consider the time derivative of $\mathbf{Y}^\top\mathbf{Y}$:

$$\begin{aligned}\frac{d}{dt}(\mathbf{Y}^\top\mathbf{Y}) &= \dot{\mathbf{Y}}^\top\mathbf{Y} + \mathbf{Y}^\top\dot{\mathbf{Y}}\\ &= (\mathbf{A}\mathbf{Y})^\top\mathbf{Y} + \mathbf{Y}^\top\mathbf{A}\mathbf{Y}\\ &= -\mathbf{Y}^\top\mathbf{A}\mathbf{Y} + \mathbf{Y}^\top\mathbf{A}\mathbf{Y}\\ &= 0\end{aligned}$$

This implies $\mathbf{Y}(t)^\top\mathbf{Y}(t)$ is constant. From this, it follows that the orthogonality is preserved, as claimed. In fact $\mathbf{I} = \mathbf{Y}(0)^\top\mathbf{Y}(0) = \mathbf{Y}(t)^\top\mathbf{Y}(t)$.

(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 (94)

Solution: The explicit Euler method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}(t_k, \mathbf{Y}_k).$$

For the given differential equation we therefore obtain

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{A}\mathbf{Y}_k.$$

The implicit Euler method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}(t_{k+1}, \mathbf{Y}_{k+1}).$$

For the given differential equation this yields

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{A}\mathbf{Y}_{k+1} \quad \Longrightarrow \quad \mathbf{Y}_{k+1} = (\mathbf{I} - h\mathbf{A})^{-1}\mathbf{Y}_k$$

Finally, the implicit mid-point method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}\big(\tfrac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{Y}_k + \mathbf{Y}_{k+1})\big),$$

hence

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}(\mathbf{Y}_k + \mathbf{Y}_{k+1}) \quad \Longrightarrow \quad \mathbf{Y}_{k+1} = \big(\mathbf{I} - \tfrac{h}{2}\mathbf{A}\big)^{-1}\big(\mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}\mathbf{Y}_k\big).$$

(1c) $\boxdot$ Investigate numerically, which one of the implemented methods preserves orthogonality in the sense of sub-problem (1a) for the ODE (94) 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).

Solution: Orthogonality is preserved only by the implicit midpoint rule. Explicit and implicit Euler methods do not preserve orthogonality. See `matrix_ode.cpp`.

From now we consider a non-linear ODE that is structurally similar to (94). We study the initial value problem

$$\dot{\mathbf{Y}} = -(\mathbf{Y} - \mathbf{Y}^\top)\mathbf{Y} =: f(\mathbf{Y}) \quad , \quad \mathbf{Y}(0) = \mathbf{Y}_0 \in \mathbb{R}^{n,n}, \tag{95}$$

whose solution is given by a *matrix-valued function* $t \mapsto \mathbf{Y}(t) \in \mathbb{R}^{n\times n}$.

(1d) $\boxdot$ Write a C++ function

```
Eigen::MatrixXd matode(const Eigen::MatrixXd & Y0,
    double T)
```

which solves (95) 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`.

Solution: The class `ode45` can take `Eigen::MatrixXd` as `StateType`. Alternatively, one can transform Matrices to Vectors (and viceversa), using the `Eigen::Map` function (similar to MATLAB own `reshape` function). See `matrix_ode.cpp`.

(1e) ⚁ Show that the function $t \mapsto \mathbf{Y}^{\top}(t)\mathbf{Y}(t)$ is constant for the exact solution $\mathbf{Y}(t)$ of (95).

HINT: Remember the general product rule [1, Eq. (2.4.9)].

Solution: By the product rule and using the fact that $\mathbf{Y}$ is a solution of the IVP we obtain

$$\begin{aligned}\frac{d}{dt}(\mathbf{Y}^{\top}(t)\mathbf{Y}(t)) =& \frac{d}{dt}(\mathbf{Y}^{\top}(t))\mathbf{Y}(t) + \mathbf{Y}^{\top}(t)\frac{d}{dt}(\mathbf{Y}(t)) \\ =& (-(\mathbf{Y}(t) - \mathbf{Y}^{\top}(t))\mathbf{Y}(t))^{\top}\mathbf{Y}(t) + \mathbf{Y}(t)(-(\mathbf{Y}(t) - \mathbf{Y}^{\top}(t))\mathbf{Y}(t)) \\ =& -\mathbf{Y}^{\top}(t)\mathbf{Y}^{\top}(t)\mathbf{Y}(t) + \mathbf{Y}^{\top}\mathbf{Y}(t)\mathbf{Y}(t) \\ & -\mathbf{Y}^{\top}\mathbf{Y}(t)\mathbf{Y}(t) + \mathbf{Y}^{\top}(t)\mathbf{Y}^{\top}(t)\mathbf{Y}(t) = 0.\end{aligned}$$

implying that the map is constant.

(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`.

Solution: Let $\mathbf{Y}_k$ be the output of `matode`. We compute $\mathbf{Y}_0^{\top}\mathbf{Y}_0 - \mathbf{Y}_k^{\top}\mathbf{Y}_k$ and check if the (Frobenius) norm is smaller than a constant times the machine `eps`. Even for an orthogonal matrix, we have to take into account round-off errors. See `matrix_ode.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 $T = 1$.

Solution: The invariant is not preserved. See `matrix_ode.cpp`.

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

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 \\ \hline & 1/6 & 1/6 & 2/3 \end{array} \tag{96}$$

(2a) ⊡ Consider the prey/predator model

$$\dot{y}_1(t) = (1 - y_2(t))y_1(t) \tag{97}$$
$$\dot{y}_2(t) = (y_1(t) - 1)y_2(t) \tag{98}$$
$$\mathbf{y}(0) = [100, 1]\,. \tag{99}$$

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 $2^{-j}, j = 2, \dots, 13$.

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`.

Solution: The scheme is of order 3. With big step sizes the scheme is unstable. At least 64 steps are needed for the solution to be stable. See `stabrk.cpp`.

(2b) ⊡ Calculate the stability function $S(z)$, $z = h\lambda$, $\lambda \in \mathbb{C}$ of the method given by the table (96).

Solution: We obtain the stability function $S(z)$ by applying our method to the model problem $\dot{y}(t) = \lambda y$, $y(0) = y_0$ and by writing the result in the form $y_1 = S(z)y_0$, where $z := h\lambda$. For the increment, we obtain

$$\begin{aligned} k_1 &= \lambda y_0, \\ k_2 &= \lambda(y_0 + hk_1) \\ &= \lambda y_0(1 + z), \\ k_3 &= \lambda(y_0 + \tfrac{h}{4}(k_1 + k_2)) \\ &= \lambda y_0(1 + \tfrac{1}{2}z + \tfrac{1}{4}z^2), \end{aligned}$$

and for the update

$$\begin{aligned} y_1 &= y_0 + \frac{h\lambda y_0}{6}\left(1 + 1 + z + 4\left(1 + \frac{1}{2}z + \frac{1}{4}z^2\right)\right) \\ &= y_0 \underbrace{\left(1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^6\right)}_{=:S(z)}. \end{aligned}$$

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

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{100}$$

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

$$\mathbf{\Phi}: \mathbb{R} \times D \to D, \quad (t, \mathbf{y}) \mapsto \mathbf{\Phi}(t, \mathbf{y})$$

satisfies

$$\frac{\partial \mathbf{\Phi}}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\mathbf{\Phi}(t, \mathbf{y})).$$

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 $\mathsf{D}_{\mathbf{y}}$

$$\frac{\partial \,\mathsf{D}_{\mathbf{y}} \mathbf{\Phi}}{\partial t}(t, \mathbf{y}) = \mathsf{D}_{\mathbf{y}} \frac{\partial \mathbf{\Phi}}{\partial t}(t, \mathbf{y}) = \mathsf{D}_{\mathbf{y}}(\mathbf{f}(\mathbf{\Phi}(t, \mathbf{y}))) = \mathsf{D}\,\mathbf{f}(\mathbf{\Phi}(t, \mathbf{y}))\, \mathsf{D}_{\mathbf{y}} \mathbf{\Phi}(t, \mathbf{y}).$$

Abbreviating $\mathbf{W}(t, \mathbf{y}) := \mathsf{D}_{\mathbf{y}} \mathbf{\Phi}(t, \mathbf{y})$ we can rewrite this as the non-autonomous ODE

$$\dot{\mathbf{W}} = \mathsf{D}\,\mathbf{f}(\mathbf{\Phi}(t, \mathbf{y}))\mathbf{W}. \tag{101}$$

Here, the state $\mathbf{y}$ can be regarded as a parameter. Since $\mathbf{\Phi}(0, \mathbf{y}) = \mathbf{y}$, we also know $\mathbf{W}(0, \mathbf{y}) = \mathbf{I}$ (identity matrix), which supplies an initial condition for (101). In fact, we can even merge (100) and (101) into the ODE

$$\frac{d}{dt}[\mathbf{y}(\cdot)\ ,\ \mathbf{W}(\cdot, \mathbf{y}_0)] = [\mathbf{f}(\mathbf{y}(t))\ ,\ \mathsf{D}\,\mathbf{f}(\mathbf{y}(t))\mathbf{W}(t, \mathbf{y}_0)]\ , \tag{102}$$

which is autonomous again.

Now let us apply (101)/(102). As in [1, Ex. 11.1.9], we consider the following autonomous Lotka-Volterra differential equation of a predator-prey model

$$\begin{aligned} \dot{u} &= (2-v)u \\ \dot{v} &= (u-1)v \end{aligned} \tag{103}$$

on the state space $D = \mathbb{R}_+^2$, $\mathbb{R}_+ = \{\xi \in \mathbb{R} : \xi > 0\}$. All the solutions of (103) 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 $\mathbf{y} \in D$, (101) represents an ODE. What is its state space?

Solution: By construction, $\mathbf{\Phi}$ is a function with values in $\mathbb{R}^d$. Thus, $\mathsf{D}_\mathbf{y}\,\mathbf{\Phi}$ has values in $\mathbb{R}^{d,d}$, and so the state space of (101) is $\mathbb{R}^{d,d}$, a space of matrices.

(3b) ⊡ What is the right hand side function for the ODE (101), in the case of the $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ given by the Lotka-Volterra ODE (103)? You may write $u(t), v(t)$ for solutions of (103).

Solution: Writing $\mathbf{y} = [u, v]^T$, the map $\mathbf{f}$ associated to (103) is $\mathbf{f}(\mathbf{y}) = [(2-v)u, (u-1)v]^T$. Therefore

$$\mathsf{D}\,\mathbf{f}(\mathbf{y}) = \begin{bmatrix} 2-v & -u \\ v & u-1 \end{bmatrix}.$$

Thus, (103) becomes

$$\dot{\mathbf{W}} = \begin{bmatrix} 2-v(t) & -u(t) \\ v(t) & u(t)-1 \end{bmatrix}\mathbf{W}. \tag{104}$$

This is a non-autonomous ODE.

(3c) ⊡ From now on we write $\mathbf{\Phi}: \mathbb{R} \times \mathbb{R}_+^2 \to \mathbb{R}_+^2$ for the evolution operator associated with (103). Based on $\mathbf{\Phi}$ 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 (103) with initial value

$$\mathbf{y}_0 = \begin{bmatrix} u(0) \\ v(0) \end{bmatrix}$$

is equal to a given value T_P.

Solution: Let $\mathbf{F}$ be defined by

$$\mathbf{F}(\mathbf{y}) := \mathbf{\Phi}(T_P, \mathbf{y}) - \mathbf{y}.$$

If the solution of the system (103) with initial value $\mathbf{y}_0$ has period T_P, then we have $\mathbf{\Phi}(T_P, \mathbf{y}_0) = \mathbf{y}_0$, so $\mathbf{F}(\mathbf{y}_0) = 0$.

(3d) We write $\mathbf{W}(T, \mathbf{y}_0)$, $T \geq 0$, $\mathbf{y}_0 \in \mathbb{R}^2_+$ for the solution of (101) for the underlying ODE (103). Express the Jacobian of $\mathbf{F}$ from (3c) by means of $\mathbf{W}$.

Solution: By definition of $\mathbf{W}$ we immediately obtain

$$\mathsf{D}\,\mathbf{F}(\mathbf{y}) = \mathbf{W}(T_P, \mathbf{y}) - \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix.

(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. 374].

Solution: If $\mathbf{y}_0 \in \mathbb{R}^2_+$ is a solution, then every state on the trajectory $\mathbf{y}_0$ will also be a solution. Only if the trajectory collapses to a point, that is, if $\mathbf{y}_0$ is a stationary point, $\mathbf{f}(\mathbf{y}_0) = 0$, we can expect uniqueness.

(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

```
std::pair<Vector2d,Matrix2d> PhiAndW(double u0, double
    v0, double T)
```

that computes $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and $\mathbf{W}(T, [u_0, v_0]^T)$. The first component of the output pair should contain $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and the second component the matrix $\mathbf{W}(T, [u_0, v_0]^T)$. See `LV_template.cpp`.

HINT: As in (102), 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).

Solution: Writing $\mathbf{w} = [u, v, W_{11}, W_{21}, W_{12}, W_{22}]^T$, by (103) and (104) we have that

$$\begin{aligned}
\dot{w}_1 &= (2 - w_2)w_1 & \dot{w}_3 &= (2 - w_2)w_3 - w_1 w_4 & \dot{w}_5 &= (2 - w_2)w_5 - w_1 w_6 \\
\dot{w}_2 &= (w_1 - 1)w_2 & \dot{w}_4 &= w_2 w_3 + (w_1 - 1)w_4 & \dot{w}_6 &= w_2 w_5 + (w_1 - 1)w_6
\end{aligned}$$

with initial conditions

$$w_1(0) = u_0, \quad w_2(0) = v_0, \quad w_3(0) = 1, \quad w_4(0) = 0, \quad w_5(0) = 0, \quad w_6(0) = 1,$$

since $\mathbf{W}(0,\mathbf{y}) = \mathsf{D}_{\mathbf{y}}\,\mathbf{\Phi}(0,\mathbf{y}) = \mathsf{D}_{\mathbf{y}}\,\mathbf{y} = \mathbf{I}$.

The implementation of the solution of this system is given in the file `LV.cpp`.

(3g) Using `PhiAndW`, write a C++ routine that determines initial conditions $u(0)$ and $v(0)$ such that the solution of the system (103) 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 $\|\mathbf{F}(\mathbf{y})\| \le 10^{-5}$. 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 $v(0) = 2.081$.

Solution: See file `LV.cpp`.

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

$$\mathbf{y}_1 = \mathbf{y}_0 + h\,\varphi\big(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0), \tag{105}$$

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 (105) consistent with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ (see [1, Def. 11.3.10])? Explain your answer.

Solution: In view of (105), consistency is equivalent to $\varphi\big(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0) = \mathbf{f}(\mathbf{y}_0)$ for $h = 0$. Since φ is not defined for the zero matrix, this should be intended in the limit as $h \to 0$. By definition of the matrix exponential we have $e^{hZ} = \mathsf{Id} + hZ + O(h^2)$ as $h \to 0$. Therefore

$$\varphi(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)) = (\exp(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)) - \mathsf{Id})(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0))^{-1} = (\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)) + O(h))\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)^{-1},$$

whence

$$\lim_{h\to 0} \varphi\big(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0) = \mathbf{f}(\mathbf{y}_0),$$

as desired.

(4b) Show that the exponential Euler single step method defined in (105) 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.

Solution: Given $\mathbf{y}(0)$, a step of size t of the method gives

$$\begin{aligned}
\mathbf{y}_1 &= \mathbf{y}(0) + t\,\varphi\big(t\ \mathsf{D}\,\mathbf{f}(\mathbf{y}(0))\big)\,\mathbf{f}(\mathbf{y}(0))\\
&= \mathbf{y}(0) + t\,\varphi\big(t\,\mathbf{A}\big)\,\mathbf{A}\,\mathbf{y}(0)\\
&= \mathbf{y}(0) + t\,\big(\exp(\mathbf{A}t) - \mathsf{Id}\big)\,(t\mathbf{A})^{-1}\,\mathbf{A}\,\mathbf{y}(0)\\
&= \mathbf{y}(0) + \exp(\mathbf{A}t)\,\mathbf{y}(0) - \mathbf{y}(0)\\
&= \exp(\mathbf{A}t)\,\mathbf{y}(0)\\
&= \mathbf{y}(t).
\end{aligned}$$

(4c) Determine the region of stability of the exponential Euler single step method defined in (105) (see [1, Def. 12.1.49]).

Solution: The discrete evolution $\mathbf{\Psi}^h$ associated to (105) is given by

$$\mathbf{\Psi}^h(y) = y + h\varphi(h\,\mathsf{D}\,\mathbf{f}(\mathbf{y}))\mathbf{f}(\mathbf{y}).$$

Thus, for a scalar linear ODE of the form $\dot{y} = \lambda y$ for $\lambda \in \mathbb{C}$ we have ($f(y) = \lambda y$, $Df(y) = \lambda$)

$$\mathbf{\Psi}^h(y) = y + h\frac{e^{h\lambda} - 1}{h\lambda}\lambda y = e^{h\lambda}y.$$

Thus, the stability function $S : \mathbb{C} \to \mathbb{C}$ is given by $S(z) = e^z$. Therefore, the region of stability of this method is

$$\mathcal{S}_{\mathbf{\Psi}} = \{z \in \mathbb{C} : |e^z| < 1\} = \{z \in \mathbb{C} : e^{\operatorname{Re} z} < 1\} = \{z \in \mathbb{C} : \operatorname{Re} z < 0\}.$$

It is useful to compare this with [1, Ex. 12.1.51].

Alternatively, one may simply appeal to (4b) to see that the method has the "ideal" region of stability as introduced in [1, § 12.3.33].

(4d) ⚁ Write a C++ function

```
template <class Function, class Function2>
Eigen::VectorXd ExpEulStep(Eigen::VectorXd y0, Function
    f, Function2 df, double h)
```

that implements (105). 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`.

Solution: See `ExpEul.cpp`.

(4e) ⚂ What is the order of the single step method (105)? 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 $k = 1, \dots, 15$. As in Problem 2 in Problem Sheet 12, for each k compute and show an approximate order of convergence.

HINT: The exact solution is

$$y(t) = \frac{y(0)}{y(0) + \big(1 - y(0)\big)e^{-t}}.$$

Solution: Error = $O(h^2)$. See `ExpEul.cpp`.

Issue date: 10.12.2015
Hand-in: – (in the boxes in front of HG G 53/54).

Version compiled on: July 16, 2016 (v. 1.0).

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

AS 2015

ETH Zürich
D-MATH

Numerical Methods for CSE

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.

Solution: See `implicit_rkintegrator.hpp`.

(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].

Solution: The Butcher scheme used corresponds to the Radau RK-SSM of order 3 (see [1, Ex. 12.3.44] for the complete scheme). Thus, the method is A-stable and L-stable.

(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.

Solution: As expected, the observed order of convergence is 3 (see [1, Ex. 12.3.44]).

Problem 2 Initial Value Problem With Cross Product

We consider the initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \mathbf{a} \times \mathbf{y} + c\mathbf{y} \times (\mathbf{a} \times \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 = [1,1,1]^\top, \tag{106}$$

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 (106).

HINT: Target the time derivative $\frac{d}{dt}\|\mathbf{y}(t)\|_2^2$ and use the product rule.

Solution: We have

$$\begin{aligned}
&(\mathbf{a} \times \mathbf{y} + c(\mathbf{y} \times (\mathbf{a} \times \mathbf{y}))) \cdot \mathbf{y} = 0 \\
&\Rightarrow D_t \|\mathbf{y}(t)\|_2^2 = 2\dot{\mathbf{y}}(t) \cdot \mathbf{y}(t) = 0 \\
&\Rightarrow \|\mathbf{y}(t)\|_2^2 = \text{const. } \forall t.
\end{aligned}$$

Thus $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}(0)\|_2 = \|\mathbf{y}_0\|_2$.

(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 $\mathbf{a} = [1,0,0]^\top$.

Solution: Using the definition of cross product, a simple but tedious calculation shows that

$$D\mathbf{f}(\mathbf{y}) = \begin{bmatrix} -ca_2y_2 - ca_3y_3 & -a_3 + 2ca_1y_2 - ca_2y_1 & a_2 - ca_3y_1 + 2ca_1y_3 \\ a_3 - ca_1y_2 + 2ca_2y_1 & -ca_3y_3 - ca_1y_1 & -a_1 + 2ca_2y_3 - ca_3y_2 \\ -a_2 + 2ca_3y_1 - ca_1y_3 & a_1 - ca_2y_3 + 2ca_3y_2 & -ca_1y_1 - ca_2y_2 \end{bmatrix}.$$

Thus for $\mathbf{y} = \mathbf{a}$ we obtain

$$D\mathbf{f}(\mathbf{a}) = \begin{bmatrix} -c(a_2^2 + a_3^2) & -a_3 + ca_1a_2 & a_2 + ca_3a_1 \\ a_3 + ca_1a_2 & -c(a_1^2 + a_3^2) & -a_1 + ca_2a_3 \\ -a_2 + ca_1a_3 & a_1 + ca_2a_3 & -c(a_1^2 + a_2^2) \end{bmatrix}.$$

A direct calculation gives that the spectrum is given by

$$\sigma(D\mathbf{f}(\mathbf{a})) = \{0, -c\,\|a\|_2^2 \pm i\,\|a\|_2\} = \{0, -c \pm i\}.$$

Therefore, the problem is stiff for large c (see [1, § 12.2.14]).

(2c) ⚁ For $\mathbf{a} = [1,0,0]^\top$, (106) 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 25.

Solution: In the plot, we see that, for the solver `ode45`, the number of steps rises with c. On the other hand, `ode23s` uses roughly the same amount of steps, irregardless of the value of c chosen.

As `ode45` is an explicit solver, it suffers under stability-based step-size limits for large $c > 0$. The implicit solver `ode23s` must however not take smaller step-sizes to satisfy the tolerance for big c. In other words: the problem becomes stiffer, the greater the parameter c is. This is expected from what we saw above.

(2d) ⚁ Formulate the non-linear equation given by the implicit mid-point rule for the initial value problem (106).

Solution: With the formula for the implicit mid-point rule

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right)$$

the formulation of (106) is given as

$$\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h} = \mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) + c\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) \times \left(\mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right)\right)$$

(2e) ⚁ Solve (106) 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?

Solution: See file `cross.cpp`. As expected from (2a), the norm of the approximate states is constant.

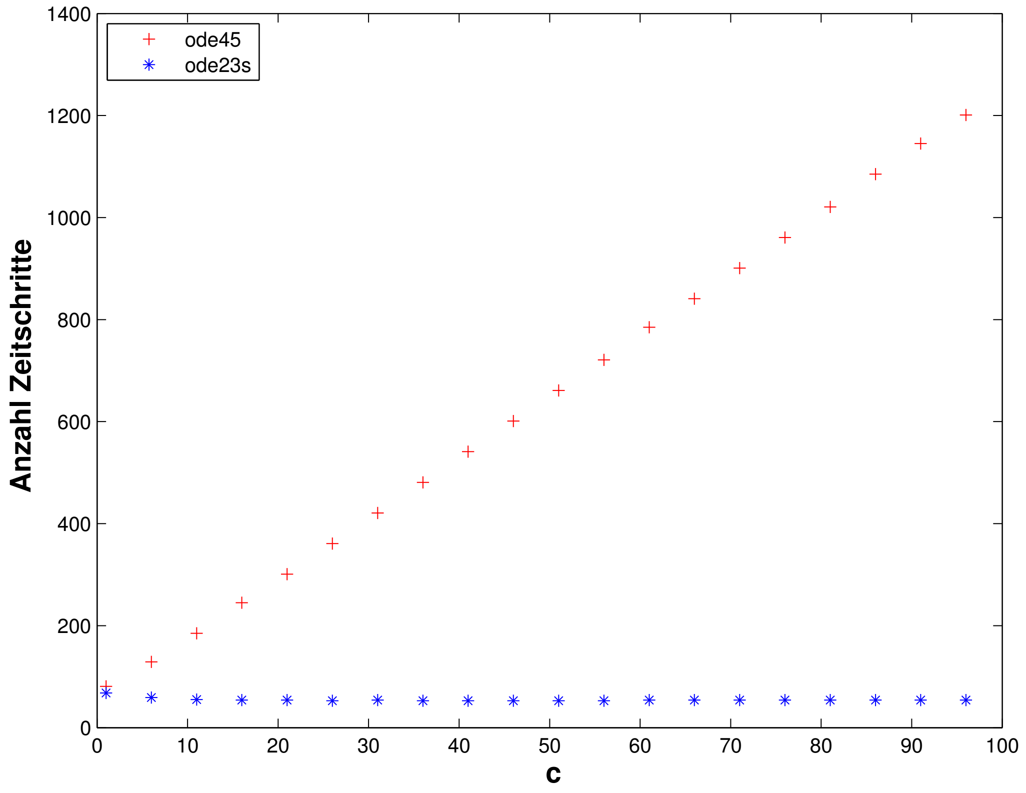


Figure 25: Subproblem (2c): number of steps used by standard MATLAB integrators in relation to the parameter c.

(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.

Solution: The linear implicit mid-point rule is obtained by developing the increment $\mathbf{k}_1$ of the implicit mid point rule by its Taylor series

$$\mathbf{k}_1 = f\left(\mathbf{y}_k + \frac{h}{2}\mathbf{k}_1\right) = \mathbf{f}(\mathbf{y}_k) + \frac{h}{2}D\mathbf{f}(\mathbf{y}_k)\mathbf{k}_1 + O(h^2)$$

and only taking the linear terms. Since the non-linear method is given by $\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1$, the linearization reads

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1^{\text{lin.}}, \qquad \mathbf{k}_1^{\text{lin.}} := \left(I - \frac{h}{2}D\mathbf{f}(\mathbf{y}_k)\right)^{-1} \mathbf{f}(\mathbf{y}_k).$$

(2g) ⚂ Implement the linear–implicit midpoint rule using the template provided in `cross_template.cpp`. Use this method to solve (106) with $\mathbf{a} = [1,0,0]^\top$, $c = 1$ 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?

Solution: See file `cross.cpp`. The sequence of the norms is not exactly constant: this is due to the approximation introduced with the linearization.

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

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{107}$$

and discretize it with a *semi-implicit* Runge-Kutta SSM (*Rosenbrock method*):

$$\begin{aligned} \mathbf{W}\mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_0) \\ \mathbf{W}\mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\mathbf{k}_1) - ah\mathbf{J}\mathbf{k}_1 \\ \mathbf{y}_1 &= \mathbf{y}_0 + h\mathbf{k}_2 \end{aligned} \tag{108}$$

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 (108), 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 $\dot{y} = \lambda y, \lambda \in \mathbb{C}$.

Solution: For a scalar ODE, the Jacobian is just the derivative w.r.t. y, whence $J = Df(y) = f'(y) = (\lambda y)' = \lambda$. The quantity $\mathbf{W}$ is, therefore, a scalar quantity as well:

$$W = 1 - ahJ = 1 - ah\lambda$$

If we plug everything together and assume h is small enough:

$$\begin{aligned}
y_1 &= y_0 + h\frac{\mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\mathbf{k}_1) - ah\mathbf{J}\mathbf{k}_1}{W} \\
&= y_0 + h\frac{\mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\frac{\mathbf{f}(\mathbf{y}_0)}{W}) - ah\mathbf{J}\frac{\mathbf{f}(\mathbf{y}_0)}{W})}{W} \\
&= y_0 + h\frac{\lambda(\mathbf{y}_0 + \frac{1}{2}h\frac{\lambda\mathbf{y}_0}{1-ah\lambda}) - \frac{ah\lambda^2 y_0}{1-ah\lambda}}{1-ah\lambda} \\
&= \frac{(1-ah\lambda)^2 + h\lambda(1-ah\lambda) + \frac{1}{2}h^2\lambda^2 - ah^2\lambda^2}{(1-ah\lambda)^2}y_0 \\
&= \frac{(1+a^2z^2-2az) + z(1-az) + \frac{1}{2}z^2 - az^2}{(1-az)^2}y_0 \\
&= \frac{1+(1-2a)z + (\frac{1}{2}-2a+a^2)z^2}{(1-az)^2}y_0
\end{aligned}$$

Since $\frac{1}{2} - 2a + a^2 = 0$, it follows

$$S(z) = \frac{1+(1-2a)z}{(1-az)^2}$$

(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 (108).

HINT: The idea behind this sub-problem is elucidated in [1, Rem. 12.1.19]. Apply [1, Lemma 12.1.21].

Solution: We compute:

$$\begin{aligned} S(0) &= 1, \\ S'(0) &= 1, \\ S''(0) &= 4a - 2a^2, \\ S'''(0) &= 18a^2 - 12a^3, \end{aligned}$$

therefore

$$S(z) = 1 + z + (2a - a^2)z^2 + (3a^2 - 2a^3)z^3 + O(z^4).$$

We can also compute the expansion of exp:

$$\exp(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + O(z^4)$$

and since $2a - a^2 = \frac{1}{2}$ but $3a^2 - 2a^3 = \frac{1}{2}(\sqrt{2} - 1) \neq \frac{1}{6}$, we have:

$$|S(z) - \exp(z)| = O(|z|^3)$$

Using [1, Lemma 12.1.21], we deduce that the maximal order q of the scheme is $q = 2$.

(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 (108).

HINT: See `rosenbrock_template.cpp`.

Solution: See `rosenbrock.cpp`.

(3d) ⚂ Explore the order of the method (108) empirically by applying it to the IVP for the limit cycle [1, Ex. 12.2.5]:

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2)\mathbf{y}, \tag{109}$$

with $\lambda = 1$ and initial state $\mathbf{y_0} = [1,1]^\top$ on $[0,10]$. Use fixed timesteps of size $h = 2^{-k}, k = 4, \dots, 10$ and compute a reference solution with $h = 2^{-12}$ step size. Monitor the maximal mesh error:

$$\max_j \|\mathbf{y}_j - \mathbf{y}(t_j)\|_2.$$

Solution: The Jacobian Df is:

$$D\mathbf{f}(\mathbf{y}) = \begin{bmatrix} \lambda(1 - \|\mathbf{y}\|^2) - 2\lambda y_0^2 & -1 - 2\lambda y_1 y_0 \\ 1 - 2\lambda y_1 y_0 & \lambda(1 - \|\mathbf{y}\|^2) - 2\lambda y_1^2 \end{bmatrix}.$$

For the implementation, cf. `rosenbrock.cpp`.

(3e) ⚃ Show that the method (108) 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:

$$\forall z \in \mathbb{C}^- |f(z)| < \sup_{\tau\in\mathbb{R}} |f(i\tau)|.$$

Solution: We start by proving that the method is A-stable [1, § 12.3.30], meaning that $S(z) < 1, \forall z \in \mathbb{C}, \operatorname{Re}(z) < 0$. First of all, we can compute the complex norm of the stability function at $z = iy, y \in \mathbb{R}$ as:

$$|S(iy)|^2 = \frac{|1 + (1-2a)iy|^2}{|(1-aiy)^2|^2} = \frac{1 + (1-2a)^2 y^2}{(1 + a^2 y^2)^2} = \frac{1 + (1 - 4a + 4a^2)y^2}{(1 + 2a^2y^2 + a^4y^4)}.$$

Notice that $1 - 4a + 4a^2 = 2a^2$, therefore:

$$|S(iy)|^2 = \frac{1 + 2a^2y^2}{1 + 2a^2y^2 + a^4y^4} < 1.$$

The norm of the function S is bounded on the imaginary axis. Observe that $|S(z)| \to 0, |z| \to \infty$, which, follows from the fact that the degree of the denominator of S is bigger than the polynomial degree of the numerator. Notice that $1 - 2a = 1 - 2\frac{1}{\sqrt{(2)}+2} = \sqrt{2} - 2$.

The only pole of the function is at $z = 1/a$, therefore the function is holomorphic on the left complex plane.

Applying the theorem of the hint on concludes that the absolute value of the function is bounded by 1 on the left complex plane.

The L-stability follows immediately with A-stability and the fact that the absolute value of the function converges to zero as $|z| \to \infty$.

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

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|cccc} c_1 & \gamma & \cdots & & 0 \\ c_2 & a_{21} & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ c_s & a_{s1} & \cdots & a_{s,s-1} & \gamma \\ \hline & b_1 & \cdots & b_{s-1} & b_s \end{array}, \tag{110}$$

with $\gamma \neq 0$.

More concretely, in this problem the scalar linear initial value problem of second order

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{111}$$

should be solved numerically using a SDIRK-method (**S**ingly **D**iagonally **I**mplicit **R**unge-

Kutta Method). It is a Runge-Kutta method described by the Butcher scheme

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & 1/2 & 1/2 \end{array} . \tag{112}$$

(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 (112) applied to the initial value problem corresponding to the differential equation $\dot{\boldsymbol{y}} = \mathbf{f}(t, \boldsymbol{y})$.

Solution: The increments $\boldsymbol{k}_i$ are given by

$$\begin{aligned} \boldsymbol{k}_1 &= f\left(t_0 + h\gamma, \boldsymbol{y}_0 + h\gamma\boldsymbol{k}_1\right), \\ \boldsymbol{k}_2 &= f\left(t_0 + h(1-\gamma), \boldsymbol{y}_0 + h(1-2\gamma)\boldsymbol{k}_1 + h\gamma\boldsymbol{k}_2\right). \end{aligned}$$

(4c) ⊡ Show that, the stability function $S(z)$ of the SDIRK-method (112) 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.

Solution: The stability function $S(z)$ of a method is derived by applying the method to the scalar, linear test equation

$$\dot{y}(t) = \lambda y(t)$$

The solution can be written as

$$y_{k+1} = S(z) y_k$$

where $S(z)$ is the stability function and $z := h\lambda$.

In the case of the SDIRK-method, we get

$$\begin{aligned} k_1 &= \lambda(y_k + h\gamma k_1) \\ k_2 &= \lambda(y_k + h(1-2\gamma)k_1 + h\gamma k_2), \end{aligned}$$

therefore

$$\begin{aligned} k_1 &= \frac{\lambda}{1-h\lambda\gamma} y_k, \\ k_2 &= \frac{\lambda}{1-h\lambda\gamma}(y_k + h(1-2\gamma)k_1). \end{aligned}$$

Furthermore

$$y_{k+1} = y_k + \frac{h}{2}(k_1 + k_2),$$

with $z := h\lambda$ and by plugging in k_1 and k_2 we arrive at

$$y_{k+1} = \underbrace{\left(1 + \frac{z}{2(1-\gamma z)}\left(2 + \frac{z(1-2\gamma)}{1-\gamma z}\right)\right)}_{=:S(z)} y_k.$$

Hence

$$S(z) = \frac{2(1-\gamma z)^2 + 2z(1-\gamma z)^2 + z^2(1-2\gamma)}{2(1-\gamma z)^2}$$

and after collecting the powers of z in the numerator we get

$$S(z) = \frac{1 + z(1-2\gamma) + z^2(\gamma^2 - 2\gamma + \frac{1}{2})}{(1-\gamma z)^2}.$$

For $\gamma = 1$ the stability function is therefore

$$S_1(z) := \frac{1 - z - \frac{z^2}{2}}{(1-z)^2}. \tag{113}$$

Verification of the A-stability of (113)**:**

By definition [1, § 12.3.30], we need to show that $|S_1(z)| \le 1$ for all $z \in \mathbb{C}^- := \{z \in$

$\mathbb{C} \mid \operatorname{Re} z < 0\}$. In order to do this we consider the stability function on the imaginary axis

$$\begin{aligned}|S_1(iy)|^2 &= \frac{|1-iy-(iy)^2/2|^2}{|1-iy|^4} \\ &= \frac{|1-iy+y/2|^2}{|1-iy|^4} \\ &= \frac{(1+y^2/2)^2+y^2}{(1+y^2)^2} \\ &= \frac{1+2y^2+y^4/4}{1+2y^2+y^4} \leq 1, \quad y \in \mathbb{R}.\end{aligned}$$

Since the only pole ($z = 1$) of the rational function $S_1(z)$ lies in the positive half plane of $\mathbb{C}$, the function S_1 is holomorphic in the left half plane. Furthermore S_1 is bounded by 1 on the boundary of this half plane (i.e. on the imaginary axis). So by the maximum principle for holomorphic functions (hint) S_1 is bounded on the entire left half plane by 1. This implies in particular that S_1 is A-stable.

Verification of the L-stability of (113)**:**
S_1 is not L-stable (cf. definition [1, § 12.3.37]), because

$$\lim_{\operatorname{Re} z \to -\infty} |S_1(z)| = \lim_{\operatorname{Re} z \to -\infty} \left| \frac{1-z-z^2/2}{1-2z+z^2} \right| = \frac{1}{2} \neq 0.$$

(4d) ⊡ Formulate (111) as an initial value problem for a linear first order system for the function $\boldsymbol{z}(t) = (y(t), \dot{y}(t))^\top$.

Solution: Define $z_1 = y$, $z_2 = \dot{y}$, then the initial value problem

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{114}$$

is equivalently to the first order system

$$\begin{aligned}\dot{z}_1 &= z_2 \\ \dot{z}_2 &= -z_1 - z_2,\end{aligned} \tag{115}$$

with initial values $z_1(0) = 1$, $z_2(0) = 0$.

(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 (112) 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`.

Solution: Let

$$\mathbf{A} := \begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix}.$$

Then

$$\begin{aligned} k_1 &= \mathbf{A}\boldsymbol{y}_0 + h\gamma\mathbf{A}k_1 \\ k_2 &= \mathbf{A}\boldsymbol{y}_0 + h(1-2\gamma)\mathbf{A}k1 + h\mathbf{A}k_2, \end{aligned}$$

so

$$\begin{aligned} k_1 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}\mathbf{A}\boldsymbol{y}_0 \\ k_2 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}(\mathbf{A}\boldsymbol{y}_0 + h(1-2\gamma)\mathbf{A}k_1). \end{aligned}$$

See the implementation in `sdirk.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 $\boldsymbol{y}_0 = [1,0]^\top$ as initial value, `T=10` as end time and `N=20,40,80,...,10240` as steps.

Solution: The numerically estimated order of convergence is 3, see `sdirk.cpp`.

Issue date: 17.12.2015

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

Version compiled on: July 16, 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.