First name		Grade
Last name		
Department		
Computer Name		
Legi No.		
Date	02.02.2016	

1	2	3	4	5	Total

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

Final exam
Numerical Methods for CSE

R. Hiptmair, G. Alberti, F. Leonardi

February 2, 2016

A "`CMake`" file is provided with the templates. To generate a "`Makefile`" for all problems, type "`cmake .`" in the folder "`~/results`". Compile your programs with "`make`". For example, in order to compile and run the C++ code related to Problem 3, type "`cmake .`", then "`make`" (or "`make problem3`" to compile only `problem3.cpp`) and finally "`./problem3`".

If you want to manually compile your code, use:

```
g++ -std=c++11 -I/usr/include/eigen3
    -Wno-deprecated-declarations filename.cpp -o programname
```

or

```
clang++ -std=c++11 -I/usr/include/eigen3
    -Wno-deprecated-declarations filename.cpp -o programname
```

The flag `-Wno-deprecated-declarations` suppresses some EIGEN warning.

For each problem requiring C++ implementation, a template file named `problemX.cpp` is provided. For your own convenience, there is a marker `TODO` in the places where you are supposed to write your own code.

Problem 1 Complex roots (16 pts.)

Given a complex number $w = u + iv$, $u, v \in \mathbb{R}$ with $v \geq 0$, its root $\sqrt{w} = x + iy$ can be defined by

$$x := \sqrt{(\sqrt{u^2 + v^2} + u)/2}, \tag{1}$$

$$y := \sqrt{(\sqrt{u^2 + v^2} - u)/2}. \tag{2}$$

(1a) For what $w \in \mathbb{C}$ will the direct implementation of (1) and (2) be vulnerable to cancellation?

Solution: As $u/v \to -\infty$ or when $v = 0$ and $u < 0$, we have $\sqrt{u^2 + v^2} \approx -u$, and so cancellations may arise in $\sqrt{u^2 + v^2} + u$. Therefore, the implementation of x will be vulnerable to cancellation. Similarly, as $u/v \to +\infty$ or when $v = 0$ and $u > 0$, we have $\sqrt{u^2 + v^2} \approx u$, and so cancellations may arise in $\sqrt{u^2 + v^2} - u$. Therefore, the implementation of y will be vulnerable to cancellation.

In conclusion, the implementation of these formulas will be vulnerable to cancellation when $|u|$ is much bigger than v.

(1b) Compute xy as an expression of u and v.

Solution: An immediate calculation shows that $xy = v/2$.

(1c) Implement a function

```
1 std::complex<double> myroot( std::complex<double> w );
```

that computes the root of `w` as given by (1) and (2) without the risk of cancellation. You may use only real arithmetic: for instance, you may not apply the function `sqrt` with complex arguments.

Test your implementation with $w = 10^{20} + 5i$ and with $w = -5 + 10^{20}i$.

DEPENDS ON: subproblems (1a) and (1b).

Solution: See implementation in `problem1_solution.cpp` and below.

```
1 #include <iostream>
2 #include <complex>
3 #include <cmath>
4
5 //! \brief Compute complex root
```

```
6  //! \param[in] w complex number with non negative imaginary
       parts
7  //! \return the square root of w with non negative real and
       imaginary parts
8  std::complex<double> myroot( std::complex<double> w ) {
9      double x,y;
10     double u = w.real();
11     double v = w.imag();
12
13     // TODO: problem 1c: construct x and y as functions of u
           and v
14
15     if (v==0) return sqrt(u);
16
17     if (u > 0) {
18         x = sqrt((sqrt(u*u+v*v)+u)/2.);
19         y = v/(2*x);
20     } else {
21         y = sqrt((sqrt(u*u+v*v)-u)/2.);
22         x = v/(2*y);
23     }
24
25     return std::complex<double> (x,y);
26 }
27
28 // Test the implementation
29 int main() {
30     std::cout << "*** PROBLEM 1, testing:" << std::endl;
31
32     std::complex<double> w(1e20,5);
33     std::cout << "The square root of " << w << " is " <<
           myroot(w) << std::endl ;
34     std::cout << "The correct square root of " << w << " is "
           << sqrt(w) <<std::endl <<std::endl;
35
36     w=std::complex<double>(-5,1e20);
37     std::cout << "The square root of " << w << " is " <<
           myroot(w) << std::endl ;
38     std::cout << "The correct square root of " << w << " is "
           << sqrt(w) <<std::endl <<std::endl;
39 }
```

Problem 2 Ellpack sparse matrix format (24 pts.)

The following class (`EllpackMat` in file `problem2.cpp`) implements the Ellpack sparse matrix format for a generic matrix $\mathbf{A} \in \mathbb{R}^{m,n}$:

```
using Triplet = Eigen::Triplet<double>;
using Triplets = std::vector<Triplet>;
using Vector = Eigen::VectorXd;
using index_t = std::size_t;

class EllpackMat {
public:
  EllpackMat(const Triplets & triplets, index_t m, index_t n);

  double operator() (index_t i, index_t j) const;

  void mvmult(const Vector &x, Vector &y) const;
  void mtvmult(const Vector &x, Vector &y) const;
private:
  std::vector<double> val;
  std::vector<index_t> col;

  index_t maxcols; // Max nnz elements per row
  index_t m,n; // Number of rows, number of columns
};
```

Here,

$$\texttt{maxcols} := \max\{\#\{(i,j) : A_{i,j} \neq 0, j = 1,\ldots,n\} : i = 1,\ldots,m\}.$$

The access operator `operator()` is implemented as follows:

```
double EllpackMat::operator()(index_t i, index_t j) const {
    assert(0 <= i && i < m && 0 <= j && j < n
           && "Index out of bounds!");

    for(index_t l = i*maxcols; l < (i+1)*maxcols; ++l) {
        if( col[l] == j ) return val[l];
    }
    return 0.;
```

```
}
```

and is well defined for `i` from `0` to `m-1` and for `j` from `0` to `n-1`.

(2a) Implement an efficient constructor that builds an object of type `EllpackMat` from the data forming a matrix in triplet format, i.e. implement the constructor with signature

```
EllpackMat(const Triplets & triplets, index_t m, index_t n);
```

where `triplets` is a `std::vector` of EIGEN triplets. The arguments `m` and `n` represent the number of rows and of columns of the matrix $\mathbf{A}$, respectively.

The data in the triplets vector must be compatible with the matrix size provided to the constructor. No assumption is made on the ordering of the triplets. Values belonging to multiple occurrences of index pairs are to be summed up. However, in this sub-problem you may assume that duplicate index pairs do not occur in the triplets vector.

HINT: An object of `Eigen::Triplets<double>` type is a structure representing a triplet (r, c, v). It provides the methods `index_t row()`, `index_t col()` and `double value()` which return r, c and v, respectively.

HINT: As first thing, you will have to determine the quantity `maxcols`.

Solution: See implementation in `problem2_solution.cpp`.

(2b) Implement an efficient method

```
void mvmult(const Vector &x, Vector &y) const;
```

that, given an input n-vector `x` and an output m-vector `y` as arguments, returns the matrix-vector product `A*x` in `y`, where `A` is the matrix represented by `*this`. The implementation must deliver optimal complexity of $O(\mathrm{nnz}(\mathbf{A}))$.

Solution: See implementation in `problem2_solution.cpp`.

(2c) Implement an efficient method

```
void mtvmult(const Vector &x, Vector &y) const;
```

that, given an input m-vector `x` and an output n-vector `y` as arguments, returns the matrix-vector product `A'*x` in `y`, where `A'` is the transposed of the matrix represented by `*this`. The implementation must deliver optimal complexity of $O(\mathrm{nnz}(\mathbf{A}))$.

Solution: See implementation in `problem2_solution.cpp`.

HINT: You can test your implementations of (2a), (2b) and (2c) using the code written in `main()`. A 3×6 EIGEN-Sparse matrix $\mathbf{A}$ (`Eigen::SparseMatrix<double> A`) is built from the triplets:

$$\{(1,2,4),(0,0,5),(1,2,6),(2,5,7),(0,4,8),(1,3,9),(2,2,10),(2,1,11),(1,0,12)\}.$$

An equivalent matrix `E` of `Ellpack` type is also built. We then use the vectors $\mathbf{x} = [4,5,6,7,8,9]^\intercal$, $\mathbf{y} = [1,2,3]^\intercal$ and compute the product $\mathbf{Ax}$ and $\mathbf{Ex}$. Finally we output the l^2 norm of the difference. We do the same for $\mathbf{A}^\intercal\mathbf{y}$ and $\mathbf{E}^\intercal\mathbf{y}$.

Solution:

```cpp
#include <iostream>

#include <vector>

#include <Eigen/Dense>
#include <Eigen/Sparse>

using Triplet = Eigen::Triplet<double>;
using Triplets = std::vector<Triplet>;
using Vector = Eigen::VectorXd;
using index_t = std::ptrdiff_t;

//! \brief Class representing a sparse matrix in Ellpack format
class EllpackMat {
public:

    EllpackMat(const Triplets & triplets, index_t m, index_t
       n);

    double operator() (index_t i, index_t j) const;

    void mvmult(const Vector &x, Vector &y) const;

    void mtvmult(const Vector &x, Vector &y) const;
private:
    std::vector<double> val; //!< Vector containing value
       corresponding to entry in col
    std::vector<index_t> col; //! Vector containing columns,
       partitioned into same-size rows

```

```
28      index_t maxcols; // Max elements per row
29      index_t m,n; // Number of rows, number of columns
30  };
31
32  //! \brief Construct  a matrix Ellpack from triplet format
33  //! \param[in] triplets vector of Eigen::Triplets
34  //! \param[in] m rows of matrix
35  //! \param[in] n columns of matrix
36  //! \param[in] maxcols max. number of nonzeros of the matrix
37  EllpackMat::EllpackMat(const Triplets & triplets, index_t m,
        index_t n)
38      : m(m), n(n) {
39      // TODO: problem2a: implement constructor building matrix
            from Eigen Triplet format
40
41      // Find maxcols, counters denotes number of non-empty
            columns for each row (assuming all triplets uniquely
42      // define one entry)
43      std::vector<unsigned int> counters(m);
44      // Fill counter with 0
45      std::fill(counters.begin(), counters.end(), 0);
46      // maxcols starts with no entry for each row
47      maxcols = 0;
48      // Loop over each triplet and increment corresponding
            counter, update maximum if needed
49      for(const Triplet& tr: triplets) {
50          if( ++counters[tr.row()] > maxcols) maxcols =
                counters[tr.row()];
51      }
52      std::cout << "Maxcols: " << maxcols << std::endl;
53
54      // Reserve space
55      col.resize(m*maxcols,-1);
56      val.resize(m*maxcols,0);
57
58      // Loop over each triplet, then find a column where to put
            the value
59      for(const Triplet& tr: triplets) {
60          assert(0 <= tr.row() && tr.row() < m && 0 <= tr.col()
                && tr.col() < n &&
61                  "Index out of bounds!");
```

```
62
63          index_t l;
64          for(l = tr.row()*maxcols; l < (tr.row()+1)*maxcols;
                ++l) {
65              // If value must be added to existing one
66              if( col[l] == tr.col() ) {
67                  val[l] += tr.value();
68                  break;
69              // If value is new (find the next empty place)
70              } else if( col[l] == -1 ) {
71                  col[l] = tr.col();
72                  val[l] = tr.value();
73                  break;
74              }
75          }
76          assert(l < (tr.row()+1)*maxcols &&
77                 "You did not reserve enough columns!");
78      }
79  }
80
81  //! \brief Retrieve value of entry at index (i,j)
82  //! \param[in] i row index i
83  //! \param[in] j column index j
84  //! \return value at (i,j)
85  double EllpackMat::operator() (index_t i, index_t j) const {
86      assert(0 <= i && i < m && 0 <= j && j < n && "Index out of
            bounds!");
87
88      for(index_t l = i*maxcols; l < (i+1)*maxcols; ++l) {
89          if( col[l] == j ) return val[l];
90      }
91      return 0.;
92  }
93
94  //! \brief Computes (*this)*x
95  //! \param[in] x vector x for mat-vec mult
96  //! \param[out] y result y = (*this)*x
97  void EllpackMat::mvmult(const Vector &x, Vector &y) const {
98      assert(x.size() == n && "Incompatible vector x size!");
99      assert(y.size() == m && "Incompatible vector y size!");
100
```

```
101     // TODO: problem 2b: implement operation y = this*x
102
103     for(index_t i = 0; i < m; ++i) {
104         y(i) = 0;
105         for(index_t l = i*maxcols; l < (i+1)*maxcols; ++l) {
106             if( col[l] == -1 ) break;
107             y(i) += x(col[l]) * val[l];
108         }
109     }
110 }
111
112 //! \brief Computes (*this)'*x, A' is transposed
113 //! \param[in] x vector x for mat-vec mult
114 //! \param[out] y result y = (*this)'*x
115 void EllpackMat::mtvmult(const Vector &x, Vector &y) const {
116     assert(x.size() == m && "Incompatible vector x size!");
117     assert(y.size() == n && "Incompatible vector y size!");
118
119     // TODO: problem 2c: implement operation y = this^T*x
120
121     y = Vector::Zero(n);
122     for(index_t i = 0; i < m; ++i) {
123         for(index_t l = i*maxcols; l < (i+1)*maxcols; ++l) {
124             if( col[l] == -1 ) break;
125             y(col[l]) += x(i) * val[l];
126         }
127     }
128 }
129
130 int main(int, char**) {
131     // Vector of triplets
132     Triplets triplets;
133
134     // Data
135     unsigned int m = 3, n = 6;
136     unsigned int ntriplets = 9;
137
138     // Reserve space for triplets
139     triplets.reserve(ntriplets);
140
141     // Fill in some triplet
```

```
142     triplets.push_back(Triplet(1,2,4));
143     triplets.push_back(Triplet(0,0,5));
144     triplets.push_back(Triplet(1,2,6));
145     triplets.push_back(Triplet(2,5,7));
146     triplets.push_back(Triplet(0,4,8));
147     triplets.push_back(Triplet(1,3,9));
148     triplets.push_back(Triplet(2,2,10));
149     triplets.push_back(Triplet(2,1,11));
150     triplets.push_back(Triplet(1,0,12));
151
152     // Build matrix (Eigen Sparse)
153     Eigen::SparseMatrix<double> S(m,n);
154     S.setFromTriplets(triplets.begin(), triplets.end());
155
156     // Build Ellpack matrix
157     EllpackMat E(triplets, m, n);
158
159     //// PROBLEM 2 TEST
160     std::cout << "*** PROBLEM 2, testing:" << std::endl;
161     std::cout << " ------------- Test of y = A^t*x
            ------------- " << std::endl;
162     Eigen::VectorXd x(6);
163     x << 4,5,6,7,8,9;
164     Eigen::VectorXd Ex = Eigen::VectorXd::Zero(m);
165     E.mvmult(x, Ex);
166     std::cout << "Sparse S*x =" << std::endl << S*x <<
            std::endl;
167     std::cout << "Ellpack E*x =" << std::endl << Ex <<
            std::endl;
168     std::cout << " ------------- Test of x = A*y -------------
            " << std::endl;
169     Eigen::VectorXd y(3);
170     y << 1,2,3;
171     Eigen::VectorXd Ey = Eigen::VectorXd::Zero(n);
172     E.mtvmult(y, Ey);
173     std::cout << "Sparse S^T*y =" << std::endl <<
            S.transpose()*y  << std::endl;
174     std::cout << "Ellpack E^T*y =" << std::endl << Ey <<
            std::endl;
175 }
```

Problem 3 Piecewise cubic Hermite interpolation (24 pts.)

We are concerned with *piecewise cubic Hermite interpolation* of given data points $(t_j, y_j) \in \mathbb{R}^2, j = 0, \ldots, n$, $n \in \mathbb{N}^*$. Assume that the nodes are ordered, namely $t_j < t_{j+1}$ for every $j = 0, \ldots, n-1$. The slopes $c_j = s'(t_j)$, $j = 0, \ldots, n$ of the interpolant s at the points t_j are computed from the data according to the formula:

$$c_j := \begin{cases} \Delta_1 & \text{for } j = 0, \\ \operatorname{minmod}(\Delta_j, \Delta_{j+1}) & \text{for } j = 1, \ldots, n-1, \\ \Delta_n & \text{for } j = n, \end{cases} \tag{3}$$

where $\Delta_j := \dfrac{y_j - y_{j-1}}{t_j - t_{j-1}}$ and

$$\operatorname{minmod}(x, y) := \begin{cases} 0 & \text{if } \operatorname{sgn}(x) \neq \operatorname{sgn}(y), \\ \min(|x|, |y|) \cdot \operatorname{sgn}(x) & \text{otherwise}, \end{cases} \tag{4}$$

with the sign function given by

$$\operatorname{sgn}(x) := \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{5}$$

(3a) Implement a C++ function:

```
double minmod(double x, double y);
```

for the computation of the function minmod as defined in (4).

Solution: See implementation in `problem3_solution.cpp`.

(3b) Implement a C++ function

```
using Vector = Eigen::VectorXd;
Vector delta(const Vector & t, const Vector & y);
```

for the computation of the quantities Δ_j, $j = 1, \ldots, n$, from the data (t_j, y_j). The arguments `t` and `y` pass the t_j and y_j, $j = 0, \ldots, n$. The return vector contains the computed values $\Delta_1, \ldots, \Delta_n$.

Solution: See implementation in `problem3_solution.cpp`.

(3c) The template file `problem3.cpp` contains the declaration of a class `mmPCHI`, reproduced here:

```
//! \brief Class representing a piecewise cubic Hermite
   interpolant s with minmod-reconstructed slopes
class mmPCHI {
public:
    //! \brief Construct the slopes c_j of the interpolant, from
       the data (t_j, y_j).
    //! \param[in] t Vector of nodes t_j, j = 0,...,n
    //! \param[in] y Vector of values y_j, j = 0,...,n
    mmPCHI(const Vector &t, const Vector &y);

    //! \brief Evaluate *this interpolant at the points
       defined by x and store the result in v
    //! \param[in] x Vector of points x_i, i = 0,...,m
    //! \return v Vector of values s(x_i), i = 0,...,m
    Vector eval(const Vector &x) const;

private:
    const Vector t, y;//!< Vectors containing nodes and values
    Vector c; //!< Vector containing slopes for the interpolant
};
```

This class implements a piecewise cubic Hermite interpolant using the `minmod` slope reconstruction (3). Some class member functions still have to be implemented. Implement an efficient constructor

```
mmPCHI::mmPCHI(const Vector &t, const Vector &y);
```

The function `mmPCHI` is the constructor of the class, in which you should compute the slopes c_j according to (3) and store the value in the corresponding class variable.

Solution: See implementation in `problem3_solution.cpp`.

(3d) Implement an efficient method:

```
Vector mmPCHI::eval(const Vector &x) const;
```

The function `eval` evaluates the interpolant defined by the class at the points in `x` and returns the values in a vector of the same length as `x`. No assumption is made on the ordering of `x`.

HINT: An efficient implementation may sort `x` first. To that end, you can use `std::sort`, in combination with the `data()` and `size()` method of the class `Vector`.

Solution: The implementation computes the value of the interpolant using local Hermite polynomial evaluation. See implementation in `problem3_solution.cpp`.

HINT: You can test your implementation using the code written in the `main()` function of `problem3.cpp`. The `main()` function will perform a convergence study applied to the function

$$f(t) := \frac{1}{1+t^2}, \qquad t \in [-5, 5]. \tag{6}$$

(3e) We use the piecewise cubic Hermite interpolation introduced above for function approximation. Consider interpolation error for the functions

$$f_1(t) := \frac{1}{1+t^2}, \qquad t \in [-5, 5], \tag{7}$$

$$f_2(t) := \frac{1}{1+|t|}, \qquad t \in [-5, 5], \tag{8}$$

and sequences of n equidistant interpolation nodes ($-5 = t_0 < \ldots < t_{n-1} = 5$) for $n = 2^2, 2^3, \ldots, 2^{11}$. In the following tables, we measure the interpolation error in the L^∞-norm on $[-5, 5]$, which is approximated by sampling on a grid of $M = 100,000$ nodes on $[-5, 5]$. One table represents the error for f_1, the other for f_2.

Assign each table to the corresponding function and explain the reasons for your choice. Determine the empirical rate of convergence for each table.

Table A		Table B	
n	error	n	error
4	0.735294	4	0.6245
8	0.337838	8	0.416167
16	0.0999997	16	0.2495
32	0.0253547	32	0.138389
64	0.00625914	64	0.0730296
128	0.00154735	128	0.037379
256	0.00038407	256	0.018731
512	9.54818e-05	512	0.00919012
1024	2.36379e-05	1024	0.00436401
2048	5.7162e-06	2048	0.00193685

Solution: The experimental rate of convergence is polynomial and limited to second-order ($O(h^2)$) in the case of a regular function (i.e. f_1) (table A), and limited to first-order ($O(h)$) in case of irregular function (i.e. f_2) (table B). See the implementation in `problem3_solution.cpp`.

```cpp
#include <iostream>

#include <vector>

#include <Eigen/Dense>

using Vector = Eigen::VectorXd;
using Matrix = Eigen::MatrixXd;

//! \brief Function for the evaluation of minmod(x,y)
double minmod(double x, double y) {
    //// TODO: problem 3a, implement this function

    return x*y <= 0. ? 0. : (x >= 0. ? std::min(x,y) :
       std::max(x,y));
}

//! \brief Function for the computation of the values Δ_j
//! \param[in] t Vector of nodes t_j, j = 0,...,n
//! \param[in] y Vector of values y_j, j = 0,...,n
//! \return Vector (Δ_1,...,Δ_n) relative to (t, y) data
Vector delta(const Vector & t, const Vector & y) {
    assert(t.size() == y.size());
    assert(t.size() > 1);

    //// TODO: problem 3b, implement this function

    Vector dlt(t.size()-1);

    for(unsigned int j = 0; j < t.size()-1; ++j) {
        dlt(j) = (y(j+1) - y(j)) / (t(j+1) - t(j));
    }

    return dlt;
}
```

```
35
36 //! \brief Class representing a pecevise cubic Hermite
      interpolant s with minmod-reconstructed slopes
37 class mmPCHI {
38 public:
39     //! \brief Construct the slopes c_j of the interpolant, from
          the data (t_j,y_j).
40     //! \param[in] t Vector of nodes t_j, j = 0,...,n
41     //! \param[in] y Vector of values y_j, j = 0,...,n
42     mmPCHI(const Vector &t, const Vector &y);
43
44     //! \brief Evaluate interpolant defined by (*this) at the
          points defined by x and store the result in v
45     //! \param[in] x Vector of points x_i, i = 0,...,m
46     //! \return v Vector of values s(x_i), i = 0,...,m
47     Vector eval(const Vector &x) const;
48 private:
49     const Vector t, y;//!< Vectors containing nodes and values
50     Vector c; //!< Vector containing slopes for the interpolant
51 };
52
53 mmPCHI::mmPCHI(const Vector &t, const Vector &y)
54 : t(t), y(y), c(t.size()) {
55
56     assert(t.size() == y.size());
57     assert(t.size() > 1);
58
59     //// TODO: problem 3c, implement this function
60
61     unsigned int n = t.size()-1;
62
63     Vector D = delta(t,y);
64
65     // Reconstruct slopes using minmod
66     c(0) = D(0);
67     for(unsigned int i = 1; i < n; ++i) {
68
69         c(i) = minmod(D(i), D(i-1));
70     }
71     c(n) = D(n-1);
72 }
```

```
73
74  Vector mmPCHI::eval( const Vector &x) const {
75      Vector v(x.size());
76
77      //// TODO: problem 3d, implement this function
78      Vector xcopy = x;
79      std::sort(xcopy.data(), xcopy.data()+xcopy.size());
80
81      unsigned int i = 0;
82      for(unsigned int j = 0; j < t.size()-1; ++j) {
83          double t1 = t(j);
84          double t2 = t(j+1);
85          double y1 = y(j);
86
87          double h = t2 - t1;
88          double a1 = y(j+1) - y1;
89          double a2 = a1 - h*c(j);
90          double a3 = h*c(j+1) - a1 - a2;
91          while( i < xcopy.size() && xcopy(i) <= t2 ) {
92              double tx = (xcopy(i) - t1) / h;
93              v(i) = y1 + (a1+(a2+a3*tx)*(tx-1.))*tx;
94              i++;
95          }
96      }
97
98      return v;
99  }
100
101 int main(int, char**) {
102     //// PROBLEM 3 TEST
103
104     auto f = [] (double t) -> double { return 1. / (1. + t*t);
        };
105     //auto f = [] (double t) -> double { return 1. / (1. +
        std::abs(t)); };
106
107     // Interval
108     const double a = -5., b = 5;
109     // Number of sampling nodes
110     const unsigned int M = 10000;
111
```

```
    // Sampling nodes and values
    const Vector x = Vector::LinSpaced(M,a,b);
    Vector v_ex(M);
    // Fill in exact values of f
    for(unsigned int i = 0; i<M; ++i) {
        v_ex(i) = f(x(i));
    }
    // Output table with error w.r.t number of nodes
    std::cout << "n" << "\t" << "L^inf err." << std::endl;
    for(unsigned int n = 4; n <= 2048; n=n<<1) {
        Vector t = Vector::LinSpaced(n,a,b);
        Vector y(n);
        for(unsigned int i = 0; i<n; ++i) {
            y(i) = f(t(i));
        }

        mmPCHI s(t,y);
        const Vector v = s.eval(x);
        std::cout << n << "\t\t" << (v -
            v_ex).lpNorm<Eigen::Infinity>() << std::endl;
    }

}
```

Problem 4 Quadrature by transformation (16 pts.)

For $f \in C([0,2])$ define

$$I(f) := \int_0^2 \sqrt{\frac{2-t}{t}} f(t)\, dt.$$

(4a) For $n \in \mathbb{N}^*$, derive a family of $2n$-point quadrature formulas

$$Q_n(f) = \sum_{j=1}^{2n} w_j^n f(c_j^n), \qquad f \in C([0,2])$$

for which $Q_n(f)$ converges to $I(f)$ exponentially as $n \to \infty$, if f has an analytic extension to a complex neighborhood of $[0,2]$.

Solution: We wish to transform the integral in $I(f)$ into an integral of a smooth function

over a bounded interval. In order to this, write

$$I(f) = \int_0^1 \frac{\sqrt{2-t}}{\sqrt{t}} f(t)\,dt + \int_1^2 \frac{\sqrt{2-t}}{\sqrt{t}} f(t)\,dt.$$

Performing the substitutions $s = \sqrt{t}$ in the first integral and $s = \sqrt{2-t}$ in the second integral we obtain

$$I(f) = 2\int_0^1 \sqrt{2-s^2} f(s^2)\,ds + 2\int_0^1 \frac{s^2}{\sqrt{2-s^2}} f(2-s^2)\,ds = 2\int_0^1 f_1(s)\,ds + 2\int_0^1 f_2(s)\,ds$$

where $f_1(s) = \sqrt{2-s^2} f(s^2)$ and $f_2(s) = \frac{s^2}{\sqrt{2-s^2}} f(2-s^2)$. With the change of variables $t = 2s-1$ we finally obtain

$$I(f) = \int_{-1}^1 f_1((t+1)/2)\,ds + \int_{-1}^1 f_2((t+1)/2)\,ds. \tag{9}$$

Applying Gauss quadrature to these integrals, we define

$$Q_n(f) = \sum_{j=1}^n \tilde{w}_j^n f_1((\xi_j^n+1)/2) + \sum_{j=1}^n \tilde{w}_j^n f_2((\xi_j^n+1)/2),$$

where ξ_j^n and $\tilde{w}_j^n$ are nodes and weights of the Gauss quadrature of order n defined on $[-1,1]$. (Alternatively, it is also possible to use the Clenshaw-Curtis quadrature rule.) By construction, we have that $Q_n(f) \to I(f)$ exponentially as $n \to \infty$ if f has an analytic extension to a neighborhood of $[1,2]$, since this implies that the integrands in (9) have an analytic extension to a neighborhood of $[-1,1]$.

Define now the nodes

$$c_j^n = \begin{cases} ((\xi_j^n+1)/2)^2 & \text{if } j = 1,\dots,n, \\ 2 - ((\xi_{j-n}^n+1)/2)^2 & \text{if } j = n+1,\dots,2n. \end{cases}$$

Therefore, the above expression can be written as

$$Q_n(f) = \sum_{j=1}^n \tilde{w}_j^n \sqrt{2-c_j^n} f(c_j^n) + \sum_{j=1}^n \tilde{w}_j^n \frac{2-c_{j+n}^n}{\sqrt{c_{j+n}^n}} f(c_{j+n}^n) = \sum_{j=1}^{2n} w_j^n f(c_j^n),$$

where the weights are given by

$$w_j^n = \begin{cases} \tilde{w}_j^n \sqrt{2-c_j^n} & \text{if } j = 1,\dots,n, \\ \tilde{w}_{j-n}^n \frac{2-c_j^n}{\sqrt{c_j^n}} & \text{if } j = n+1,\dots,2n. \end{cases}$$

(4b) Given $n > 0$, determine the maximal $d \in \mathbb{N}$ such that $I(p) = Q_n(p)$ for every polynomial $p \in \mathcal{P}_{d-1}$.

DEPENDS ON: subproblem (4a).

Solution: The above expression is never exact for polynomials f, because of the presence of the square roots in (9). In other words, $d = 0$, for which $\mathcal{P}_{-1} = \varnothing$.

Problem 5 Mono-implicit Runge-Kutta single step method (28 pts.)

A so-called s-stage mono-implicit Runge-Kutta single step method (MIRK) for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subset \mathbb{R}^d \to \mathbb{R}^d$, is defined as:

$$\begin{aligned} \mathbf{g}_i &:= (1 - v_i)\mathbf{y}_0 + v_i\mathbf{y}_1 + h\sum_{j=1}^{i-1} d_{i,j}\mathbf{f}(\mathbf{g}_j), \quad i = 1, \dots s\,, \\ \mathbf{y}_1 &:= \mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{f}(\mathbf{g}_j) \end{aligned} \tag{10}$$

for suitable coefficients $v_i, b_i, d_{i,j} \in \mathbb{R}$.

(5a) Single step methods defined by (10) belong to the class of implicit Runge-Kutta methods. Write down the corresponding Butcher scheme in terms of the coefficients $v_i, b_i, d_{i,j}$.

Solution: Let us define $d_{i,j} := 0$ for $j \geq i$. Replacing $\mathbf{y}_1$ in the equation for $\mathbf{g}_i$, one obtains:

$$\begin{aligned} \mathbf{g}_i &:= (1 - v_i)\mathbf{y}_0 + v_i\left(\mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{f}(\mathbf{g}_j)\right) + h\sum_{j=1}^{i-1} d_{i,j}\mathbf{f}(\mathbf{g}_j) = \mathbf{y}_0 + h\sum_{j=1}^{s}(v_i b_j + d_{i,j})\mathbf{f}(\mathbf{g}_j), \\ \mathbf{y}_1 &:= \mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{f}(\mathbf{g}_j) \end{aligned} \tag{11}$$

Set $\mathbf{k}_i := \mathbf{f}(\mathbf{g}_i)$. Then, $\mathbf{k}_i = f(\mathbf{y}_0 + h\sum_{j=1}^{s} \overbrace{(v_i b_j + d_{i,j})}^{=:a_{i,j}}\mathbf{k}_j)$ and

$$\mathbf{y}_1 := \mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{k}_j \tag{12}$$

Therefore, the Butcher scheme becomes:

$$\begin{array}{c|c} \mathbf{c} & \mathbf{D} + \mathbf{v}\mathbf{b}^\top \\ \hline & \mathbf{b}^\top, \end{array}$$

where $(\mathbf{D})_{i,j} = d_{i,j}$, $\mathbf{b} = [b_1, \dots, b_s]^\top$, $\mathbf{v} = [v_1, \dots, v_s]^\top$ and $c_i = \sum_{j=1}^{s} a_{i,j}$.

(5b) Compute the stability function of a MIRK scheme defined as in (10) for $s = 2$.

Solution: We apply the MIRK scheme to the ODE $y' = \lambda y$. Denote by $\mathbf{g} := [g_1, g_2]^\top$ and $z := h\lambda$, then:

$$\mathbf{g} = (\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1 + z\mathbf{D}\mathbf{g}$$
$$(\mathbf{I} - z\mathbf{D})\mathbf{g} = (\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1$$

where $\mathbf{1} = [1, \dots, 1]^\top$. Consider

$$y_1 = y_0 + z\mathbf{b}^\top\mathbf{g} = y_0 + z\mathbf{b}^\top(\mathbf{I} - z\mathbf{D})^{-1}((\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1).$$

Then

$$(1 - z\mathbf{b}^\top(\mathbf{I} - z\mathbf{D})^{-1}\mathbf{v})y_1 = (1 + z\mathbf{b}^\top(\mathbf{I} - z\mathbf{D})^{-1}(\mathbf{1} - \mathbf{v}))y_0,$$

whence

$$S(z) = \frac{1 + z\mathbf{b}^\top(\mathbf{I} - z\mathbf{D})^{-1}(\mathbf{1} - \mathbf{v})}{1 - z\mathbf{b}^\top(\mathbf{I} - z\mathbf{D})^{-1}\mathbf{v}}.$$

Alternative: use formula from Thm, 12.3.27 from the course together with the Butcher scheme derived in the previous sub-problem.

(5c) Now, we consider the special case of a scalar ODE ($d = 1$) and $s = 2$. Abbreviating $\mathbf{z} := [g_1, g_2, y_1]^\top$, rewrite (10) as a non-linear system of equations in the form $\mathbf{F}(\mathbf{z}) = \mathbf{0}$ for an explicitly specified suitable function $\mathbf{F} : \mathbb{R}^3 \to \mathbb{R}^3$.

Solution:

$$\mathbf{F}\left(\begin{bmatrix} g_1 \\ g_2 \\ y_1 \end{bmatrix}\right) = \begin{bmatrix} g_1 - (1 - v_1)y_0 - v_1 y_1 \\ g_2 - (1 - v_2)y_0 - v_2 y_1 - h d_{2,1} f(g_1) \\ y_1 - y_0 - h\sum_{j=1}^{2} b_j f(g_j) \end{bmatrix}$$

(5d) Find the Jacobian $D\mathbf{F}(\mathbf{z})$ of the function $\mathbf{F}$ (in terms of $d_{i,j}, v_i, b_i$ and the derivative of f) from the previous sub-problem.

DEPENDS ON: subproblem (5c).

Solution:

$$\mathbf{DF}\left(\begin{bmatrix} g_1 \\ g_2 \\ y_1 \end{bmatrix}\right) = \mathbf{I}_3 - \begin{bmatrix} 0 & 0 & v_1 \\ h d_{2,1} f'(g_1) & 0 & v_2 \\ h b_1 f'(g_1) & h b_2 f'(g_2) & 0 \end{bmatrix}.$$

In the next steps, we will implement the MIRK scheme for $d = 1, s = 2$.

(5e) Implement a function

```
using Vector = Eigen::VectorXd;
template <class Func, class Jac>
void newton2steps(const Func & F, const Jac & DF,
                  Vector & z);
```

that approximates the solution of $\mathbf{F}(\mathbf{z}) = \mathbf{0}$ by performing two steps of the Newton method applied to $\mathbf{F}(\mathbf{z}) = \mathbf{0}$. Use `z` to pass the initial guess and to return the approximated solution. Objects of type `Func` and `Jac` have to supply suitable evaluation operators `operator()`.

Solution: See implementation in `problem5_solution.cpp`.

(5f) Consider the particular MIRK scheme given by the coefficients:

$$v_1 = 1, \quad v_2 = \frac{344}{2025}, \quad d_{21} = -\frac{164}{2025}, \quad b_1 = \frac{37}{82}, \quad b_2 = \frac{45}{82}. \tag{13}$$

Using the function `newton2steps`, implement a function

```
template <class Func, class Jac>
double MIRKstep(const Func & f, const Jac & df,
                double y0, double h);
```

that realizes one step of the MIRK scheme defined by (10) for $s = 2$ and a scalar ODE, that is, $d = 1$. The right hand side function $\mathbf{f}$ and its Jacobian are passed through `f` and `df`. The solution of the nonlinear system arising from (10) is approximated using two Newton steps, namely by using the function `newton2steps`. The initial guess has to be chosen appropriately!

DEPENDS ON: subproblems (5c) and (5d).

Solution: See implementation in `problem5_solution.cpp`.

(5g) Implement a function

```
template <class Func, class Jac>
double MIRKsolve(const Func & f, const Jac & df,
                 double y0, double T, unsigned int N);
```

for the solution of a scalar ODE up to time T, using N equidistant steps of the mono-implicit Runge-Kutta single step method defined by (13). The initial value is passed in `y0`.

Solution: See implementation in `problem5_solution.cpp`.

(5h) Apply your implementation to the IVP

$$\dot{y} = 1 + y^2, y(0) = 0 \tag{14}$$

on $[0, 1]$. The file `problem5.cpp` contains a partial template for this sub-problem. The exact solution of (14) is $y_{ex}(t) := \tan t$. Compute the solution y_n at $T = 1$ with a sequence of uniform temporal meshes with $n = 4, \ldots, 512$ intervals. Compute and output the error $|y_n(1) - y_{ex}(1)|$ and determine the rate of convergence of the scheme.

DEPENDS ON: subproblems (5c), (5d), (5e), (5f) and (5g).

Solution: The experimental rate of convergence of the methods is $O(h^2)$. See implementation in `problem5_solution.cpp`.

Solution:

```cpp
#include <iostream>

#include <vector>

#include <Eigen/Dense>

using Vector = Eigen::VectorXd;
using Matrix = Eigen::MatrixXd;

//! \brief Perform 2 steps of newton method applied to F and its jacobian DF
//! \tparam Func type for function F
//! \tparam Jac type for jacobian DF of F
//! \param[in] F function F, for which F(z) = 0 is needed
//! \param[in] DF Jacobian DF of the function F
//! \param[in,out] z initial guess and final approximation for F(z) = 0
template <class Func, class Jac>
void newton2steps(const Func & F, const Jac & DF, Vector & z) {
    // TODO: problem 5e: two newton steps

    Vector znew = z - DF(z).lu().solve(F(z));
    z = znew - DF(znew).lu().solve(F(znew));
}

```

```cpp
//! \brief Perform a single step of the MIRK scheme applied to the scalar ODE y' = f(y)
//! \tparam Func type for function f
//! \tparam Jac type for jacobian df of f
//! \param[in] f function f, as in y' = f(y)
//! \param[in] df Jacobian df of the function f
//! \param[in] y0 previous value
//! \param[in] h step-size
//! \return value y1 at next step
template <class Func, class Jac>
double MIRKstep(const Func & f, const Jac & df, double y0, double h) {
    const double v1  = 1;
    const double v2  = 344./2025.;
    const double d21 = -164./2025.;
    const double b1  = 37./82.;
    const double b2  = 45./82.;

    // TODO: problem 5f: implement MIRK step

    auto F = [&f, &y0, &v1, &v2, &d21, &b1, &b2, &h] (const Vector & z) -> Vector {
        Vector ret(3);
        ret << z(0) - (1-v1)*y0 - v1*z(2),
               z(1) - (1-v2)*y0 - v2*z(2) - h*d21*f(z(0)),
               z(2) - y0 - h*(b1*f(z(0))+b2*f(z(1)));
        return ret;
    };
    auto DF = [&df, &v1, &v2, &d21, &b1, &b2, &h] (const Vector & z) -> Matrix {
        Matrix M(3,3);
        M << 0,0, v1,
             h*d21*df(z(0)), 0,  v2,
             h*b1*df(z(0)),h*b2*df(z(1)),0;
        return Matrix::Identity(3,3) - M;
    };
    Vector z(3);
    z << 0,0,y0; // FIXME: initial data
    newton2steps(F, DF, z);

    return z(2);
```

```
61 }
62
63 //! \brief Solve an ODE y' = f(y) using MIRK scheme on
       equidistant steps
64 //! \tparam Func type for function f
65 //! \tparam Jac type for jacobian df of f
66 //! \param[in] f function f, as in y' = f(y)
67 //! \param[in] df Jacobian df of the function f
68 //! \param[in] y0 initial value
69 //! \param[in] T final time
70 //! \param[in] N number of steps
71 //! \return value approximating y(T)
72 template <class Func, class Jac>
73 double MIRKsolve(const Func & f, const Jac & df, double y0,
       double T, unsigned int N) {
74     // TODO: problem 5g: implement MIRK solver
75
76     const double h = T / N;
77     double ynext = y0;
78     for(unsigned int i = 0; i < N; ++i) {
79         ynext = MIRKstep(f, df, ynext, h);
80     }
81     return ynext;
82 }
83
84 int main(int, char**) {
85
86     auto f = [] (double y) -> double { return 1 + y*y; };
87     auto df = [] (double y) -> double { return 2*y; };
88
89     const double y0 = 0.;
90     const double T = 1.;
91
92     const double yex = tan(1);
93
94     //// PROBLEM 5h TEST
95     std::cout << "*** PROBLEM 5h:" << std::endl;
96     // TODO: problem 5h: solve IVP y' = f(y) up to T
97
98     std::cout << "N" << "\t" << "yend" << "\t" << "err" <<
         std::endl;
```

```
99      for(unsigned int N = 4; N < 512; N=N<<1) {
100         double yend = MIRKsolve(f, df, y0, T, N);
101         double err = std::abs(yex - yend);
102         std::cout << N << "\t" << yend << "\t" << err <<
                std::endl;
103     }
104 }
```