First name		Grade
Last name		
Department		
Computer Name		
Legi No.		
Date	12.08.2016	

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

August 12, 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`".

In order to compile and run the C++ code related to a single problem, e.g. Problem 3, type "`make problem3`". Execute the program using "`./problem3`".

If you want to manually compile your code, use:

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

or

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

The flags `-Wno-deprecated-declarations -Wno-ignored-attributes` suppress some unwanted 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. All templates should compile even if left unchanged.

Problem 1 Symmetric Gauss-Seidel iteration (20 pts.)

For a square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, define $\mathbf{D_A}, \mathbf{L_A}, \mathbf{U_A} \in \mathbb{R}^{n,n}$ by:

$$(\mathbf{D_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i = j, \\ 0, & i \neq j \end{cases}, (\mathbf{L_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i > j, \\ 0, & i \leq j \end{cases}, (\mathbf{U_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i < j, \\ 0, & i \geq j \end{cases}. \tag{1}$$

The symmetric Gauss-Seidel iteration associated with the linear system of equations $\mathbf{Ax} = \mathbf{b}$ is defined as

$$\mathbf{x}^{(k+1)} = (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{b} - (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}^{(k)}), \tag{2}$$

where $\mathbf{x}^{(k)}, \mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ ($k \geq 0$) and $x^{(0)}$ is a given initial guess.

(1a) Give a necessary and sufficient condition on $\mathbf{A}$ such that the iteration (2) is well-defined.

Solution: The matrices $\mathbf{U_A} + \mathbf{D_A}$ and $\mathbf{L_A} + \mathbf{D_A}$ must be invertible, which is equivalent to $\mathbf{D_A}$ being invertible. This is equivalent to the matrix $\mathbf{A}$ having no zeros on the diagonal.

(1b) Assume that (2) is well-defined. Show that a fixed point $\mathbf{x} \in \mathbb{R}^n$ of the iteration (2) is a solution of the linear system of equations $\mathbf{Ax} = \mathbf{b}$.

Solution: Let $\mathbf{x}$ be such fixed point, then

$$\mathbf{x} = (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{b} - (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) \tag{3}$$

Left-multiply by $\mathbf{U_A} + \mathbf{D_A} = \mathbf{A} - \mathbf{L_A}$ (invertible!):

$$\begin{aligned} (\mathbf{U_A} + \mathbf{D_A})\mathbf{x} &= \mathbf{b} - \mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) && (4) \\ &= \mathbf{b} - (\mathbf{L_A} + \mathbf{D_A} - \mathbf{D_A})(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) && (5) \\ &= \mathbf{U_A}\mathbf{x} + \mathbf{D_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) && (6) \end{aligned}$$

Hence,

$$\begin{aligned} \mathbf{D_A}\mathbf{x} &= \mathbf{D_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) && (7) \\ \mathbf{x} &= (\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) && (8) \\ (\mathbf{L_A} + \mathbf{D_A})\mathbf{x} &= \mathbf{b} - \mathbf{U_A}\mathbf{x}, && (9) \end{aligned}$$

which is equivalent to $\mathbf{Ax} = \mathbf{b}$, since $\mathbf{L_A} + \mathbf{D_A} + \mathbf{U_A} = \mathbf{A}$ by construction.

(1c) Implement a C++ function (in `problem1.cpp`)

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

void GSIt(const Matrix & A, const Vector & b, Vector & x,
   double rtol);
```

solving the linear system $\mathbf{Ax} = \mathbf{b}$ using the iterative scheme (2). To that end, apply the iterative scheme to an initial guess $\mathbf{x}^{(0)}$ provided in `x`. The approximated solution given by the final iterate is then stored in `x` as an output.

Use a correction based termination criterion with relative tolerance `rtol` using the Euclidean vector norm.

Solution: See implementation in `problem1_solution.cpp`.

(1d) Test your implementation (use `n = 9` and `rtol = 10e-8`) with the linear system given by

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 0 & \dots & 0 \\ 2 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & 2 & 3 \end{bmatrix} \in \mathbb{R}^{n,n}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^{n}. \tag{10}$$

Output the l^2-norm of the *residual* of the approximated solution. Use $\mathbf{b}$ as your initial guess.

Solution: See implementation in `problem1_solution.cpp`.

(1e) Using the same matrix $\mathbf{A}$ and the same r.h.s. vector $\mathbf{b}$ as above (1d), in Table 1 we have tabulated the quantity $\|\mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\|_2$ for $k = 1, \dots, 20$.

Describe qualitatively and quantitatively the convergence of the iterative scheme with respect to the number of iterations k.

Solution: Asymptotic linear convergence can be expected after an initial phase where the error decreases much faster. The quotient (rate of convergence) $|\mathbf{x}^{(k)} - \mathbf{x}^*|/|\mathbf{x}^{(k-1)} - \mathbf{x}^*|$ approaches 0.679048.

```
#include <iostream>

#include <Eigen/Dense>
```

k	$\|\mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\|_2$	k	$\|\mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\|_2$
1	0.172631	11	0.00341563
2	0.0623049	12	0.00234255
3	0.0464605	13	0.00160173
4	0.0360226	14	0.00109288
5	0.0272568	15	0.000744595
6	0.0200715	16	0.000506784
7	0.0144525	17	0.000344682
8	0.0102313	18	0.000234316
9	0.00715417	19	0.000159234
10	0.00495865	20	0.000108185

Table 1: Euclidean norms of error vectors for 20 iterates

```
4
5  using Matrix = Eigen::MatrixXd;
6  using Vector = Eigen::VectorXd;
7
8  //! \brief Use symmetric Gauss-Seidel iterations to
       approximate the solution of the system Ax = b
9  //! \param[in] A system matrix to be decomposed, must be
       invertible (L + D + U)
10 //! \param[in] b r.h.s. vector
11 //! \param[in,out] x initial guess as input and last value of
       iteration as output
12 //! \param[in] rtol relative tolerance for termination criteria
13 void GSIt(const Matrix & A, const Vector & b, Vector & x,
       double rtol) {
14
15     // Extract strictly triangular part
16     auto U = Eigen::TriangularView<Matrix,
           Eigen::StrictlyUpper>(A);
17     auto L = Eigen::TriangularView<Matrix,
           Eigen::StrictlyLower>(A);
18
19     // Extract non-strictly triangular parts (we only require
           L+D and U+D)
20     auto UpD = Eigen::TriangularView<Matrix, Eigen::Upper>(A);
21     auto LpD = Eigen::TriangularView<Matrix, Eigen::Lower>(A);
22
```

```
23      // Temporary storage
24      Vector temp(x.size());
25      Vector* xold = &x;
26      Vector* xnew = &temp;
27
28      // Counter for iterations
29      unsigned int k = 0;
30      // Error of this and previous iteration
31      double err;
32  //     double errold = 0.;
33      do {
34          // Apply iteration scheme
35          *xnew = UpD.solve(b) - UpD.solve(L*LpD.solve(b -
                U**xold));
36          //Compute error
37          err = (*xold - *xnew).norm();
38          // Output error and rate for problem 1e
39  //          std::cout << k++ << "\t& " << (A*(*xnew) -
    b).norm() << "\t& " << (*xold - *xnew).norm() / errold <<
    "\t\\\\" << std::endl;
40
41          // Proceed to next step
42          std::swap(xold, xnew);
43  //          errold = err;
44      } while( err > rtol*(*xnew).norm() ); // Termination
            criteria when tolerance reached
45
46      x = *xnew; // Output was last value of iterations
47
48      return;
49  }
50
51  int main(int, char**) {
52      // Build matrix and r.h.s.
53      unsigned int n = 9;
54
55      Matrix A = Matrix::Zero(n,n);
56      for(unsigned int i = 0; i < n; ++i) {
57          if(i > 0) A(i,i-1) = 2;
58          A(i,i) = 3;
59          if(i < n-1) A(i,i+1) = 1;
```

```
60      }
61      Vector b = Vector::Constant(n,1);
62
63      //// PROBLEM 1d
64      std::cout << "*** PROBLEM 1d:" << std::endl;
65
66      Vector x = b;
67      GSIt(A, b, x, 10e-8);
68
69      double residual = (A*x - b).norm();
70
71      std::cout << "Residual = " << residual << std::endl;
72  }
```

Problem 2 Efficient sparse solver (24 pts.)

Fix $n \in \mathbb{N}$, $n \geq 2$, and let $1 \leq i_0, j_0 \leq n, i_0 > j_0$ and $c_i \in \mathbb{R}$ for $i = 1, \ldots, n-1$. We consider a $n \times n$ linear system of equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n,n}$ is given by

$$(\mathbf{A})_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ c_i & \text{if } i = j-1, \\ 1 & \text{if } i = i_0,\ j = j_0, \\ 0 & \text{otherwise,} \end{cases}$$

for $1 \leq i, j \leq n$.

(2a) Efficiently construct the matrix $\mathbf{A}$ defined above as an EIGEN matrix of type `Eigen::SparseMatrix<double>` using an intermediate triplet format. To that end, implement a function

```
1  using Vector = Eigen::VectorXd;
2  using Matrix = Eigen::SparseMatrix<double>;
3
4  Matrix buildA(const Vector & c, unsigned int i0, unsigned int
       j0);
```

whose, given the coefficients c_i in `c` and the indices i_0 and j_0, returns the sparse matrix `A`.

Solution: See implementation in `problem2_solution.cpp`.

(2b) Implement a function

```
Vector solveLSE(const Vector & c, const Vector & b,
          unsigned int i0, unsigned int j0);
```

that computes the solution $\mathbf{x} \in \mathbb{R}^n$ of the system $\mathbf{Ax} = \mathbf{b}$ with optimal asymptotic complexity $O(n)$.

HINT: If $(\mathbf{A})_{i_0,j_0}$ were zero, $\mathbf{A}$ would be a banded upper triangular matrix and the task would be easy. Regard $\mathbf{A}$ as a small modification of such a matrix.

Solution: We can either employ a rank-1 modification technique using the Sherman-Morrison-Woodbury formula, or use an elementary Gaussian elimination. See implementation in `problem2_solution.cpp`.

```
#include <iostream>

#include <vector>

#include <Eigen/Sparse>
#include <Eigen/SparseLU>

using Triplet = Eigen::Triplet<double>;
using Triplets = std::vector<Triplet>;

using Vector = Eigen::VectorXd;
using Matrix = Eigen::SparseMatrix<double>;

//! \brief Efficiently construct the sparse matrix A given c,
   i_0 and j_0
//! \param[in] c contains entries c_i for matrix A
//! \param[in] i0 row index i_0
//! \param[in] j0 column index j_0
//! \return Sparse matrix A
Matrix buildA(const Vector & c, unsigned int i0, unsigned int
   j0) {
    assert(i0 > j0);

    unsigned int n = c.size() + 1;
    Matrix A(n,n);
    Triplets triplets;

    unsigned int ntriplets = 2*n;
```

```
27
28      // Reserve space
29      triplets.reserve(ntriplets);
30
31      // Build triplets vector
32      for(unsigned int i = 0; i < n; ++i) {
33          triplets.push_back(Triplet(i,i,1));
34          if(i < n-1) triplets.push_back(Triplet(i,i+1,c[i]));
35      }
36      triplets.push_back(Triplet(i0,j0,1));
37
38      // Construct sparse matrix from its triplets
39
40      A.setFromTriplets(triplets.begin(), triplets.end());
41      return A;
42  }
43
44  //! \brief Solve system Ax = b with optimal complexity O(n)
45  //! \param[in] c Entries for matrix A
46  //! \param[in] b r.h.s. vector
47  //! \param[in] i0 index
48  //! \param[in] j0 index
49  //! \return Solution x, s.t. Ax = b
50  Vector solveLSE(const Vector & c, const Vector & b, unsigned
        int i0, unsigned int j0) {
51      assert(c.size() == b.size()-1 && "Size mismatch!");
52      assert(i0 > j0);
53
54      //// PROBLEM 2b
55
56      // Allocate solution vector
57      Vector ret(b.size());
58
59      unsigned int n = b.size();
60
61      Triplets triplets;
62
63      unsigned int ntriplets = 2*n;
64
65      // Reserve space
66      triplets.reserve(ntriplets);
```

```
67
68      // Build triplets vector
69      for(unsigned int i = 0; i < n; ++i) {
70          triplets.push_back(Triplet(i,i,1));
71          if(i < n-1) triplets.push_back(Triplet(i,i+1,c[i]));
72      }
73
74      // Construct sparse matrix from its triplets
75      Matrix A(n,n);
76      A.setFromTriplets(triplets.begin(), triplets.end());
77
78      // Vectors u and v
79      Vector u = Vector::Zero(n);
80      Eigen::MatrixXd v = Eigen::MatrixXd::Zero(1,n);
81      u( i0 ) = 1;
82      v( 0, j0 ) = 1;
83
84      // Apply SMW formula
85      Vector Ainv_b = A.triangularView<Eigen::Upper>().solve( b
            );
86      ret = Ainv_b - A.triangularView<Eigen::Upper>().solve( u *
            (v * Ainv_b)(0) ) / (1. + (v *
            A.triangularView<Eigen::Upper>().solve( u ))(0) );
87
88      return ret;
89  }
90
91  int main(int, char**) {
92      // Setup data for problem
93      unsigned int n = 150;
94
95      unsigned int i0 = 5, j0 = 4;
96
97      Vector b = Vector::Random(n); // Random vector for b
98      Vector c = Vector::Random(n-1); // Random vector for c
99
100     //// PROBLEM 2a
101     std::cout << "*** PROBLEM 2a:" << std::endl;
102
103     Matrix A = buildA(c, i0, j0);
104
```

```
105     // Solve sparse system using sparse LU and our own routine
106     A.makeCompressed();
107     Eigen::SparseLU<Matrix> splu;
108     splu.analyzePattern(A);
109     splu.factorize(A);
110
111     std::cout << "Error: " << std::endl << (
            solveLSE(c,b,i0,j0) - splu.solve(b) ).norm() <<
            std::endl;
112 }
```

Problem 3 Not-a-knot cubic spline interpolation (24 pts.)

We are given an interval $[a,b] \subset \mathbb{R}$ and a knot set

$$\mathcal{M} = \{a = t_0 < t_1 < \cdots < t_{n-1} < t_n = b\}$$

for some $n \in \mathbb{N} \setminus \{0\}$. Let $\mathcal{S}_{3,\mathcal{M}}$ denote the space of cubic spline functions on $\mathcal{M}$, and define

$$\tilde{\mathcal{S}}_{3,\mathcal{M}} = \{s \in \mathcal{S}_{3,\mathcal{M}} : s''' \text{ is continuous at } t_1 \text{ and at } t_{n-1}\}.$$

(3a) Derive a linear system of equations whose solution yields the slopes $\tilde{s}'(t_j)$, $j = 0,\ldots,n$, of $\tilde{s} \in \tilde{\mathcal{S}}_{3,\mathcal{M}}$ satisfying the interpolation conditions $\tilde{s}(t_j) = y_j$ for given $y_j \in \mathbb{R}$, $j = 0,\ldots,n$.

Solution: We know that (see (3.5.9) of the lecture notes), for general spline interpolants, the slopes $c_j := \tilde{s}'(t_j)$ satisfy

$$\begin{bmatrix} b_0 & a_1 & b_1 & 0 & \cdots & & 0 \\ 0 & b_1 & a_2 & b_2 & \ddots & & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & b_{n-3} & a_{n-2} & b_{n-2} & 0 \\ 0 & & \cdots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 3\left(\frac{y_1-y_0}{h_1^2} + \frac{y_2-y_1}{h_2^2}\right) \\ \vdots \\ 3\left(\frac{y_{n-1}-y_{n-2}}{h_{n-1}^2} + \frac{y_n-y_{n-1}}{h_n^2}\right) \end{bmatrix},$$

where

$$h_i = t_i - t_{i-1}, \qquad a_i = \frac{2}{h_i} + \frac{2}{h_{i+1}} \qquad \text{and} \qquad b_i = \frac{1}{h_{i+1}}.$$

This is an underdetermined linear system with $n+1$ unknowns and $n-1$ equations. The two missing equations determining the splines in $\tilde{\mathcal{S}}_{3,\mathcal{M}}$ will be obtained by imposing the continuity conditions on $\tilde{s}'''$ at t_1 and at t_{n-1}.

By differentiating three times the expression in (3.5.5) of the lecture notes for $\tilde{s}_{|[t_{j-1},t_j]}$, we obtain

$$\tilde{s}'''_{|[t_{j-1},t_j]}(t) = 6h_j^{-3}(2y_{j-1} - 2y_j + h_j c_{j-1} + h_j c_j).$$

Therefore, imposing $\tilde{s}'''_{|[t_0,t_1]}(t_1) = \tilde{s}'''_{|[t_1,t_2]}(t_1)$ yields

$$h_1^{-2}c_0 + (h_1^{-2} - h_2^{-2})c_1 - h_2^{-2}c_2 = 2(h_2^{-3}(y_1 - y_2) + h_1^{-3}(y_1 - y_0)).$$

Similarly, imposing the continuity condition in t_{n-1} yields

$$h_{n-1}^{-2}c_{n-2} + (h_{n-1}^{-2} - h_n^{-2})c_{n-1} - h_n^{-2}c_n = 2(h_n^{-3}(y_{n-1} - y_n) + h_{n-1}^{-3}(y_{n-1} - y_{n-2})).$$

Finally, adding these two equations to the initial system gives

$$\begin{bmatrix} b_0^2 & b_0^2 - b_1^2 & -b_1^2 & 0 & \cdots & & 0 \\ b_0 & a_1 & b_1 & 0 & \cdots & & 0 \\ 0 & b_1 & a_2 & b_2 & \ddots & & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & b_{n-3} & a_{n-2} & b_{n-2} & 0 \\ 0 & & \cdots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & & \cdots & 0 & b_{n-2}^2 & b_{n-2}^2 - b_{n-1}^2 & -b_{n-1}^2 \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 2\left(\frac{y_1-y_0}{h_1^3} + \frac{y_1-y_2}{h_2^3}\right) \\ 3\left(\frac{y_1-y_0}{h_1^2} + \frac{y_2-y_1}{h_2^2}\right) \\ \vdots \\ 3\left(\frac{y_{n-1}-y_{n-2}}{h_{n-1}^2} + \frac{y_n-y_{n-1}}{h_n^2}\right) \\ 2\left(\frac{y_{n-1}-y_{n-2}}{h_{n-1}^3} + \frac{y_{n-1}-y_n}{h_n^3}\right) \end{bmatrix},$$

as desired.

(3b) The provided C++ function (in `problem3.cpp`)

```
using Vector = Eigen::VectorXd;

void natsplineslopes (const Vector &t, const Vector &y, Vector
    &c)
```

computes the slopes $c_j := s'(t_j)$, $j = 0, \ldots, n$, of the *natural* cubic spline interpolant $s \in \mathcal{S}_{3,\mathcal{M}}$ through the data points (t_j, y_j), $j = 0, \ldots, n$.

Based on `natsplineslopes`, implement a corresponding function (in `problem3.cpp`)

```
void notknotslopes (const Vector &t, const Vector &y, Vector
    &c)
```

that computes the slopes $c_j := \tilde{s}'(t_j)$, $j = 0, \ldots, n$, of $\tilde{s} \in \tilde{\mathcal{S}}_{3,\mathcal{M}}$ satisfying $\tilde{s}(t_j) = y_j$ for given $y_j \in \mathbb{R}$, $j = 0, \ldots, n$.

Solution: See the implementation in `problem3_solution.cpp`.

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

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

//! \brief Computes the slopes for natural cubic spline interpolation
//! \param[in] t Vector of nodes
//! \param[in] y Vector of values at the nodes
//! \param[out] c Vector of slopes at the nodes
void natsplineslopes (const Vector &t, const Vector &y, Vector &c) {
    // Size check
    assert( ( t.size() == y.size() ) && "Error: mismatched size of t and y!");

    // n+1 is the number of conditions (t goes from t_0 to t_n)
    int n = t.size() - 1;

    Vector h(n);
    // Vector containing increments (from the right)
    for(int i = 0; i < n; ++i) {
        h(i) = t(i+1) - t(i);
        // Check that t is sorted
        assert( ( h(i) > 0 ) && "Error: array t must be sorted!");
    }

    // System matrix and rhs as in (3.5.9), we remove first and last row (known by natural contition)
    Eigen::SparseMatrix<double> A(n+1,n+1);
    Vector b(n+1);

    // WARNING: sparse reserve space
    A.reserve(3);

```

```
36      // Fill in natural conditions (3.5.10) for matrix
37      A.coeffRef(0,0) = 2./h(0);
38      A.coeffRef(0,1) = 1./h(0);
39      A.coeffRef(n,n-1) = 1./h(n-1);
40      A.coeffRef(n,n) = 2./h(n-1);
41
42      // Reuse computation for rhs
43      double bold = (y(1) - y(0)) / (h(0)*h(0));
44      b(0) = 3.*bold; // Fill in natural conditions (3.5.10)
45      // Fill matrix A and rhs b
46      for(int i = 1; i < n; ++i) {
47          // Fill in a
48          A.coeffRef(i,i-1) = 1./h(i-1);
49          A.coeffRef(i,i) = 2./h(i-1) + 2./h(i);
50          A.coeffRef(i,i+1) = 1./h(i);
51
52          // Reuse computation for rhs b
53          double bnew = (y(i+1) - y(i)) / (h(i)*h(i));
54          b(i) = 3. * (bnew + bold);
55          bold = bnew;
56      }
57      b(n) = 3.*bold; // Fill in natural conditions (3.5.10)
58      // Compress the matrix
59      A.makeCompressed();
60      std::cout << A;
61
62      // Factorize A and solve system A*c(1:end) = b
63      Eigen::SparseLU<Eigen::SparseMatrix<double>> lu;
64      lu.compute(A);
65      c = lu.solve(b);
66  }
67
68  //! \brief Computes the slopes for not-a-knot cubic spline
        interpolation
69  //! \param[in] t Vector of nodes
70  //! \param[in] y Vector of values at the nodes
71  //! \param[out] c Vector of slopes at the nodes
72  void notknotsplines (const Vector &t, const Vector &y, Vector
        &c) {
73      // Size check
74      assert( ( t.size() == y.size() ) && "Error: mismatched
```

```
        size of t and y!");

    // n+1 is the number of conditions (t goes from t_0 to t_n)
    int n = t.size() - 1;

    Vector h(n);
    // Vector containing increments (from the right)
    for(int i = 0; i < n; ++i) {
        h(i) = t(i+1) - t(i);
        // Check that t is sorted
        assert( ( h(i) > 0 ) && "Error: array t must be
            sorted!");
    }

    // System matrix and rhs as in (3.5.9), adding the two
        remaining conditions
    Eigen::SparseMatrix<double> A(n+1,n+1);
    Vector b(n+1);

    // WARNING: sparse reserve space
    A.reserve(3);

    // Fill in additional equations
    A.coeffRef(0,0) = 1./(h(0)*h(0));
    A.coeffRef(0,1) = 1./(h(0)*h(0)) - 1./(h(1)*h(1));
    A.coeffRef(0,2) = - 1./(h(1)*h(1));
    A.coeffRef(n,n-2) = 1./(h(n-2)*h(n-2));
    A.coeffRef(n,n-1) = 1./(h(n-2)*h(n-2)) -
        1./(h(n-1)*h(n-1));
    A.coeffRef(n,n) = - 1./(h(n-1)*h(n-1));

    // Reuse computation for rhs
    double bold = (y(1) - y(0)) / (h(0)*h(0));
    b(0) = 2.*((y(1)-y(0))/(h(0)*h(0)*h(0)) +
        (y(1)-y(2))/(h(1)*h(1)*h(1)));
    // Fill matrix A and rhs b
    for(int i = 1; i < n; ++i) {
        // Fill in a
        A.coeffRef(i,i-1) = 1./h(i-1);
        A.coeffRef(i,i) = 2./h(i-1) + 2./h(i);
        A.coeffRef(i,i+1) = 1./h(i);
```

```
111
112         // Reuse computation for rhs b
113         double bnew = (y(i+1) - y(i)) / (h(i)*h(i));
114         b(i) = 3. * (bnew + bold);
115         bold = bnew;
116     }
117     b(n) = 2.*((y(n-1)-y(n-2))/(h(n-2)*h(n-2)*h(n-2)) +
        (y(n-1)-y(n))/(h(n-1)*h(n-1)*h(n-1)));
118     // Compress the matrix
119     A.makeCompressed();
120     std::cout << A;
121
122     // Factorize A and solve system A*c(1:end) = b
123     Eigen::SparseLU<Eigen::SparseMatrix<double>> lu;
124     lu.compute(A);
125     c = lu.solve(b);
126 }
127
128 int main() {
129     std::cout << "Nothing to do!" << std::endl;
130 }
```

Problem 4 On the Gauss points (24 pts.)

Given $f \in C^0([0,1])$, the integral

$$g(x) := \int_0^1 e^{|x-y|} f(y)\, dy$$

defines a function $g \in C^0([0,1])$. Throughout this problem, we approximate the integral by means of n-point Gauss quadrature, which yields a function $g_n(x)$.

(4a) Let $\{\xi_j^n\}_{j=1}^n$ be the nodes for the Gauss quadrature on the interval $[0,1]$ and write w_j^n for the corresponding weights. Assume that the nodes are ordered, namely $\xi_j^n < \xi_{j+1}^n$ for every $j = 1, \ldots, n-1$. We can write

$$(g_n(\xi_l^n))_{l=1}^n = \mathbf{M}\ (f(\xi_j^n))_{j=1}^n,$$

for a suitable matrix $\mathbf{M} \in \mathbb{R}^{n,n}$. Give a formula for the entries of $\mathbf{M}$.

Solution: Applying Gauss quadrature to the expression for $g(\xi_l^n)$ we obtain $g_n(\xi_l^n) = \sum_{j=1}^n w_j^n e^{|\xi_l^n - \xi_j^n|} f(\xi_j^n)$, whence $M_{lj} = w_j^n e^{|\xi_l^n - \xi_j^n|}$ for $l, j = 1, \ldots, n$.

(4b) Implement a function (in `problem4.cpp`)

```
using Vector = Eigen::VectorXd;

template<typename Function>
Vector comp_g_gausspts(Function f, unsigned int n)
```

that computes the n-vector $(g_n(\xi_l^n))_{l=1}^n$ with optimal complexity $O(n)$ (excluding the computation of the Gauss nodes and weights), where `f` is an object with an evaluation operator `double operator()(double x)`, e.g. a lambda function, that represents the function f.

HINT: You may use the provided function (in `problem4.cpp`)

```
void gaussrule(int n, Vector & w, Vector & xi)
```

that computes the weights `w` and the ordered nodes `xi` relative to the `n`-point Gauss quadrature on the interval $[-1,1]$.

Solution: In order to write a function with optimal complexity, we need to use the structure of the matrix $\mathbf{M}$. Write

$$g_n(\xi_l^n) = \sum_{j=1}^{n} w_j^n e^{|\xi_l^n - \xi_j^n|} f(\xi_j^n) = \sum_{j=1}^{l-1} w_j^n e^{\xi_l^n} e^{-\xi_j^n} f(\xi_j^n) + \sum_{j=l}^{n} w_j^n e^{-\xi_l^n} e^{\xi_j^n} f(\xi_j^n).$$

Setting $p_l^n = \sum_{j=1}^{l-1} w_j^n e^{-\xi_j^n} f(\xi_j^n)$ and $q_l^n = \sum_{j=l}^{n} w_j^n e^{\xi_j^n} f(\xi_j^n)$, this identity can be rewritten as

$$g_n(\xi_l^n) = e^{\xi_l^n} p_l^n + e^{-\xi_l^n} q_l^n.$$

This expression for $l = 1, \dots, n$ can be computed with $O(n)$ operations. It remains to compute p_l^n and q_l^n in $O(n)$ operations. This can be done by using these recursive formulas, that immediately follow from the definition: for every $l = 1, \dots, n-1$ we have

$$\begin{aligned} p_1^n &= 0, & q_n^n &= w_n^n e^{\xi_n^n} f(\xi_n^n), \\ p_{l+1}^n &= p_l^n + w_l^n e^{-\xi_l^n} f(\xi_l^n), & q_l^n &= q_{l+1}^n + w_l^n e^{\xi_l^n} f(\xi_l^n). \end{aligned}$$

See the implementation in `problem4_solution.cpp`.

(4c) Test your implementation by computing $g(\xi_{11}^{21})$ ($\xi_{11}^{21} = 1/2$) for $f(y) = e^{-|0.5-y|}$. What result do you expect?

Solution: For reasons of symmetry, we have $\xi_{11}^{21} = 1/2$, and so $g_{21}(\xi_{11}^{21})$ should be equal to 1 for $f(y) = e^{-|0.5-y|}$, since Gauss quadrature is exact for constant functions.

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

using Vector = Eigen::VectorXd;

//! \brief Golub-Welsh implementation 5.3.35
//! \param[in] n number of Gauss nodes
//! \param[out] w weights for interval [-1,1]
//! \param[out] xi ordered nodes for interval [-1,1]
void gaussrule(int n, Eigen::VectorXd & w, Eigen::VectorXd &
    xi) {
     w.resize(n);
     xi.resize(n);
     if( n == 0 ) {
          xi(0) = 0;
          w(0) = 2;
     } else {
          Vector b(n-1);
          Eigen::MatrixXd J = Eigen::MatrixXd::Zero(n,n);

          for(int i = 1; i < n; ++i) {
               double d = (i) / sqrt(4. * i * i - 1.);
               J(i,i-1) = d;
               J(i-1,i) = d;
          }

          Eigen::EigenSolver<Eigen::MatrixXd> eig(J);

          xi = eig.eigenvalues().real();
          w = 2 * eig.eigenvectors().real()
                                    .topRows<1>()
                                    .cwiseProduct(
                                         eig.eigenvectors()
                                            .real()
                                            .topRows<1>()
                                    );
     }

     std::vector<std::pair<double,double>> P;
```

```
40      P.reserve(n);
41      for(unsigned int i = 0; i < n; ++i) {
42          P.push_back(std::pair<double,double>(xi(i),w(i)));
43      }
44      std::sort(P.begin(), P.end());
45      for(unsigned int i = 0; i < n; ++i) {
46          xi(i) = std::get<0>(P[i]);
47          w(i) = std::get<1>(P[i]);
48      }
49  }
50
51  //! \brief Compute the function g in the Gauss nodes
52  //! \param[in] f object with an evaluation operator (e.g. a
        lambda function) representing the function f
53  //! \param[in] n number of nodes
54  //! \param[out] Eigen::VectorXd containing the function g
        calculated in the Gauss nodes.
55  template<typename Function>
56  Eigen::VectorXd comp_g_gausspts(Function f, unsigned int n) {
57
58      Vector g(n);
59      Vector w(n), xi(n), p(n), q(n), expxi(n), fxi(n);
60      gaussrule(n, w, xi);  // Compute Gauss nodes and weights
            relative to [-1,1]
61      w = w/2.;  // Rescale the weights to [0,1]
62      xi = (xi + Eigen::MatrixXd::Ones(n,1))/2.;  // Rescale the
            nodes to [0,1]
63
64      for (int l=0; l<n; l++) {
65          fxi(l) = f(xi(l));
66          expxi(l) = exp(xi(l));
67      }
68
69      // Initialize the vectors p and q
70      p(0) = 0;
71      q(n-1) = w(n-1) * expxi(n-1) * fxi(n-1);
72
73      // Fill in the vectors p and q (O(n) complexity)
74      for (int l=0; l<n-1; l++) {
75          p(l+1) = p(l) + w(l)/expxi(l)*fxi(l);
76          q(n-2-l) = q(n-1-l) + w(n-2-l)*expxi(n-2-l)*fxi(n-2-l);
```

```
77      }
78
79      // Finally, constructing the output (O(n) complexity)
80
81      g = (p.array() * expxi.array() + q.array() /
            expxi.array()).matrix();
82      return g;
83  }
84
85  // Test the implementation by calculating g(xi^21_10) for
        f(y)=exp(-|0.5-y|).
86  int main() {
87      int n = 21;
88      auto f =  [] (double y) {return exp(-std::abs(.5-y)); };
89      Eigen::VectorXd g = comp_g_gausspts(f,n);
90      std::cout << "g(xi^" << n << "_" << (n+1)/2 << ") = " <<
            g((n+1)/2-1) << std::endl;
91  }
```

Problem 5 Construction of an evolution operator (28 pts.)

Let Ψ^h define the discrete evolution of an order p Runge-Kutta single step method for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \mathbf{f} : D \subseteq \mathbb{R}^d \to \mathbb{R}^d$. We define a new evolution operator:

$$\tilde{\Psi}^h := \frac{1}{1-2^p}\left(\Psi^h - 2^p \cdot (\Psi^{h/2} \circ \Psi^{h/2})\right), \tag{11}$$

where $\circ$ denotes the composition of mappings.

(5a) Let Ψ^h be the evolution operator of the explicit Euler method. Give the explicit formula for $\tilde{\Psi}^h$.

Solution: Writing the forward Euler method as $\Psi^h(\mathbf{y}_0) := \mathbf{y}_0 + h\mathbf{f}(\mathbf{y}_0)$ (and $p = 1$), we obtain:

$$\begin{aligned}
\tilde{\Psi}^h(\mathbf{y}_0) &= \frac{1}{1-2^p}\left(\mathbf{y}_0 + hf(\mathbf{y}_0) - 2^p \cdot (\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0) + \frac{h}{2}f(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right) \\
&= \mathbf{y}_0 - \left(h(1-1)f(\mathbf{y}_0) - hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right) \\
&= \mathbf{y}_0 + hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0)).
\end{aligned}$$

Remark 1. This is the evolution operator of the explicit trapezoidal method, a 2-stage explicit Runge-Kutta method.

(5b) Implement a C++ function (in `problem5.cpp`)

```
using Vector = Eigen::VectorXd;

template <class Operator>
Vector psitilde(const Operator &Psi, unsigned int p,
          double h, const Vector &y0);
```

that returns $\tilde{\Psi}^h \mathbf{y}_0$ when given the underlying Ψ.

Objects of type `Operator` (here and in the following) must provide an evaluation operator with signature:

```
Vector operator()(double h, const Vector &y);
```

providing the evaluation of $\Psi^h(\mathbf{y})$. A suitable C++ lambda function satisfies this requirement.

Solution: See implementation in `problem5_solution.cpp`.

(5c) Implement a C++ function (in `problem5.cpp`)

```
template <class Operator>
std::vector<Vector> odeintequi(const Operator &Psi,
     double T, const Vector &y0, unsigned int N);
```

for the approximation of the solution of the initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \mathbf{y}(0) = \mathbf{y}_0$. The function uses a single step method defined by the discrete evolution operator Ψ (given as `Psi`) on N equidistant timesteps with final time $T > 0$. The function returns the approximated value at each step (including y_0: $y_0, y_1, \ldots$) in a `std::vector<Vector>`.

HINT: You can use `std::vector<Vector>::push_back` to add elements to the end of a `std::vector<Vector>`.

Solution: See implementation in `problem5_solution.cpp`.

(5d) In the case of the IVP

$$\dot{y} = 1 + y^2, \quad y(0) = 0, \tag{12}$$

with exact solution $y_{ex}(t) = \tan(t)$, determine empirically (using `odeintequi`) the order of the single step method induced by $\tilde{\Psi}^h$, when Ψ is the discrete evolution operator for the explicit Euler method. Monitor the error $|y_N(1) - y_{ex}(1)|$ at final time $T = 1$, where y_N is the solution approximated through $\tilde{\Psi}^h$ using N uniforms steps with $N = 2^q, q = 2, \ldots, 12$.

Solution: The experimental convergence order is 2 ($O(h^2)$), the order of convergence of forward Euler is $p = 1$. See implementation in `problem5_solution.cpp`.

(5e) In general, the method defined by $\tilde{\Psi}^h$ has order $p + 1$. Thus, it can be used for adaptive timestep control and prediction.

Complete the implementation of a function (found in `problem5.cpp`)

```
template <class Operator>
std::vector<Vector> odeintssctrl(const Operator &Psi,
                        double T, const Vector &y0,
                        double h0, unsigned int p,
                        double reltol, double abstol,
                        double hmin);
```

for the approximation of the solution of the IVP by means of adaptive timestepping based on Ψ and $\tilde{\Psi}$, where Ψ is passed through the argument `Psi`. Step rejection and stepsize correction and prediction has to be employed. The argument `T` supplies the final time, `y0` the initial state, `h0` an initial stepsize, `p` the order to the discrete evolution Ψ, `reltol` and `abstol` the respective tolerances, and `hmin` a minimal stepsize that will trigger premature termination. Compute the solution obtained using this function applied to the IVP (12) up to time $T = 1$ with the following data: `h0 = 1/100`, `reltol = 10e-8`, `abstol = 10e-8`, `hmin = 10e-6`. Output the approximated solution at time $T = 1$.

Solution: See implementation in `problem5_solution.cpp`.

```
#include <iostream>

#include <vector>

#include <Eigen/Dense>

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

//! \brief Evolves the vector y0 using the evolution operator
   \tilde{\Psi} with step-size y0
```

```
11 //! \tparam Operator type for evolution operator (e.g. lambda
       function type)
12 //! \param[in] Psi original evolution operator, must have
       operator(double, const Vector&)
13 //! \param[in] p parameter p for construction of Psi tilde
14 //! \param[in] h step-size
15 //! \param[in] y0 previous step
16 //! \return Evolved step $\tilde{\Psi}^h y0$
17 template <class Operator>
18 Vector psitilde(const Operator& Psi, unsigned int p, double h,
       const Vector & y0) {
19     return ( Psi(h,y0) - (2<<(p))*Psi(h/2., Psi(h/2.,y0)) ) /
           (1. - (2<<(p)));
20 }
21
22 //! \brief Evolves the vector y0 using the evolution operator
       from time 0 to T using equidistant steps
23 //! \tparam Operator type for evolution operator (e.g. lambda
       function type)
24 //! \param[in] Psi original evolution operator, must have
       operator(double, const Vector&)
25 //! \param[in] T final time
26 //! \param[in] y0 initial data
27 //! \param[in] N number of steps
28 //! \return Vector of all steps y_0, y_1, ...
29 template <class Operator>
30 std::vector<Vector> odeintequi(const Operator& Psi, double T,
       const Vector &y0, int N) {
31     double h = T / N;
32     double t = 0.;
33     std::vector<Vector> Y;
34     Y.reserve(N+1);
35     Y.push_back(y0);
36     Vector y = y0;
37
38     while( t < T ) {
39         y = Psi(h,Y.back());
40
41         Y.push_back(y);
42         t += std::min(T-t,h);
```

```
43      }
44
45      return Y;
46  }
47
48  //! \brief Evolves the vector y0 using the evolution operator
        from time 0 to T using adaptive error control
49  //! \tparam Operator type for evolution operator (e.g. lambda
        function type)
50  //! \param[in] Psi original evolution operator, must have
        operator(double, const Vector&)
51  //! \param[in] T final time
52  //! \param[in] y0 initial data
53  //! \param[in] h0 initial step size
54  //! \param[in] p parameter p for construction of Psi tilde
55  //! \param[in] reltol relative tolerance for error control
56  //! \param[in] abstol absolute tolerance for error control
57  //! \param[in] p parameter p for construction of Psi tilde
58  //! \param[in] hmin minimal step size
59  //! \return Vector of all steps y_0, y_1, ...
60  template <class Operator>
61  std::vector<Vector> odeintssctrl(const Operator& Psi, double
        T, const Vector &y0, double h0,
62                                   unsigned int p, double
                                         reltol, double abstol,
                                         double hmin) {
63      // Time tracker
64      double t = 0.;
65      // Step size tracker
66      double h = h0;
67      // Vector of snapshots
68      std::vector<Vector> Y;
69      // Save initial data
70      Y.push_back(y0);
71      // Start with y0 as value
72      Vector y = y0;
73
74      // Kill if final time reached or min step size reached
75      while( t < T && h > hmin ) {
76          // Compute with two methods
77          Vector y_high = psitilde(Psi, p, h, y);
```

```
78          Vector y_low = Psi(h,y);
79          // Estimate error
80          double err_est = (y_high - y_low).norm();
81
82          // Compute tolerance (min of reltol and abstol)
83          double tol = std::max(reltol*y_high.norm(),abstol);
84
85          // Accept step if true (error smaller than tol)
86          if( err_est < tol ) {
87              t += h;
88
89              y = y_high;
90              Y.push_back(y_high);
91
92          }
93          // Predict step-size
94          h *= std::max(.5, std::min(2., std::pow(tol/err_est,
                1./(p+1.))));
95          // Force time if needed
96          h = std::min(T-t, h);
97      }
98      if( t < T ) std::cerr << "Failed to reach final time!" <<
            std::endl;
99
100     return Y;
101 }
102
103 int main(int, char**) {
104
105     auto f = [] (const Vector &y) -> Vector { return
            Vector::Ones(1) + y*y; };
106     Vector y0 = Vector::Zero(1);
107     double T = 1.;
108     auto y_ex = [] (double t) -> Vector { Vector y(1); y <<
            tan(t); return y; };
109
110     unsigned p = 1;
111
112     auto Psi = [&f] (double h, const Vector & y0) -> Vector {
            return y0 + h*f(y0); };
113     auto PsiTilde = [&Psi, &f, &p] (double h, const Vector &
```

```
        y0) -> Vector {  return psitilde(Psi, p, h, y0); };
114
115     //// PROBLEM 5d
116     std::cout << "*** PROBLEM 5d:" << std::endl;
117
118     std::cout << "Error table for equidistant steps:" <<
            std::endl;
119     std::cout << "N" << "\t" << "Error" << std::endl;
120     for(int N = 4; N < 4096; N=N<<1 ) {
121          std::vector<Vector> Y = odeintequi(PsiTilde, T, y0, N);
122          double err = (Y.back() - y_ex(1)).norm();
123          std::cout << N << "\t" << err << std::endl;
124     }
125
126     //// PROBLEM 5e
127     std::cout << "*** PROBLEM 5e:" << std::endl;
128
129     double h0 = 1./100.;
130     std::vector<Vector> Y;
131     double err = 0;
132     Y = odeintssctrl(Psi, T, y0, h0, p, 10e-8, 10e-8, 10e-6);
133     err = (Y.back() - y_ex(1)).norm();
134     std::cout << "Adaptive error control results:" <<
            std::endl;
135     std::cout << "Error" << "\t" << "No. Steps" << "\t" <<
            "y(1)" << "\t" << "y(ex)" << std::endl;
136     std::cout << err << "\t" << Y.size() << "\t" << Y.back()
            << "\t" << y_ex(1) << std::endl;
137 }
```