S. Mishra
K. O. Lye
L. Scarabosio

Autumn Term 2015

Computational Methods for Engineering Applications II

ETH Zürich
D-MATH

Homework Problem Sheet 3

This assignment sheet contains some tasks marked as **Core problems**. If you hand them in (see deadline at the end of the problem sheet), these tasks will be corrected. Full mark for the total of the core problems in all assignments will give a 20% bonus on the total points in the final exam. This is really a bonus, which means that at the exam you can still get the highest grade without having the bonus points (of course then you need to score more points at the exam).

The total number of points for the Core problems of this sheet is **4 points**. (The total number of points over all assignments will be around 20).

Problem 3.1 Finite Differences for Poisson Equation in 1D

We consider the 1D Poisson equation with homogeneous Dirichlet boundary conditions:

$$\begin{aligned} -u''(x) &= f(x), \quad \forall x \in \Omega = (0,1) \\ u(0) &= u(1) = 0. \end{aligned} \tag{3.1.1}$$

We want to discretize (3.1.1) using the Finite Differences method, namely the centered finite differences.

To this aim, we subdivide the interval $[0,1]$ in $N+1$ subintervals using equispaced grid points $\{x_0 = 0, x_1, \ldots, x_N, x_{N+1} = 1\}$.

The discretized problem can be written as a linear system

$$\mathbf{A}\mathbf{u} = \mathbf{F}, \tag{3.1.2}$$

where $\mathbf{A}$ is a $N \times N$ matrix, $\mathbf{F}$ a $N \times 1$ vector and $\mathbf{u}$ the $N \times 1$ vector containing of unknowns $u_j \approx u(x_j)$, $j = 1, \ldots, N$, the approximate values of the function u at the grid points.
Let us denote by $h = |x_1 - x_0|$ the meshsize.

(3.1a) Refresh your mind on what we saw in class on the Finite Difference scheme and in particular on the central finite differences.
Write the matrix $\mathbf{A}$ and the right-handside $\mathbf{L}$. For the right-handside, write it in terms of a generic force term $f(x)$ in (3.1.1).

Solution: We have:

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & -1 & 2 \end{pmatrix} \qquad \mathbf{F} = h^2 \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \\ f(x_N) \end{pmatrix}$$

(3.1b) In the template file `finite_difference.cpp`, implement the function

`void` createPoissonMatrix(SparseMatrix& A, `int` N),

where `typedef` Eigen::SparseMatrix<`double`> SparseMatrix. This function computes the matrix $\mathbf{A}$ for (3.1.2). Here the input parameter `N` denotes the number of *interior* grid points. Assume that the size of the input matrix `A` has not been initialized.

Solution: See listing 3.1 for the code.

Listing 3.1: Implementation for `createPoissonMatrix`

```
//!  Create the 1D Poisson matrix
//!  @param[out] A will contain the Poisson matrix
//!  @param[in] N the number of points
void createPoissonMatrix(SparseMatrix& A, int N) {
    A.resize(N, N);
    std::vector<Triplet> triplets;
    triplets.reserve(N + 2 * N - 2);
    for (int i = 0; i < N; ++i) {
        triplets.push_back(Triplet(i, i, 2));
        if (i > 0) {
            triplets.push_back(Triplet(i, i - 1, -1));
        }
        if (i < N - 1){
            triplets.push_back(Triplet(i, i + 1, -1));
        }
    }

    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

(3.1c) In the template file `finite_difference.cpp`, implement the function

`void` createRHS(Vector& rhs, FunctionPointer f, `int` N, `double` dx),

where `typedef double`(*FunctionPointer)(`double`) and `typedef` Eigen::VectorXd Vector. This function computes the right-handside $\mathbf{F}$ for (3.1.2). The input parameter `f` is the function pointer for the right-handside $f(x)$, `N` is again the number of interior grid points, and `dx` is the length of a cell. Assume that the size of the input vector `rhs` has not been initialized.

Solution: See listing 3.2 for the code.

Listing 3.2: Implementation for `createRHS`

```
//!  Create the right hand side for the poisson problem.
//!  @note This scales the right hand side (ie.  dx^2 * f(x))
//!
//!  @param[out] rhs will contain the right hand side
//!  @param[in] f function pointer to f
//!  @param[in] N the number of points to use
//!  @param[in] dx the cell length
void createRHS(Vector& rhs, FunctionPointer f, int N, double dx) {
    rhs.resize(N);
    for (int i = 0; i < N; ++i) {
        const double x = (i + 1) * dx;
        rhs[i] = dx * dx * f(x);
    }
}
```

(3.1d) In the template file `finite_difference.cpp`, implement the function

`void poissonSolve(Vector& u, FunctionPointer f, int N),`

to solve the Poisson problem (3.1.1).
The input parameters `f` and `N` are as in subproblem (3.1c). The vector `u` is assumed to have not been initialized in size, and at the end of the routine it has to correspond to the array $\{u_h(x_j)\}_{j=1}^N$ containing the approximate values of the solution u at the interior grid points $\{x_j\}_{j=1}^N$.

HINT: Use the routines from subproblems (3.1b) and (3.1c).

Solution: See listing 3.3 for the code.

Listing 3.3: Implementation for `poissonSolve`

```
//!  Solves the Poisson equation
//!
//!  -u''(x) = f(x)
//!
//!  on [0,1] with boundary values u(0) = u(1) = 0.
//!
//!  @param[out] u should contain the solution u at the end
//!  @param[in] f should be a function pointer to f
//!  @param[in] N as in the exercise
void poissonSolve(Vector& u, FunctionPointer f, int N) {
    double dx = 1.0 / (N + 1);

    SparseMatrix A;
    createPoissonMatrix(A, N);

```

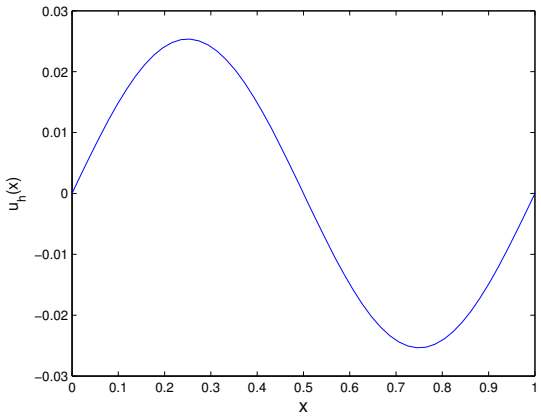

Figure 3.1: Plot for subproblem (3.1e)

```
16      Vector rhs;
17      createRHS(rhs, f, N, dx);
18
19      Eigen::SparseLU<SparseMatrix> solver;
20      solver.compute(A);
21
22      if ( solver.info() != Eigen::Success) {
23          throw std::runtime_error("Could not decompose the
              matrix");
24      }
25      u.resize(N + 2);
26      u.segment(1, N) = solver.solve(rhs);
27      u[0] = 0;
28      u[N+1] = 0;
29  }
```

(3.1e) Run the routine `poissonSolve` for $f(x) = \sin(2\pi x)$ and $N = 50$ and plot the solution.

Solution:

See Fig. 3.1 for the plot.

We saw in the lecture that the centered finite difference schemes satisfies is stable and consistent, and thus it converges to the exact solution u to (3.1.1) when the mesh is refined.
Here we are going to test the convergence of our scheme through a convergence study.

(3.1f) In the template file `finite_difference.cpp`, implement the function

```
void poissonConvergence( std::vector<double>& errors, std::vector<int>& resolutions,
    FunctionPointer f)
```

,

to perform the convergence study. The input argument `f` is a function pointer to the right-hanside $f(x)$. The vectors `resolutions` and `errors`, assumed to be passed in input with uninitialized

size, must contain, in output, the array $[N_1, \ldots, N_6]$ of numbers of degrees of freedom and the array $[e_1, \ldots, e_6]$ of computed errors, respectively.
As error between the discrete solution u_h and the exact solution u, we consider the maximum norm error $\|u - u_{h_i}\|_\infty = \max_{x\in[0,1]} |u - u_h|$, $i = 1, \ldots, 6$; we can compute this error just in an approximate way: we consider a very fine grid with meshsize $h_{ref} = \frac{1}{2^{10}}$ and grid points $x_0 = 0, x_1 = h_{ref}, \ldots, x_{N+1} = 1$ and approximate the maximum norm by

$$\|u - u_h\|_\infty \approx \max_{x_0 \ldots x_{N+1}} |u(x_i) - u_h(x_i)| \tag{3.1.3}$$

The standard steps for a convergence study then:

1. compute the *exact solution* u to (3.1.1);
2. start from a meshsize $h_1 = \frac{1}{4}$, corresponding to $N_1 = 3$ interior grid points;
3. compute the discrete solution u_{h_1} to (3.1.1);
4. compute the error $e_1 \approx \|u - u_{h_1}\|_\infty$; inside each mesh interval, consider the linear interpolant for u_h;
5. refine the grid, considering $h_2 = \frac{h_1}{2} = \frac{1}{8}$ and repeat the algorithm from step 3;
6. repeat the previous step till $h_6 = \frac{h_1}{2^5}$.

Solution: See listing 3.4 for the code.

Listing 3.4: Implementation for `poissonConvergence`

```
//!  Computes error for a range of cell lengths and stores them to
   errors
//!  @param[out] errors the errors computed
//!  @param[out] resolutions the resolutions used
void poissonConvergence( std::vector<double>& errors,
                         std::vector<int>& resolutions) {

    const int startK = 2;
    const int endK = 9;
    errors.resize(endK - startK);
    resolutions.resize(errors.size());
    for (int k = startK; k < endK; ++k) {
        const int N = 1 << k - 1;
        Vector u;
        poissonSolve(u, F, N);

        double maxError = 0;
        const double dx = 1.0 / (N + 1);
        for (int i = 0; i < N + 2; ++i) {
            const double x = i * dx;
            const double error = std::abs(u[i] - exact(x));
            maxError = std::max(error, maxError);
```

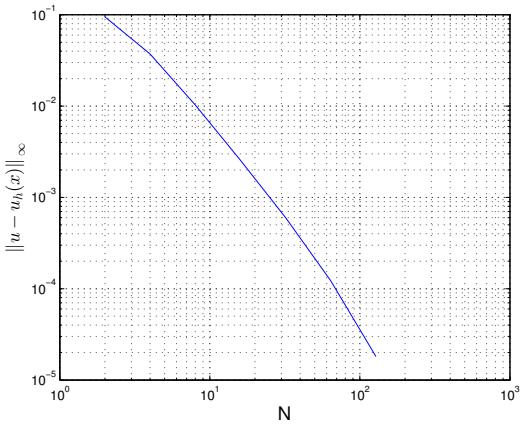

Figure 3.2: Convergence plot for subproblem (3.1g).

```
22
23              }
24              errors[k - startK] = maxError;
25              resolutions[k - startK] = N;
26          }
27
28  }
```

(3.1g) Run the routine `poissonConvergence` for $f(x) = \sin(2\pi x)$.
Make a double logarithmic plot of the errors $e_1, \ldots, e_6$ versus the resolutions $N_1, \ldots, N_6$. What do you observe? Which is the order of convergence?

Solution: See Fig.3.2 for the plot. Doing a linear fitting of the logarithm of the errors versus the logarithm of the resolutions, we can observe an order of convergence of around 2 with respect to the degrees of freedom. Since the meshsizes are given by $h_i = \frac{1}{N_i+1}$ for $i = 1, \ldots, 6$, we also have second order convergence with respect to the meshsize, as expected from the theory.

Problem 3.2 Finite Differences for Poisson Equation in 2D

In this problem we consider the Finite Differences discretization of the Poisson problem on the unit square:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega := (0,1)^2, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{3.2.1}$$

for a bounded and continuous function $f \in \mathcal{C}^0(\overline{\Omega})$.

We consider a regular tensor product grid with meshwidth $h := (N+1)^{-1}$ and we assume a lexicographic numbering of the interior vertices of the mesh as depicted in Fig.3.3.

We consider the 5-point stencil finite difference scheme for the operator $-\Delta$ described by the 5-points stencil shown in Fig. 3.4.

(3.2a) Write the system

$$\mathbf{A}\mathbf{u} = \mathbf{F} \tag{3.2.2}$$

corresponding to the discretization of (3.2.1) using the stencil in Fig. 3.4, specifying the matrix $\mathbf{A}$ and the vectors $\mathbf{F}$ and $\mathbf{u}$.

Solution: The equations for the discretized system are:

$$\frac{-u_{j-1} + 4u_j - u_{j+1}}{h^2} = f(x_j, y_j), \quad \text{for } j = 1, \ldots, N, \tag{3.2.3}$$

where (x_j, y_j) denote the coordinates of the node j according to the lexicographic order of Fig. 3.3, and u_j, $j = 0, \ldots, N+1$, denotes the discrete solution, with $u_0 = u_{N+1} = 0$.

Thus, writing the equations as a system, we have a block tridiagonal matrix $\mathbf{A} \in \mathbb{R}^{N^2 \times N^2}$

$$\mathbf{A} = \frac{1}{h^2} \begin{pmatrix} \mathbf{B} & -\mathbf{I} & 0 & \ldots & \ldots & 0 \\ -\mathbf{I} & \mathbf{B} & -\mathbf{I} & 0 & \ldots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ 0 & \ldots & 0 & -\mathbf{I} & \mathbf{B} & -\mathbf{I} \\ 0 & \ldots & \ldots & 0 & -\mathbf{I} & \mathbf{B} \end{pmatrix}$$

where $\mathbf{I} \in \mathbb{R}^{N \times N}$ is the identity matrix, and $\mathbf{B} \in \mathbb{R}^{N \times N}$ is the tridiagonal matrix

$$\mathbf{B} = \begin{pmatrix} 4 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 4 & -1 & 0 & \ldots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ 0 & \ldots & 0 & -1 & 4 & -1 \\ 0 & \ldots & \ldots & 0 & -1 & 4 \end{pmatrix}$$

The vectors $\mathbf{F}$ and $\mathbf{u}$ are given by $(\mathbf{F})_j = f(x_j)$ and $(\mathbf{u})_j = u_j$, $j = 1, \ldots, N$.

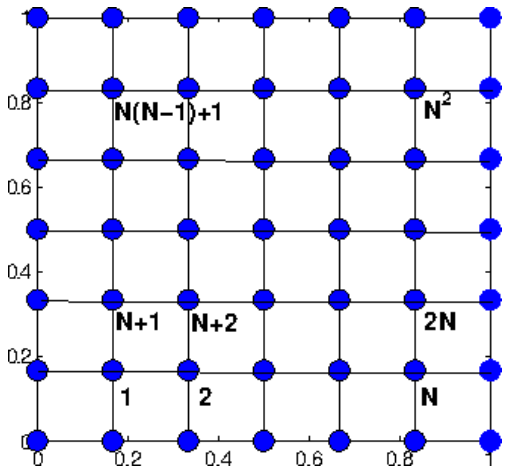

Figure 3.3: Lexikographic numbering of vertices of the equidistant tensor product mesh.

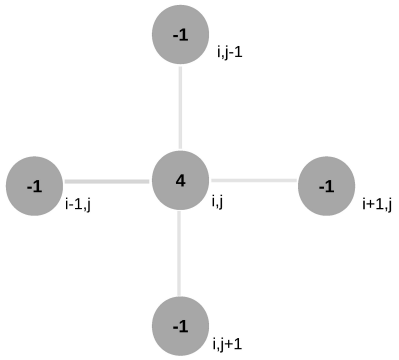

Figure 3.4: 5-point stencil used in this problem.

(3.2b) **(Core problem)** In the template file `finite_difference.cpp`, implement the function

```
void createPoissonMatrix2D(SparseMatrix& A, int N),
```

to construct the matrix $\mathbf{A}$ in (3.2.2), where N denotes the number of interior grid points along one dimension, with `typedef Eigen::SparseMatrix<double> SparseMatrix`. Assume the matrix `A` to have an uninitialized size at the beginning.

Solution: See Listing 3.5 for the code.

Listing 3.5: Implementation for `createPoissonMatrix2D`

```
//!  Create the Poisson matrix for 2D finite difference.
//!  @param[out] A will be the Poisson matrix (as in the exercise)
//!  @param[in] N number of elements in the x-direction
void createPoissonMatrix2D(SparseMatrix& A, int N) {
    std::vector<Triplet> triplets;
    A.resize(N*N, N*N);
    triplets.reserve(5*N*N-4*N);
    for (int i = 0; i < N*N; ++i) {
        triplets.push_back(Triplet(i, i, 4));
        if (i % N != 0) {
            triplets.push_back(Triplet(i, i-1, -1));
        }
        if (i % N != N - 1) {
            triplets.push_back(Triplet(i, i+1, -1));
        }
        if (i >= N) {
            triplets.push_back(Triplet(i, i-N, -1));
        }
        if (i < N*N - N ) {
            triplets.push_back(Triplet(i, i+N, -1));
```

```
        }
    }

    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

(3.2c) **(Core problem)** In the template file `finite_difference.cpp`, implement the function

```
void createRHS(Vector& rhs, FunctionPointer f, int N, double dx),
```

to build the vector $\mathbf{F}$ in (3.2.2), with `typedef Eigen::VectorXd Vector` and `typedef double(*FunctionPointer)(double, double)`. The argument `f` is a function pointer to the function f in (3.2.1), `N` is the number of interior grid points and `dx` is cell width. Again, assume that the vector `rhs` has uninitialized size when passed in input.

Solution: See Listing 3.6 for the code.

Listing 3.6: Implementation for `createRHS`

```
//!  Create the Right hand side for the 2D finite difference
//!  @param[out] rhs will at the end contain the right hand side
//!  @param[in] f the right hand side function f
//!  @param[in] N the number of points in the x direction
//!  @param[in] dx the cell width
void createRHS(Vector& rhs, FunctionPointer f, int N, double
   dx) {
    rhs.resize(N * N);

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            const double x = (i + 1) * dx;
            const double y = (j + 1) * dx;
            rhs[i * N + j] = dx * dx * f(x, y);
        }
    }
}
```

(3.2d) **(Core problem)** In the template file `finite_difference.cpp`, implement the function

```
void poissonSolve(Vector& u, FunctionPointer f, int N),
```

to solve the system (3.2.2), with `u` the vector containing the values of the approximate solution at all the grid points, *including those on the boundary*, and the other arguments as in the previous subproblems.

Solution: See Listing 3.7 for the code.

Listing 3.7: Implementation for `poissonSolve`

```cpp
//!  Solve the Poisson equation in 2D
//!  @param[out] u will contain the solution u
//!  @param[in] f the function pointer to f
//!  @param[in] N the number of points to use (in x direction)
void poissonSolve(Vector& u, FunctionPointer f, int N) {
    double dx = 1.0 / (N + 1);

    SparseMatrix A;
    createPoissonMatrix2D(A, N);

    Vector rhs;
    createRHS(rhs, f, N, dx);

    Eigen::SparseLU<SparseMatrix> solver;
    solver.compute(A);

    if ( solver.info() != Eigen::Success) {
        throw std::runtime_error("Could not decompose the
            matrix");
    }
    u.resize((N + 2) * (N + 2));
    u.setZero();

    Vector innerU = solver.solve(rhs);

    // Copy vector to inner u.
    for (int i = 1; i < N + 1; ++i) {
        for (int j = 1; j < N + 1; ++j) {
            u[i * (N + 2) + j] = innerU[(i - 1) * N + j - 1];
        }
    }

}
```

(3.2e) Plot the discrete solution that you get from subproblem (3.2d) for $f(x, y) = 8\pi^2 \sin(2\pi x) \sin(2\pi y)$ and $N = 50$, and compare it to the exact solution $u(x, y) = \sin(2\pi x) \sin(2\pi y)$.

Solution: See Fig. 3.5 for the discrete solution.

Problem 3.3 Linear Finite Elements in 1D

In class the Galerkin discretization of a 2-point boundary value problem by means of trial and test spaces of merely continuous piecewise linear functions was discussed. In this problem, we

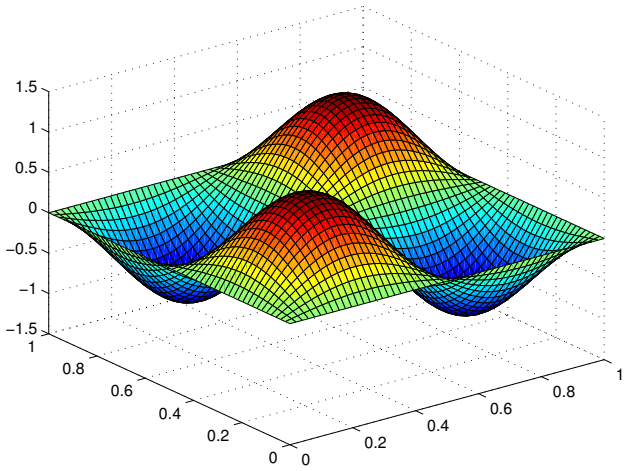

Figure 3.5: Plot for subproblem (3.2e).

practise the crucial steps for the linear variational problem

$$u \in H_0^1([a,b]): \quad \int_a^b \frac{\mathrm{d}u}{\mathrm{d}x}(x)\,\frac{\mathrm{d}v}{\mathrm{d}x}(x) = \int_a^b f(x)v(x)\,\mathrm{d}x, \quad \forall v \in H_0^1([a,b]), \tag{3.3.1}$$

where $-\infty < a < b < \infty$ and $f \in \mathcal{C}^0([a,b])$. Please note that both trial and test functions vanish at the endpoints of the interval, as indicated by the subscript "0" in the symbol for the function space.

(3.3a) Derive the Galerkin matrix for (3.3.1), when using the trial and test space $\mathcal{S}^0_{1,0}(\mathcal{M})$ of continuous, piecewise linear functions on an *equidistant* mesh $\mathcal{M}$ with $N \in \mathbb{N}$ interior nodes. The standard basis of hat functions is to be used.

Solution: Let $\Omega = [a,b]$ and $\mathcal{M} := \{(x_{j-1}, x_j) \,|\, 1 \le j \le N+1\}$ be an equidistant mesh, meaning that $x_j = a + jh$ with $h = \frac{b-a}{N+1}$.

We first write the discretized version of the equation:

$$\int_a^b \frac{\mathrm{d}u_N}{\mathrm{d}x}(x)\frac{\mathrm{d}v_N}{\mathrm{d}x}(x) = \int_a^b f(x)v_N(x)\,\mathrm{d}x$$

The discrete solution u_N is given by $u_N = \sum_{i=1}^N \mu_i b_N^i(x)$, with $\boldsymbol{\mu} = \{\mu_i\}_{i=1}^N$ the unknown vector of coefficients. Thus, rewriting the previous system in terms of the basis $\{b_N^i, 1 \le i \le N\}$ of hat functions, we obtain:

$$\sum_{i=1}^N \int_a^b \frac{\mathrm{d}b_N^i}{\mathrm{d}x}(x)\frac{\mathrm{d}b_N^k}{\mathrm{d}x}(x)\,\mathrm{d}x\mu_i = \int_a^b f(x)b_N^k(x)\,\mathrm{d}x,\ k \in \{1,...,N\}$$

This can be translated to a system of linear equations of the form $\mathbf{A}\boldsymbol{\mu} = \mathbf{F}$ where

$$\mathbf{A}_{i,k} = \int_a^b \frac{\mathrm{d}b_N^i}{\mathrm{d}x}(x)\frac{\mathrm{d}b_N^k}{\mathrm{d}x}(x)\,\mathrm{d}x,$$

and

$$\mathbf{F}_k = \int_a^b f(x) b_N^k(x)\,\mathrm{d}x.$$

As shown in the lecture notes, the Galerkin matrix $\mathbf{A}$ has the following form:

$$\mathbf{A} = \frac{1}{h}\begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & & \\ 0 & -1 & \dots & \dots & \\ \dots & & \dots & 2 & -1 \\ 0 & & & -1 & 2 \end{pmatrix}.$$

(3.3b) In the template file `fem.cpp`, implement the function

```
void createGalerkinMatrixWithBoundary(SparseMatrix& A, int N, double dx)
```

(with `typedef Eigen::SparseMatrix<double> SparseMatrix`), that computes the matrix `A` as in subproblem (3.3a). The argument `N` denotes the number of interior grid points, and `dx` denotes the cell length. For later purposes, the matrix `A` has to contain the entries associated to the two boundary basis functions $b_N^0(x)$ and b_{N+1}^N, too, leading to a $(N+2)\times(N+2)$ matrix.

Solution: See listing 3.8 for the code.

Listing 3.8: createGalerkinMatrixWithBoundary

```
1  //!  Create the Galerkin matrix for 1D with boundary terms
2  //!  @param[out] A will be the Galerkin matrix (as in the exercise)
3  //!  @param[in] N as in the exercise
4  //!  @param[in] dx the cell length
5  void createGalerkinMatrixWithBoundary(SparseMatrix& A, int N,
       double dx) {
6      std::vector<Triplet> triplets;
7      A.resize(N + 2, N + 2);
8
9      // Reserve the space we will allocate
10     triplets.reserve((N+2) + 2 * N - 2);
11
12     for (int i = 0; i < N + 2; ++i) {
13         if (i > 0 && i < N + 1) { // This is the last and first
               element (boundary)
14             triplets.push_back(Triplet(i, i, 2.0/dx));
15         } else { // Inner diagonal
16             triplets.push_back(Triplet(i, i, 1.0/dx));
17         }
18
19         if (i > 0) { // Lower diagonal, should not be present on
               first row.
20             triplets.push_back(Triplet(i, i-1, -1.0/dx));
21         }
22         if (i < N + 1) { // Upper diagonal, should not be present
               on last row.
```

```
            triplets.push_back(Triplet(i, i+1, -1.0/dx));
        }
    }

    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

(3.3c) To obtain the right-hand side vector of the linear system arising from the Galerkin discretization of (3.3.1) as described in subproblem (3.3a), one relies on the composite trapezoidal rule on $\mathcal{M} = \{x_0 = a, x_1, \ldots, x_N, x_{N+1} = b\}$ for numerical quadrature:

$$\int_a^b f(x)\,\mathrm{d}x \approx f(a)\frac{h}{2} + h\sum_{i=1}^{N} f(x_i) + f(b)\frac{h}{2}, \tag{3.3.2}$$

with h the meshsize.

In the template file `fem.cpp`, implement the function

```
void createRHS(Vector& rhs, FunctionPointer f, int N, double dx, const Vector& x)
```

(where `typedef Eigen::VectorXd Vector` and `typedef double(*FunctionPointer)(double)`), that computes the right-hand side for (3.3.1) when discretising with the standard basis of hat functions for linear finite elements, and when using the trapezoidal quadrature rule to compute the integrals. The argument `f` is a function pointer to the function f, the vector `x` contains the gridpoints, including the endpoints, and the other arguments are as in subproblem (3.3b).

Solution: Using the trapezoidal rule, one gets

$$\mathbf{F}_k = \int_a^b f(x) b_N^k(x)\,\mathrm{d}x \simeq h\hat{\sum}_{i=0}^{N+1} f(x_i) b_N^k(x_i), \quad k = 1, \ldots, N,$$

where the symbol $\hat{\sum}$ denotes half weight on the first and last summand. Since $b^k(x_i) \neq 0$ only for $i = k$, the sum collapses nicely into $\mathbf{F}_k \simeq h f(x_k)$ for $1 \leq k \leq N$.

See Listing 3.9 for the code.

Listing 3.9: createRHS

```
//!  Creates the right hand side for the finite element method in
    the exericse
//!  @param[out] rhs the resulting right hand side (or phi in the
    exercise)
//!  @param[in] f the function pointer to f
//!  @param[in] dx the cell length
//!  @param[in] x the x points
void createRHS(Vector& rhs, FunctionPointer f, int N, double
    dx, const Vector& x) {
    rhs.resize(N + 2);

    rhs[0] = 0.5 * dx * f(x[0]);
```

```
10
11      for (int i = 1; i < N; ++i) {
12          rhs[i] = dx * f(x[i]);
13      }
14
15      rhs[N+1] = 0.5 * dx * f(x[N+1]);
16
17  }
```

(3.3d) In the template file `fem.cpp`, implement the function

void femSolve(Vector& u, Vector& x, FunctionPointer f, **int** N, **double** a, **double** b, **double** ua = 0.0, **double** ub = 0.0)

that computes and stores in `u` the values of the Galerkin solution $u_N \in \mathcal{S}^0_{1,0}(\mathcal{M})$ at the nodes of the mesh $\mathcal{M}$ and returns them in the row vector `u`. The arguments `a` and `b` supply the domain $\Omega = [a, b]$, whereas `f` is a function pointer to the source function f. The argument `N` passes the number of interior nodes of the equidistant mesh. In the vector `u`, include also the boundary values of u.

Solution: See Listing 3.10 for the code, ignoring lines 39-43.

Listing 3.10: femSolve

```
1   //!  Solve the equation
2   //!
3   //!  −u''(x) = f(x)
4   //!
5   //!  using FEM on the interval [a, b].
6   //!
7   //!  @param[out] u at the end, will have all the values of u
8   //!  @param[out] x will be the x points
9   //!  @param[in] f should be a pointer to the function f
10  //!  @param[in] N number of inner cells to use
11  //!  @param[in] a endpoint on left side
12  //!  @param[in] b endpoint on right side
13  //!  @param[in] ua boundary value at a
14  //!  @param[in] ub boundary value at b
15  void femSolve(Vector& u, Vector& x, FunctionPointer f, int N,
        double a, double b,
16                  double ua = 0.0, double ub = 0.0) {
17
18      double dx = (b − a) / (N + 1);
19
20      // Fill x vector
21      x.setLinSpaced(N+2, a, b);
22
23      SparseMatrix A;
```

```
24      createGalerkinMatrixWithBoundary(A, N, dx);
25
26      Vector phi;
27      createRHS(phi, f, N, dx, x);
28
29      Eigen::SparseLU<SparseMatrix> solver;
30      solver.compute(A.block(1, 1, N, N));
31
32      if ( solver.info() != Eigen::Success) {
33          throw std::runtime_error("Could not decompose the
                matrix");
34      }
35
36      u.resize((N + 2));
37      u.setZero();
38
39      // Do boundary conditions
40      u[0] = ua;
41      u[N+1] = ub;
42
43      phi -= A * u;
44
45      u.segment(1, N) = solver.solve(phi.segment(1,N));
46
47  }
```

(3.3e) State and justify the asymptotic computational complexity of `linfegalerkinsol` in terms of the problem size parameter N.

Solution: The system matrix is *tridiagonal* and s.p.d. and, thus, Gaussian elimination without pivoting/Cholesky factorization can solve it with an asymptotic computational effort of $\mathcal{O}(N)$.

(3.3f) Plot the Galerkin solution u_N for $\Omega := [-\pi, \pi]$, $f(x) = \sin(x)$, and $N = 50, 100, 200$. To validate your code compare u_N with the exact analytic solution $u(x) = \sin(x)$.

Solution: The resulting graph should look like the following:

(3.3g) Extend your above implementation of `femSolve` using the optional arguments `ua`, `ub` that specify *boundary values* for the solution u of (3.3.1), supposing now u_a,$u_b \neq 0$. This means that now we seek to solve (3.3.1) under the constraints $u(a) = u_a$, $u(b) = u_b$.
HINT: Use the offset function technique to arrive at a modified right-hand side of the linear system of equations that incorporates the values u_a and u_b.

Solution: See Listing 3.10, lines 39-43, for the code.

(3.3h) Plot the Galerkin solution u_N for $\Omega := [-\pi, \pi]$, $f(x) = \cos(x)$, $u_a = -1$, $u_b = \frac{1}{2}$ and $N = 50$. To validate your code compare u_N with the exact analytic solution $u(x) = \cos(x) + \frac{3}{4\pi}x + \frac{3}{4}$.

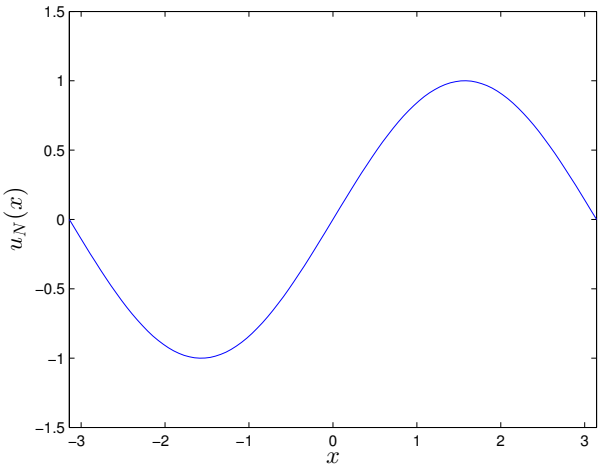

Figure 3.6: Plot for subproblem (3.3f).

Solution: The resulting graph should look like the following:

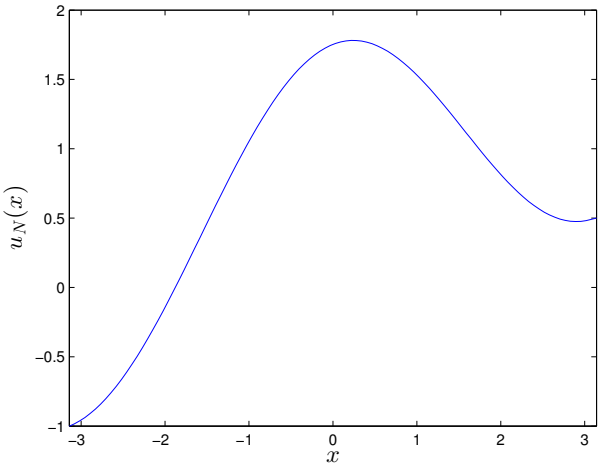

Figure 3.7: Plot for subproblem (3.3h).

Published on November 5.
To be submitted on November 17.

Last modified on November 24, 2015