Series 2

Numerical methods for PDEs
Last edited: March 16, 2017
Due date: 2017-04-04 at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=3089`.

Exercise 1 Linear Finite Elements in 1D

In class the finite element 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 practise the crucial steps for the linear variational problem

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

where $-\infty < a < b < \infty$ and $f \in C^0([a,b])$. 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.

1a)

Derive the stiffness matrix for (1), when using the trial and test space 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.

1b)

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

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

(with `typedef Eigen::SparseMatrix<double> SparseMatrix`), that computes the matrix `A` as in subproblem **1a)**. 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.

1c)

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

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

with h the mesh size.

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 (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 **1b)**.

1d)

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 finite element solution u_N 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.

1e)

State and justify the asymptotic computational complexity of `linfesol` in terms of the problem size parameter N.

1f)

Plot the finite element 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 $u(x) = \sin(x)$.

1g)

Extend your above implementation of `femSolve` using the optional arguments `ua`, `ub` that specify *boundary values* for the solution u of (1), supposing now u_a,$u_b \neq 0$. This means that now we seek to solve (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.

1h)

Plot the finite element 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 $u(x) = \cos(x) + \frac{3}{4\pi}x + \frac{3}{4}$.

Exercise 2 Linear Finite Elements for stationary reaction-diffusion equation in 2D

We consider the problem

$$-\nabla \cdot (\sigma \nabla u) + ru = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{3}$$
$$u(\boldsymbol{x}) = g(\boldsymbol{x}) \quad \text{on } \partial\Omega \tag{4}$$

where $f \in L^2(\Omega)$, $r \in \mathbb{R}_+$ is some positive constant and $\sigma : \Omega \longrightarrow \mathbb{R}_+$, $\sigma \in \mathcal{C}^1(\Omega)$.

In the folder `unittest` you can find routines to test your implementation tasks for this problem.

2a)

Write the variational formulation for (3)-(4).

We solve (3)-(4) by means of *linear finite elements* on triangular meshes of Ω. Let us denote by φ_N^i, $i = 0, \ldots, N-1$ the finite element basis functions (hat functions) associated to the vertices of a given mesh, with $N = N_V$ the total number of vertices. The finite element solution u_N to (3) can thus be expressed as

$$u_N(\boldsymbol{x}) = \sum_{i=0}^{N-1} \mu_i \varphi_N^i(\boldsymbol{x}), \tag{5}$$

where $\boldsymbol{\mu} = \{\mu_i\}_{i=0}^{N-1}$ is the vector of coefficients. Notice that we don't know μ_i if i corresponds to an interior vertex, but we know that $\mu_i = g(x_i)$ if x_i is a vertex on the boundary $\partial\Omega$.

Hint: Here and in the following, we use zero-based indices in contrast to the lecture notes.

Inserting φ_N^i, $i = 0, \ldots, N-1$ as test functions in the variational formulation from subproblem **2a)** we obtain the linear system of equations

$$\mathbf{A}\boldsymbol{\mu} = \mathbf{F}, \tag{6}$$

with $\mathbf{A} \in \mathbb{R}^{N \times N}$ and $\mathbf{F} \in \mathbb{R}^N$.

2b)

Write an expression for the entries of $\mathbf{A}$ and $\mathbf{F}$ in (6).

2c)

Complete the template file `shape.hpp` implementing the function

```
inline double lambda(int i, double x, double y)
```

which computes the the value a local shape function $\lambda_i(\boldsymbol{x})$, with i that can assume the values $0, 1$ or 2, on the reference element depicted in Fig. 1 at the point $\boldsymbol{x} = (x, y)$.

The convention for the local numbering of the shape functions is that $\lambda_i(\boldsymbol{x}_j) = \delta_{i,j}$, $i, j = 0, 1, 2$, with $\delta_{i,j}$ denoting the Kronecker delta.

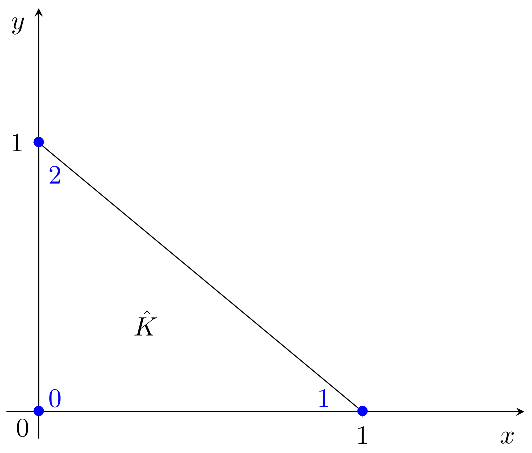

Figure 1: Reference element $\hat{K}$ for 2D linear finite elements.

2d)

Complete the template file `grad_shape.hpp` implementing the function

```
inline Eigen::Vector2d gradientLambda(const int i, double x, double y)
```

which returns the value of the derivatives (i.e. the gradient) of a local shape functions $\lambda_i(\boldsymbol{x})$, with i that can assume the values $0, 1$ or 2, on the reference element depicted in Fig. 1 at the point $\boldsymbol{x} = (x, y)$.

The routine `makeCoordinateTransform` contained in the file `coordinate_transform.hpp` computes the Jacobian matrix of the *linear* map $\Phi_l : \mathbb{R}^2 \to \mathbb{R}^2$ such that

$$\Phi_l \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{12} \end{pmatrix} = \boldsymbol{a}_1, \quad \Phi_l \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a_{21} \\ a_{22} \end{pmatrix} = \boldsymbol{a}_2,$$

where $\boldsymbol{a}_1, \boldsymbol{a}_2 \in \mathbb{R}^2$ are the two input arguments.

2e)

(**Core problem**) Complete the template file `stiffness_matrix.hpp` implementing the routine

```
template<class MatrixType, class Point>
void computeStiffnessMatrix(MatrixType& stiffnessMatrix,
                            const Point& a, const Point& b, const Point& c,
                            const std::function<double(double, double)>& sigma,
                            const double r)
```

that returns the *element stiffness matrix* for the bilinear form associated to (3) and for the triangle with vertices `a`, `b` and `c`.

Use the provided function `integrate`. It takes a function $f(x, y)$ as a parameter, and it returns the value of $\int_K f(x, y) dV$, where K is the triangle with vertices in $(0, 0)$, $(1, 0)$ and $(0, 1)$.

Hint: Do not forget to take into account the proper coordinate transformations!

Hint: Use the routine `gradientLambda` from subproblem **2d)** to compute the gradients and the routine `makeCoordinateTransform` to transform the gradients and to obtain the area of a triangle.

The routine `integrate` in the file `integrate.hpp` uses a quadrature rule to compute the approximate value of $\int_{\hat{K}} f(\hat{\boldsymbol{x}})\, d\hat{\boldsymbol{x}}$, where f is a function, passed as input argument.

2f)

Complete the template file `load_vector.hpp` implementing the routine

```
template<class Vector, class Point>
void computeLoadVector(Vector& loadVector, const Point& a, const Point& b,
                       const Point& c, const std::function<double(double, double)>& f)
```

that returns the *element load vector* for the linear form associated to (3), for the triangle with vertices `a`, `b` and `c`, and where `f` is a function handler to the right-hand side of (3).

Hint: Use the routine `lambda` from subproblem **2c)** to compute values of the shape functions on the reference element, and the routines `makeCoordinateTransform` and `integrate` from the handout to map the points to the physical triangle and to compute the integrals.

2g)

Complete the template file `stiffness_matrix_assembly.hpp` implementing the routine

```
template<class Matrix>
void assembleStiffnessMatrix(Matrix& A, const Eigen::MatrixXd& vertices,
                             const Eigen::MatrixXi& triangles,
                             const std::function<double(double, double)>& sigma,
                             double r)
```

to compute the finite element matrix $\mathbf{A}$ as in (6). The input argument `vertices` is a $N_V \times 3$ matrix of which the i-th row contains the coordinates of the i-th mesh vertex, $i = 0, \ldots, N_V - 1$, with N_V the number of vertices. The input argument `triangles` is a $N_T \times 3$ matrix where the i-th row contains the *indices* of the vertices of the i-th triangle, $i = 0, \ldots, N_T - 1$, with N_T the number of triangles in the mesh.

Hint: Use the routine computeStiffnessMatrix from subproblem **2e)** to compute the local stiffness matrix associated to each element.

Hint: Use the sparse format to store the matrix A.

2h)

Complete the template file `load_vector_assembly.hpp` implementing the routine

```
void assembleLoadVector(Eigen::VectorXd& F, const Eigen::MatrixXd& vertices,
                        const Eigen::MatrixXi& triangles,
                        const std::function<double(double, double)>& f)
```

to compute the right-hand side vector **F** as in (6). The input arguments vertices and triangles are as in subproblem **2g)**, and f is as in subproblem **2f)**.

Hint: Proceed in a similar way as for assembleStiffnessMatrix and use the routine computeLoadVector from subproblem **2f)**.

The routine

```
void setDirichletBoundary(Eigen::VectorXd& u, Eigen::VectorXi& interiorVertexIndices,
                          const Eigen::MatrixXd& vertices,
                          const Eigen::MatrixXi& triangles,
                          const std::function<double(double, double)>& g)
```

implemented in the file `dirichlet_boundary.hpp` provided in the handout does the following:

- it gets in input the matrices vertices and triangles as defined in subproblem **2g)** and the function handle g to the boundary data, i.e. to g such that $u = g$ on $\partial\Omega$;
- it returns in the vector interiorVertexIndices the indices of the interior vertices, that is of the vertices that are *not* on the boundary $\partial\Omega$;
- if $\boldsymbol{x}_i$ is a vertex on the boundary, then it sets u(i)=$g(\boldsymbol{x}_i)$.

2i)

Complete the template file `fem_solve.hpp` with the implementation of the function

```
int solveFiniteElement(Vector& u, const Eigen::MatrixXd& vertices,
                       const Eigen::MatrixXi& triangles,
                       const std::function<double(double, double)>& f,
                       const std::function<double(double, double)>& sigma,
                       const std::function<double(double, double)>& g, double r)
```

This function takes in input the matrices vertices, triangles as defined in the previous subproblems, the function handle f to the right-hand side f, the function handle sigma as σ in (3), and the function handle g to the boundary data. The output argument u has to contain, at the end of the function, the finite element solution u_N to (3). The function returns the number of degrees of freedom (namely, the number of interior vertices).

Hint: Use the routines assembleStiffnessMatrix and assembleLoadVector from subproblems **2g)** and **2h)**, respectively, to obtain the matrix $\mathbf{A}$ and the vector $\mathbf{F}$ as in (6), and then use the provided routine setDirichletBoundary to set the boundary values of u to the corresponding values of g and to select the free degrees of freedom.

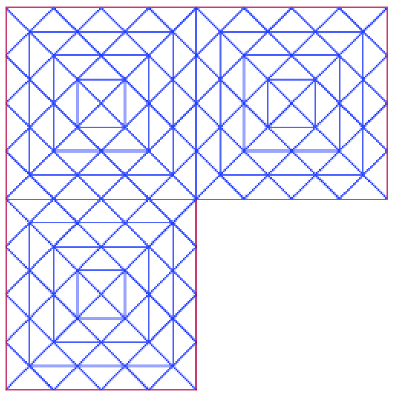

Figure 2: Domain for subproblems **2j)** and **2k)**.

2j)

Run the routine `solveL` contained in the file `Lshape.hpp` to compute the finite element solution to (3)-(4) when Ω is the L-shaped domain $\Omega = (-1,1)^2 \setminus ((0,1) \times (-1,0))$, as depicted in Fig. 2, and $r = 0.5$. The forcing term is given by $f(\boldsymbol{x}) = 0$, $\sigma \equiv 1$, the boundary condition by $g = u_{|\partial\Omega}$, with u the exact solution, which, in polar coordinates, is given by $u(r, \vartheta) = r^{\frac{2}{3}} \sin(\frac{2}{3}\vartheta)$, for $r \geq 0$ and $\vartheta \in [0, \frac{3}{2}\pi]$. The mesh used is Lshape_5.mesh. Use then the routine plot_on_mesh.py to produce a plot of the solution.

The functions `computeL2Difference` and `computeH1Difference`, contained in the files L2_norm.hpp and H1_norm.hpp, respectively, are defined as

```
double computeL2Difference(const Eigen::MatrixXd& vertices,
                           const Eigen::MatrixXi& triangles,
                           const Eigen::VectorXd& u1,
                           const std::function<double(double, double)>& u2)
```

```
double computeH1Difference(const Eigen::MatrixXd& vertices,
                           const Eigen::MatrixXi& triangles,
                           const Eigen::VectorXd& u1,
                           const std::function<Eigen::Vector2d(double, double)>& u2grad)
```

In both cases, the argument `u1` is considered to be a vector corresponding to a linear finite element function u_1, and thus containing the value of this function in the vertices of a mesh. The argument u2 in computeL2Difference is a function handle to a scalar function u_2 which is known analytically. The argument u2grad in computeH1Difference is a function handle to a function gradient ∇u_2, supposed to be known analytically. Then the routines computeL2Difference and computeH1Difference compute an approximation to $\|u_1 - u_2\|_{L^2(\Omega)}$ and $|u_1 - u_2|_{H^1(\Omega)} = \|\nabla u_1 - \nabla u_2\|_{L^2(\Omega)}$, respectively.

2k)

(Core problem) Complete the template file `convergence.hpp` by implementing the routine

```
void convergenceAnalysis(const std::string& baseMeshName, int maxLevel
                         const std::function<double(double, double)> f,
                         const std::function<double(double, double)>& sigma,
                         const std::function<double(double, double)> g, double r,
                         const std::function<double(double, double)> exactSol,
                         const std::function<Eigen::Vector2d(double,double)> exactSol_grad
    ↪ )
```

to compute the convergence analysis for a sequence of meshes `baseMeshName_X`, with X=0.. maxLevel. This means, for each mesh you will compute the difference between the exact solution and the finite elements solution you get with the given input in terms of the L^2 and H^1-norms. This routine should write a file with the number of degrees of freedom used at each convergence step, a file with the computed L^2-error norms and a file with the computed H^1-error seminorms.

Use the functions `computeL2Difference` and `computeH1Difference` in order to do this.

Hint: Use the function `solveFiniteElement` to get the number of degrees of freedom on each mesh.

2l)

Run the routine convergenceAnalysis to perform a convergence study for the finite element solution to (3)-(4) for the meshes Lshape and square and the data contained in `fem2d.cpp`.

Which convergence orders with respect to the number of degrees of freedom do you observe?