Series 2 Warmup

Numerical methods for PDEs
Last edited: March 15, 2017
Due date: None at 23:59

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

This is a warmup problem. You do **NOT need to hand in** this problem.

Exercise 1 Linear Finite Elements for the Poisson equation in 2D

We consider the problem

$$-\Delta u = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{1}$$

$$u(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega \tag{2}$$

where $f \in L^2(\Omega)$.

1a)

Write the variational formulation for (1)-(2).

We solve (1)-(2) 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 (1) can thus be expressed as

$$u_N(\boldsymbol{x}) = \sum_{i=0}^{N-1} \mu_i \varphi_N^i(\boldsymbol{x}), \tag{3}$$

where $\boldsymbol{\mu} = \{\mu_i\}_{i=0}^{N-1}$ is the vector of coefficients. Notice that we don't know μ_i if i is an interior vertex, but we know that $\mu_i = 0$ if 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, $i = 0, \dots, N-1$ as test functions in the variational formulation from subproblem **1a)** we obtain the linear system of equations

$$\mathbf{A}\boldsymbol{\mu} = \mathbf{F}, \tag{4}$$

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

1b)

Write an expression for the entries of $\mathbf{A}$ and $\mathbf{F}$ in (4).

1c)

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.

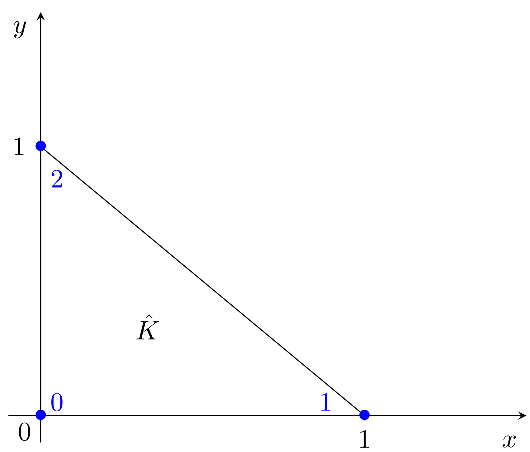

Figure 1: Reference element $\hat{K}$ for 2D linear finite elements.

1d)

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.

1e)

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)
```

that returns the *element stiffness matrix* for the bilinear form associated to (1) and for the triangle with vertices `a`, `b` and `c`.

Hint: Use the routine `gradientLambda` from subproblem **1d)** to compute the gradients and the routine `makeCoordinateTransform` to transform the gradients and to obtain the area of a triangle.

Hint: You do not have to analytically compute the integrals for the product of basis functions; instead, you can 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 $(0, 1)$. Do not forget to take into account the proper coordinate transforms!

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.

1f)

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 (1), for the triangle with vertices a, b and c, and where f is a function handler to the right-hand side of (1).

Hint: Use the routine `lambda` from subproblem **1c)** 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.

1g)

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)
```

to compute the finite element matrix $\mathbf{A}$ as in (4). 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 **1e)** to compute the local stiffness matrix associated to each element.

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

1h)

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 $\mathbf{F}$ as in (4). The input arguments vertices and triangles are as in subproblem **1g)**, and f is an in subproblem **1f)**.

Hint: Proceed in a similar way as for assembleStiffnessMatrix and use the routine computeLoadVector from subproblem **1f)**.

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 **1g)** and the function handle g to the boundary data, i.e. to g such that $u = g$ on $\partial\Omega$ (in our case $g \equiv 0$);
- 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)=$g(\boldsymbol{x}_i)$, that is, in our case, it sets to 0 the entries of the vector u corresponding to vertices on the boundary.

1i)

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)
```

This function takes in input the matrices vertices, triangles as defined in the previous subproblems, and the function handle f to the right-hand side f in (1). The output argument u has to contain, at the end of the function, the finite element solution u_N to (1).

Hint: Use the routines assembleStiffnessMatrix and assembleLoadVector from subproblems **1g)** and **1h)**, respectively, to obtain the matrix $\mathbf{A}$ and the vector $\mathbf{F}$ as in (4), and then use the provided routine setDirichletBoundary to set the boundary values of u to zero and to select the free degrees of freedom.

1j)

Run the code in the file `fem2d.cpp` to compute the finite element solution to (1) when $\Omega = [0, 1]^2$ is the unit square, the forcing term is given by $f(\boldsymbol{x}) = 2\pi^2 \sin(\pi x)\sin(\pi y)$ and the mesh is square_5.
↪ mesh. Use then the routine plot_on_mesh.py to produce a plot of the solution.