S. Mishra
K. O. Lye
L. Scarabosio

Autumn Term 2015

# Computational Methods for Engineering Applications II

ETH Zürich
D-MATH

## Homework Problem Sheet 4

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 **8 points**. (The total number of points over all assignments will be around 20).

## Problem 4.1 Linear Finite Elements for the Poisson equation in 2D

We consider the problem

$$-\Delta u = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{4.1.1}$$

$$u(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega \tag{4.1.2}$$

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

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

**(4.1a)** Write the variational formulation for (4.1.1)-(4.1.2).

We solve (4.1.1)-(4.1.2) by means of Galerkin discretization based on *piecewise linear finite elements* on triangular meshes of $\Omega$. Let us denote by $\varphi_N^i$, $i = 0, \dots, N-1$ 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 (4.1.1) can thus be expressed as

$$u_N(\boldsymbol{x}) = \sum_{i=0}^{N-1} \mu_i \varphi_N^i(\boldsymbol{x}), \tag{4.1.3}$$

where $\boldsymbol{\mu} = \{\mu_i\}_{i=0}^{N-1}$ is the vector of coefficients. Notice that we don't know $\mu_i$ 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 $\varphi_N^i$, $i = 0, \dots, N-1$ as test functions in the variational formulation from subproblem (4.1a) we obtain the linear system of equations

$$\mathbf{A}\boldsymbol{\mu} = \mathbf{F}, \tag{4.1.4}$$

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

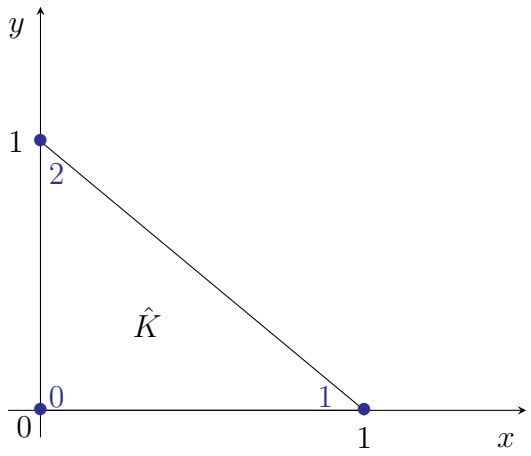


Figure 4.1: Reference element $\hat{K}$ for 2D linear finite elements.

**(4.1b)** Write an expression for the entries of $\mathbf{A}$ and $\mathbf{F}$ in (4.1.4).

**(4.1c)** (**Core problem**) 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. 4.1 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.

**(4.1d)** (**Core problem**) 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. 4.1 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.

**(4.1e)** (**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)
```

that returns the *element stiffness matrix* for the bilinear form associated to (4.1.1) and for the triangle with vertices a, b and c.

HINT: Use the routine gradientLambda from subproblem (4.1d) 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 $\int_{\hat{K}} f(\hat{\boldsymbol{x}})\,\mathrm{d}\hat{\boldsymbol{x}}$, where $f$ is a function, passed as input argument.

**(4.1f)** (**Core problem**) 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 (4.1.1), for the triangle with vertices a, b and c, and where f is a function handler to the right-hand side of (4.1.1).

HINT: Use the routine lambda from subproblem (4.1c) 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.

**(4.1g)** (**Core problem**) 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 (4.1.4). 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 (4.1e) to compute the local stiffness matrix associated to each element.

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

**(4.1h)** (**Core problem**) 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 (4.1.4). The input arguments vertices and triangles are as in subproblem (4.1g), and f is an in subproblem (4.1f).

HINT: Proceed in a similar way as for assembleStiffnessMatrix and use the routine computeLoadVector from subproblem (4.1f).

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 (4.1g) 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.

**(4.1i)** (**Core problem**) 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)>& g) .

This function takes in input the matrices vertices, triangles as defined in the previous subproblems, the function handle f to the right-hand side $f$ in (4.1.1) and the function handle g to the boundary data. For the moment, consider g to be the zero function. The output argument u has to contain, at the end of the function, the finite element solution $u_N$ to (4.1.1).

HINT: Use the routines assembleStiffnessMatrix and assembleLoadVector from subproblems (4.1g) and (4.1h), respectively, to obtain the matrix $\mathbf{A}$ and the vector $\mathbf{F}$ as in (4.1.4), and then use the provided routine setDirichletBoundary to set the boundary values of u to zero and to select the free degrees of freedom.

**(4.1j)** (**Core problem**) Run the routine solveSquare contained in the file square.hpp to compute the finite element solution to (4.1.1) 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.

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 $\nabla u_2$, 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.

The routine convergenceAnalysis contained in the file convergence.hpp performs a convergence study calling the routines computeL2Difference and computeH1Difference. This function writes 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.

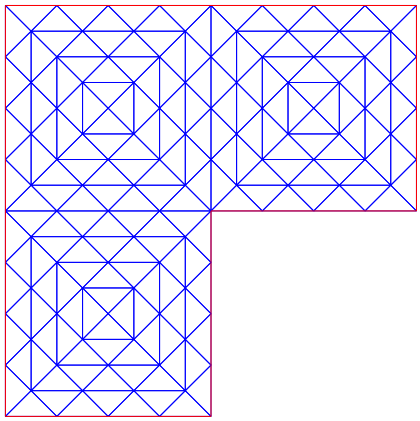

Figure 4.2: Domain for subproblems (4.1m)-(4.1n).

**(4.1k)** Run the routine convergenceAnalysis to perform a convergence study for the finite element solution to (4.1.1)-(4.1.2). For both errors, make a double logarithmic plot of error versus number of degrees of freedom. Which convergence orders with respect to the number of degrees of freedom do you observe?

We now consider the problem

$$-\Delta u = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{4.1.5}$$

$$u(\boldsymbol{x}) = g(\boldsymbol{x}) \quad \text{on } \partial\Omega \tag{4.1.6}$$

where $f \in L^2(\Omega)$ and now $g$ is a continuous function which is in general not zero on the boundary.

**(4.1l)** Modify your implementation of the routine solveFiniteElement implemented in subproblem (4.1i) in order to compute the finite element solution to (4.1.5)-(4.1.6).

HINT: Use the information contained in the output of the routine setDirichletBoundary and use the offset function technique.

**(4.1m)** Run the routine solveL contained in the file Lshape.hpp to compute the finite element solution to (4.1.5)-(4.1.6) when $\Omega$ is the L-shaped domain $\Omega = (-1,1)^2 \setminus ((0,1) \times (-1,0))$, as depicted in Fig. 4.2. The forcing term is given by $f(\boldsymbol{x}) = 0$, 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.

**(4.1n)** Run the routine convergenceAnalysis to perform a convergence study for the finite element solution to (4.1.5)-(4.1.6). For both errors, make a double logarithmic plot of error versus number of degrees of freedom. Which convergence orders with respect to the number of degrees of freedom do you observe?

## Problem 4.2 Green's Formula

To derive the variational formulation from a PDE, we needed multi-dimensional integration by parts as expressed through Green's first formula. In this problem we study the derivation of Green's formula, thus practicing elementary vector analysis and the application of Gauss' theorem.

Now prove Green's formula for $\Omega \subset \mathbb{R}^2$

$$-\int_\Omega \nabla \cdot \mathbf{j}\, v \,\mathrm{d}\mathbf{x} = -\int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S + \int_\Omega \mathbf{j} \cdot \nabla v \,\mathrm{d}\mathbf{x},$$

where $\mathbf{j} \in \mathcal{C}^1(\overline{\Omega})^2$ and $v \in \mathcal{C}^1(\overline{\Omega})$.

HINT: Use Gauss' theorem.

$$\int_\Omega \nabla \cdot \mathbf{F} \,\mathrm{d}\mathbf{x} = \int_{\partial\Omega} \mathbf{F} \cdot \mathbf{n} \,\mathrm{d}S,$$

where $\mathbf{F} \in \mathcal{C}^1(\overline{\Omega})^2$.

Published on November 25.
To be submitted on December 11.

Last modified on December 3, 2015