Series 2 Warmup

ETH zürich

Numerical methods for PDEs
Last edited: March 14, 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).

Solution: We multiply both the left handside and right handside of (1) by a test function v. Applying Green's formula for integration by parts on the left handside we get:

$$-\int_\Omega \Delta u(\boldsymbol{x}) v(\boldsymbol{x}) = \int_\Omega \nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x})\, d\boldsymbol{x} - \int_{\partial\Omega} \frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{x}) v(\boldsymbol{x})\, d\boldsymbol{x}.$$

Since u satisfies Dirichlet boundary conditions, the test functions belong to $H_0^1(\Omega)$ and thus the

boundary integral in the above expression vanishes. The variational formulation results then:

$$\text{Find } u \in V = H_0^1(\Omega) \text{ such that}$$
$$\int_\Omega \nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, d\boldsymbol{x} \text{ for all } v \in H_0^1(\Omega),$$

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, $i = 0, \ldots, N-1$ 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).

Solution: We have

$$\mathbf{A}_{ij} = \int_\Omega \nabla \varphi_N^j(\boldsymbol{x}) \cdot \nabla \varphi_N^i(\boldsymbol{x})\, d\boldsymbol{x} \quad \text{and } \mathbf{F}_i = \int_\Omega f(\boldsymbol{x}) \varphi_N^i(\boldsymbol{x})\, d\boldsymbol{x},$$

for $i, j = 0, \ldots, N-1$.

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.

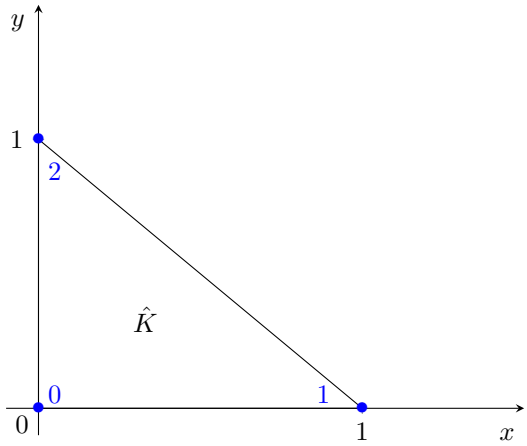

Figure 1: Reference element $\hat{K}$ for 2D linear finite elements.

Solution: See listing 1 for the code.

Listing 1: Implementation for lambda

```
#pragma once


//! The shape function (on the reference element)
//!
//! We have three shape functions.
//!
//! lambda(0, x, y) should be 1 in the point (0,0) and zero in (1,0) and (0,1)
//! lambda(1, x, y) should be 1 in the point (1,0) and zero in (0,0) and (0,1)
//! lambda(2, x, y) should be 1 in the point (0,1) and zero in (0,0) and (1,0)
//!
//! @param i integer between 0 and 2 (inclusive). Decides which shape function to
    ↪  return.
//! @param x x coordinate in the reference element.
//! @param y y coordinate in the reference element.
inline double lambda(int i, double x, double y) {
  //// NPDE_START_TEMPLATE
    if (i == 0) {
        return 1 - x - y;
    } else if (i == 1) {
        return x;
    } else {
```

```
        return y;
    }
    //// NPDE_RETURN_TEMPLATE
    //// NPDE_END_TEMPLATE
}
```

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 $\boldsymbol{x} = (x, y)$.

Solution: See listing 2 for the code.

Listing 2: Implementation for gradientLambda

```
#pragma once
#include <Eigen/Core>


//! The gradient of the shape function (on the reference element)
//!
//! We have three shape functions
//!
//! @param i integer between 0 and 2 (inclusive). Decides which shape function to
    ↪ return.
//! @param x x coordinate in the reference element.
//! @param y y coordinate in the reference element.
inline Eigen::Vector2d gradientLambda(const int i, double x, double y) {
  //// NPDE_START_TEMPLATE
    return Eigen::Vector2d(-1 + (i > 0) + (i==1),
                           -1 + (i > 0) + (i==2));
    //// NPDE_END_TEMPLATE
    return Eigen::Vector2d(0,0); //remove when implemented
}
```

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!

Solution: See listing 3 for the code.

Listing 3: Implementation for computeStiffnessMatrix

```
//! Evaluate the stiffness matrix on the triangle spanned by
//! the points (a, b, c).
//!
//! Here, the stiffness matrix A is a 3x3 matrix
//!
//! $$A_{ij} = \int_{K} ( \nabla \lambda_i^K(x, y) \cdot \nabla \lambda_j^K(x, y)
    ↪ \; dV$$
//!
//! where $K$ is the triangle spanned by (a, b, c).
//!
//! @param[out] stiffnessMatrix should be a 3x3 matrix
//! At                      the end, will contain the integrals above.
//!
//! @param[in] a the first corner of the triangle
//! @param[in] b the second corner of the triangle
//! @param[in] c the third corner of the triangle
template<class MatrixType, class Point>
void computeStiffnessMatrix(MatrixType& stiffnessMatrix,
                            const Point& a,
                            const Point& b,
                            const Point& c)
{
```

```
    Eigen::Matrix2d coordinateTransform = makeCoordinateTransform(b - a, c - a);
    double volumeFactor = std::abs(coordinateTransform.determinant());
    Eigen::Matrix2d elementMap = coordinateTransform.inverse().transpose();
    //// NPDE_START_TEMPLATE
    for (int i = 0; i < 3; ++i) {
        for (int j = i; j < 3; ++j) {

            stiffnessMatrix(i, j) = integrate([&](double x, double y) {
                Eigen::Vector2d gradLambdaI = elementMap * gradientLambda(i, x, y);
                Eigen::Vector2d gradLambdaJ = elementMap * gradientLambda(j, x, y);

                auto lambdaI = lambda(i, x, y);
                auto lambdaJ = lambda(j, x, y);

                return volumeFactor * gradLambdaI.dot(gradLambdaJ);

            });
        }
    }

    // Make symmetric (we did not need to compute these value above)
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < i; ++j) {
            stiffnessMatrix(i, j) = stiffnessMatrix(j, i);
        }
    }
    //// NPDE_END_TEMPLATE


}
```

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.

Solution: See listing 4 for the code.

Listing 4: Implementation for computeLoadVector

```
//! Evaluate the load vector on the triangle spanned by
//! the points (a, b, c).
//!
//! Here, the load vector is a vector $(v_i)$ of
//! three components, where
//!
//! $$v_i = \int_{K} \lambda_i^K(x, y) f(x, y) \; dV$$
//!
//! where $K$ is the triangle spanned by (a, b, c).
//!
//! @param[out] loadVector should be a vector of length 3.
//! At                      the end, will contain the integrals above.
//!
//! @param[in] a the first corner of the triangle
//! @param[in] b the second corner of the triangle
//! @param[in] c the third corner of the triangle
//! @param[in] f the function f (LHS).
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)
{
    Eigen::Matrix2d coordinateTransform = makeCoordinateTransform(b - a, c - a);
    double volumeFactor = std::abs(coordinateTransform.determinant());
    //// NPDE_START_TEMPLATE
    for (int i = 0; i < 3; ++i) {
        loadVector(i) = integrate([&](double x, double y) {
            Eigen::Vector2d z = coordinateTransform * Eigen::Vector2d(x, y) + Eigen
                ↪ ::Vector2d(a(0), a(1));
            return f(z(0), z(1)) * lambda(i, x, y) * volumeFactor;
        });
    }

    //// NPDE_END_TEMPLATE

}
```

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.

Solution: See listing 5 for the code.

Listing 5: Implementation for assembleStiffnessMatrix

```
//! Assemble the stiffness matrix
//! for the linear system
//!
//! @param[out] A will at the end contain the Galerkin matrix
//! @param[in] vertices a list of triangle vertices
//! @param[in] triangles a list of triangles
template<class Matrix>
void assembleStiffnessMatrix(Matrix& A, const Eigen::MatrixXd& vertices,
                             const Eigen::MatrixXi& triangles)
{

    const int numberOfElements = triangles.rows();
    A.resize(vertices.rows(), vertices.rows());

    std::vector<Triplet> triplets;

    triplets.reserve(numberOfElements * 3 * 3);
    //// NPDE_START_TEMPLATE
    for (int i = 0; i < numberOfElements; ++i) {
        auto& indexSet = triangles.row(i);

        const auto& a = vertices.row(indexSet(0));
        const auto& b = vertices.row(indexSet(1));
        const auto& c = vertices.row(indexSet(2));
```

```
        Eigen::Matrix3d stiffnessMatrix;
        computeStiffnessMatrix(stiffnessMatrix, a, b, c);

        for (int n = 0; n < 3; ++n) {
            for (int m = 0; m < 3; ++m) {
                auto triplet = Triplet(indexSet(n), indexSet(m), stiffnessMatrix(n,
                    ↪ m));
                triplets.push_back(triplet);
            }
        }
    }
    //// NPDE_END_TEMPLATE
    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

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)**.

Solution: See listing 6 for the code.

Listing 6: Implementation for assembleLoadVector

```
//! Assemble the load vector into the full right hand side
//! for the linear system
//!
//! @param[out] F will at the end contain the RHS values for each vertex.
//! @param[in] vertices a list of triangle vertices
//! @param[in] triangles a list of triangles
//! @param[in] f the RHS function f.
void assembleLoadVector(Eigen::VectorXd& F,
                        const Eigen::MatrixXd& vertices,
                        const Eigen::MatrixXi& triangles,
                        const std::function<double(double, double)>& f)
{
    const int numberOfElements = triangles.rows();
```

```
    F.resize(vertices.rows());
    F.setZero();
    //// NPDE_START_TEMPLATE
    for (int i = 0; i < numberOfElements; ++i) {
        const auto& indexSet = triangles.row(i);

        const auto& a = vertices.row(indexSet(0));
        const auto& b = vertices.row(indexSet(1));
        const auto& c = vertices.row(indexSet(2));

        Eigen::Vector3d elementVector;
        computeLoadVector(elementVector, a, b, c, f);

        for (int i = 0; i < 3; ++i) {
           F(indexSet(i)) += elementVector(i);
        }
    }
    //// NPDE_END_TEMPLATE
}
```

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 **A** and the vector **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.

Solution: See listing 7 for the code.

Listing 7: Implementation for solveFiniteElement

```
//! Solve the FEM system.
//!
//! @param[out] u will at the end contain the FEM solution.
//! @param[in] vertices list of triangle vertices for the mesh
//! @param[in] triangles list of triangles (described by indices)
//! @param[in] f the RHS f (as in the exercise)
//! return number of degrees of freedom (without the boundary dofs)
int solveFiniteElement(Vector& u,
    const Eigen::MatrixXd& vertices,
    const Eigen::MatrixXi& triangles,
    const std::function<double(double, double)>& f)
{
    SparseMatrix A;
    //// NPDE_START_TEMPLATE
    assembleStiffnessMatrix(A, vertices, triangles);
    //// NPDE_END_TEMPLATE

    Vector F;
    //// NPDE_START_TEMPLATE
    assembleLoadVector(F, vertices, triangles, f);
    //// NPDE_END_TEMPLATE

    u.resize(vertices.rows());
    u.setZero();
    Eigen::VectorXi interiorVertexIndices;

    auto zerobc = [](double x, double y){ return 0;};
    // set homogeneous Dirichlet Boundary conditions
    //// NPDE_START_TEMPLATE
    setDirichletBoundary(u, interiorVertexIndices, vertices, triangles, zerobc);
    F -= A * u;
    //// NPDE_END_TEMPLATE

    SparseMatrix AInterior;
```

```
    igl::slice(A, interiorVertexIndices, interiorVertexIndices, AInterior);
    Eigen::SimplicialLDLT<SparseMatrix> solver;

    Vector FInterior;

    igl::slice(F, interiorVertexIndices, FInterior);

    //initialize solver for AInterior
    //// NPDE_START_TEMPLATE
    solver.compute(AInterior);

    if (solver.info() != Eigen::Success) {
        throw std::runtime_error("Could not decompose the matrix");
    }
    //// NPDE_END_TEMPLATE

    //solve interior system
    //// NPDE_START_TEMPLATE
    Vector uInterior = solver.solve(FInterior);
    igl::slice_into(uInterior, interiorVertexIndices, u);
    //// NPDE_END_TEMPLATE

    return interiorVertexIndices.size();

}
```

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.

Solution: See Fig. 2 for the plot.

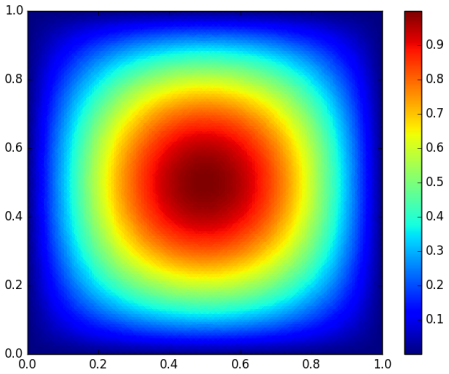

Figure 2: Solution plot for subproblem **1j)**.