Series 5

ETH zürich

Computational Methods for
Engineering Applications
Last edited: November 19, 2020
Due date: November 29 at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=13412`.

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

Hint: This exercise has *unit tests* which can be used to test your solution. To run the unit tests, run the executable `unittest`. Note that correct unit tests are *not* a guarantee for a correct solution. In some rare cases, the solution can be correct even though the unit tests do not pass (always check the output values, and if in doubt, ask the teaching assistant!)

1a)

Write the variational formulation for (5)-(6).

We solve (5)-(6) by means of *linear finite elements* on triangular meshes of Ω. Let us denote by φ_i^N, $i = 0, \ldots, N-1$ 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 (5) can thus be expressed as

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

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

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

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 (11).

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. 2 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.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestShapeFunction`.

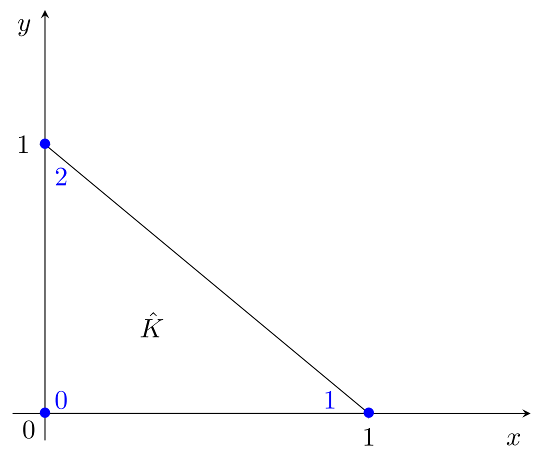

Figure 1: Reference element $\hat{K}$ for 2D linear finite elements.

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. 2 at the point $\boldsymbol{x} = (x, y)$.**Hint:** You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestGradientShapeFunction`.

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)

(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 (5) and for the triangle with vertices `a`, `b` and `c`.

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.

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!

Hint: You will need to give a parameter f to `integrate` representing the function to be integrated. You can define your own routine for that, or you can use an "anonymous function" (or "lambda expression"), e.g.:

```
auto f = [&](double x, double y){ return /*something depending on (x,y), i, j...*/};
```

which produces a function pointer in object `f` (that one can call as a normal function).

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestStiffnessMatrix`.

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)

(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 (5), for the triangle with vertices `a`, `b` and `c`, and where `f` is a function handler to the right-hand side of (5).

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.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestElementVector`.

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 (11). The input argument vertices is a $N_V \times 2$ 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.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestAssembleStiffnessMatrix`.

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 (11). The input arguments vertices and triangles are as in subproblem **2g)**, and f is an in subproblem **2f)**.

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

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestAssembleLoadVector`.

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 $\partial\Omega$ (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)

(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)
```

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 (5). The output argument u has to contain, at the end of the function, the finite element solution u_N to (5).

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 (11), and then use the provided routine setDirichletBoundary to set the boundary values of u to zero and to select the free degrees of freedom.

Hint: You will need to give a parameter g to `setDirichletBoundary` representing the boundary condition. In our case, this is an identically zero function. You could define your own routine for that, or you can use an "anonymous function" (or "lambda expression"), e.g.:

```
auto zerobc = [](double x, double y){ return 0;};
```

which produces a function pointer in object `zerobc` (that one can call as a normal function).

1j)

Run the code in the file `fem2d.cpp` to compute the finite element solution to (5) 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.

Exercise 2 Quadratic Finite Elements for the Poisson equation in 2D

We consider the problem

$$-\Delta u = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{5}$$
$$u(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega \tag{6}$$

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

We know that its variational formulation is given by: Find $u \in V = H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, d\boldsymbol{x}, \quad \text{for all } v \in H_0^1(\Omega). \tag{7}$$

We solve (7) by means of *quadratic finite elements* on triangular meshes $\mathcal{M}$ of Ω. Consequently, we consider the following finite-dimensional subspace of $H_0^1(\Omega)$:

$$V^h = \big\{ w\colon \Omega \to \mathbb{R}\colon w \text{ is continuous}, w = 0 \text{ on } \partial\Omega,$$
$$\text{and } w|_K \text{ is a } \textbf{second order polynomial} \ \ \forall\, K \in \mathcal{M} \big\}.$$

This means we now consider two types of basis functions:

- The ones associated to the vertices of the given mesh

$$b_i(\boldsymbol{x}_j) := \begin{cases} 1, & i = j \\ 0, & else \end{cases}, \quad i = 0, ..., N_V - 1, \tag{8}$$

  with N_V the total number of vertices.

- The basis functions associated to the midpoint $\mathbf{m}_i$ of each edges i of the given mesh

$$\psi_i(\mathbf{m}_j) := \begin{cases} 1, & i = j \\ 0, & else \end{cases}, \quad i = 0, ..., N_E - 1, \tag{9}$$

  with N_E the total number of edges.

We therefore have degrees of freedom associated to vertices and edges, and a total number of $N = N_V + N_E$ basis functions. Let us order our basis functions by first considering vertices and then edges. In other words

$$\varphi_i^N := \begin{cases} b_i, & i = 0, \ldots, N_V - 1 \\ \psi_{i-N_V}, & i = N_V, \ldots, N - 1. \end{cases}$$

The finite element solution u_N to (5) can thus be expressed as

$$
\begin{aligned}
u_N(\boldsymbol{x}) &= \sum_{i=0}^{N_V-1} \mu_i b_i(\boldsymbol{x}) + \sum_{i=0}^{N_E-1} \mu_{i+N_V} \psi_i(\boldsymbol{x}) \\
&= \sum_{i=0}^{N-1} \mu_i \varphi_i^N(\boldsymbol{x}),
\end{aligned} \tag{10}
$$

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 degree of freedom, but we know that $\mu_i = 0$ if i is a vertex or edge on the boundary $\partial\Omega$.

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

Inserting φ_i^N, $i = 0, \ldots, N-1$ as test functions in the variational formulation from (7) we obtain the linear system of equations

$$
\mathbf{A}\boldsymbol{\mu} = \mathbf{F}, \tag{11}
$$

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

2a)

Write an expression in terms of b_i, $i = 0, \ldots, N_V - 1$ and ψ_i, $i = 0, \ldots, N_E - 1$, for the entries of $\mathbf{A}$ and $\mathbf{F}$ in (11).

The convention for the local numbering of the shape functions is given in the following figure

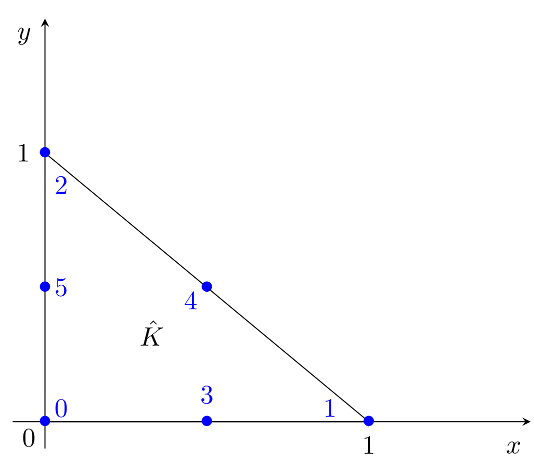

Figure 2: Reference element $\hat{K}$ for 2D quadratic finite elements.

2b)

(**Core problem**) Complete the template file `shape.hpp` implementing the function

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

which computes the the value a local shape function $\varphi_i^K(\boldsymbol{x})$, with $i = 0, \ldots, 5$, on the reference element depicted in Fig. 2 at the point $\boldsymbol{x} = (x, y)$.

Use your previous *linear* finite elements implementation and the following formulas to complete this task:

$$
\begin{aligned}
\varphi_0^K(\boldsymbol{x}) &= (2\lambda_0(\boldsymbol{x}) - 1)\lambda_0(\boldsymbol{x}), \\
\varphi_1^K(\boldsymbol{x}) &= (2\lambda_1(\boldsymbol{x}) - 1)\lambda_1(\boldsymbol{x}), \\
\varphi_2^K(\boldsymbol{x}) &= (2\lambda_2(\boldsymbol{x}) - 1)\lambda_2(\boldsymbol{x}), \\
\varphi_3^K(\boldsymbol{x}) &= 4\lambda_0(\boldsymbol{x})\lambda_1(\boldsymbol{x}), \\
\varphi_4^K(\boldsymbol{x}) &= 4\lambda_1(\boldsymbol{x})\lambda_2(\boldsymbol{x}), \\
\varphi_5^K(\boldsymbol{x}) &= 4\lambda_0(\boldsymbol{x})\lambda_2(\boldsymbol{x}),
\end{aligned}
$$

where $\lambda_i, i = 0, \ldots, 2$ are the *linear* local shape functions.

2c)

(**Core problem**) Compute the gradients of the local shape functions described above and complete the template file `grad_shape.hpp` implementing the function

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

which returns the value of the derivatives (i.e. the gradient) of a local shape functions $\varphi_i^K(\boldsymbol{x})$, with $i = 0, \ldots, 5$, on the reference element depicted in Fig. 2 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.

2d)

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 (5) and for the triangle with vertices `a`, `b` and `c`. Notice that for *quadratic* finite elements you should obtain a 6×6 element stiffness matrix.

Hint: Use the routine `gradientShapefun` from subproblem **2c)** 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.

2e)

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 in the right-hand side of (7), for the triangle with vertices `a`, `b` and `c`, and where `f` is a function handler to the right-hand side of (5).

Hint: Use the routine `shapefun` from subproblem **2b)** 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.

2f)

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

```
template<class Matrix>
void assembleStiffnessMatrix(Matrix& A, const Eigen::MatrixXd& vertices,
                             const Eigen::MatrixXi& dofs, const int& N)
```

to compute the finite element matrix $\mathbf{A}$ as in (11). 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 dofs is a $N_T \times 6$ matrix where the i-th row contains the *indices* of the vertices and edges of the i-th triangle, $i = 0, \ldots, N_T - 1$, with N_T the number of triangles in the mesh. Finally, the input N gives the number of degrees of freedom (i.e. $N = N_V + N_E$).

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

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

2g)

(**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& dofs, const int& N,
                        const std::function<double(double, double)>& f)
```

to compute the right-hand side vector $\mathbf{F}$ as in (11). The input arguments vertices, dofs and N are as in subproblem **2f)**, and f is an in subproblem **2e)**.

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

The routine

```
void setDirichletBoundary(Eigen::VectorXd& u,
                          Eigen::VectorXi& interiorDofs,
                          QDofs& quadraticDofs,
                          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 class quadraticDofs that has the information about the vertices and edges, 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 interiorDofs the indices of the interior degrees od freedom, that is of the vertices and edges that are *not* on the boundary $\partial\Omega$;
- if $\boldsymbol{x}_i$ is a node 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.

2h)

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

```
int solveFiniteElement(Vector& u, const QDofs& quadraticDofs,
                       const std::function<double(double, double)>& f)
```

This function takes in input the class quadraticDofs that has the information about the vertices and edges (as in the previous subproblems), and the function handle `f` to the right-hand side f in (5). The output argument `u` has to contain, at the end of the function, the finite element solution u_N to (7).

Hint: Use the routines assembleStiffnessMatrix and assembleLoadVector from subproblems **2f)** and **2g)**, respectively, to obtain the matrix $\mathbf{A}$ and the vector $\mathbf{F}$ as in (11), and then use the provided routine setDirichletBoundary to set the boundary values of `u` to zero and to select the free degrees of freedom.

2i)

Run the code in the file `fem2d.cpp` to compute the finite element solution to (5) 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.

Hint: This should look like the solution in series 2 warmup.

2j)

(**Core problem**) Complete the functions

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

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

contained in the files L2_norm.hpp and H1_norm.hpp, respectively.

In both cases, the argument `u1` is considered to be a vector corresponding to a *quadratic* 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)

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)> g,
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 (5)-(6) for the mesh square and the data contained in `fem2d.cpp`.

Which convergence order with respect to the number of degrees of freedom do you observe? Compare it with your *linear* finite element implementation.