Series 3

ETH zürich

Numerical methods for PDEs
Last edited: April 14, 2017
Due date: 2017-05-02 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 Quadratic Finite Elements for the Poisson equation in 2D

We consider the problem

$$-\Delta u = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{1}$$
$$u(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega \tag{2}$$

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

$$\int_\Omega \nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, d\boldsymbol{x}, \quad \text{for all } v \in H_0^1(\Omega). \tag{3}$$

We solve (3) 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

$$b_i(\boldsymbol{x}_j) := \begin{cases} 1, & i = j \\ 0, & else \end{cases}, \quad i = 0, ..., N_V - 1, \tag{4}$$

  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

$$\psi_i(\mathbf{m}_j) := \begin{cases} 1, & i = j \\ 0, & else \end{cases}, \quad i = 0, ..., N_E - 1, \tag{5}$$

  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

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

The finite element solution u_N to (1) can thus be expressed as

$$\begin{aligned} u_N(\boldsymbol{x}) &= \sum_{i=0}^{N_V-1} \mu_i b_i(\boldsymbol{x}) + \sum_{i=0}^{N_E-1} \mu_{i+N_V} \psi_i(\boldsymbol{x}) \\ &= \sum_{i=0}^{N-1} \mu_i \varphi_i^N(\boldsymbol{x}), \end{aligned} \tag{6}$$

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, $i = 0, \dots, N-1$ as test functions in the variational formulation from (3) we obtain the linear system of equations

$$\mathbf{A}\boldsymbol{\mu} = \mathbf{F}, \tag{7}$$

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

1a)

Write an expression in terms of b_i, $i = 0, \dots, N_V - 1$ and ψ_i, $i = 0, \dots, N_E - 1$, for the entries of $\mathbf{A}$ and $\mathbf{F}$ in (7).

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

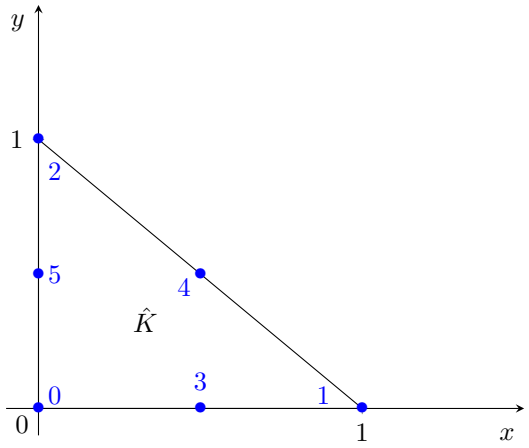

Figure 1: Reference element $\hat{K}$ for 2D quadratic finite elements.

1b)

(**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. 1 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.

1c)

(**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. 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.

1d)

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`. Notice that for *quadratic* finite elements you should obtain a 6×6 element stiffness matrix.

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

1e)

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 (3), 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 `shapefun` from subproblem **1b)** 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.

1f)

(**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 (7). 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 **1d)** to compute the local stiffness matrix associated to each element.

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

1g)

(**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 (7). The input arguments vertices, dofs and N are as in subproblem **1f)**, and f is an in subproblem **1e)**.

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

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.

1h)

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 (1). The output argument u has to contain, at the end of the function, the finite element solution u_N to (3).

Hint: Use the routines assembleStiffnessMatrix and assembleLoadVector from subproblems **1f)** and **1g)**, respectively, to obtain the matrix $\mathbf{A}$ and the vector $\mathbf{F}$ as in (7), and then use the provided routine setDirichletBoundary to set the boundary values of u to zero and to select the free degrees of freedom.

1i)

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.

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

1j)

(**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.

1k)

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.

1l)

Run the routine convergenceAnalysis to perform a convergence study for the finite element solution to (1)-(2) 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.

Exercise 2 Finite-Differences for the Heat Equation on a disc

In this exercise we will solve the heat equation on the unit disc

$$\Omega := B(0,1) := \{x \in \mathbb{R}^2 \mid \|x\| < 1\}$$

. Concretely, we consider the heat equation in two space dimensions given as

$$\begin{cases} \dfrac{\partial u}{\partial t} - \Delta u = 0 & \text{on } \Omega \times [0,T] \\ u|_{\partial\Omega} = 0 & \\ u(x,y,0) = u_0(x,y) & x,y \in \Omega \end{cases} \tag{8}$$

We remind you of the polar coordinate transformation

$$x(r,\theta) = r\cos(\theta) \qquad y(r,\theta) = r\sin(\theta) \qquad (r,\theta) \in [0,\infty) \times [0,2\pi). \tag{9}$$

For a given function $u : \Omega \times [0,T] \to \mathbb{R}$, define

$$\tilde{u}(r,\theta,t) = u(r\cos\theta, r\sin\theta, t) \qquad (r,\theta,t) \in [0,1) \times [0,2\pi) \times [0,T].$$

2a)

Assume u is a smooth solution of (8), show that $\tilde{u}$ satisfies

$$\frac{\partial \tilde{u}}{\partial t} - \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \tilde{u}}{\partial r}\right) - \frac{1}{r^2}\frac{\partial^2 \tilde{u}}{\partial \theta^2} = 0 \qquad \text{on } [0,r) \times [0,2\pi) \times [0,T] \tag{10}$$

2b)

For the rest of the problem, we assume u is radially symmetric, that is

$$\frac{\partial \tilde{u}}{\partial \theta} = 0.$$

Notice that this reduces the problem to a one dimensional partial differential equation

$$\frac{\partial \tilde{u}}{\partial t} - \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \tilde{u}}{\partial r}\right) = 0 \qquad \text{on } [0, r) \times [0, T] \tag{11}$$

Assume u is a smooth solution of (11), show that

$$\frac{\partial u}{\partial r}(0, t) = 0 \qquad \text{for all } t > 0. \tag{12}$$

Hint: $\frac{\partial u}{\partial t}$ and $\frac{\partial^2 u}{\partial r^2}$ will be bounded around 0 since it's a continuous functions on a compact (bounded and closed) domain.

2c)

We discretize the domain into $N + 2$ points, where we set $\Delta r = \frac{1}{N+1}$ and

$$r_i = i\Delta r \qquad i = 0, \ldots, N + 1.$$

We discretize the temporal domain into $M + 1$ points, and we choose

$$t^n = n\Delta t \qquad n = 0, \ldots, M$$

where

$$\Delta t = \frac{1}{M}.$$

We let U_j^n be an approximation of $\tilde{u}$ at r_i, t^n, that is

$$U_j^n \approx \tilde{u}(r_i, t^n) \qquad i = 0, \ldots, N + 1, n = 0, \ldots, M.$$

State the boundary conditions for U_{N+1}^n and U_0^n, as well as the initial value U_j^0.

Hint: You can use the approximation

$$\frac{\partial u}{\partial r}(r_i, t^n) \approx \frac{U_{i+1}^n - U_i^n}{\Delta r}.$$

2d)

Discretize (11) using the discretization defined in the previous exercise. For the time discretization, use Forward-Euler, that is

$$\frac{\partial \tilde{u}}{\partial t} \approx \frac{U_i^{n+1} - U_i^n}{\Delta t}.$$

Hint: Use

$$\frac{\partial}{\partial r}\left(r\frac{\partial \tilde{u}}{\partial r}\right)(r_i, t^n) \approx \frac{r_{i+\frac{1}{2}}\frac{\partial \tilde{u}}{\partial r}(r_{i+\frac{1}{2}}, t^n) - r_{i-\frac{1}{2}}\frac{\partial \tilde{u}}{\partial r}(r_{i-\frac{1}{2}}, t^n)}{\Delta r}$$

where

$$r_{i\pm\frac{1}{2}} = r_i \pm \Delta r/2.$$

2e)

Recall that we say the scheme obeys the maximum condition if

$$\max_i U_i^n \leq \max_i U_i^0 \qquad n = 1, 2, \ldots.$$

Find necessary conditions on Δt for the maximum condition to be fulfilled for the scheme derived in the previous exercise.

2f)

(**Core problem**) Implement the C++ function

```
//! Does one Forward-Euler timestep of the heat equation
//!
//! @param[out] u at the end, u will contain the values at time t^{n+1}
//! @param[in] uPrevous should contain the values at time t^n
//! @param[in] dr the cell length in r direction
//! @param[in] dt the timestep size
void stepHeatEquation(Eigen::VectorXd& u,
        const Eigen::VectorXd& uPrevious,
        const double dr,
        const double dt);
```

The function should increment the solution from u^n (given in `uPrevious`) to u^{n+1}(to be written to `u`).

For a template, see `heqateq_polar/heateq_polar.cpp`.

Hint: Remember to handle the boundary conditions!

2g)

Implement a C++ function

```
//! Gives an approximation to the heat equation with the given initial data
//!
//! @param initialData represents the function \tilde{u}_0.
//!
//! @param shouldStop is a function taking as first
//! parameter      the current value of u, and
//! as       second value the current time t.
//! The       simulation should run until shouldStop(u,t) == true. That is
//!
//! \code{.cpp}
//! while(!shouldStop(u,t))     { /* Do one more timestep */ }
//! \endcode
//!
//! @param N the number of inner points
//! @param cfl the constant C with which we choose the timestep size. We set
//! \code{.cpp}
//! dt       = cfl*dr*dr
//! \endcode
Eigen::VectorXd solveHeatEquation(
        const std::function<double(double)>& initialData,
        const std::function<bool(const Eigen::VectorXd&, double)>& shouldStop,
        const int N,
        double cfl = 0.5
)
```

That solves the heat equation. The simulation should run until `shouldStop(u,t)` returns `true`.

2h)

In the function `main` in `heateq_polar/heateq_polar.cpp`, we call the function with $N = 20$ and `cfl=0.5` and `cfl=0.51` and simulate until $t = 1$. The initial value is given as $u_0(x, y) = 20$. Run the program, and plot the result for `cfl=0.5` and `cfl=0.51`. What happens when `cfl=0.51`? Explain your results.

Hint: When does the solution obey the maximum principle?

2i)

(**Core problem**) For practical purposes, it's often convenient to run the simulation until a certain criterion is met, rather than running the simulation to a fixed time t. In this subtask, we will run the simulation until the maximum of U reaches the value 4. In a real world use case, this would correspond to finding out when the temperature reaches the given threshold.

In the function `main` in `heateq_polar/heateq_polar.cpp`, call `solveHeatEquation` where the parameters are

- `initialData` the same as in the previous exercise
- `shouldStop` a function that checks if the maximum of the solution is 4 or less.
- `N`=500
- `cfl`=0.5

At what time did the solution reach the maximum value of 4?

Remark 1. *This task only rewards bonus points if it is solved through using the* `shouldStop` *function parameter correctly.* ***No*** *alteration of* `solveHeatEquation` *should be needed for this exercise.*

Hint: You do **not** need to alter `solveHeatEquation` to solve this exercise.

2j)

(**Core problem**) Perform a convergence study against a reference solution. Implement your solution in the method

```
void convergenceStudy()
```

Use the initial data

$$\tilde{u}_0(r) = 1 - r^2 \cos(r).$$

For the reference solution, use $N = 2^10 - 2$. Set the final time $T = 0.025$, and measure the error when

$$N = 2^k - 2 \qquad k = 4, 5, \ldots, 8.$$

What is the order of convergence?