Scientific Computing for PDEs
Homework set 2: A finite element solver in 2D

Deliverable. *This assignment comes with a 2D Finite Element solver written in Python. The code has some missing parts and you are asked to hand the program with the filled in parts.*

Introduction

The objective of this exercise is to write a Finte Element solver for the Poisson problem

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \Omega = (-1,1)^2 \\ u &= 0 \qquad \text{on } \partial\Omega. \end{aligned}$$

In order to do so, we consider the weak formulation: Find $u \in H_0^1(\Omega)$ such that

$$a(u,v) = L(v) \qquad \forall v \in H_0^1(\Omega),$$

where

$$a(u,v) := (\nabla u, \nabla v)_{L_2(\Omega)}, \qquad\qquad L(v) := (f,v)_{L_2(\Omega)}.$$

To solve this problem numerically, we use the Galerkin method involving a $\mathbb{P}_1$ finite element subspace $V_{0,h}$ of $H_0^1(\Omega)$. To construct this subspace, we use a triangular mesh of Ω and use on each of these triangles a polynomial of degree 1.

This exercise guides you to implement your solver for the above problem. You are provided with a code that is built in the spirit of most modern finite element libraries. Your task is to understand the code, and to fill in some missing parts.

The main steps in the code are:

1. reading a given mesh of the domain Ω,
2. assembling the stiffness matrix and right-hand side,
3. solving the resulting linear system to obtain a $\mathbb{P}_1$-FEM approximation to the solution,
4. visualizing the approximate solution.

You will essentially work on step 2 but it will be necessary to understand most of the other parts. Below are some further explanations.

1 The grid format

The triangular mesh $\mathcal{T}_h$ of the domain Ω is given in a text file that contains the following data:

- The number N_v of vertices in the mesh. Remember that

  $$N_v = N_b + N_\circ$$

  where N_b is the number of vertices at the boundary and $N_\circ$ is the number of vertices in the interior.

- The number n of triangles.
- A line containing only the keyword `Vertices`, followed by N_v lines containing pairs of real numbers representing the coordinates of the vertices.
- A line containing only the keyword `Triangles`, followed by n lines of triangles. Each triangle is represented by a triplet of indices into the list of vertices. The list of triangles ends with a line containing only the keyword `EndTriangles`.
- Numerical data in a line are seperated by spaces or tabulators which are also allowed at the begin and end of each line.
- The file can contain comment lines that start with the `#` character. Those lines should be skipped when reading in the mesh.

For your convenience, you are provided with a file `grid.py` which contains a class `Grid` that is meant to store a mesh and has a class method `Grid.read_from_file` which already implements reading the aforementioned file format. You can thus instantiate the class by writing

```
grid = Grid.read_from_file(filename)
```

where `filename` contains the path to the text file of the mesh. The `Grid` class also provides properties `num_vertices` and `num_triangles` to query the number of vertices and triangles and it can be iterated over, yielding all triangles in the order in which they were given in the input file.

Advice. *To understand how an object from the `Grid` class works, you can try to read the grid in a separate Jupyter notebook, explore its attributes, and see how to iterate over them. You are advised to proceed similarly with the other classes.*

2 Handling degrees of freedom to assemble the stiffness matrix

The **stiffness matrix** is the discretization of the bilinear form $a(\cdot,\cdot)$. Here, we use that

$$V_{0,h} = \operatorname{span}\{\phi_i\}_{i=1}^{N_v},$$

where each $\phi_i : \overline{\Omega} \to \mathbb{R}$ is the hat function based at node i. Each hat function is built from Lagrange polynomials $\varphi_i^{(T)} : T \to \mathbb{R}$ defined locally on each triangle T as we have seen in the lecture. You may have noticed that we have included all inner and boundary nodes in the definition of $V_{0,h}$ instead of only the internal nodes. This is because it will be more convenient to first work with all the nodes, and then set the degrees of freedom of the boundary nodes to zero in a post-processing step (more on this later).

The solution u_h of the Poisson equation can be expressed as

$$u_h = \sum_{i=1}^{N_v} c_i \phi_i, \qquad c_i \in \mathbb{R}$$

with some yet to determine coefficients c_i. When we plug this basis representation of u_h into our variational problem, we obtain

$$\begin{aligned} a\left(\sum_{i=1}^{N_v} c_i\phi_i, \phi_j\right) &= L(\phi_j) \qquad \forall j \in \{1,\dots,N_v\} \\ \Leftrightarrow \sum_{i=1}^{N_v} a(\phi_i,\phi_j)c_i &= L(\phi_j) \qquad \forall j \in \{1,\dots,N_v\}. \end{aligned}$$

This system of N_v linear equations for N_v unknown coefficients c_i can be rewritten to

$$Ac = b$$

with stiffness matrix

$$A = (a_{i,j})_{1 \le i,j \le m} \in \mathbb{R}^{N_v \times N_v} \quad \text{with } a_{i,j} := a(\phi_i, \phi_j)$$

and right-hand side vector

$$b = (L(\phi_1), \dots, L(\phi_{N_v}))^T \in \mathbb{R}^{N_v}.$$

The matrix A is symmetric, positive definite, and sparse which allows for efficient solution of the system even in the case of fine meshes with many unknowns.

The sparsity of A is due to the fact that each entry $a_{i,j}$ is only non-zero if the supports of ϕ_i and ϕ_j overlap and each basis function has its support on a small number of neighboring triangles. Since the ϕ_i are built from a Lagrange basis, we can associate with each ϕ_i a node $q_i \in \mathbb{R}^2$ at which ϕ_i evaluates to 1. Then we obtain

$$a_{i,j} = (\nabla\phi_i, \nabla\phi_j)_{L_2(\Omega)} = \sum_{\{T \in \mathcal{T}_h | q_i, q_j \in T\}} \int_T \nabla\phi_i \nabla\phi_j \,\mathrm{d}x = \sum_{\{T \in \mathcal{T}_h | q_i, q_j \in T\}} \int_T \nabla\varphi_i^{(T)} \nabla\varphi_j^{(T)} \,\mathrm{d}x$$

and

$$b_j = (f, \phi_j)_{L_2(\Omega)} = \sum_{\{T \in \mathcal{T}_h | q_j \in T\}} \int_T f\phi_j \,\mathrm{d}x. = \sum_{\{T \in \mathcal{T}_h | q_j \in T\}} \int_T f\ \varphi_j^{(T)} \,\mathrm{d}x.$$

Since it is cumbersome to find for all the nodes q_i, q_j all the triangles that contain them, and since for most pairs of nodes the resulting set of triangles would be empty anyways, it is more efficient to construct the entries of A and b by iterating over all triangles and then adding the contributions of all local Lagrange polynomials to the right entries in A and b. We next explain how to do so.

As already mentioned before, a `Grid` object can be iterated over to obtain the triangles of the grid in a pre-defined order. So you can use

```
for (i, T) in enumerate(grid):
```

to iterate over triangles, and set a 'canonical' indexation attached to each triangle thanks to the i index that `enumerate` gives you.

By calling the function from `fe.py`

```
fe = create_fe(d)
```

you can create a Lagrange basis $\{\varphi_1^{\widehat{T}}, \varphi_2^{\widehat{T}}, \varphi_3^{\widehat{T}}\}$ of degree $m = 1$ on the reference triangle $\widehat{T}$ with nodes $\hat{q}_1 = (0,0)$, $\hat{q}_2 = (1,0)$ and $\hat{q}_3 = (0,1)$. Inside the reference triangle we use barycentric coordinates $\lambda \in \Sigma_3$ which make it easy to transform a local coordinate λ on the reference triangle to a global coordinate in a triangle $T(Q)$ of our mesh by combining the cartesian coordinates $Q = \{q_i\}_{i=1}^3$ of its vertices with λ to the global cartesian coordinate $\sum_{i=1}^3 \lambda_i q_i \in T$. To distinguish between cartesian and barycentric coordinates the provided code uses the types `Point` and `BPoint` which in the end are both wrappers around NumPy arrays.

The result `fe` of `create_fe(d)` has `len(fe)=3` reference basis function $\varphi_i^{(\widehat{T})}$. Each basis function is associated with node $\hat{q}_i$ with barycentric coordinates given by `fe.node(i)`, $i = 1, \dots,$ `len(fe)`. Each basis function $\varphi_i^{\widehat{T}}$ can be evaluated at a given barycentric coordinate $\lambda \in \Sigma_3$ with `fe.value(i, λ)`. We thus have

$$\texttt{fe.value(i, fe.node(j))} = \delta_{i,j}, \quad 1 \le i, j \le 3.$$

The gradient of the $\varphi_i^{(\widehat{T})}$ can be determined in a similar manner with `fe.grad(i, λ)`. We emphasize that this information is given for the reference triangle. To evaluate the corresponding basis functions $\varphi_i^{(T)}$ for other triangles T, we have to implement the affine transformation $F_T : \widehat{T} \to T$ from the reference to the target triangle. This is handled by the class `TriangleTransform` (more on this later).

To assemble A and b, we will do a nested loop: in an outer loop, we iterate over all triangles T_k, $k = 1, \ldots, n$ and, for each triangle T_k, we iterate over all local Lagrange basis functions $\varphi_i^{(T)}$, $i = 1, \ldots,$ `len(fe)`. For each pair of indices (k, i), we need to find the index of the corresponding node in order to fill in each component in the right entries of A and b (which are indexed by the nodes).

For this mapping from the **local degrees of freedom** (k, i) to the **global degrees of freedom** (the node indices in the mesh $\mathcal{T}_h$), the file `dof_handler.py` already contains the main elements of a class `DofHandler` that does the mapping via its method `local_to_global(k, i)`. Your task is to complete the `__init__` method by creating a dictionary that maps pairs of triangle index k and local Lagrange node index i to the index p of the associated global node in `nodes`.

Task. *Within the `__init__` method of `DofHandler`, implement a dictionary that maps the indices (k, i) of T_k and $\varphi_i^{(T)}$ to the index p of the corresponding node in `grid.nodes` of the mesh.*

3 Assembling the stiffness matrix

Now that we have a `DofHandler` to find the right row and column in A corresponding to some local DoFs, we can finally calculate the local integrals

$$\int_T \nabla\phi_i \nabla\phi_j \,\mathrm{d}x = \int_T \nabla\varphi_i^{(T)} \nabla\varphi_j^{(T)} \,\mathrm{d}x$$

and

$$\int_T f\phi_j \,\mathrm{d}x = \int_T f\varphi_j^{(T)} \,\mathrm{d}x$$

on the triangles T of our mesh. As we have seen in the lecture, this is done by making a change of variable to the reference triangle $\widehat{T}$, and using a quadrature in $\widehat{T}$.

For your convenience, the file `quadrature.py` contains a function `create_quadrature_data(d)` that creates the quadrature points and weights for a quadrature of degree `d` $= 3$ or 5 on the reference triangle.

As already mentioned earlier, the values and gradients returned by `fe.value(i, λ)` and `fe.grad(i, λ)` have to be transformed to the triangle geometry, which can be done with routines from the class `TriangleTransform`. This class is an implementation of $F_T^{-1} : T \to \widehat{T}$. In particular, `transform_grad(fe.grad(i, λ))` gives the Jacobian of F_T^{-1} after the transformation to the reference domain. With these ingredients, you can now iterate over all triangles and add their local contributions to A and b.

Task. *Within the function `galerkin_discretization` in `fesolver.py`, implement the matrix A and the right-hand side b. At this stage, you should not worry about the boundary values. They will be addressed in the next step.*

As anticipated earlier, our finite dimensional space $V_{0,h}$ is not really a subspace of $H_0^1(\Omega)$ because it contains functions that may be non-zero on the boundary. While we could build those **boundary values** into the space $V_{0,h}$ by removing basis functions with Lagrange nodes

on the boundary, this would make the previous step of assembling A and b more complicated. Therefore, we enforce the boundary values by instead modifying A and b in a way that ensures that the coefficients c_i corresponding to nodal basis functions ϕ_i on the boundary, have value 0. This can be easily done by setting all entries in the corresponding rows of A to 0, except for the diagonal entries which we set to 1. Additionally we set the corresponding rows in b to 0, which will then force $c_i = 0$ on the boundary. This method destroys the symmetry of A, but it is easily restored by doing Gaußian elimination with each modified row i. In our case, this amounts to also zeroing out all non-diagonal entries of the ith column of A.

Task. *Implement a post-process of the matrix A to enforce the Dirichlet boundary conditions. To test which nodes of the mesh are at the boundary, you can use the function* `is_point_on_boundary` *that is provided as input to* `galerkin_discretization`.

Finally, we have to solve the discretized system $Ac = b$. Here, it makes sense to choose a solver that makes use of the sparsity of A. For smaller systems, like in our case, the direct solver `spsolve` from `scipy.sparse.linalg` works well, and this is what the current code uses. For larger systems, an iterative solver, like Conjugate Gradient iteration, will be faster and more stable.

4 Running the code and visualizing the results

The main file of the code is `fesolver.py`. Its function `main` is the entry-point to start reading the code. In `fesolver.py`, you can find the skeleton of a function `galerkin_discretization` that should assemble and return A and b.

You can test your program by running

```
python3 fesolver.py
```

The program will read the mesh, create the Galerkin discretization of the Poisson equation with right hand side $f = 1$, solve it and then plot the solution.

If you like, you can play with the different meshes provided in the `data` folder, and with the right-hand side f.

Here is the solution that you should obtain for $f = 1$ with the coarse mesh, and with the fine mesh.

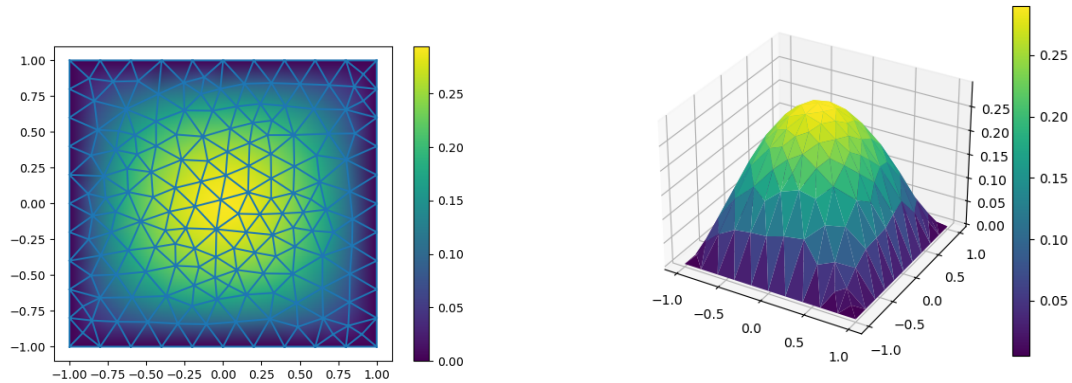

Figure 1: FEM solution for $f = 1$ with the coarse mesh.

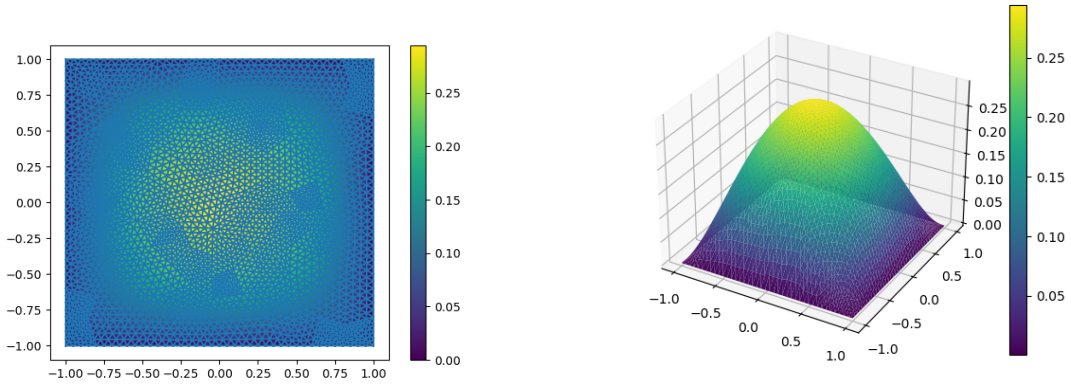

Figure 2: FEM solution for $f = 1$ with the fine mesh.