Numerical Methods for Partial Differential Equations
A.Y. 2025/2026

Laboratory 02

Finite Element method for the Poisson equation in 2D

One of the key features of `deal.II` is that it supports *dimension-independent programming*, that is it is possible and relatively easy to write code that works in any physical dimension (1D, 2D, 3D). This is realized through the extensive use of `C++` templates. Indeed, many `deal.II` classes are actually class templates taking as arguments the physical dimension of the problem.

In this lecture, we will demonstrate this by moving from 1D to 2D problems. As we will see, this mostly requires to change the value of the variable indicating the physical dimension (called `dim` in previous lectures) from 1 to 2.

Exercise 1.

Let $\Omega = (0,1) \times (0,1)$, and let us decompose its boundary $\partial\Omega$ as follows (see Figure 1):

$$\begin{aligned}
\Gamma_0 &= \{x = 0, y \in (0,1)\} \ , \\
\Gamma_1 &= \{x = 1, y \in (0,1)\} \ , \\
\Gamma_2 &= \{x \in (0,1), y = 0\} \ , \\
\Gamma_3 &= \{x \in (0,1), y = 1\} \ .
\end{aligned}$$

Let us consider the following Poisson problem with mixed Dirichlet-Neumann boundary conditions:

$$\begin{cases}
-\nabla \cdot (\mu \ \nabla u) = f & \mathbf{x} \in \Omega, & \text{(1a)} \\
u = g & \text{on } \Gamma_0 \cup \Gamma_1, & \text{(1b)} \\
\mu \nabla u \cdot \mathbf{n} = h & \text{on } \Gamma_2 \cup \Gamma_3, & \text{(1c)}
\end{cases}$$

where $\mu(\mathbf{x}) = 1$, $f(\mathbf{x}) = -5$, $g(\mathbf{x}) = x + y$ and $h(\mathbf{x}) = y$.

1.1. Write the weak formulation and the Galerkin formulation of problem (1).

1.2. Implement in `deal.II` a finite element solver for (1), using triangular elements.

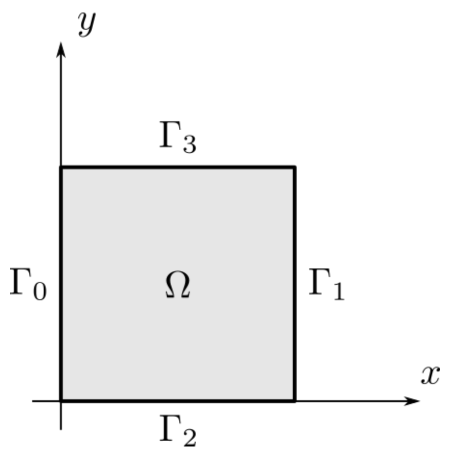

Figure 1: Domain Ω and partition of its boundary into Γ_0, Γ_1, Γ_2 and Γ_3.

Exercise 2.

Let $\Omega = (0,1) \times (0,1)$, as depicted in Figure 1, and let us consider the following diffusion-reaction problem with homogeneous Dirichlet boundary conditions:

$$\begin{cases} -\nabla \cdot (\mu \, \nabla u) + \sigma u = f & \mathbf{x} \in \Omega, \quad (2a) \\ u = 0 & \text{on } \partial\Omega, \quad (2b) \end{cases}$$

where $\mathbf{x} = (x,y)^T$, $\mu(\mathbf{x}) = 1$, $\sigma = 1$ and

$$f(\mathbf{x}) = (20\pi^2 + 1)\sin(2\pi x)\sin(4\pi y) \; .$$

The exact solution to this problem is

$$u_{\text{ex}}(x,y) = \sin(2\pi x)\sin(4\pi y) \; .$$

2.1. Write the weak formulation, the Galerkin formulation and the finite element formulation of (2)

2.2. Implement a finite element solver for problem (2). The solver should read the mesh from file (four differently refined meshes are provided as `mesh/mesh-square-*.msh`).

2.3. Using the four meshes provided, study the convergence of the solver for polynomials of degree $r = 1$ and of degree $r = 2$. Plot the error in the L^2 and H^1 norms against h, knowing that for every mesh file `mesh/mesh-square-`N`.msh`, the mesh size equals $h = 1/N$.

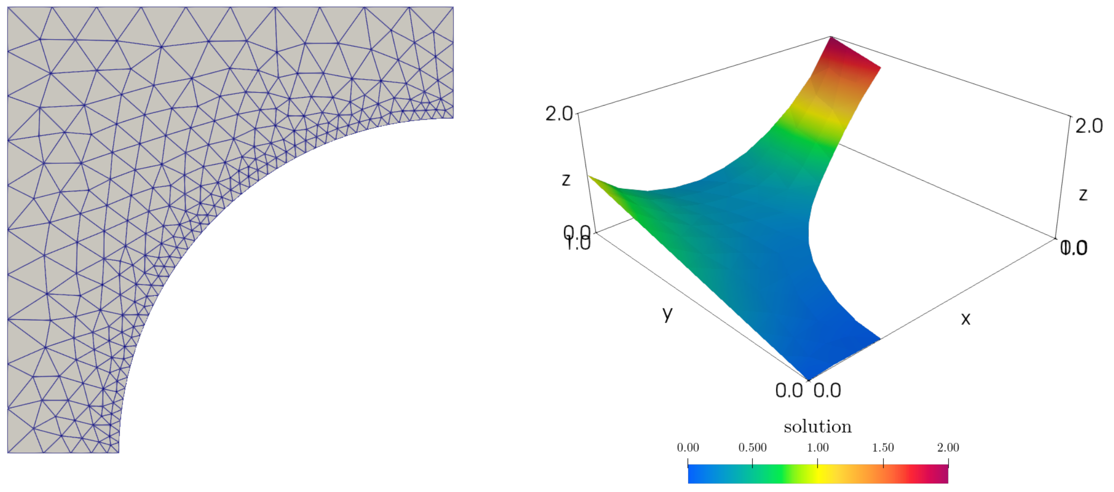

Figure 2: (a) Example of non-trivial mesh generated using `gmsh`. (b) Example of numerical solution of a Poisson problem defined in the mesh represented in (a).

Mesh generation with external tools

For problems defined on non-trivial domains (thus for most practical applications), it is generally not possible to construct the mesh directly from inside the C++ code.

Therefore, external tools are usually required to generate the mesh. Free/open source examples include `gmsh` (`https://gmsh.info/`) and VMTK (`http://www.vmtk.org/`), and most commercial finite elements software relies on some graphical meshing tool.

As an example, you can find in the `examples/gmsh` folder a `gmsh` script that generates the mesh shown in Figure 2a. You can edit the script with any text editor, and load it into `gmsh` (either from the command line or from its GUI). The `gmsh` website offers a series of tutorials at this link: `https://gmsh.info/doc/texinfo/gmsh.html#Gmsh-tutorial`. See Figure 2b for an example of numerical solution computed on Figure 2a.

Some general guidelines for mesh generation are the following:

- geometrical accuracy: the mesh should have enough elements so that the geometrical features of the domain are accurately represented;
- solution accuracy: the mesh should have enough elements so that the numerical solution computed on it is accurate. This might also mean that the elements in some regions are smaller than elements in other regions, to capture sharp gradients or to enhance the accuracy in regions of particular interest;
- mesh quality: the elements of the mesh should be as regular as possible (i.e. as close as possible to regular triangles). If elements are distorted, the constant

appearing in the error estimate increases, and the accuracy becomes worse. Moreover, the condition number of the linear system resulting from the discretization becomes larger, and if elements are inverted (i.e. they degenerate to have zero or negative area/volume) the discrete problem becomes ill-posed.

Generating a mesh satisfying these three requirements is a complex task, especially for 3D problems, and often the mesh generation step is one of the most time consuming ones in the finite element pipeline.

Possibilities for extension

Space adaptivity. One of the key features of `deal.II` is it support of *space adaptivity*, i.e. refining the mesh only in regions where the solution is less accurate (i.e. where an *a posteriori error estimate* indicates that the solution is inaccurate). This allows to strike an excellent compromise between accuracy and problem size (and thus computational cost). Based on `deal.II`'s step 6 tutorial (`https://dealii.org/9.5.0/doxygen/deal.II/step_6.html`), modify the code of Exercise 1 to use adaptive space refinement. Compare the results with those of Exercise 1 in terms of error against the number of degrees of freedom. Beware that, as of now, adaptivity is only supported for quadrilateral (or hexahedral) meshes.