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:

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

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).

Solution. Let us introduce the function spaces

$$V = \{v \in H^1(\Omega) \text{ such that } v = g \text{ on } \Gamma_0 \cup \Gamma_1\} \;, \tag{2}$$

$$V_0 = \{v \in H^1(\Omega) \text{ such that } v = 0 \text{ on } \Gamma_0 \cup \Gamma_1\} \;. \tag{3}$$

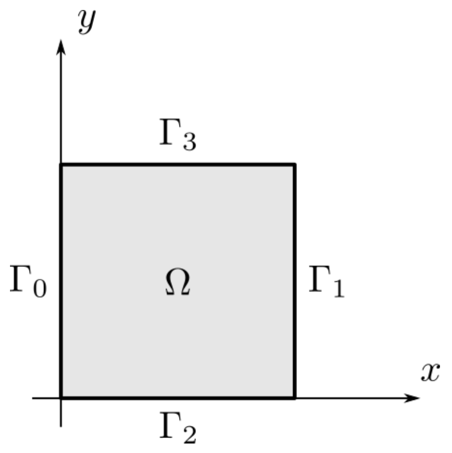


Figure 1: Domain Ω and partition of its boundary into Γ_0, Γ_1, Γ_2 and Γ_3.

Taking $v \in V_0$, multiplying it to (1a) and integrating over Ω, we get

$$\begin{aligned}\int_\Omega -\boldsymbol{\nabla}\cdot(\mu\boldsymbol{\nabla}u)\,v\,d\mathbf{x} &= \int_\Omega f\,v\,d\mathbf{x}\ ,\\ \int_\Omega \mu\,\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}v\,d\mathbf{x} - \int_{\partial\Omega}\mu\,v\,\boldsymbol{\nabla}u\cdot\mathbf{n}\,d\sigma &= \int_\Omega f\,v\,d\mathbf{x}\ ,\\ \int_\Omega \mu\,\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}v\,d\mathbf{x} - \int_{\Gamma_2\cup\Gamma_3}\mu\,v\,\boldsymbol{\nabla}u\cdot\mathbf{n}\,d\sigma &= \int_\Omega f\,v\,d\mathbf{x}\ .\end{aligned}$$

Now, we write $u = u_0 + R_g$, where $u_0 \in V_0$ and $R_g \in V$ is an arbitrary *lifting function* such that $R_g = g$ on $\Gamma_0 \cup \Gamma_1$. Then, the above becomes:

$$\underbrace{\int_\Omega \mu\,\boldsymbol{\nabla}u_0\cdot\boldsymbol{\nabla}v\,d\mathbf{x}}_{a(u_0,v)} = \underbrace{\int_\Omega f\,v\,d\mathbf{x} + \int_{\Gamma_2} v\,h\,d\sigma + \int_{\Gamma_3} v\,h\,d\sigma - \int_\Omega \mu\boldsymbol{\nabla}R_g\cdot\boldsymbol{\nabla}v\,d\mathbf{x}}_{F(v)}\ .$$

The weak formulation reads:

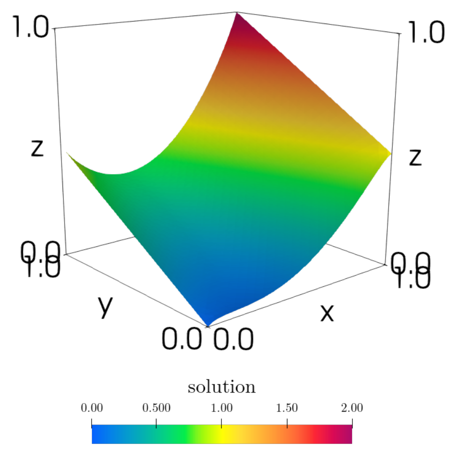
$$\text{find } u_0 \in V_0\ :\ a(u_0,v) = F(v) \text{ for all } v \in V_0\ .$$

Introducing a finite dimensional approximation $V_{0,h}$ of V_0, the Galerkin formulation reads:

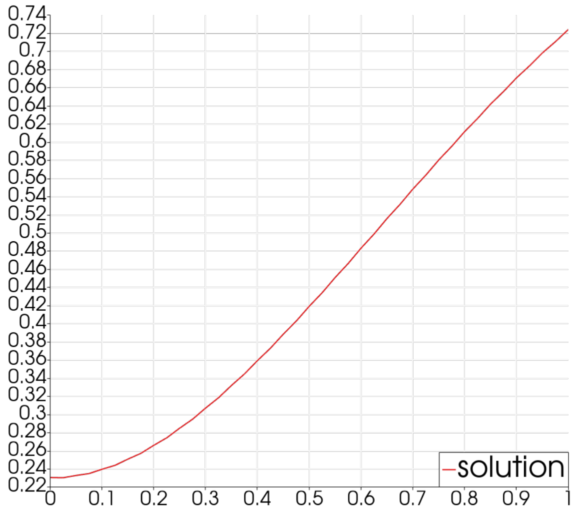
$$\text{find } u_{0,h} \in V_{0,h}\ :\ a(u_{0,h},v_h) = F(v_h) \text{ for all } v_h \in V_{0,h}\ .$$

The finite element formulation is obtained by defining a triangular mesh over Ω and taking $V_{0,h} = X_h^r \cap V_0$.

With this approach (i.e. using the lifting function R_g), we can use the Lax-Milgram theorem to prove the well-posedness of the problem (since the test and trial function belong to the same function space). However, while it is possible to use the same



(a)



(b)

Figure 2: (a) Approximate solution to problem (1) using linear finite elements and 20 subdivisions for mesh generation. The plot was obtained by appling the "Warp by scalar" Paraview filter. (b) Plot of the solution over the line $x = 0.5$. The plot was obtained by applying the "Plot over line" Paraview filter.

technique in the implementation, this is not very convenient. Instead, we equivalently impose non-homogeneous strongly by modifying the rows of the linear system associated to the *degrees of freedom* (DoFs) on the Dirichlet boundary.

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

Solution. See the file `src/exercise-01.cpp` for the implementation.

Most of `deal.II` classes are dimension independent, meaning that they work in the same way (i.e. expose the same interface) regardless of whether they are used for 1D, 2D or 3D problems. The spatial dimension is provided as a template parameter (usually called `dim`).

We start from the code for the 1D Poisson problem. To move from 1D to 2D, we begin by renaming the class `Poisson1D` to `Poisson2D`, and we change the value of `Poisson2D::dim` from 1 to 2. This already does most of the work.

Then, we need to address mesh generation. `deal.II` was born to solve problems using finite elements on quadrilateral (and hexahedral) grids, and the support for triangular (and tetrahedral) meshes was added at a later stage. For this reason, the functions in the `GridGenerator` namespace mostly generate quadrilateral or hexahedral meshes. A quadrilateral mesh can be converted to a triangular one by using the function `GridGenerator::convert_hypercube_to_simplex_mesh`, although this has partial support as of `deal.II` v9.5.1. Instead, we load the mesh from an externally generated

`.msh` file, through the `GridIn` class and its `read_msh` method. The class supports different mesh formats, which are described in the associated page of `deal.II`'s documentation. See below for a short note on mesh generation.

The above steps are sufficient to convert the 1D solver into a 2D solver (and the same approach allows to easily obtain a 3D solver). On top of that, we need to make the modifications related to the data (forcing term, boundary conditions, coefficients) for this specific test case.

Dirichlet boundary conditions are managed in the same way as done in 1D, by modifying the linear system after it is assembled. Neumann boundary conditions, instead, require the computation of some additional integrals on the boundary, so that the method `assemble` needs to be modified accordingly. Since we need to compute integrals on boundary edges, we need a new `Quadrature` object (`quadrature_boundary`). Moreover, we need to evaluate shape functions on boundary edges, and we do so with a `FEValues`-like object (`FEFaceValues<dim> fe_values_boundary`).

The approximate solution is shown in Figure 2.

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:

$$\begin{cases} -\boldsymbol{\nabla} \cdot (\mu\, \boldsymbol{\nabla} u) + \sigma u = f & \mathbf{x} \in \Omega, \qquad (4a) \\ u = 0 & \text{on } \partial\Omega, \qquad (4b) \end{cases}$$

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 (4)

Solution. We proceed as usual: let $V = H_0^1(\Omega)$ and $v \in V$. We multiply v to (4a) and integrate over Ω:

$$\int_\Omega -\boldsymbol{\nabla} \cdot (\mu\, \boldsymbol{\nabla} u)\, v d\mathbf{x} + \int_\Omega \sigma u v d\mathbf{x} = \int_\Omega f v\ ,$$
$$\int_\Omega \mu\, \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v d\mathbf{x} - \underbrace{\int_{\partial\Omega} \mu(\boldsymbol{\nabla} u \cdot \mathbf{n}) v d\gamma}_{=0} + \int_\Omega \sigma u v d\mathbf{x} = \int_\Omega f v d\mathbf{x}\ .$$

Let us define

$$a(u,v) = \int_\Omega \mu \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v d\mathbf{x} + \int_\Omega \sigma u v d\mathbf{x}\ ,$$
$$F(v) = \int_\Omega f v d\mathbf{x}\ .$$

The weak formulation reads:

$$\text{find } u \in V \quad : \quad a(u,v) = F(v) \quad \forall v \in V\ . \qquad (5)$$

Let us introduce a mesh in Ω, and let $V_h = X_h^r \cap V$ be the finite element space. Let φ_i be its basis functions, for $i = 1, 2, \ldots, N_h$. After restricting (5) to V_h and writing the solution as a linear combination of the basis, we obtain

$$\sum_{j=1}^{N_h} U_j \left(\int_\Omega \mu \boldsymbol{\nabla} \varphi_i \cdot \boldsymbol{\nabla} \varphi_j d\mathbf{x} + \int_\Omega \sigma \varphi_i \varphi_j d\mathbf{x} \right) = F(\varphi_i) \qquad i = 1, 2, \ldots, N_h\ ,$$

which can be written as a linear system:

$$\begin{gathered}
A\mathbf{u} = \mathbf{f} \ , \\
A_{ij} = \int_\Omega \mu \boldsymbol{\nabla} \varphi_i \cdot \boldsymbol{\nabla} \varphi_j d\mathbf{x} + \int_\Omega \sigma \varphi_i \varphi_j d\mathbf{x} \ , \\
\mathbf{f}_i = F(\varphi_i) \ .
\end{gathered}$$

2.2. Implement a finite element solver for problem (4). The solver should read the mesh from file (four differently refined meshes are provided as `mesh/mesh-square-*.msh`).

Solution. See the file `src/exercise-02.cpp` for the implementation. An overview of the differences with respect to previous laboratory is provided below.

The major change is the introduction of the reaction term. This requires to change the assembly, adding to the local matrix the contribution of

$$\int_\Omega \sigma \varphi_i \varphi_j d\mathbf{x} \ .$$

Finally, we need to impose the appropriate boundary conditions (homogeneous Dirichlet, in this case), and change the definition of the forcing term.

The numerical solution is shown in Figure 3.

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 $h = 1/N$.

Solution. See the file `src/lab-04.cpp` for the implementation.

To study the convergence, we need to define a class `ExactSolution` to represent u_{ex}. Moreover, we need to define a method `Poisson2D::compute_error` to compute the error. The latter has the same implementation as seen in Laboratory 2, with two significant differences:

- the quadrature formula is defined using the class `QGaussSimplex<dim>`, so that it works for triangles;
- we need to provide an extra argument to the function `integrate_difference` that explicitly describes the mapping ϕ_c from the reference element to the physical elements. This mapping is constructed using the class `MappingFE`.

Then, the `main` function is modified to perform the convergence analysis and produce a CSV file `convergence.csv`, whose content can then be plotted to analyze the convergence results. See Figure 4 for the result. For both $r = 1$ and $r = 2$, we observe the expected convergence orders in both the L^2 and H^1 norms.

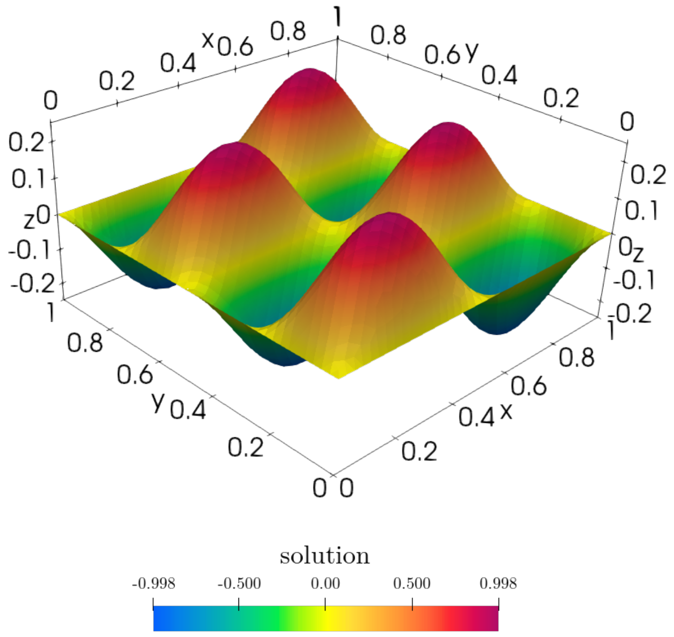


Figure 3: Numerical solution to (4), computed using linear finite elements on mesh `mesh-square-40.msh`.

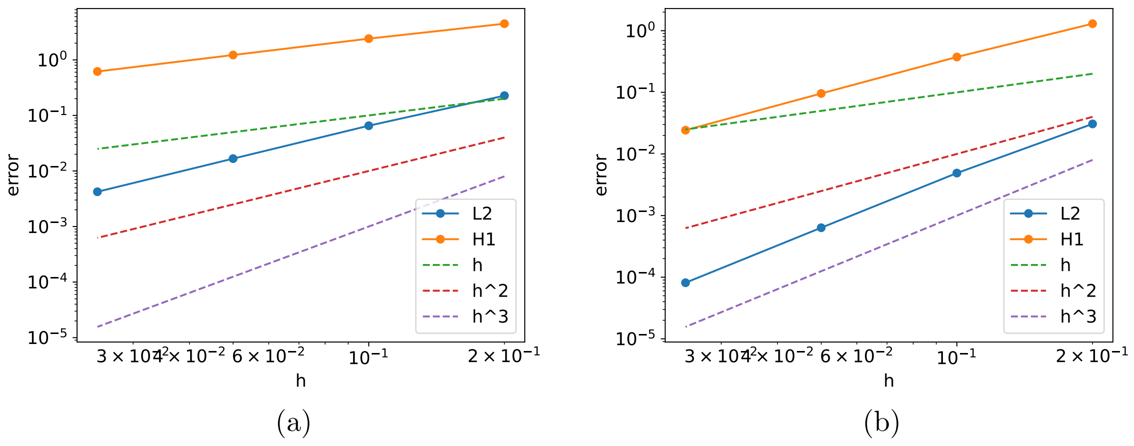
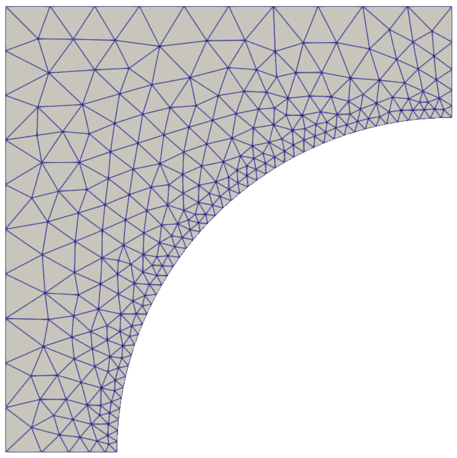
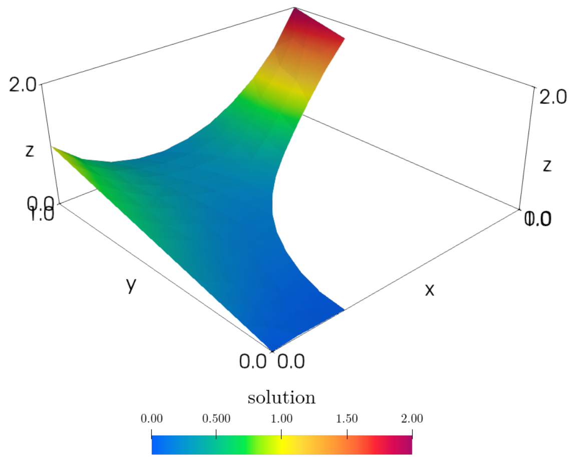


Figure 4: Exercise 2. Error in the L^2 and H^1 norms as a function of h for (a) $r = 1$ and (b) $r = 2$.


(a)



(b)

Figure 5: (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 5a. 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 5b for an example of numerical solution computed on Figure 5a.

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.