Numerical Methods for Partial Differential Equations
A.Y. 2024/2025

Laboratory 03

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:

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

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

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

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

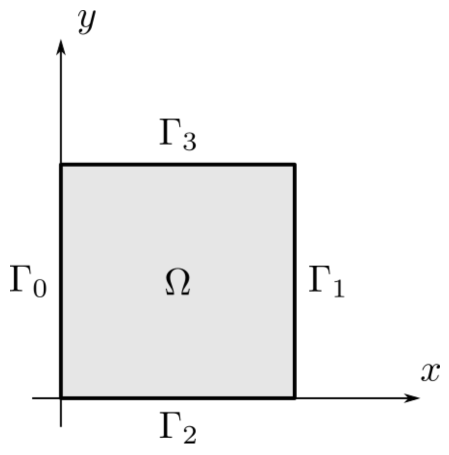


Figure 1: (a) Domain Ω and partition of its boundary into Γ_0, Γ_1, Γ_2 and Γ_3. (b) Mesh obtained by generating a square with 20 subdivisions in `deal.II` and then converting it to a triangular mesh.

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 $\Gamma_0 \cup \Gamma_1$.

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

$$\int_\Omega -\boldsymbol{\nabla} \cdot (\mu \boldsymbol{\nabla} u)\, v\, d\mathbf{x} = \int_\Omega f\, v\, d\mathbf{x}\ ,$$

$$\int_\Omega \mu\, \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v\, d\mathbf{x} - \int_{\partial\Omega} \mu\, v\, \boldsymbol{\nabla} u \cdot \mathbf{n}\, d\sigma = \int_\Omega f\, v\, d\mathbf{x}\ ,$$

$$\int_\Omega \mu\, \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v\, d\mathbf{x} - \int_{\Gamma_2 \cup \Gamma_3} \mu\, v\, \boldsymbol{\nabla} u \cdot \mathbf{n}\, d\sigma = \int_\Omega f\, v\, d\mathbf{x}\ ,$$

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

Then, 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

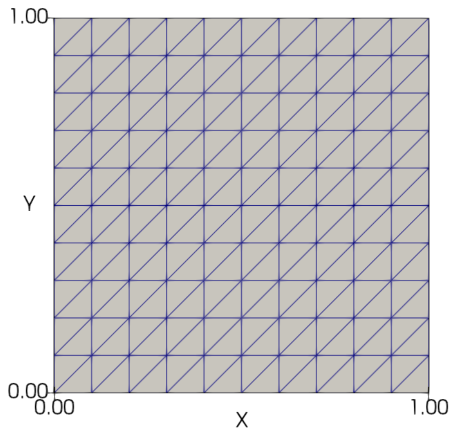


Figure 2: Square mesh obtained in `gmsh`.

belong to the same function space). However, while it is possible to use the same formulation in the implementation, this is not particularly convenient. Therefore, we equivalently impose non-homogeneous strong

Notice that, while in the derivation above the weak formulation imposed non-homogeneous Dirichlet boundary conditions through the lifting function R_g, at the level of implementation we will impose the conditions algebraically, as seen in previous lectures and laboratory sessions.

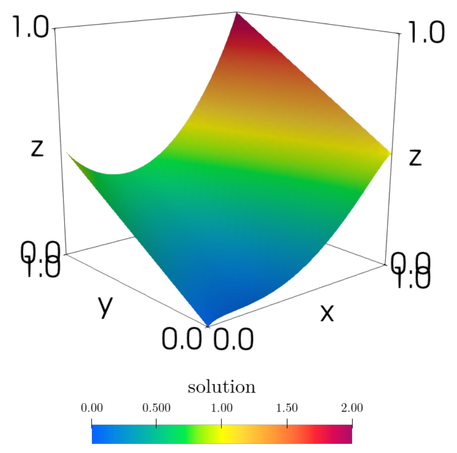
1.2. Implement in `deal.II` a finite element solver for (1), using triangular elements.

Solution. See the file `src/lab-03.cpp` for the implementation. Detailed information is given as comments in the source file, while below we comment on the differences with respect to previously implemented solvers in 1D.

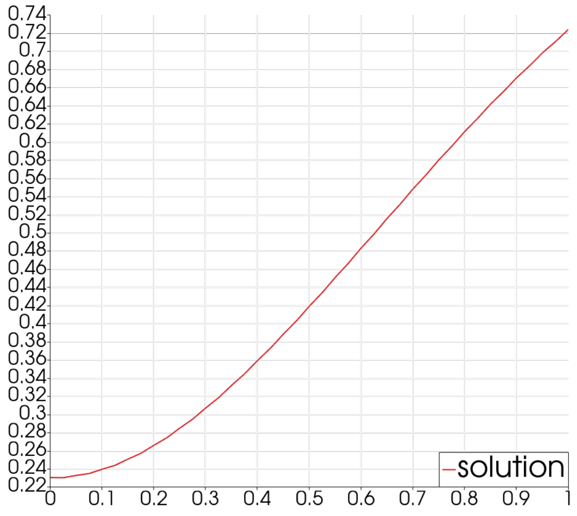
Most of `deal.II` classes are dimension independent, meaning that they work in the same way (i.e. expose the same interface) regardless of whether they are used for 1D, 2D or 3D problems. The spatial dimension is provided as a template parameter (usually called `dim`).

To move from 1D to 2D, we begin by renaming the class `Poisson1D` to `Poisson2D`, then 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 only generate quadrilateral or hexahedral



(a)



(b)

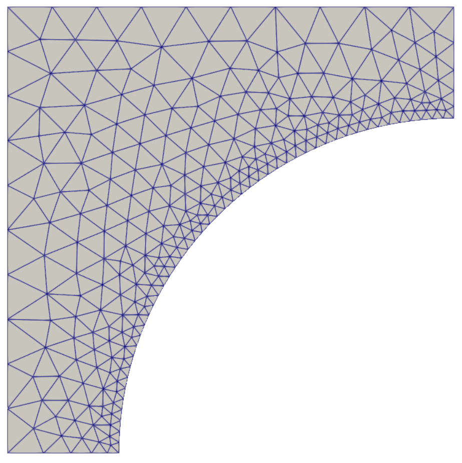
Figure 3: (a) Approximate solution to problem (1) using linear finite elements and 20 subdivisions for mesh generation. The plot was obtained by appling the "Warp by scalar" Paraview filter. (b) Plot of the solution over the line $x = 0.5$. The plot was obtained by applying the "Plot over line" Paraview filter.

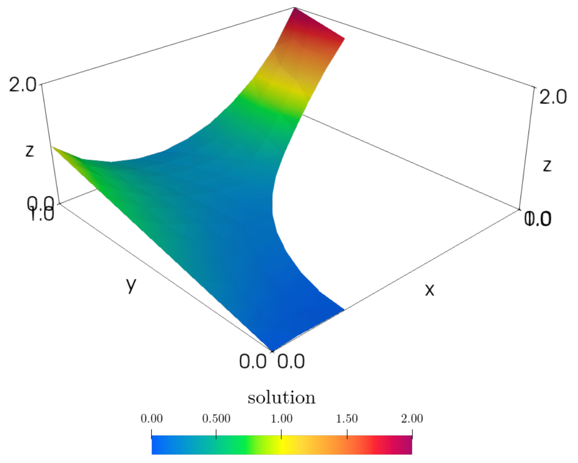
meshes. A quadrilateral mesh can be converted to a triangular one by using the function `GridGenerator::convert_hypercube_to_simplex_mesh`, although this has only limited support. Instead, we load the mesh from an externally generated `.msh` file, through the `GridIn` class and its `read_msh` method. 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.

We have to use Lagrangian finite elements on triangles: these are implemented by the class `FE_SimplexP<dim>` (whereas the class `FE_Q<dim>` that we used before works for quadrilaterals). Therefore, we change the construction of the finite element space to `fe = std::make_unique<FE_SimplexP<dim>>(r)`. Similarly, we have to use a quadrature formula suitable for triangular element. This is provided by the class `QGaussSimplex<dim>`, which has the same interface of `QGauss<dim>` (which, instead, works for quadrilateral elements).

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 boundary conditions 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,


(a)



(b)

Figure 4: (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).

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 3.

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 4a. 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 4b for an example of numerical solution computed on Figure 4a.

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.