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

Laboratory 05
Finite Element method for the diffusion-reaction equation in 3D

Exercise 1.

Let $\Omega = (0,1)^3$, be the unit cube and let us consider the following diffusion-reaction problem:

$$\begin{cases} -\boldsymbol{\nabla} \cdot (\mu \, \boldsymbol{\nabla} u) + \sigma u = f & \mathbf{x} \in \Omega, \quad (1a) \\ u = 0 & \text{on } \partial\Omega, \quad (1b) \end{cases}$$

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

$$\mu(\mathbf{x}) = \begin{cases} 100 & \text{if } x < \dfrac{1}{2}\,, \\ 1 & \text{if } x \geq \dfrac{1}{2}\,. \end{cases}$$

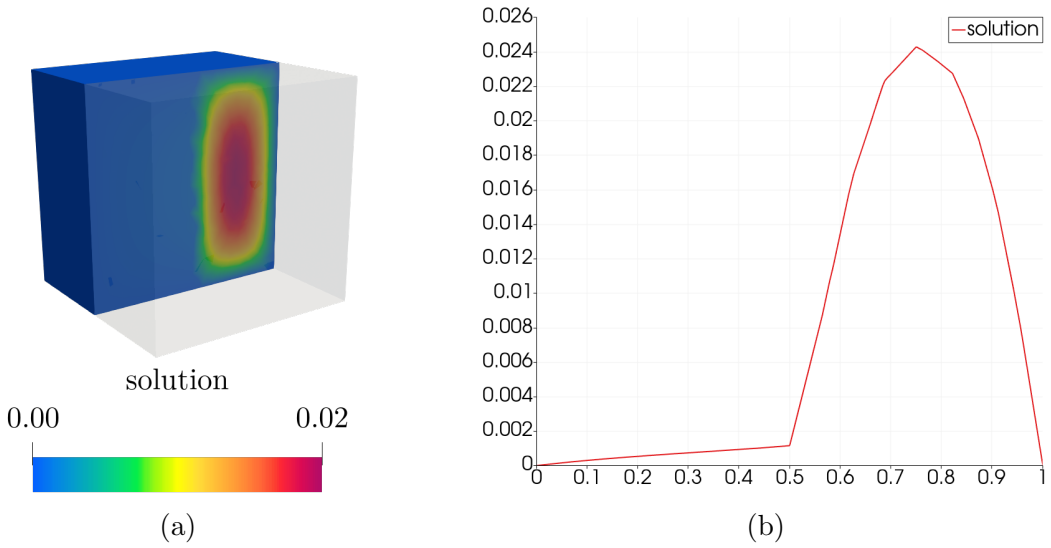
1.1. Starting from the code of Laboratories 3 and 4, implement a finite element solver for problem (1). The solver should read the mesh from file. Use the mesh `mesh/mesh-cube-10.msh` (four differently refined meshes are provided as `mesh/mesh-cube-*.msh`).

Solution. See the file `src/lab-05.cpp` for the implementation. The only differences with respect to previous code is that the static variable `dim` is now set to 3 (instead of 2), and the definitions of the forcing term and diffusion coefficient are updated.

The numerical solution is shown in Figure 1.

1.2. Compute the solution on the mesh `mesh/mesh-cube-40.msh` (i.e. $N + 1 = 40$), with linear finite elements.

Solution. When trying to compute the solution in this setting, we obtain the following error message: `Iterative method reported convergence failure in step 1000.`



(a) (b)

Figure 1: (a) Numerical solution to (1), computed using linear finite elements on mesh `mesh-cube-20.msh`. The domain was clipped along the $x - z$ plane. (b) Plot of the solution along the line $y = z = 0.5$.

`The residual in the last step was 9.88979e-09.` This indicates that, after 1000 iterations, the linear solver did not satisfy the prescribed tolerance on the residual.

The reason behind this behavior is the wide range of values in the diffusion coefficient μ. In practical terms, after discretization, some of the equations in the linear system are scaled by a factor 100, whereas other equations are scaled by a factor 1. This leads to an ill-conditioned system, that requires many iterations for its solution.

One possible workaround to this is to increase the maximum number of iterations, say to 10 000. By doing so, convergence is obtained after 1046 CG iterations. However, this approach has a large computational cost (more iterations means more computations are performed).

Instead, we can improve the performance of the solver by introducing a *preconditioner*. The simplest possible preconditioner is the Jacobi preconditioner, implemented by the class `PreconditionJacobi`. The preconditioneris defined as follows: the system matrix A is decomposed as $A = D + L + U$, where D is diagonal, U is upper-triangular and L is lower-triangular. Then, $P_{\text{J}} = D$. With the Jacobi preconditioner, the linear solver reaches convergence in 124 iterations.

We can try and improve over the Jacobi preconditioner, for instance by taking the SOR preconditioner $P_{\text{SOR}} = \frac{1}{r}(D + rL)$, with r a relaxation parameter. For $r = 1$, the SOR preconditioner becomes the Gauss-Seidel (GS) preconditioner $P_{\text{GS}} = D + L$. The SOR preconditioner is implemented by the class `PreconditionSOR`. This preconditioner, however, is not symmetric: we can no longer use the CG solver, but we need to resort

to GMRES (implemented by the `SolverGMRES` class and offering the same interface as `SolverCG`). The GMRES with the GS preconditioner reaches convergence in 83 iterations.

Finally, there exists a symmetric variant of the SOR preconditioner, known as SSOR and implemented by the class `PreconditionSSOR`. By using the CG method with the SSOR preconditioner, the solver converges in only 52 iterations.

More advanced (and possibly more efficient) preconditioners are offered by the Trilinos and PETSc libraries, for which `deal.II` provides wrappers. However, those classes can only be used if using Trilinos or PETSc also for the linear algebra (i.e. for managing matrices and vectors), while for the moment we are using the linear algebra classes provided by `deal.II` itself.

Finally, here we are comparing the different solvers and preconditioners in terms of the number of iterations. However, the computational time, i.e. the total time needed to compute the solution, would be a more appropriate performance metric. Since we are working on laptops and personal computers (possibly within a Docker container) and on a relatively small problem, time measurements are unreliable, so we use iterations as a proxy for computational cost. However, in real life problems, the efficiency of solvers and preconditioners must be determined in terms of the computational time.

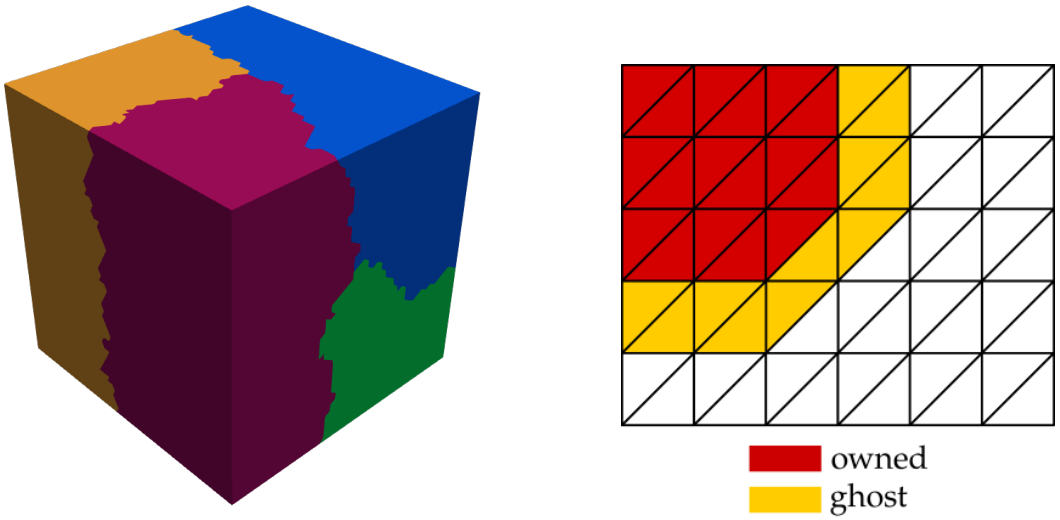
Parallelization of the finite element solver

Finite element computations for real applications are usually characterized by a large scale: they often require millions or billions of elements for the spatial discretization to properly capture the geometry of the computational domain and/or to accurately resolve the solution. The corresponding discrete problem has millions or billions of unknowns. Therefore, it is often impossible to solve such problems on single core machines. To deal with such problems, it is imperative to use parallel computing techniques. Finite element software is usually parallelized using the MPI paradigm.

A minimal example of parallel FE solver is implemented in the files `src/lab-05_-parallel.cpp`, `src/Poisson3D_parallel.hpp` and `src/Poisson3D_parallel.hpp`. In the following, we provide an overview of the differences between the parallel implementation and the serial one.

The mesh is distributed over multiple processes in the MPI pool: each element is assigned to one process (which is then said to *own* that element; see Figure 2a). In practice, each process usually has to access information on elements that are adjacent to the ones it own. Therefore, each process also stores a layer of elements around the owned ones (Figure 2b), known as the *ghost elements*. The owned and ghost elements form the *locally relevant elements*.

In `deal.II`, the partitioned triangulation is constructed by first creating the triangula-



(a) (b)

Figure 2: (a) Partitioning of the cube domain over four processes. Elements are colored according to the owner process. (b) Owned and ghost elements for one process.

tion in serial, and then distributing it across all processes (see `Poisson3DParallel::setup_system`).

The partitioning of the mesh implies a partitioning of the degrees of freedom (DoFs). Degrees of freedom that correspond to elements owned by a certain process are also owned by that process. DoFs that lie on the boundary between two parallel partitions are arbitrarily attributed to one of the two adjacent partitions. As a result, each DoF is owned by one and only one process. `deal.II` offers the `IndexSet` class (which represents a set of indices, as the name suggests) to keep track of the DoFs that the current proces owns. In a similar way, the DoFs that are associated to relevant elements are known as *relevant DoFs*. The relevant DoFs are a superset of the owned DoFs.

Linear algebra objects (matrices and vectors, but also sparsity patterns) are distributed according to the DoFs owned by each process. The matrices are distributed by row (e.g., the process with rank 0 owns the first n_0 rows, the process with rank 1 the following n_1 rows, and so on). Distributed linear algebra is not implemented directly by `deal.II`, but are provided by the library Trilinos, which `deal.II` wraps in an easy-to-use interface within the namespace `TrilinosWrappers`. When initializing these objects, we need to provide to the `reinit` method the `IndexSet` of locally owned DoFs, to specify the partitioning (see `Poisson3DParallel::setup_system`).

The `assemble` method is almost identical to the serial case: each process performs the same operations on the elements it owns. The most important difference is that we have to skip cells that we do not own (since information about them is not stored by

current process). Moreover, it is possible that a process writes to rows of the matrix it does not own. This happens, for instance, when visiting an element that is on the boundary of the partition for that process. Therefore, at the end of the assembly loop, we need to communicate between processes, to make sure that all processes store all contributions for the rows they own. This is done by calling the `compress` method on both the system matrix and right-hand side vector.

The linear solver step is also almost unchanged. The major difference is that, since we are using Trilinos as linear algebra backend, we need to use linear solver and preconditioner classes that are compatible with it. For the solver, this amounts to changing its template parameter. For the preconditioner, we need to use a different class altogether, provided by the `TrilinosWrappers` namespace.

Also the output step requires some changes to account for the fact that each process knows only a part of the solution. We use a different file format that supports writing files in parallel: each process writes a VTU file with the elements it owns, and then a PVTU file is written to link them all together (you should open the PVTU file in Paraview). A few extra arguments are needed to write the file, see `Poisson3DParallel::output`. Each process needs a vector that stores not only the owned DoFs, but also the relevant ones: we construct one for this purpose within the `output` method.

Finally, writing the standard output directly to `std::cout` should generally be avoided in parallel computations with MPI, since every process would write the output, resulting in multiple repetitions of each message. A simple workaround could be `if (mpi_rank == 0) std::cout << ...`: `deal.II` offers a class that wraps this in a lighter interface, the `ConditionalOStream` class. We define one as member of `Poisson3DParallel` (`ConditionalOStream pcout`), and then replace all output to `std::cout` with outputs to `pcout`.