Poisson Tutorial 1 PSD simulation of nontrivial manufactured tests

Rania Saadi, Mohd Afeef Badri

Abstract

This document details a multiple tutorials of 'poisson' module of PSD in a more verbos manner. The idea is to showcase how to adapt mesh in sequential or parallel, for a 2D or 3D domain.

Theory

Poisson problem is a boundary value problem in FEM which is often used to demonstrate the solvers correctness or used to develop many fundamental tools that are later used for other complex problems. Poisson's problem itself can be used to approximate certain physcial phenomena such as, heat trasfer, electrostatics, basic aerodynamics, etc.

The Poisson equation is a PDE which is defined by the following boundary-value problem,

$$\Delta u(x) = -f(x) \quad \forall x \in \Omega,$$

$$u(x) = u_{\mathrm{D}}(x) \quad \forall x \in \partial\Omega,$$

here, $u(x)$ is the unknown variable, $f(x)$ is a known function also known as source term, Δ is the Laplace operator, Ω is the spatial domain in space x and is bounded by boundary $\partial\Omega$. The Poisson problem here, includes known value u_{D} on its border $\partial\Omega$.

The starting point for FEM approximation for a PDE is its *variational form*. The variational formulation for Poission's problem reads, find $u \in \mathcal{V}$:

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x \qquad \forall v \in \hat{\mathcal{V}}.$$

For this formulation, the trial function u and test function v are defined in functional spaces

$$\mathcal{V} = \{v \in H^1(\Omega) | v = u_{\mathrm{D}} \quad \text{on} \quad \partial\Omega\},$$

$$\hat{\mathcal{V}} = \{v \in H^1(\Omega) | v = 0 \quad \text{on} \quad \partial\Omega\}.$$

Here, $H^1(\Omega)$ is the Sobolev space containing functions v such that these are square integrable with continuous derivatives over Ω. Note that, the Sobolev space allows functions with discontinuous derivatives. This weaker continuity requirement in our weak formulation (caused by the integration by parts) is of great importance when it comes to constructing the finite element function space. In particular, it allows the use of piecewise polynomial function spaces. This means that the function spaces are constructed by stitching together polynomial functions on simple domains such as intervals, triangles, quadrilaterals, tetrahedra and hexahedra. In principal these simpler domains are obtained by converting our domain Ω to its meshed form Ω^h, such that h defines the size of the mesh.

The variational problem is a *continuous problem*: it defines the solution in the infinite-dimensional function space . The finite element method for the Poisson equation finds an approximate solution u^h of the variational problem by replacing the infinite-dimensional function spaces $\mathcal{V}$ by *discrete* (finite dimensional) trial and test spaces $\mathcal{V}^h$ and $\hat{\mathcal{V}}^h$. The discrete variational problem reads: Find $u^h \in \mathcal{V}^h$ such that

$$\int_\Omega^h \nabla u^h \cdot \nabla v^h \, \mathrm{d}x = \int_\Omega^h f v^h \, \mathrm{d}x \qquad \forall v^h \in \hat{\mathcal{V}}^h.$$

2D example with discontinuous solution

To showcase the usage of Poisson, we solve for the following manufactured solution on a 2D rectangular mesh:

$$u = \tanh(-100(y - 0.5 - 0.25\sin(2\pi x))) + \tanh(100(y - x))$$

Such a function is really challenging as sudden jumps in solution are expected which can only be captured by very fine mesh. This function will also be used to define the Dirichlet condition on all borders. Analytical Laplacian of this function gives us our source term:

$$\begin{aligned} f = {} & 100\pi^2 \sin(2\pi x)\left(\tanh^2\left(-100\left(y - 0.5 - 0.25\sin(2\pi x)\right)\right) - 1\right) - 5000\pi^2 \cos^2(2\pi x) \\ & \left(1 - \tanh^2\left(-100\left(y - 0.5 - 0.25\sin(2\pi x)\right)\right)\right)\tanh\left(-100\left(y - 0.6 - 0.25\sin(2\pi x)\right)\right) \\ & +20000\left(\tanh^2\left(-100\left(y - 0.5 - 0.25\sin(2\pi x)\right)\right) - 1\right)\tanh\left(-100\left(y - 0.5 - 0.25\sin(2\pi x)\right)\right) \\ & +20000\left(\tanh^2\left(100\left(y - x\right)\right) - 1\right)\tanh\left(100\left(y - x\right)\right) \\ & -20000\left(1 - \tanh^2\left(100\left(y - x\right)\right)\right)\tanh\left(100\left(y - x\right)\right) \end{aligned}$$

To get the PSD simulation going, we start by PSD pre-processing to specify the tools and methods to use and the desired type of output. For instance, the following command communicates to PSD that the problem we want to solve is a Poisson problem, in two dimensions, we will be using sequential code, and that we want to post-proccess the solution u.

```
PSD_PreProcess -dimension 2 -problem poisson -postprocess u -sequential
```

This will produce a set of .edp files with Main.edp to be run by PSD_Solve_Seq for sequential solving.

```
PSD_Solve_Seq Main.edp -mesh ./../Meshes/2D/bar.msh
```

Further editing can be done on the produced files to adjust to a problem, mainly on ControlParameters.edp where all the problem specific parameters are written. For this example, the default provided ones are suitable. This simulation will be done on the mesh ./../Meshes/2D/bar.msh.

To vizualise the solution, we launch paraview and open the file solution0.vtu. The result that we see is not the expected solution as the mesh is too coarse to capture the variations in the function.

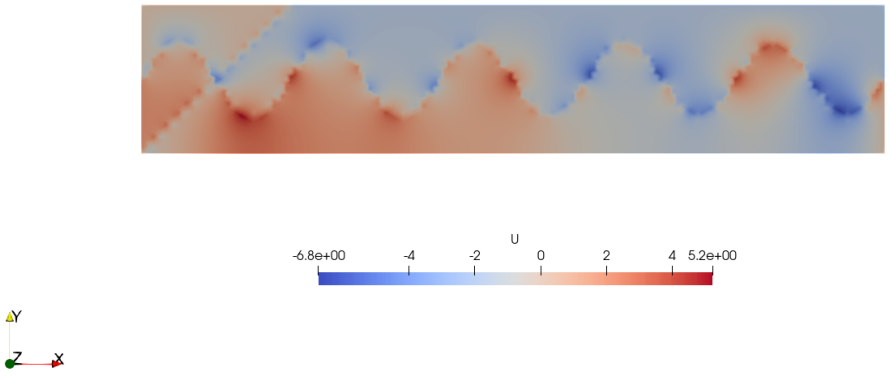


Figure 1: Solution on a coarse mesh

2D example with discontinuous solution and automatic mesh adaption in sequential

One could uniformly refine the mesh and solve this problem with a very fine mesh, this is not the goal of this tutorial, instead of using a finer uniform mesh, we will perform mesh adaption to search for an optimized mesh that is suitable for this simulation. In order to do that, we add the flag -adaptmesh and specify some more parameters. Parameters such as the number of iterations for adaption, the minimum and maximum element sizes can be edited in ControlParameters.edp.

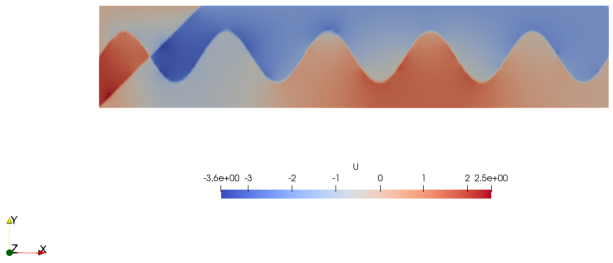
```
PSD_PreProcess -dimension 2 -problem poisson -adaptmesh -adaptmesh_metric_backend freefem \
-adaptmesh_backend freefem -postprocess u -sequential
```

nd freefem -postprocess u -sequential \end{lstlisting}

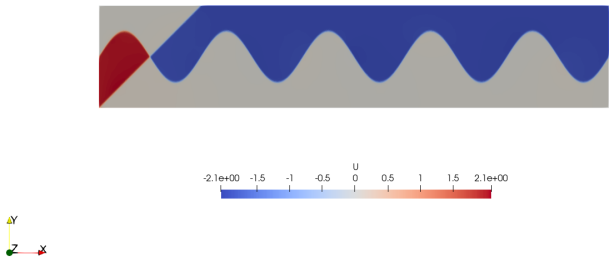
For this code we have demanded for native FreeFEM kernel for metric calculation and FreeFEM handles the adaption. Metric and adaption are two main processes involved to perform automatic mesh adaption given a solution. Just like above we will now solve and visulize the solution, for solving:

```
PSD_Solve_Seq Main.edp -mesh ./../Meshes/2D/bar.msh
```

In order to visualize the solution, we open the group of solution files and view the results of adaption iterations one by one. We start getting a satisfactory approximated solution by the fourth iteration.



(a) Iteration 2



(b) Iteration 4

Figure 2: Solution on an adapted mesh

2D example with discontinuous solution and automatic mesh adaption in sequential with different backends

We can also run with different backends. Each have their respective parameters documented on ControlParameters.edp. These are the valid configurations:

- FreeFEM as mesh backend combined with FreeFEM as metric backend

```
PSD_PreProcess -dimension 2 -problem poisson -adaptmesh -adaptmesh_metric_backend freefem \
-adaptmesh_backend freefem -postprocess u -sequential
```

nd freefem -postprocess u -sequential \end{lstlisting}

- MMG as mesh backend combined with FreeFEM as metric backend

```
PSD_PreProcess -dimension 2 -problem poisson -adaptmesh -adaptmesh_metric_backend freefem \
-adaptmesh_backend mmg -postprocess u -sequential
```

nd mmg -postprocess u -sequential \end{lstlisting}

- MMG as mesh backend combined with MshMet as metric backend. Notice that with our example, further adjustments are needed for a MshMet metric backend. MshMet metrics do not scale with respect to the given mesh like FreeFEM metrics. For better refinement, we suggest setting hmin to 0.001, nnu to 1 and eps to 0.02.

```
PSD_PreProcess -dimension 2 -problem poisson -adaptmesh -adaptmesh_metric_backend mshmet \
-adaptmesh_backend mmg -postprocess u -sequential
```

nd mmg -postprocess u -sequential \end{lstlisting}

We can also reach a good approximation of the solution by enabling anisotropy through the -adaptmesh_type flag.

```
PSD_PreProcess -dimension 2 -problem poisson -adaptmesh -adaptmesh_metric_backend freefem \
-adaptmesh_backend mmg -adaptmesh_type anisotropic -postprocess u -sequential
```

nd mmg -adaptmesh_type anisotropic -postprocess u -sequential \end{lstlisting}

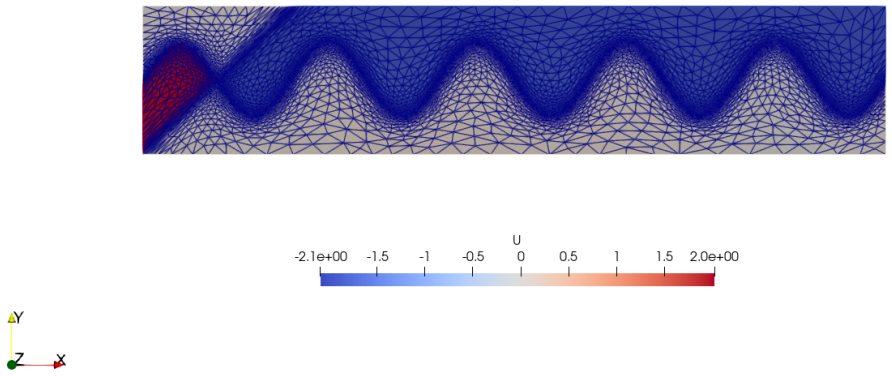


Figure 3: Solution on an anisotropic adapted mesh

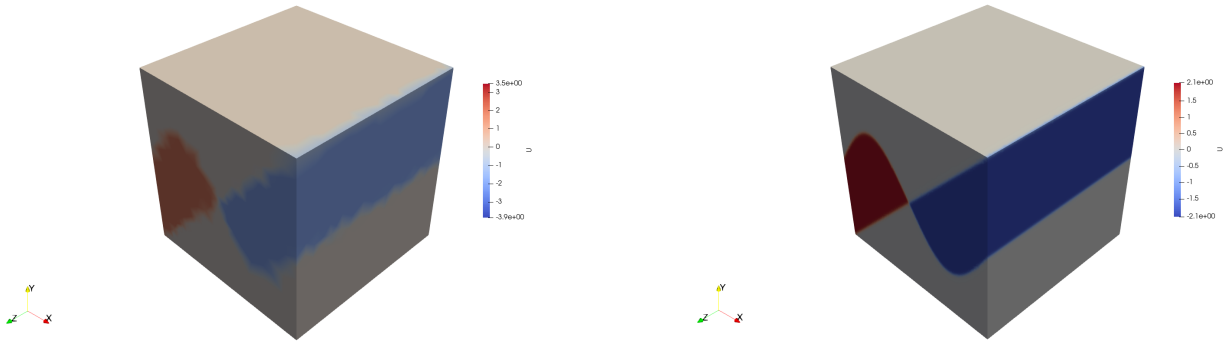
3D example with discontinuous solution and automatic mesh adaption in sequential and parallel

We solve the same problem but on a 3D cube keeping the solution constant across the z axis. Adaption in 3D is done using mshmet for metric calculation and mmg for mesh adaption. In ControlParameters.edp, set the number of adaption iterations to 5 (adaptIter).

```
PSD_PreProcess -dimension 3 -problem poisson -adaptmesh -adaptmesh_metric_backend mshmet \
-adaptmesh_backend mmg -postprocess u -sequential
```

nd mmg -postprocess u -sequential \end{lstlisting}

```
PSD_Solve_Seq Main.edp -mesh ./../Meshes/3D/cube.msh
```



(a) Iteration 2

(b) Iteration 4

Figure 4: Solution on a 3D adapted mesh

Notice how solving and adaption take a lot of time in this example; it is generally the case when we move to 3D as we have an additional axis and more elements on the initial mesh. This is why we prefer to parallelize the process. We omit the -sequential flag to enable domain-decomposition, i.e, parallel computing, and change the adaption backend to parmmg. We specify the number of procs to PSD_Solve using the flag -np. Parallel solving can also be done in 2D but is generally not necessary while parallel adaption (parmmg) is only available in 3D.

```
PSD_PreProcess -dimension 3 -problem poisson -adaptmesh -adaptmesh_metric_backend mshmet \
-adaptmesh_backend parmmg -postprocess u
```

nd parmmg -postprocess u \end{lstlisting}

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/3D/cube.msh -v 0
```

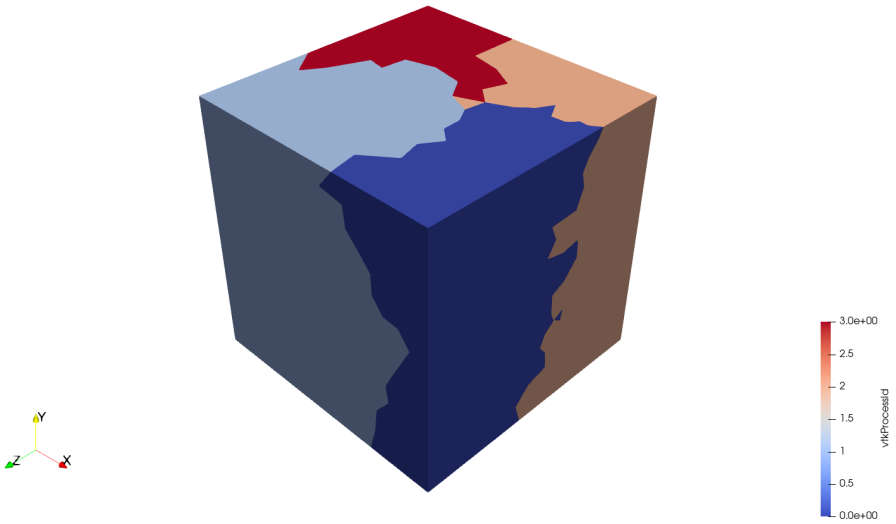


Figure 5: Partitioned 3D mesh

We now have two parameters representing a number of iterations in the ControlParameters.edp file. adaptIter is the number of adaption iterations, while parMmgIter is the number of inner iterations that ParMmg does to counter the effect of partitioning on the results, that is by repartitioning and remeshing. parMmgIter should be set to a value that balances between speed and accuracy. The current default value works well for this example.

3D example with discontinuous solution and automatic mesh adaption in parallel with different strategies

For certain mesh sizes and functions, it can be beneficial to group processes (only for the adaption step) on a bigger partition of the mesh. It was not the case in the example above. The number of groups can either be fixed across iterations and unrelated to the mesh (it can be changed in ControlParameters.edp), or be adaptive to the number of elements across iterations; these are the respective commands of the two methods.

- user controlled regrouping method

```
PSD_PreProcess -dimension 3 -problem poisson -adaptmesh -adaptmesh_metric_backend mshmet \
-adaptmesh_backend parmmg -adaptmesh_parmmg_method partition_regrouping -postprocess u
```

nd parmmg -adaptmesh_parmmg_method partition_regrouping -postprocess u \end{lstlisting}

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/3D/cube.msh
```

- Adhoc regrouping method.

```
PSD_PreProcess -dimension 3 -problem poisson -adaptmesh -adaptmesh_metric_backend mshmet \
-adaptmesh_backend parmmg -adaptmesh_parmmg_method partition_automatic_regrouping -postprocess u
```

nd parmmg -adaptmesh_parmmg_method partition_automatic_regrouping -postprocess u \end{lstlisting}

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/3D/cube.msh
```