Homework 1
Programming for Engineers

Corbella Coll, Xavier
xcorbellacoll@gmail.com

Komala Sheshachala, Sanjay
sanjayks01@gmail.com

February 29, 2016

https://github.com/sanjayks01/Homework1.git

Abstract

A steady-state scalar diffusion equation is solved with inlet, outlet and homogeneus essential boundary conditions as defined in the problem statement. A single MATLAB code capable of solving the problem for given 2D/3D geometry using tri/quad or tetra/hexa elements is developed. As an additional feature, coupling with preprocessing capabilities of GiD is introduced, enabling the user to input desired geometry with user-defined boundary conditions. This also allows for mesh refinement up to desired level of solution accuracy based on which a convergence study was performed to comment on the accuracy of the solution.

1 Introduction

Scalar diffusion is observed in many physical phenomena such as spatial temperature distribution, mass diffusion, etc. We solve the problem numerically using FEM with MATLAB as our tool. The equations governing the physics are given below:

$$\nabla \cdot (\mu \nabla u) = s \quad \text{in} \quad \Omega \tag{1}$$

$$u = 1 \quad \text{on} \quad \Gamma_{inlet} \tag{2}$$

$$u = 1 \quad \text{on} \quad \Gamma_{outlet} \tag{3}$$

$$(\mu \nabla u) \cdot \mathbf{n} = 0 \quad \text{otherwise} \tag{4}$$

Figure 1: Geometry and boundary conditions

Source term s = 0 and diffusivity $\mu = 1$ are given. The geometry with boundary conditions for the given problem are shown in Fig. 1.

In this document, we begin with describing the code structure and the versioning strategy followed. We then elucidate the method of using the code. We then move on to detail the testing procedure followed, the challenges encountered and overcome. In the final section, we make the code more versatile by providing additional GiD integration. We explain how GiD was utilized to expand the preprocess capability for the MATLAB solver and use it to comment on solution accuracy with mesh refinement.

2 Code Development

This section deals with explaining the main structure of the code. Although the additional features of GiD integration is explained mainly in the succeeding section, we briefly explain how it is linked to the main code here. Also, we guide you through the work-flow of the code, highlighting the challenges encountered while formulating it and how it was overcome.

2.1 Code description and usage

The main file is 'main.m'. This functions ask for the name of the input files provided by the user input for the specific problem to be solved and outputs a .vtk file that can be read by Paraview for post-processing. The name of the input files accounting for the meshes and boundary conditions provided in the statement of the problem are the following:

1. **C2D3:** Continuum 2D element with 3 dof (linear triangle)
2. **C2D6:** Continuum 2D element with 6 dof (quadratic triangle)
3. **C2D4:** Continuum 2D element with 4 dof (linear quad)

4. **C2D8:** Continuum 2D element with 8 dof (quadratic quad)
5. **C3D4:** Continuum 3D element with 4 dof (linear tetrahedra)
6. **C3D10:** Continuum 3D element with 10 dof (quadratic tetrahedra)
7. **C3D8:** Continuum 3D element with 8 dof (linear hexahedra)
8. **C3D20:** Continuum 3D element with 20 dof (quadratic hexahedra)

Every problem needs 4 input files: a file containing the connectivity matrix ('elem_filename'), another containing the coordinates of the nodes ('nodes_filename') and two more containing the nodes where Dirichlet boundary conditions are imposed ('inlet_filename.dat' for inlet, 'outlet_filename.dat' for outlet). The input files for the problems defined above are contained in the 'Model' subfolder. Nodal coordinates, connectivity, nodes of the input and output boundaries are obtained from these files which are of plain-text format.

After asking for the name of the input files, the program reads them, extracts the information related to the number and type of elements, number of nodes and dimensions of the problem. Once the element type to be used is known, the code calls the suitable Matlab function containing the reference element data. These functions can be found in the 'Elements' subfolder. They are labelled according to the dimension and number of nodes and contain the coordinates and weights of the Gauss points, values of shape functions and shape function derivatives, which are required for computing the stiffness matrix and force vector. It is to be noted that the variable 'type' is required for writing the .vtk file for Paraview, and not necessary for computing the solution.

These input data are used to define the general system of equations without boundary conditions. To do so, the function 'CreateMatrix.m' is called. This function builds the elemental contributions calling the functions 'MatEl2D.m' for 2D or 'MatEl3D.m' for 3D problems, and then assembles the global system.

After that, the boundary conditions are implemented and the linear system is ready to be solved. After solving the system, the results are written onto a .vtk file using the template files 'geo2D_vtk.m' and 'geo3D_vtk.m'. The structure of these files were determined from the desired output file from the solver required for Paraview.

For other user generated (using GiD) preprocess files containing mesh data with boundary conditions, the subfolder 'GiD' has selected examples which where created using GiD. The implementation and utilisation of a 'GiD' problem type is explained further in section 3.

Hence the program performs the calculations for problem geometry and user-defined geometry and boundary conditions. The 'main.m' file gives

a one click solution in the form of .vtk file that can be visualized using Paraview. For the sake of convenience, we have included all the .vtk files generated for the different cases for ready solution visualization. They are placed in the 'Paraview' subfolder.

2.2 Code versioning and testing

The code was versioned and tested using 'github'. The versioning strategy followed is as indicated below:

1. The solution was first solved for 2D cases (problems beginning with C2). This included adding new geometry files and boundary condition files from the files provided in the statement of the problem. Also the Matlab functions to account for the shape functions and shape functions derivatives for each of the element types required to solve the problems provided were designed. The generic 'geo2D_vtk.m' file had to be designed to create the output files for the various 2D cases.

2. The solution was extended to 3D cases (problems beginning with C3). This required adding the files for input, reference element files and the 'geo3D_vtk.m' file for 3D cases.

3. The next addition was the GiD files and examples. Also tweaking the code for troubleshooting (correcting the incorrect definition of shape functions for C2D6)

4. Code beautification to improve readability and adding comments to code block to improve understanding was carried out. Files were organized into relevant folders for easy segregation.

5. In the final stage, files related to convergence study of GiD examples were added.

Testing was carried out before each commit to github. This eliminated carrying forward bugs in the code. This was done by running the examples to make sure correct results were obtained.

Several challenges were encountered and overcome during the project. They turned into meaningful learning outcomes for the authors. These can be summarized as below:

1. To generate the vtk files, we had to write a generic program 'geo2D_vtk.m' and 'geo3D_vtk.m' which required us to learn the vtk documentation of Paraview.

2. The code organization required 'cleaning' of input files to make them ready for use in the code. This showed us the importance of code interoperability and how this can be a big bottleneck in huge projects.

3. Creation of Elements with specific format of output of variables such as shape function, its derivative, Gauss point definition required to study the solver requirement of these variables. Learning this gave deep insight into the working of the solver as a whole.

4. As the project grew in size, we learnt to organization and versioning is key for an smooth execution of tasks.

5. Addition of GiD problemtype was a problem extension that provided us with an opportunity to enhance our experience with GiD and appreciate its utility as a preprocess tool.

3 Additional Capabilities

3.1 Pre-process: GiD

The Matlab code developed can be easily used to solve for the steady-state scalar equation when having the 4 required input data files. However, generating these files for a new model is quite hard, especially when a high number of nodes must be used. This problem can be solved using a pre-process software that can create a geometry, apply the boundary conditions, mesh the domain following the parameters given by the user and then write all this information into the input files used by the Matlab code. One of the softwares that can do this is GiD. GiD is a pre/post-processor that can be customized to be used for different solvers, and presents a wide range of possibilities and features.

In this work, a simple GiD problem type was implemented. This problem type is included in the github folder, and its name is Homework1.gid. In order to be used, it must be copied into the "problemtypes" folder in the GiD installation folder.

This problem type can be used to generate the data files that will be read by the solver for 2D and 3D geometries with simple boundary conditions (1 inlet, 1 outlet and homogeneous natural boundary conditions).

To solve a problem using GiD and the solver implemented in Matlab, the next steps must be followed:

1. Create a 2D or 3D geometry.

2. Apply boundary conditions: Inlet and outlet.

3. Select the type of element and its size and mesh the domain.

Figure 2: Location of the point x=0,y=0

4. Click the "Calculate" button and GiD will automatically write the input files for the solver.

5. Copy the input file to the Solver's folder and solve the problem using the Matlab code.

3.2 Convergence study

The GiD problem type implemented was used to study the convergence of the different elements for 2D problems (linear and quadratic triangles and quadrilaterals). To do so, a steady-state diffusion problem was solved in the same domain as before (see Fig. 1), and the solution obtained at x=0,y=0 (see Fig. 2) was compared for different element types and sizes.
The results obtained are depicted in Fig. 3 and Fig. 4. The relative errors are computed with respect to the results obtained using quadratic triangles and 80563 nodes.
The convergence plots show that the accuracy obtained with second order elements is better than for linear elements, especially for finer meshes. If comparing the linear elements, the errors obtained with the 4-node quadrilateral are larger than those obtained using 3-node triangles. The same behaviour is observed for quadratic elements: The accuracy obtained with the 8-node quadrilateral is lower than the accuracy of the 6-node triangle. The order of convergence can be approximated as the slope of the logarithmic plot. As is shown in the figures, the average slope obtained with the linear elements is around 0.8, so they can be considered as linear. For the 8-node quadrilateral, the average slope obtained is 1.5, while for the 6-node triangles it is 1.8, almost second order.

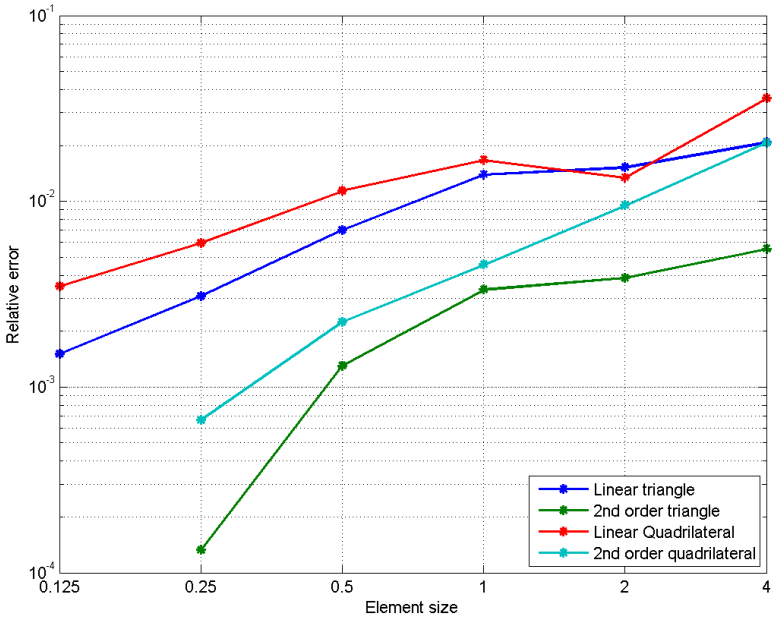

Figure 3: Relative error vs Element Size

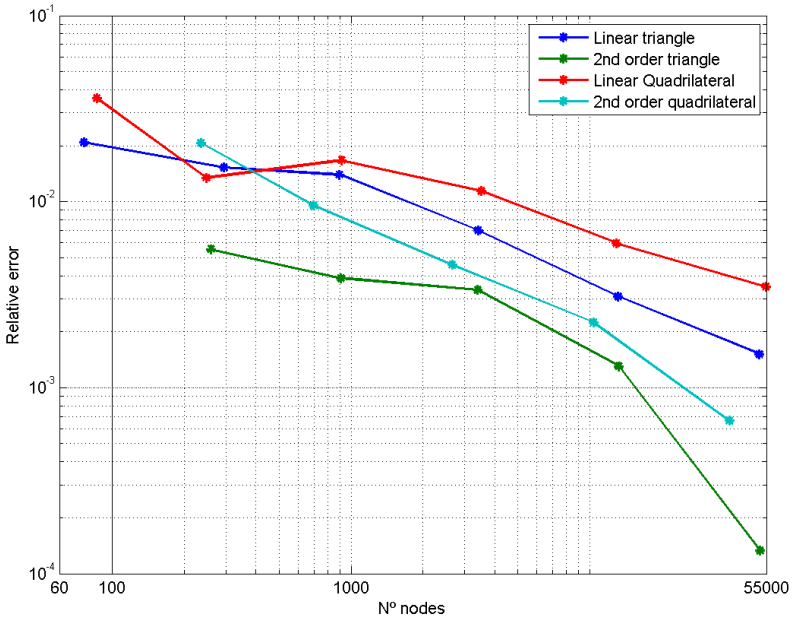

Figure 4: Relative error vs number of nodes

4 Results for the given models

The results obtained with the 8 different meshes given are depicted here. The results obtained for all the cases look similar.

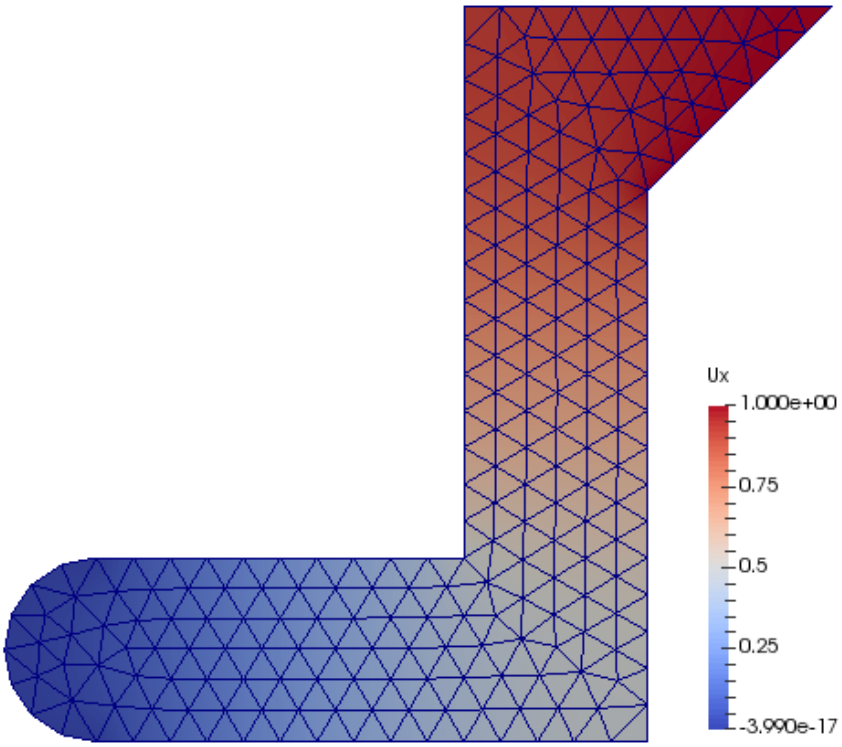

Figure 5: u for linear triangles

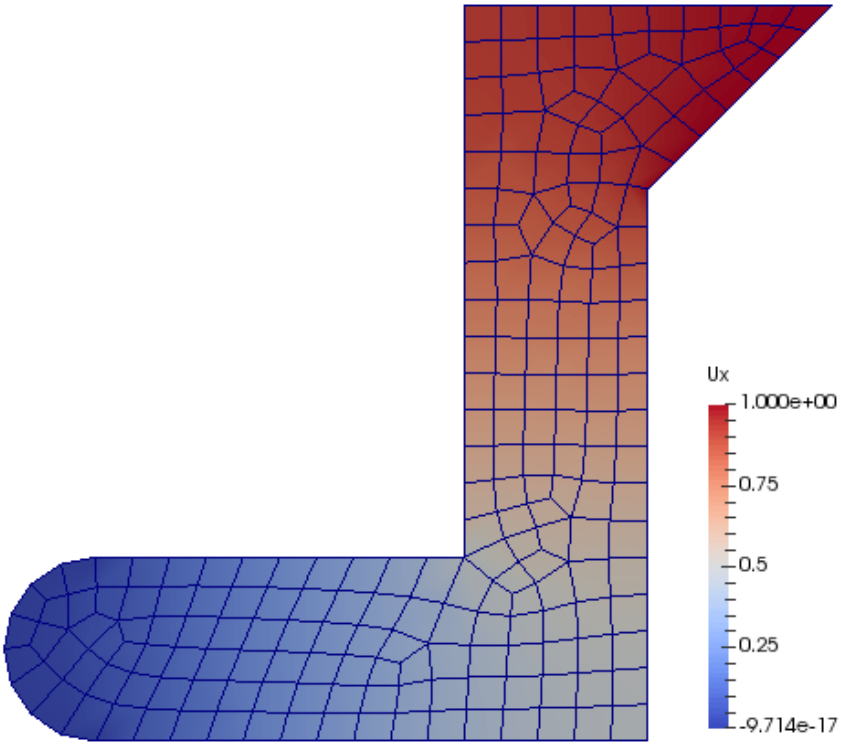

Figure 6: u for linear quad

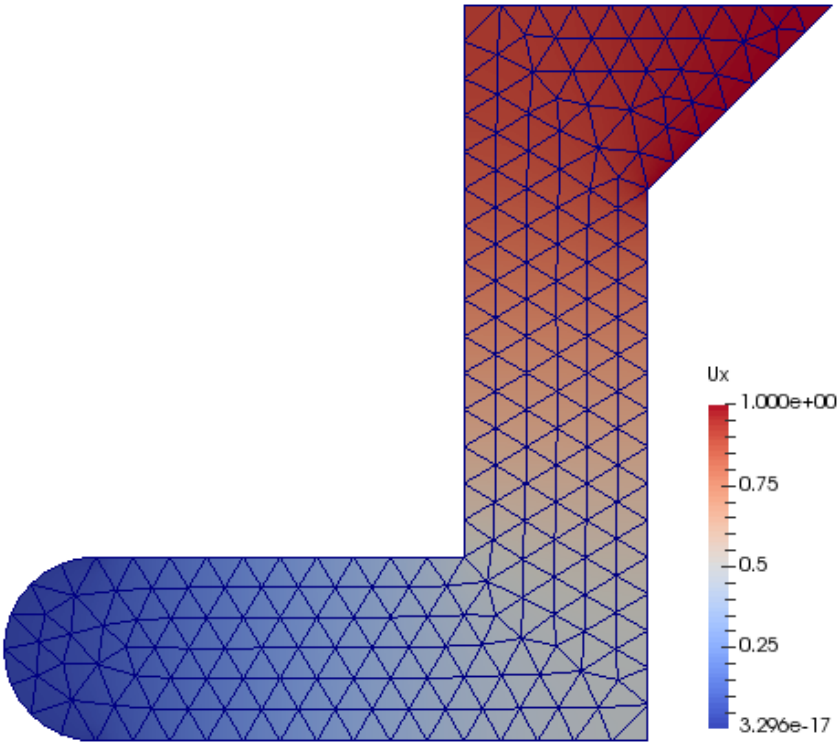

Figure 7: u for 6-node triangles

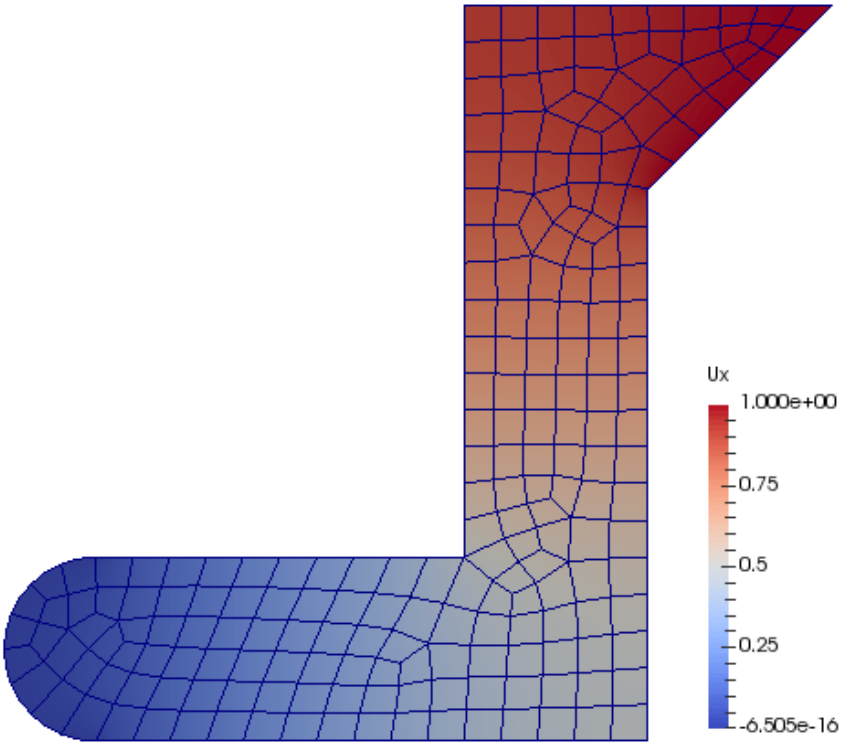

Figure 8: u for 8-nodes quads

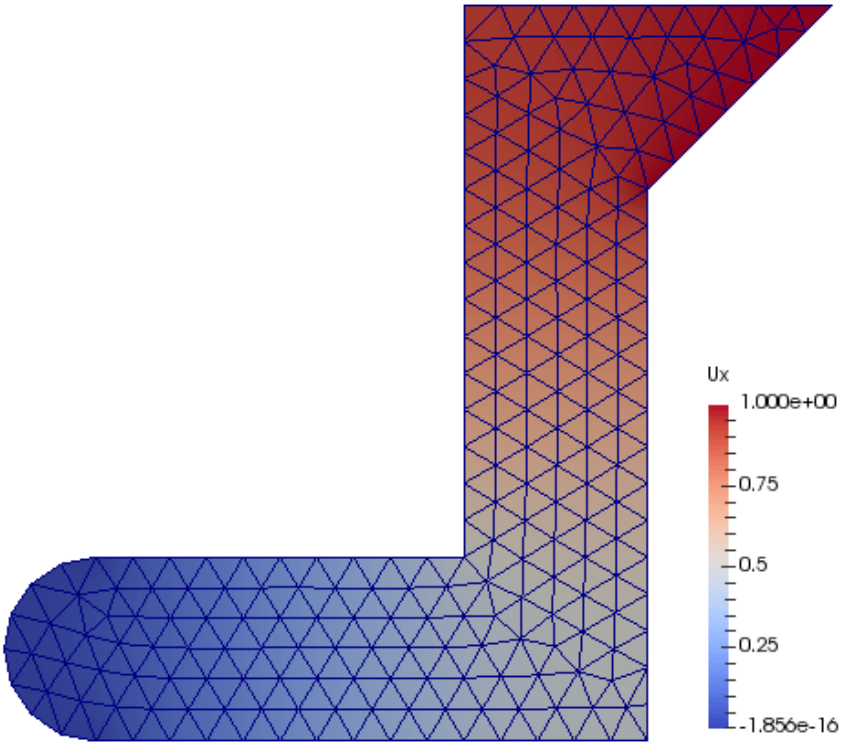

Figure 9: u for linear tetrahedras

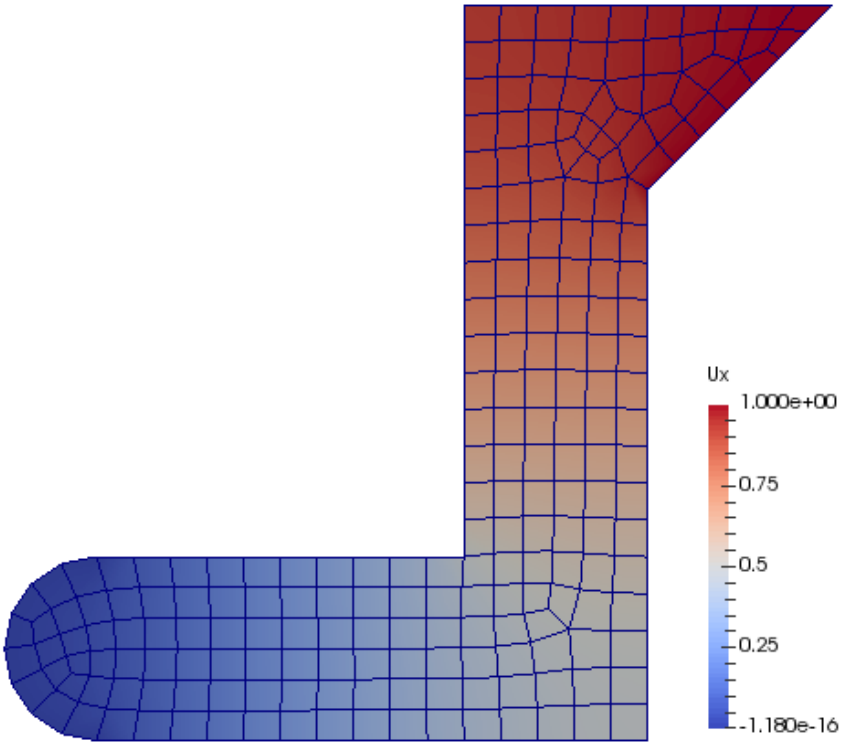

Figure 10: u for linear hexahedras

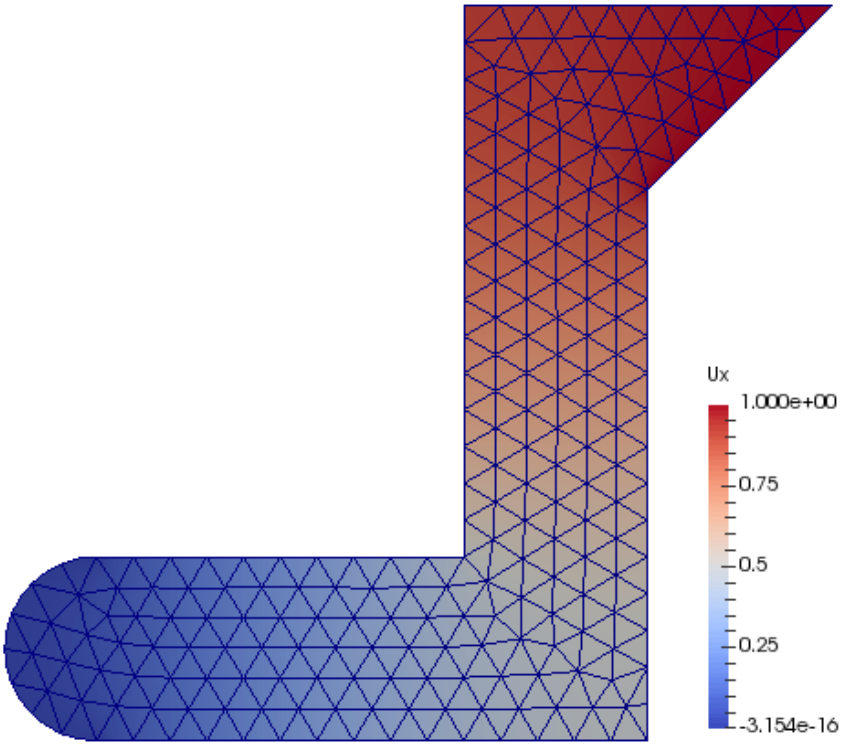

Figure 11: u for 10-node tetrahedras

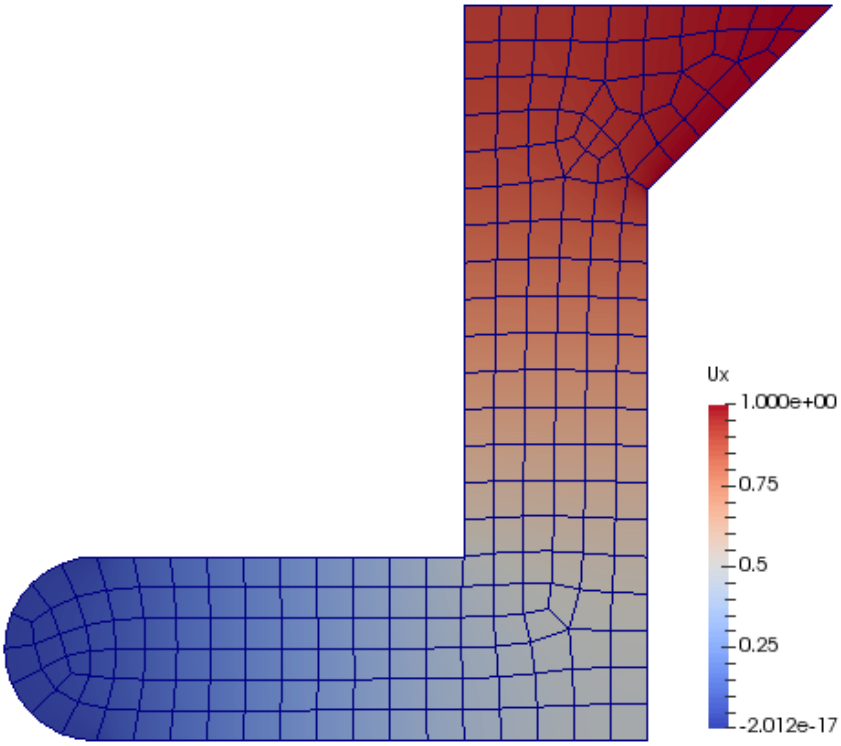

Figure 12: u for 20-nodes hexahedras