Programmig for Engineers and Scientists

Martí Burcet, Adrià Galofré

March 1st, 2016

1 Problem

The aim of this code is to solve the Poisson equation:

$$\begin{cases} \nabla \cdot (\nu \nabla u) = s & \text{in } \Omega, \\ u = 1 & \text{on } \Gamma_{Inlet}, \\ u = 0 & \text{on } \Gamma_{Outlet}, \\ (\nu \nabla u) \cdot \mathbf{n} = 0 & \text{otherwise} \end{cases} \tag{1}$$

with a given source term s=0 and a diffusivity equals v=1.

The code we were asked to develop had to be able to read different geometries (both 2D and 3D), boundary conditions and solve the problem with different types of elements.

2 Starting point

In order to start the project it was given to us a template main code that controls the data and solved a FEM Poisson problem for a given geometry in 2D. A part of the main code we were also provided with the necessary programs to solve this problem that then we extended to different element types and dimension. These codes are:

- **CreateMatrix.m**: this program receives as inputs the information of the element: shape functions and its derivatives, number of nodes of the element, position of the integration points as well as weights for the numerical integration, matrix of nodal coordinates and matrix of connectivities of the elements.
- **MatEl.m**: this function is called inside the *CreateMatrix.m* and computes the integrals of the element stiffness matrices in the isoparametric domain with the numerical Gauss quadrature correspondent to the order of the interpolation functions. This function receives as inputs the element nodal coordinates, the shape functions and derivatives, and the Gauss points and weights.
- **Isopar**: the function is called inside the *CreateMatrix.m* and evaluates the isoparametric transformation from global cartesian coordinates to local isoparametric coordinates to interpolate the function in the element.
- **SourceTerm.m**: also inside *CreateMatrix.m* the element force vector is computed calling the function *SourceTerm.m* that in our case is equal to 0.
- **Elements.m**: this function contains already coded the shape functions, shape

functions derivatives and integration points for all the elements that we were asked to use but in a single file. We have taken advantage of the already coded functions even though some were wrong and we had to changed.

- **Element coordinates and connectivities**: for testing reasons we were also provided with a nodes coordinates matrix as well as with a connectivities matrix for each of the elements to implement. The boundary conditions were also given.

3 Description of the main code

3.1 Initial approach

As we have said, we were provided with a program *elements.m* that have already coded the necessary information for all the elements that we were asked in one file. In the first moment we thought of split the function in the subfunctions that it contains (*shapefunctions*, *shapefunctionsderivs*, *numberofintegrationpoints*, *integrationpoints*, *integrationweights*) and call each one directly from the main code.

The problem from our point of view of this approach is that in each of this functions there should be lots of *if* commands, what we thought will make the program run slow. For this reason we decided to create a function for each of the element types that provides directly all the relevant information of the element just calling it. Doing this we reduced the number of conditionals in the running and we expected the program to run faster.

In addition the advantage of this structure of the code is that it easier to test each of the functions separately without changing the main code. This will also be an advantage if we want to extend the code to more element types and geometries. In that case the only thing we should do is to add the function of the element in the folder and also the nodes coordinates and element connectivities in its folder.

Now we will describe step by strep the different functions that our main code call in order to solve the problem.

3.2 Input files

The first thing done was to implement that the code can read a mesh and boundary conditions from an input file, and also to be able to solve 2D/3D geometries. To do so we have extended a FE given at the initial classes of PES. So the code needs the next different input files to run:

1. **nodes.dat**: In this file an array where the first column is the number identifier of the node, and the next columns are the x and y coordinates for 2D geometry or the x,y and z coordinates for a 3D geometry. These data files are placed on the folder *nodes* and have to be called by screen entering in the main code.

2. **elements.dat**: With similar structure of the *nodes.dat* file, here, connectivities for every node are described. The files are placed in the folder *elements* and are asked in the main code by keyboard.

3. **group.dat**: This kind of file is required for the boundary conditions. Inside groups of nodes for inlet and outlet BC are given. The files are saved in the folder *groups* and are

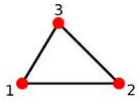
loaded automatically once detected the element type.

3.3 Element functions

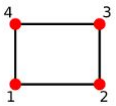
Also the use of different 2D or 3D types of elements was required. So depending on the *nodes.dat* and *elements.dat* and element or another is used. It means that entering the right nodes and connectivities is the way to choose an element to solve the problem. we also give the opportunity to run the example provided by the master choosing the element type for a given geometry.

These element functions give as outputs the shape funciton matrix, shape function derivatives matrix, the vector of position of the Gauss points for the numerical integration and the weights. The next element types were implemented:

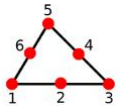
1. **C2D3**: 2D Triangular linear element with 3 nodes, one in each vertex.



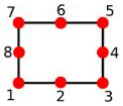
2. **C2D4**: 2D Quadrilateral linear element with 4 nodes.



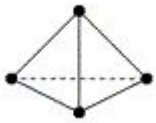
3. **C2D6**: 2D Triangular quadratic element with 6 nodes.



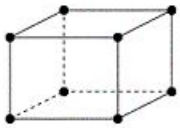
4. **C2D8**: 2D 8-noded Quadrilateral quadratic element.



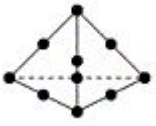
5. **C3D4**: 3D 4-noded tetrahedral linear element.



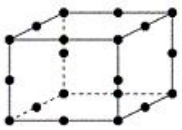
6. **C3D8**: 3D 8-noded hexahedral linear element.



7. **C3D10**: 3D 10-noded tetrahedral quadratic element.



8. **C3D20**: 3D 20-noded hexahedral element.



3.4 Solving scheme

Once the element is chosen we proceed to solving the problem. This is done by the following path: the function *CreateMatrix* (provided) constructs the element matrices according to the element connectivities. At the same time, this function calls inside the function *MatEl* that computes the integrals of the stiffness matrix and force vector.

The function *MatEl* was modified to be able to compute 3D elements, by extending the Jacobian matrix to three coordinates. The function *Isopar* does the isoparametric transformation of the nodes.

Finally in the main code the boundary conditions are introduced (previously loaded from the *.mat* files) and the system is solved.

3.5 Writing the output files

Finally the results that in Matlab are given in the vector **Temp** are written in a *.vtk* file that writes a text file that can be opened and visualized in Paraview. The writing of these files is done by the functions **geoXXX.m**.

4 Testing the code

To test the code and the implementation of the elements 8 different nodes, elements and group files corresponding to each element implementation where tested. All the files correspond to the next geometry:

4.1 Results

The results obtained with the code are presented in the next points, with a brief comparison between them. In Figures ?? and ?? we can see the comparison between the results obtained with a mesh of linear triangles and linear quadrilaterals. We can see that the result files are well read in Paraview so we can assure that the output files were well implemented.

Regarding the results of the different elements, we can see that they are equal by looking at the figures. Both 2D and 3D cases are equally well executed in Matlab. Even though we havent presented the quadratic cases, the precision of them compared to the linear cases aren't significant.

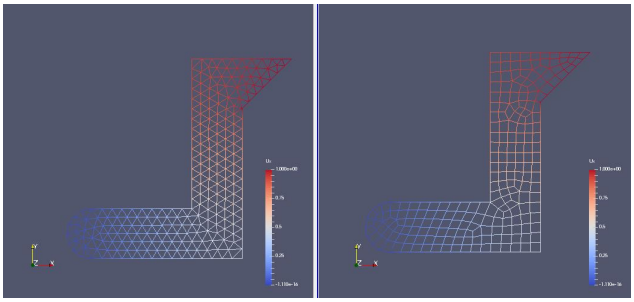


Figure 1: Comparative between linear triangle and linear quadrilateral.

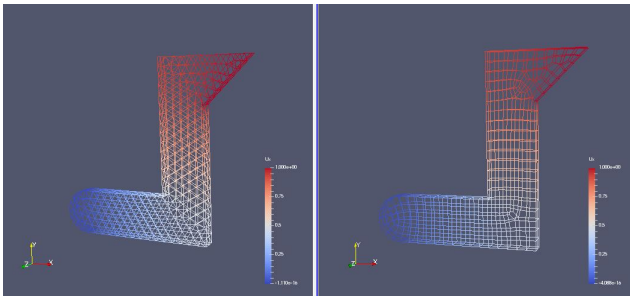


Figure 2: Comparative between linear tetrahedral and linear hexahedral.

4.2 Running time

In the following table we can see the comparison of running time of the different elements. We can see that, in general terms, the quadrilateral elements are faster than the triangular once. Of course this is a very small example and the run time is not significant, but it is always good to prove that for larger problems. On the other hand the quadratic approaches are slower than the linear ones because they use more nodes for the interpolation. Finally of course 3D meshes are slower than plane ones as expected.

Element	Time (seconds)
C2D3	0.194118
C2D4	0.191742
C2D6	0.495336
C2D8	0.454595
C3D4	1.159373
C3D8	0.780758
C3D10	10.085582
C3D20	5.051605

5 Conclusions

After faced the problem of coding a FEM model in Matlab we have realised that the structure of the program is very important because lots of functions plays a role and it is crucial to be able to test each one separately. For this reason we decided to code different programs for different elements that can be integrated in a general code. Our structure is easily modifiable and extended to more complex elements and geometries because it is only needed to add the correspondent input files to the folder and allow the main code to use them.

About the results we have obtained for the example problem, the precision for this small problem is not affected too much by the mesh and element because the mesh is small enough. An important conclusion of the comparison between meshes is that, even for this easy geometry the element type really affects the computing time. It is observed that quadrilateral elements are in general faster than triangular ones, so if possible, they are recommendable.

6 Link

The main code and all the implementationts can be found in:
https://github.com/adriagalofre/HW1_Burcet_Marti_Galofre_Adria.git