UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona, 19.03.2018

Implementation of a FEM code

Homework 1 part B

PROGRAMMING FOR ENGINEERS AND SCIENTISTS
2017/18

Professor | Sergio Zlotnik

Students | Marin Enrico

Pan Lei

Rubino Marcello

SUMMARY

1. Introduction

The purpose of this work is to implement an FE code in Matlab to solve a typical fluid dynamic problem, the definition of the velocity field described by the Poisson equation in 2D dimension.
If the flow is irrotational, the velocity can be expressed in terms of a velocity potential. The velocity field, therefore, is constructed from the velocity generated by a scalar potential *u* generated by the mass sources.
So, we have developed a MATLAB program to solve the following Poisson equation:

$$\begin{aligned} &\nabla \cdot (k(x)\nabla u) = 0 \qquad in\ \Omega \\ &\nabla u \cdot \boldsymbol{n} = g \qquad on\ \Gamma_N \\ &u = u_0 \qquad on\ \partial\Omega \setminus \Gamma_N \end{aligned}$$

where Ω is the computational domain, $\partial\Omega$ is the boundary, $\boldsymbol{n}$ is the outward unit normal vector and $k(x)$ is a not constant material property.

One method to solve the Poisson equation is the weak formulation, we derive it and multiply it by the weighting function ω:

$$\int_\Omega \nabla\omega \cdot (k(x)\nabla u)d\Omega = \int_{\Gamma_N} \omega g d\Gamma$$

We need to find a space of functions, $H_0^1(\Omega)$, where the derivative of functions are square integrable:

$\omega \in H_0^1(\Omega)$: $\int_\Omega \nabla\omega \cdot (k(x)\nabla u)d\Omega = \int_{\Gamma_N} \omega g d\Gamma$
$\forall\ \omega \in H_0^1(\Omega)$

Afterwards, applying Galerkin method it is possible to get the solution equation:

$$\mathrm{Ku} = \mathrm{f}$$

Where:

$$K_{ij} = \int_\Omega \nabla N_i \cdot \left(k(x)\nabla N_j\right)d\Omega$$

$$f_i = \int_{\Gamma_N} \nabla N_i \cdot g d\Gamma$$

In our program, we have considered three types of elements: 1-D isoparametric element, isoparametric quadrilateral element and isoparametric triangular element. The 1-D isoparametric element is used for calculating the integration of the Neumann boundary conditions.
By combining the isoparametric element and Gaussian quadrature, we can get:

$$\int_\Omega \nabla N_i \cdot \left(k(x)\nabla N_j\right)d\Omega = \sum_{g=1}^{ng} \omega_g (\boldsymbol{J}(z_g)^{-1}\nabla_{\xi\eta}N_i(z_g)) \cdot (\mu\boldsymbol{J}(z_g)^{-1}\nabla_{\xi\eta}N_j(z_g))|\boldsymbol{J}(z_g)|$$

$$
\mathrm{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{pmatrix}
$$

Where z_g is the Gauss points for 2-D elements.

$$
\int_{\Gamma_N} \nabla N_i \cdot g d\Gamma = \sum_{g=1}^{ng\prime} \omega_g{}'(\boldsymbol{J}(z_g{}')^{-1} \nabla_{\xi\eta} N_i(z_g{}')\, |\boldsymbol{J}(z_g{}')|
$$

Where $z_g{}'$ is the Gauss points for 1-D elements, $\boldsymbol{J}$ is the Jacobi Matrix.

$$
\mathrm{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{pmatrix}
$$

And:

$$
x = \textstyle\sum_1^n N_i(\xi, \eta) x_i;
$$

$$
y = \sum_1^n N_i(\xi, \eta) y_i
$$

2.Main code

2.1 Data

We loaded five datasets in our Matlab code: “T3”, “T6”, “Q4”, “Q8”and “Q9” which contain T, connectivity, and X, nodal coordinates, matrices. T is referred to a triangular mesh analysis, Q is referred to a quadrilateral one whereas the number describes how many nodes has each element of the mesh.

We decided to apply the second example given in the slide as a domain input:

$$
\begin{cases} \triangle u = 0 \quad \text{in } [0,1] \times [0,1] \\ \dfrac{\partial u}{\partial n}(x,0) = -1 \\ \dfrac{\partial u}{\partial n}(x,1) = 1 \\ u(0,y) = y \\ u(1,y) = 1 + y \end{cases} \qquad \boxed{u(x,y) = x + y}
$$

Other data implemented are the Neumann constant “g”, given by the example, the scalar variable “nondes” equal to the total number of nodes, the scalar variable “noelem” equal to the total number of elements, a matrix “k” with a number of rows equal to “noelem” and two columns, describing the material property, and a matrix “elementInfos” made by T and X matrices to describe all the elements.

More specifically, the matrix “k” is implemented with a cycle “if-else” to consider both the constant condition and the linear condition; if we are in first case the matrix become a vector of ones.

As a last step we called the function “faces”, explained in the next paragraph, that gives “elementFaces”, a tensor. It contains the number of the element in the first index and in the second and third the ID of the faces.

Subsequently we created the structure of the mesh storing together the data loaded (X, “elementInfos”, “elementFaces”) and assigning them to the substructures “thisMesh.nodalCoords”, “thisMesh.elementInfos” and “thisMesh.elementFaces” respectively.

To complete the coding of the mesh we called the function “shape” to implement the 2D dimension shape functions and their gradients with an input equal to 1. These two matrix are then stored and assigned to the structures “thisMesh.shapeFunc” and “thisMesh.shapeGrad”.

2.2 Boundary Conditions

Above all we wrote the geometrical Boundary Condition using the Matlab function “find” to catch all the zeros and all the ones on the two columns from the matrix of the structure “thisMesh.nodalCoords”. Using these indexes we achieved four vectors with 21 rows: x_0, y_0, x_1 and y_1.

After that, we created four scalar variables to measure the length of each side of the mesh, called “length_nodes” avoiding the double counting of the mesh’s corners.
To develop Dirichlet Boundary Condition we created a matrix “nodeBC_Dir” with the values of x_1 and x_0 in the first column (taking into account the deleting of the last member of them) and a second column made by “nodeBC_val1” and “nodeBC_val2”, two vectors that express the formula given by the hypothesis $u(1, y) = 1 + y$.
The same method was applied to write Neumann Boundary Condition but considering y_0 and y_1 values to build “BC_Neu_nodes” first column and the value of g, with the sign ask according to the hypothesis, in the second one.

The next step was linking the nodes with Neumann Boundary Condition to the faces; and to achieve this we did a double cycle “for” and a single “if-else” applied to the tensor “thisMesh.elementFaces” (equal to “elementFaces”): the first cycle “for” runs along the first dimension, the second one runs along the second dimension with and “if-else” internal cycle where it is applied the function “ismember”. This Matlab function checks if there are Neumann Boundary Conditions both in the first and second row of the third dimension and, only if it is true, it is assigned the Neumann constant g in a new third row of the third dimension, if not it is assigned a zero.

2.3 Global Stiffness Matrix

Initially we implemented a new squared matrix K of zeros with the dimensions equal to “nonodes”, then we called three functions, described below, within a cycle “for”: “gaussianQuadrature” assigns different Gaussian characteristics to the dataset according to the type of the analysis (linear with triangles, quadratic with triangles, linear with quadrilaterals and quadratic with quadrilaterals); “elementMatrix” calculates the element stiffness matrix on the Gaussian points and adds contributions of the gradient of shape functions and the material contribution; “ConnectivityMatrix” creates the connectivity matrix in both the constant and hypothesis of the material’s property.
Subsequently it is coded the final equation to obtain K:
K = K + (Connectivity Matrix')*(Element Stiffness Matrix)*(Connectivity Matrix)
Finally, it is plotted the sparsity pattern with the function “spy”.

2.4 Global Rhs vector

As a first step we created the shape 1D with a function based on the input of Neumann equal to 1, then we initialized a vector characterized by a dimension equal to “nonodes” and only ones as components.
As before we ran three functions, within a cycle “for” along each element, called “gaussianQuadrature1D”, “NeumannIntegration” and “ConnectivityMatrix” to obtain the global Rhs vector “f”. It is important to notice that “NeumannIntegration” calculates the local rhs vector for each element.

2.5 Lagrange Multiplers

To enlarge the system, we created a new scalar variable “nDir” equal to the number of “nodeBC_Dir” matrix’s row. Subsequently we coded a new identity matrix “A” with a number of rows equal to “nDir” and a number of columns equal to “nonodes” and, then, a new vector “b” equal to second column of “nodeBC_Dir”.
With these elements it was possible to create the final larger system with a new bigger “K”, “Ktot” matrix, and a bigger “f”, “ftot” vector, and find the final solution, “sol”, as a ratio between “Ktot” and “ftot”.

2.6 Postprocess

The final objective was the calculation of pressure vector and velocity matrix, our solutions of the problem.
Vector pressure is obtained selecting only the first part of “sol” vector, equal to the length of “nonodes”.
To compute the velocity matrix we created a cycle “for” with three functions, after the initialization. “ConnectivityMatrix” and “gaussianQuadrature” are used before, whereas “velocityElement” is it new and it is used to calculate the velocity field of each element as a matrix.
The final equation is:
velocity = velocity + (Connectivity Matrix')*(velocity_elem)

3. Functions

In this paragraph we are going to describe schematically all the functions implemented in the code.

3.1 Faces

The inputs of the function are “elementInfos”, “noelem” and “matprob”.
The output is “elementFaces”.
This function is implemented to create a tensor that describes the faces of each element of the mesh. It combines the number of the single element with its ID in fact the first dimension refers to the element, the second and third one refer to the face ID.
It contains one principal cycle “if-else” which subdivides the case of “k”, material property, constant and linear. Inside each of these there are four possible description of the material’s size.

3.2 Shape

The input of the function is “inputShape”.
The outputs of the function are “shapeFunc”, ”shapeGrad”.
This function gives the 2D dimension shape functions and their gradients in the reference system only if the input shape is defined.
There is a cycle “if-else” to divide the two possibilities described before, and the shaped functions are implemented for all the possible meshes (triangular with three nodes, triangular with six nodes, quadrilateral with four nodes and quadrilateral with nine nodes).

3.3 Gaussian Quadrature

The inputs of the function are “elementInfos”, “el”, “matprob”.
The outputs of the function are “GaussPointWeight”, “GaussPointLocation”, “GaussNoPoints”.
This function gives different information about the Gaussian Quadrature to the database for each case of the problem.
There is a cycle “if-else” to divide the hypothesis of material’s property constant from the linear one, and, inside this cycle, four different analyses are developed (linear with triangles, quadratic with triangles, linear with quadrilaterals and quadratic with quadrilaterals).

3.4 KGauss

The inputs of the function are “k0”, “k1”, “gaussCsi”, “gaussEta”, “GaussNoPoints”, “shapeFunc”, ”nodalCoords”, “elementInfos”, “el”.
The output of the function is “k_gauss”.
It is used by “Element Matrix”.
This function calculates k(x) in each Gauss node taking into account the shape functions.

There is a cycle “for” to pass through each node and, inside, an “if” with three “else” to select the proper shape function.

3.5 Jacobian Matrix

The inputs of the function are “GaussNoPoints”, “gaussCsi”, “gaussEta”, “nodalCoords”, “elementInfos”, “shapeGrad”, “el”.
The outputs of the function are “InvJacobianMatrix”, “DetJacobianMatrix”.
It is used by “Element Matrix” and “VelocityElement”.
It calculates the Jacobian Matrix, the inverse and the determinant in the Gauss points.
There is a cycle “if-else” to divide the hypothesis of material’s property constant from the linear one. Inside this, there is a loop “for” that passes through each node of the element with an “if” and three “else” to calculate the derivative of the shape functions in any possible order of Gauss points.

3.6 Derived Shape Functions

Inputs of the function are “elementInfos”, “shapeGrad”, “GaussNoPoints”, “gaussCsi” and “gaussEta”.
The output is “deltaN”.
It is used by “elementMatrix” and “VelocityElement”.
This function calculates the derivatives of the shape functions in the Gauss points and writes them into a matrix.
As in the Jacobian Matrix, there is a cycle “if-else” to divide the hypothesis of material’s property constant from the linear one. Inside this, there is a loop “for” that passes through each node of the element with an “if” and three “else” to calculate the derivative of the shape functions in any possible order of Gauss points.

3.7 Element Matrix

The inputs of the function are “elementInfos”, “nodalCoords”, “shapeFunc”, “shapeGrad”, “GaussPointWeight”, “GaussPointLocation”, “GaussNoPoints”, “el” and “matprob”.
The output of the function is “ElementStiffnessMatrix”.
It uses “jacobianMatrix”, “derivatedShapeFunctions”, “kGauss”.
It calculates the element stiffness matrix on the Gaussian points and adds contributions of the gradient of shape functions and the material contribution.
There is a cycle “if-else” to divide the hypothesis of material’s property constant from the linear one, and, inside there is another cycle “for” to calculate the matrix along each Gaussian points.

3.8 Connectivity Matrix

The inputs of the function are “el”, “elementInfos”, “nodalCoords”, “matprob”.
The output of the function is “Conn”.
It creates the Connectivity matrix, a square matrix of zeros and ones used to represent a finite graph.
There is a cycle “if-else” to divide the hypothesis of material’s property constant from the linear one and, inside, another cycle “for” to assign the ones when two nodes are connected.

3.9 Determinant Jacobian 1D

The inputs are “sizeN”, “dN”, “faceNodesCoords”, “gaussCsi”.
The output is “detJac”.
It is used by “NeumannIntegration” function.
This function implements the determinant of the Jacobian for 1D dimension Mesh.
There is cycle “if-else” to distinguish from a linear to a quadratic mesh analysis.

3.10 Shape 1D

The input is “inputNeumann”.
The output is “shapeFunc1D”.
This function created the shape functions and the gradients in 1D dimension only if the input Neumann is defined with a cycle “if-else”.

3.11 Gaussian Quadrature 1D

The inputs are “elementInfos”, “el”, “matprob”.
The outputs are “GaussPointWeight1D”, “GaussPointLocation1D”, “GaussNoPoints1D”.
As the Gaussian Quadrature function, this function gives different information about the Gaussian Quadrature to the database for each case of the problem but in 1D dimension.
There is a cycle “if-else” to divide the hypothesis of material’s property constant from the linear one, and, inside this cycle, three different analyses are developed (two linears with different sizes and a quadratic one).

3.12 Neumann Integration

The inputs are “elementInfos”, “elementFaces”, “shapeFunc1D”, “GaussPointWeight1D”, “GaussPointLocation1D”, “GaussNoPoints1D”, “el”, “matprob”.
The output is “rhsVector_element”.
It uses “DeterminantJacobian1D”.
This function gives the local rhs vector for each element.
There is a cycle “if-else” to divide the hypothesis of material’s property constant from the linear one, and, inside this cycle, are uploaded the right shape functions to calculate the contribution of each face inside the same element. Furthermore, it is computed a cycle “if” with three “else” inside to add the contribution of each face to the nodes of the element.

3.13 Velocity Element

Inputs of the function are: “el”, “pressure”, “Conn”, “nodalCoords”, “elementInfos”, “shapeGrad”, “GaussPointLocation”, “GaussNoPoints” and “matprob”.

The output is “velocity_elem”.

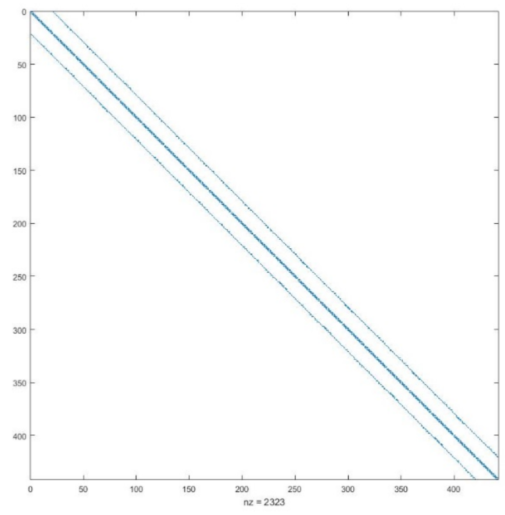
It uses the “jacobianMatrix”and “DerivatedShapeFunctions”.
We coded this function to calculate the velocity field of the domain. There are two situations for k(x): constant or linear. Obviously, in each case, the result of the velocity field is different.

3.14 Create Vtk

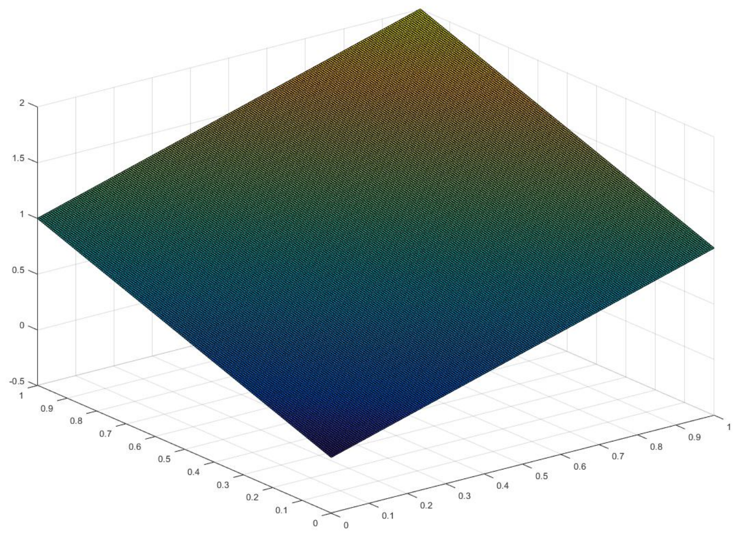
The inputs of the function are “nodes”, “elem”, “pressure”, “velocity”, “presName”, “velName”.
The output of the function is “vtk”.
It creates a “.vtk” file to describe the postprocess as a text file.

4. Plots

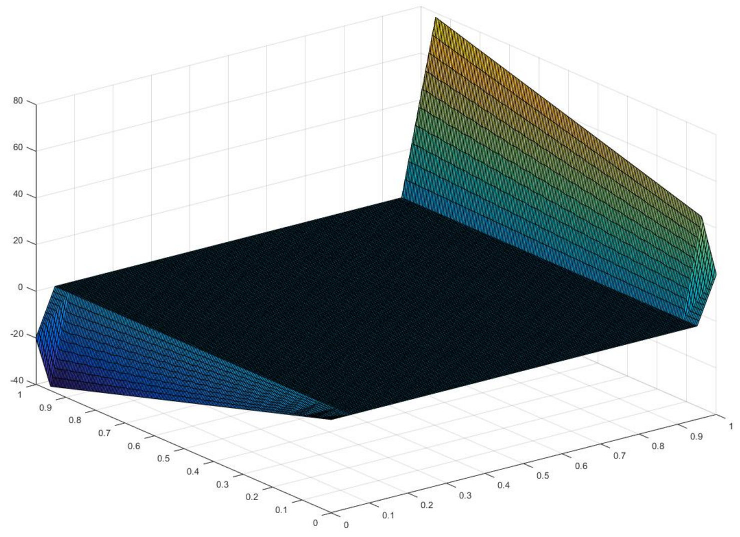
We created four plots to describe the results obtained in the case of linear triangular analysis:



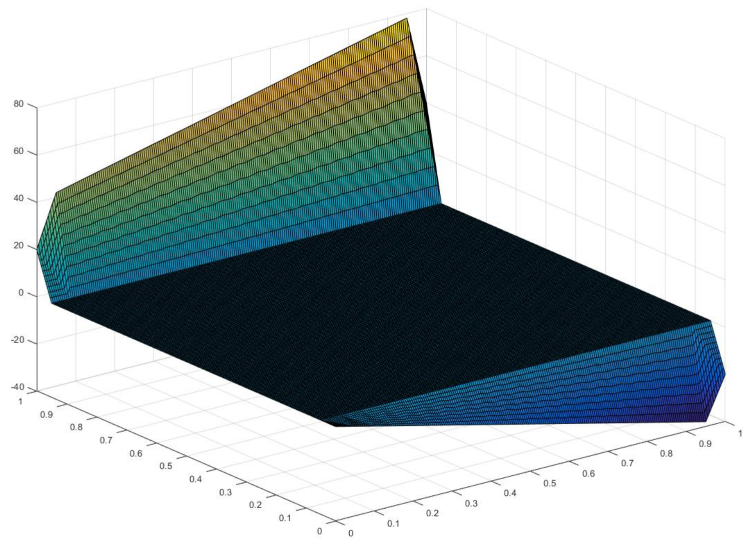
It describes the sparsity pattern of the Global stiffness matrix K, with 2323 non zero elements.



The second figure is a 3D dimensional plot of the pressure along the domain.



The third figure represents the velocity vector in the domain along the spatial direction x, considering the linear hypothesis.



The last figure describes the velocity vector in the domain along the spatial direction y considering the linear hypothesis. It is possible to notice that there are negative and positive values along y axe.

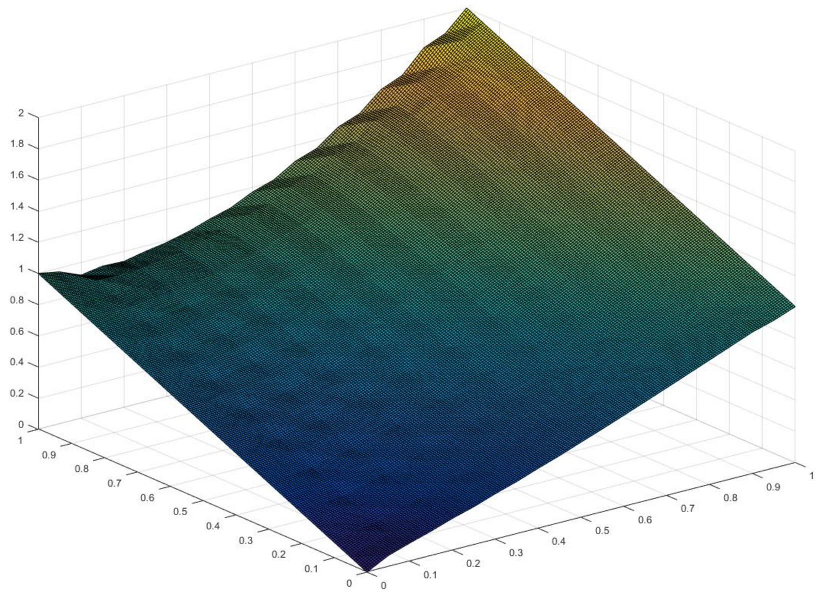
5. Improvement of the first assignment

We applied new strategies during the writing of the code, respect to the first approach, to achieve a general improvement:

- the functions “imposBC”, “compFvector” and “SolvingSystem” have been added into the main function FEM. The reason of this modification is the efficiency and ordering of the code, in fact these three functions are very short individually and it is not necessary to write them separately.
- We have added three functions to calculate Neumann boundary conditions such as “shape1D”, “gaussQuadrature” and “DeterminantJacobian1D”
- We have added the function “velocityElement” to calculate the velocity along x and y directions.
- We have also added the function “k_gauss”. This is due to k(x), the property of the material. This characteristic could be a non-constant variable, as in our problem, which means that we need to consider its changes on the domain. Therefore, we have written this function to calculate k(x) in the Gauss points.

6. Point to be improved

We noticed that with a quadratic analysis the solution does not appear correct as we found in the linear case. Tt is possible to see this evidence in the 3D plot of the pressure in the domain (quadrilateral and quadratic analysis):



2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1