Programming for Engineerings and Scientist
Universitat Politecnica de Catalunya

Assignment 2.b

Federico Parisi
Jor Fergus Dal
Nadim Saridar
Aren Khaloian

14-06-2020

Contents

1 Abstract **1**

2 Statement **2**

3 The Code **4**

3.1 Code Changes 4

3.2 Eigen Library 4

3.3 Classes 5

3.3.1 Shape Functions 5

3.3.2 Stiffness Matrix 6

3.3.3 Force Vector 6

3.4 Main 7

3.4.1 Input/Output 9

4 Problems **11**

4.1 Eigen Problem 11

4.2 Class and Structure Problems 11

5 References **12**

1. Abstract

In this report it will be described a finite elements (FE) code in C++. There will be shown the definition of the classes in C++ language, then how they will interact each other in the main code.

Due to problems with the compiler and with some additional functions, we have not been able in writing a complete working FEM code. This is the reason why in this report, will be reported only the main components. Moreover, the problems encountered during the development of the above mentioned code are reported in a specific chapter.

2. Statement

In fluid dynamics, potential flow describes the velocity field as the gradient of a scalar function: the velocity potential. As a result, a potential flow is characterised by an irrotational velocity field, which is a valid approximation for several applications. In this homework, you have to determine the potential flow around an object solving the following two-dimensional Poisson problem:

$$\begin{cases} \Delta u = 0 \; in \; \Omega, \\ \nabla u \cdot \mathbf{n} = -1 \; on \; \Gamma_{in} = 0 \; X \; (0,1), \\ \nabla u \cdot \mathbf{n} = 1 \; on \; \Gamma_{out} = 1 \; X \; (0,1), \\ \nabla u \cdot \mathbf{n} = 0 \; on \; \partial\Omega \backslash (\Gamma_{in} \bigcup \Gamma_{out}, \\ u(0,0) = 0 \end{cases} \tag{2.1}$$

where Ω is the computational domain shown in figure 1, $\partial\Omega$ its boundary and $\mathbf{n}$ is the outward unit normal vector. The velocity field is obtained in terms of this potential as:

$$v_x = \frac{\partial u}{\partial x} \; v_y = \frac{\partial u}{\partial y} \tag{2.2}$$

You have to account for the following features in the code:

- Load the computational mesh from input files
- Use triangular and quadrilateral elements, of degree one and two
- Write the results in a vtk file that can be used to display results in Paraview
- Perform a convergence analysis for the different elements, using as a reference solution the one on the finer mesh you can compute with

Several meshes are provided to test your code. In particular, for each type of element you are given 5 different meshes. Note that the i-th mesh has approximately the same number of nodes for all the element's types and you can make comparisons of the solutions, running time, accuracy, etc. Please enclose a report describing your code, how to use it, how you tested it, etc. Take into account that the structure, presentation, readability, etc. of the program will be evaluated (not only correctness) together with the report.

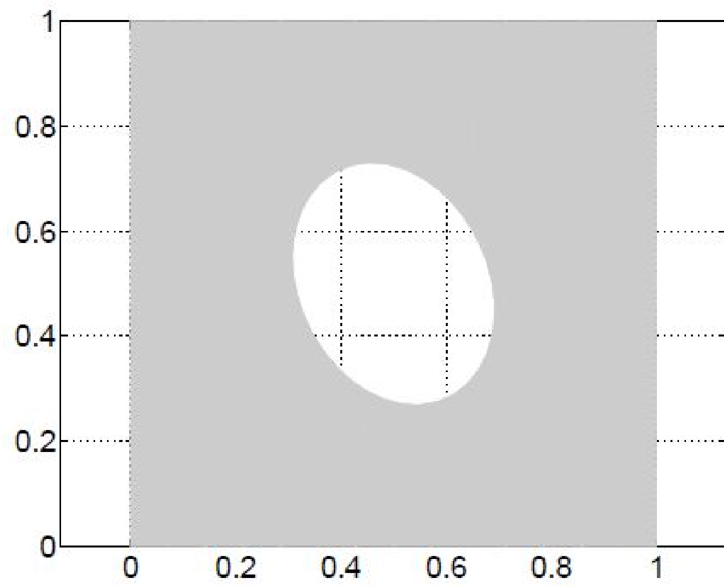

Figure 2.1: Scheme of the Geometry

3. The Code

Following the structure explained in the previous assignment, and considering the nature of the C++ language, the code has been structured in object classes.

These classes are described below and shown the implementation in C++.

Unfortunately we encountered some issues in adding the Eigen library in the Qt creator environment. We managed to add this library in another environment, called Visual Studio. The problem about this second compiler is that is not very straight forward to use, and so we didn't manage to implement the code.

However, in this report we will show the lines of C++ code which define the classes, taking into account the Eigen cases working.

3.1 Code Changes

From the previous code design, some changes have been done. Instead of having two classes, we now have five. One major reason for this change is that we decided to use the Eigen library, which already has a very smart an efficient matrix class definition. This lead us to rethink which classes to include. In particular we now have a problem class which incorporates the reoccurring variables they are inherited by other classes. Moreover, we have a shape function/reference element class which gives the relevant shape functions and jacobian matrix. And finally we now also have a separate class with the boundary conditions.

3.2 Eigen Library

In this section we will explain a little bit the Eigen library and its functions, in order to have an idea of why the code is hugely more complex to implement without this feature.

This library allows the users to operate within matrices and complex numbers. One of the main differences between MATLAB language and C++ are some predefined functions. For example Matlab can operate between matrices without any library implementation needed. On the other hand, C++ needs libraries for everything is needed to do. For examples it doesn't support operations between matrices, decompositions etc, if not with the suited library.

Unfortunately, in a Finite Elements code, the operations are based on operations between matrices. If we think at the final system to be solved:

$$Ku = f$$

in which K is the global stiffness matrix, and it is assembled as a composition of elements matrices. Doing this without the Eigen library is highly complicated and requires defining our own matrix classes and linear operations.

This is the reason why we will consider this library working even if it is not. By the way we will not have any consistent results, so here will be reported only a rough structure of a Finite Element solver.

Once this library working, it can be tested.

In Figure 3.1 is shown the error message in Qt creator regarding the missing Eigen library.

```
1   #include <string>
2   #include <vector>
3   #include <iostream>
4   #include <fstream>
5   #include <math.h>
6   #include <iomanip>
7   #include <Eigen/Dense>                    'Eigen/Dense' file not found
8   #include <Eigen/Sparse>                   'Eigen/Sparse' file not found
9
10  int main(){
11
12      int numMesh;
13      int elemType; // 0 for tri 1 for quad
14      int elemDegree; // 0 for linear 1 for quad
15
```

Figure 3.1: Eigen library not found

3.3 Classes

In this section are reported the classes as have been thought while designing the code. They are made by a public part, and a private one.

The public part will be the one related to the data we want to share with other functions, or the ones we need. The private part will be hidden during the run of the main code, they can be information or processes. These data will be created and destroyed after being used. This is one of the big advantages of an object oriented language.

In order to better design the code, it has been created a class called *problem* in which are stored the problem variables as the element type, interpolation degree, coordinates and connectivity matrices. These are values that will be used frequently and so other classes will inherit them. This class is shown here:

```
23  class problem{
24      //This structure contains all the crucial variables we will be using over and over.
25      //So later classes will inherit from this
26  public:
27      int numnodes;
28      int numelements;
29      int elemdim; //3 for tri lin, 4 for quad lin, 6 for tri quadrat and 8 for quad quadrat
30      MatrixXd X(numnodes, 2);                          unknown type
31      MatrixXd T(numelements,elemdim);
32  };
```

Figure 3.2: Problem Class

3.3.1 Shape Functions

In this class are defined the shape functions that will be used during the program execution.

```
35  class ref_element: public problem{
36  public:
37      //This class inherits from the problem structure. Later, once the X and T matrices are assembled,
38      //the assemble metrices here can be called to contruct the shape functions and their derivatives.
39      //These functions use the elemdim to figure out which shape functions to return.
40      MatrixXd SF;
41      MatrixXd dSFxi;
42      MatrixXd dSFeta;
43      void assembleSF();
44      void assembledSFxi();
45      void assembledSFeta();
46  };
```

Figure 3.3: Shape Functions Class

In the public part will be stored the information related to shape function and its derivatives. In the same part is going to be defined which kind of elements we are working with, so to create the appropriate matrix. This assemble function will be in the public part as well, in order to be able to create with an external procedure the right derivatives and functions. These assemble functions possess all the shape functions and derivatives for each element case, and by using the elemdim variable, it sets the appropriate values to the matrices of this class.

3.3.2 Stiffness Matrix

In this class is going to be computed the assembly of the stiffness matrix. In its public section it has the assemble function, which can be called to assemble the matrix K also defined in the public part. This assemble function calls the two private functions to assemble and use the jacobian and the gauss points to create the K. In addition, since it inherits form the ref_elem class, we will have the shape functions at hand too.

The Gauss quadrature function will be formed by two arrays storing the values of the Gauss points, so dimensions (k,n) in which k depends on the element and will be computed as before, similarly to the assemble function. While n is the space dimensions in which we are working on, in our case will be equal to 2. The second array will be a vector storing the Gauss weights and will have the dimension of k.

```
48  class stiffnessMatrix: public ref_element{
49  public:
50      MatrixXd K;                                    unknown ty
51      void assembleStiffnessMatrix();
52  private:
53      MatrixXd jaco;                                 unknown ty
54      void assembleJaco(); //Has the Jacobian matrices for the different cases, and uses the elemdim to return the right Jacobian. This jacobian is then used to assemble the stiffness matrix.
55      MatrixXd Gauss;                                unknown ty
56      void assembleGauss(); //Has the Gauss points for the different cases, and uses the elemdim to return the right Gauss points. These Gauss points are then used to assemble the stiffness matrix.
57  };
```

Figure 3.4: K-matrix Class

3.3.3 Force Vector

This class takes the values from the boundary structure. In the assemble function, it will be as an input the number of nodes in which the boundary values are applied.

```
58  class forceVector: public boundaries {
59  public:
60      VectorXd assembleForceVector(int numnodes);
61  };
```

Figure 3.5: Force Vector Class

The boundary class will store the values of the Dirichlet and Neumann conditions. They will be added to the force vector. This structure is shown below:

```
15  class boundaries{
16  public:
17      double Neu_0 = -1;
18      double Neu_1 = 1;
19      double Neu = 0;
20      double Dir = 0;
21  };
```

Figure 3.6: Boundary Class

3.4 Main

While executing the main, it is firstly called the requested mesh. So it is called a outer file .dat as an input. Then this file is stored in a variable called fin. The path to the files in interest will be requested from the user via the console, as well as the element shape and degrees of freedom of the interpolation. We decided to do it this way as it's an easy and straight forward way to make the code work for different users.

In order to read the files, a library called from $\#include < fstream >$ is needed. This library allows us to call an external file and copy it in a variable. Then it has to be analyzed and stored all the data inside the .dat file. These files correspond to the coordinate matrix and the connectivity matrix. By the dimensions of the matrices stored in these files we can deduct the number of nodes and elements, as well as the number of nodes per element. Still, this information is directly prompted from the user.

These data are stored in a file that can also be executed in MATLAB, so we did, to visualize the mesh and to understand how the values are stored in the .dat files. In the following Figures (3.7 and 3.8) are shown the mesh and the data storage.

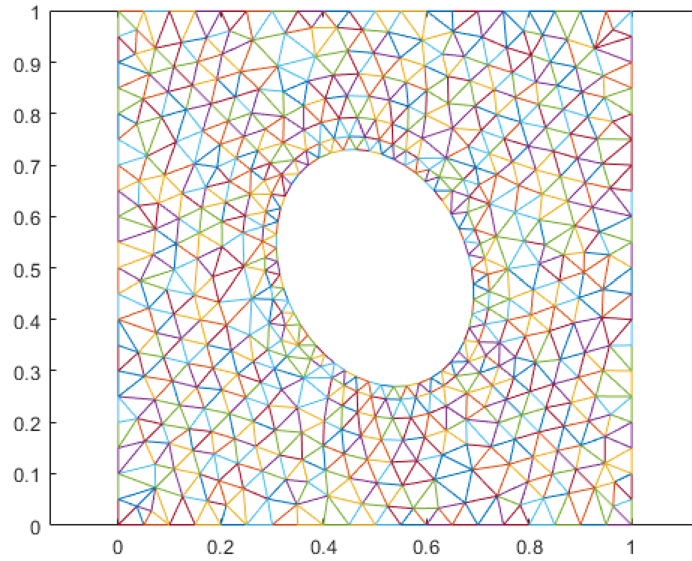

Figure 3.7: Mesh

[Nodes]

510x3 double

	1	2	3
1	1	1	1
2	2	1	0.9500
3	3	0.9500	1
4	4	0.9637	0.9637
5	5	1	0.9000
6	6	0.9000	1
7	7	0.9273	0.9273
8	8	0.8744	0.9446
9	9	0.9446	0.8737
10	10	1	0.8500
11	11	0.8500	1
12	12	0.8910	0.8910
13	13	0.9467	0.8317
14	14	0.8264	0.9535
15	15	0.9025	0.8423
16	16	0.8410	0.9008
17	17	0.8000	1
18	18	1	0.8000
19	19	0.8546	0.8546
20	20	0.8037	0.9141

[Elements]

889x3 double

	1	2	3
1	210	203	216
2	191	182	186
3	172	164	166
4	203	191	200
5	182	172	177
6	164	154	155
7	154	146	144
8	138	130	121
9	122	119	107
10	146	138	133
11	130	122	112
12	119	114	104
13	114	113	98
14	113	115	100
15	186	182	177
16	166	164	155
17	177	172	166
18	191	186	200
19	155	154	144
20	144	146	133

Figure 3.8: Data Storage

As can be seen, the data are stored in a matrix form. This means that when calling the above mentioned file

in the main code, it is needed to perform a for loop in order to go through all the lines of the document and store the data in a matrix. The code for this function is reported below.

3.4.1 Input/Output

```
103 //Opening node file and assembling coordinate matrix, X
104
105     //Open .dat file containing coordinates of nodes
106     ifstream fin;
107     fin.open(location_nodes);
108
109     //Retrieving the number of nodes
110     fin.seekg(0, fin.end);
111     prob_var.numnodes = fin.tellg();
112     prob_var.numnodes = prob_var.numnodes/3;
113
114
115     //Use load of Node file to initialize matrix X. Node file has three columns, first stating node index, second stating x-coordinate, and third stating y-coordinate.
116
117     for (int i=0; i<prob_var.numnodes; i++){
118             fin >> trash >> prob_var.X(i,0) >> prob_var.X(i,1)
119     }
120
121     fin.close();
122
```

Figure 3.9: Nodes Assembling

```
124 //Opening elements file and assembling connectivity matrix, T
125
126     //Open .dat file containing elements
127     fin.open(location_elements);
128
129     //Retrieving the number of elements
130     fin.seekg(0, fin.end);
131     prob_var.numelements = fin.tellg();
132     prob_var.numelements = prob_var.numelements/prob_var.elemdim;
133
134
135     //Use load of Element file to initialize matrix T. Every line of T corresponds to one element, and the columns the nodes belong to the element.
136
137
138     if (prob_var.elemdim == 3)
139     {
140         for (int i=0; i<prob_var.numelements; i++){
141                 fin >> prob_var.T(i,0) >> prob_var.T(i,1) >> prob_var.T(i,2);
142         }
143     }
144     else if (prob_var.elemdim == 4)
145     {
146         for (int i=0; i<prob_var.numelements; i++){
147                 fin >> prob_var.T(i,0) >> prob_var.T(i,1) >> prob_var.T(i,2) >> prob_var.T(i,3);
148         }
149     }
150     else if (prob_var.elemdim == 6)
151     {
152         for (int i=0; i<prob_var.numelements; i++){
153                 fin >> prob_var.T(i,0) >> prob_var.T(i,1) >> prob_var.T(i,2) >> prob_var.T(i,3) >> prob_var.T(i,4) >> prob_var.T(i,5);
154         }
155     }
156     else if (prob_var.elemdim == 8)
157     {
158      for (int i=0; i<prob_var.numelements; i++){
159             fin >> prob_var.T(i,0) >> prob_var.T(i,1) >> prob_var.T(i,2) >> prob_var.T(i,3) >> prob_var.T(i,4) >> prob_var.T(i,5) >> prob_var.T(i,6) >> prob_var.T(i,7);
160      }
161     }
162
163     fin.close();
164
```

Figure 3.10: Element Assembling

As can be seen from the previous lines of code, the main is asking the user to copy-paste the path of the required .dat file and to define the interpolation degrees and type of elements, in order to make the code working properly. There can be easily seen the loop for storing the data into the fin variable. The data is

stored directly in the prob_var object so as to be easily accessible. The trash variable just stores the number of the node and is not used to assemble the coordinate matrix.

After this, all the functions which need to use nodes position or elements type, can be used.

Next, the code creates instances of the stiffness matrix, force vectors and nodal solution. These instances of the stiffness and force vectors are then assembled using the functions of the respective classes. Then the code will call the Gauss quadrature functions and the boundary conditions classes in order to finally compute the global K matrix and solve the problem.

In order to do so, a reduction of the system is needed after applying the Dirichlet BC. Moreover it is needed to compute the inverse of the K-matrix in order to solve in the same way as the backslash operator in MATLAB. These functions are all included in Eigen library as operations between matrices.

4. Problems

In this section will be reported the problems encountered during the development of the assignment.

Because of the problem mentioned, the code is not complete as writing all the missing parts without being sure on the previous ones is useless.

The missing parts are only the procedures regarding the main solver, explained in Section 3.4.1.

4.1 Eigen Problem

As said at the beginning, unfortunately the Eigen library has not been loaded in the Qt compiler. This lead to an impossibility in testing the code. It has been tried to follow various tutorials and the instructions both of the Qt webpage, as well as the Eigen one, without success.

It has been tried to implement the code from the environment Qt creator, to the Visual Studio one, but because of the small time, and an undervaluation of the problems, we didn't manage to use this compiler. On the other hand, we have noticed that online is available a big amount of material for using this other IDE.

4.2 Class and Structure Problems

Not being able to plug the Eigen library, made impossible to test our classes as they are based on matrices creation or destruction. As output, all of them have matrix and all of them perform operation between matrices.

We can't be sure that our classes or structures have been well defined as the code is not running. By the way we can say that in compiling the code, the errors seem to come from the missing library only, so we believe that the implementation is right.

5. References

- https://www.qt.io/
- https://podgorskiy.com/spblog/304/
- https://www.cimne.com/2437/master-on-numerical-methods-in-engineering-(upc)-a/spring-semestre-2/programming-for-engineers-and-scientists
- http://eigen.tuxfamily.org/index.php?title=Main-PageCompiler-support
- https://www.youtube.com/watch?v=37QvVrJnwQwt=48s