Programming for Engineers and Scientists

Master in Numerical Methods in Engineering

Assignment 2b: “FE Program written in C++”

Professor: Amir Abdulahi

Students: Marcello Rubino – Enrico Marin – Lei Pan

Date: 10 June 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

1987 - 2017 CIMNE 30 years International Centre for Numerical Methods in Engineering

1. Introduction

In this report a C++ program which solves a FE method will be illustrated. In particular, the problem we going to solve is the equation of potential flow like following:

$$\begin{cases} \Delta \mathrm{u} = 0 \quad \text{in} \quad \Omega, \\ \nabla \mathrm{u} \cdot \mathbf{n} = -1 \;\; \text{on} \;\; \Gamma_{in} = 0 \times (0,1), \\ \nabla \mathrm{u} \cdot \mathbf{n} = 1 \quad \text{on} \;\; \Gamma_{out} = 1 \times (0,1), \\ \nabla \mathrm{u} \cdot \mathbf{n} = 0 \quad \text{on} \;\; \partial\Omega \;\backslash(\Gamma_{in} \cup \Gamma_{out}), \\ \mathrm{u}(0,0) = 0 \end{cases}$$

Where the 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:

$$v_x = \frac{\partial u}{\partial x} \qquad\qquad v_y = \frac{\partial u}{\partial y}$$

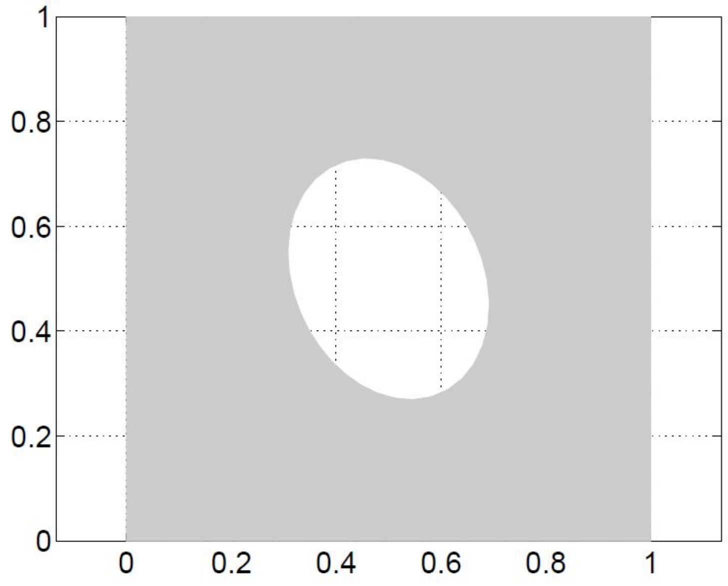


Figure 1. The computational domain

2. Structure of the code (presentation of headers and classes)

The code implemented in C++ is structured in a way that is more usable for many different problems. The classes are using all the main methods needed for the solution of a Finite Element code. Below a description of each of them will be illustrated:

<u>2.1 Class *Matr*</u>

This class represents all the possible main operations that can be done with the matrices such like matrix-vector products, matrix-matrix product or inverse of a matrix (for the Jacobian). A more user-friendly notation has been implemented (Matlab kind) to make more easy the implementation of the rest of the code. The interface of the header file is illustrated below.

```
#include <stdio.h>
#include "vec.h"

class Matr
{
protected: // Protected, access only for derived classes
    double** internalData; // // Entries of the Matrix
    int internalRows, internalCols; // Dimensions of the Matrix
public:

    // Standard
    Matr(); // Default Constructor
    Matr(const Matr& otherMatr); // Copy Constructor
    Matr(int numRows, int numCols); // Constructor of a matrix of size "internalRows x internalCols"
    Matr(int size); // Constructor of a squared matrix of size "size x size"
    Matr(char* filename); // Read file and write Constructor/Operator
    ~Matr(); // Destructor

    // Main
    int GetNumberOfRows() const; // Declear the function which gets the number of Rows from the matrix
    int GetNumberOfColumns() const; // Declear the function which gets the number of Columns from the matrix
    double& operator()(int i, int j); // MatLab default indexing (i,j)

    // Operations
    //overloaded assignment operator
    Matr& operator=(const Matr& otherMatr); // Equality of matrices (Assignment)
    Matr operator+() const; // + unary (m2 = +m1)
    Matr operator-() const; // - unary (m2 = -m1)
    Matr operator+(const Matr& m1) const; // Sum of Matrices
    Matr operator-(const Matr& m1) const; // Difference of Matrices
    Matr operator*(double a) const; // Product with a scalar
    double CalculateDeterminant() const; // Calculate the determinant
    double Trace() const; // Calculate the trace of the matrix


    // Friendship - usable by others
    friend Vec operator*(const Matr& m,const Vec& v); // Calculation of m*v
    friend Vec operator*(const Vec& v,const Matr& m); // Calculation of v*m
    friend Matr operator*(const Matr& m,const Matr& v); // Calculation of m*m
    friend Matr Inverse(Matr &m); // Invet the matrix
};

// Interface of friends operators (for the others)
Vec operator*(const Matr& m, const Vec& v);
Vec operator*(const Vec& v, const Matr& m);
Matr operator*(const Matr& m,const Matr& v);
```

2.2 Class *Vec*

As seen above class *Matr* uses another basic class called *Vec*. This class does all the possible and most important methods (operations) with the vector. It's usable everywhere in the code like *Matr*. In this case too, the Matlab indexing has been implemented. Below the header is illustrated.

```
#include <stdio.h>

class Vec
{
protected: // Protected, access only for derived classes
    double* internalData; // Components of the vector
    int internalSize; // Size of the vector
public:

    // Standard
    Vec(); // Default Constructor
    Vec(const Vec& otherVec); // Copy Constructor
    Vec(int size); // Constructor of a vector of size "size"
    ~Vec(); // Destructor
    // Main
    int GetSize() const; // Declear the function which gets the size from the vector
    double& operator[](int i); // C++ default indexing
    double Read(int i) const; // Declear the function which reads the vector (using C++ default indexing)
    double& operator()(int i); // MatLab default indexing

    // Operations
    Vec& operator=(const Vec& otherVec); // Equality of vectors (Assignment)
    Vec operator+() const; // + unary (v2 = +v1)
    Vec operator-() const; // - unary (v2 = -v1)
    Vec operator+(const Vec& v1) const; // Sum of Vectors
    Vec operator-(const Vec& v1) const; // Difference of Vectors
    Vec operator*(double a) const; // Product with a scalar
    double ScalarPr(const Vec& v1,const Vec& v2) const; // Scalar product
    double CalculateNorm(int p=2) const; // Norm calculation

    // Friendship - usable by others
    friend int length(const Vec& v); // Length of vector (friend)
};

int length(const Vec& v); // Interface of length (friend)

#endif // VEC_H
```

2.3 Class *LinearSystem*

This class is a basic classes which uses the LU method in order to compute a linear system and compute the solution.

```
#include "vec.h"
#include "matr.h"

class LinearSystem
{
private:
    int mSize; // size of linear system
    Matr* mpA;  // matrix for linear system
    Vec* mpb;  // vector for linear system

    // Only allow constructor that specifies matrix and
    // vector to be used.  Copy constructor is private.
    LinearSystem(const LinearSystem& otherLinearSystem){};
public:
    LinearSystem(const Matr &A, const Vec &b);

    // destructor frees memory allocated
    ~LinearSystem();

    // Method for solving system
    virtual Vec Solve();
};

#endif // SOLVELINEARSYSTEM_H
```

2.4 Class *Solve*

This class implements the solve of the linear system of the FE problem. Before using the method from LinearSystem class, adds to the variables the Lagrangian multipliers coming from the Dirichlet boundary conditions given, in order to make the global matrix K invertible. Below the interface and the methods are illustrated.

```
#include "assembleprocess.h"
#include "matr.h"
#include "vec.h"
#include "solvelinearsystem.h"


class Solve : private AssembleProcess
{
public:
Vec SolveSystem(Matr &X,Matr &T,Matr &In,Matr &Out,Matr &Diric,Vec &InfoTop);
private:
Vec sol;
};

#endif // SOLVE_H
```

2.5 Class *AssembleProcess*

This class implements the assemble process for both the global stiffness matrix K (using the elemental stiffness matrix K_e computed in class *Element* and the right hand side vector F. In this case (and this makes the code usable only for this particular 2D problem, the right hand side vector is completely implemented in this class. In fact, it's computed by the assembling of the Neumann integration method done only on Inflow and Outflow elemental. It uses the shape functions and the first derivatives (1-dimensional) already computed in the correct Gauss points from class *shapeFunctions*. Below the interface has been reported:

```
#include "matr.h"
#include "element.h"
#include "vec.h"

class AssembleProcess: public element
{
public:
Matr globalK(Vec &InfoTop, Matr &X, Matr &T);
Vec globalF(Vec &InfoTop, Matr &X, Matr &T, Matr &In, Matr &Out);
protected:
Matr K;
Vec f;
};

#endif // ASSEMBLEPROCESS_H
```

2.6 Class *Element*

Here the elemental stiffness matrix is being computed. It collects information about the kind of the problem (2D or 3D both are implemented) and the type of elements (collects the correct shape functions and their first derivatives computed in the Gauss points from class *shapeFunctions*). Below the header file of the class is illustrated:

```
#include "gausspoints.h"
#include "meshtopology.h"
#include "shapefunctions.h"
#include "matr.h"
#include "vec.h"

class element : public shapeFunctions
{
public:
   Matr ElementMatrix( Vec &InfoTop, Vec& Te, Matr &Xe);
protected:
   Matr Ke;
};

#endif // ELEMENT_H
```

2.7 Class *shapeFunctions*

As already explained this class has methods that calculate the shape Functions and the partial first derivatives at the correct Gauss points taken from methods in class *gaussPoints*. Each shape Function and its own partial derivatives have been reported manually for both linear and quadratic elements, as well as for 2D or 3D case for a possible future reusability. It does the same calculation for 1 dimension less than the problem one. The interface is the following:

```
#include "vec.h"
#include "matr.h"
#include "gausspoints.h"
#include "meshtopology.h"

class shapeFunctions: public gaussPoints
{
protected:
    Matr N;
    Matr dN;
    Matr Nlow;
    Matr dNlow;
public:
    Matr CalcShapeFunctions( Vec &InfoTop); // Calculates the shape functions at the given Gauss Points
    Matr CalcDerShapeFunctions( Vec &InfoTop); // Calculates the first derivatives shape functions at the given Gauss Points
    Matr CalcLowerDimShapeFunctions( Vec &InfoTop); // Calculates the shape functions for Neumann at the given Gauss Points
    Matr CalcLowerDimDerShapeFunctions( Vec &InfoTop); // Calculates the first derivatives shape functions for Neumann at the given
};

#endif // SHAPEFUNCTIONS_H
```

2.8 Class *gaussPoints*

This class contains the methods linked with the information concerning the Gauss points (such as number of points needed, the weights, the position in the natural coordinates). Like the previous class, even this class contains any possible case (2D/3D – linear/quadratic/serendipity) and it gets the right variables depending on the problem. It computes the same for 1 dimension less than the problem one. Here below the interface:

```
class gaussPoints
{
protected:
   int ngauss;
   int order;
   Matr PosGauss;
   Vec weight_v;
   Matr PosGaussLow;
   Vec weight_v_low;
public:
Vec GetWeight(Vec &InfoTop);
int GetOrder(Vec& InfoTop);
Matr GetPointLocation(Vec& InfoTop);
Matr GetLowerDimPointLocation(Vec &InfoTop);
Vec GetLowerDimWeight(Vec &InfoTop);
};

#endif // GAUSSPOINTS_H
```

2.9 Class *meshTopology*

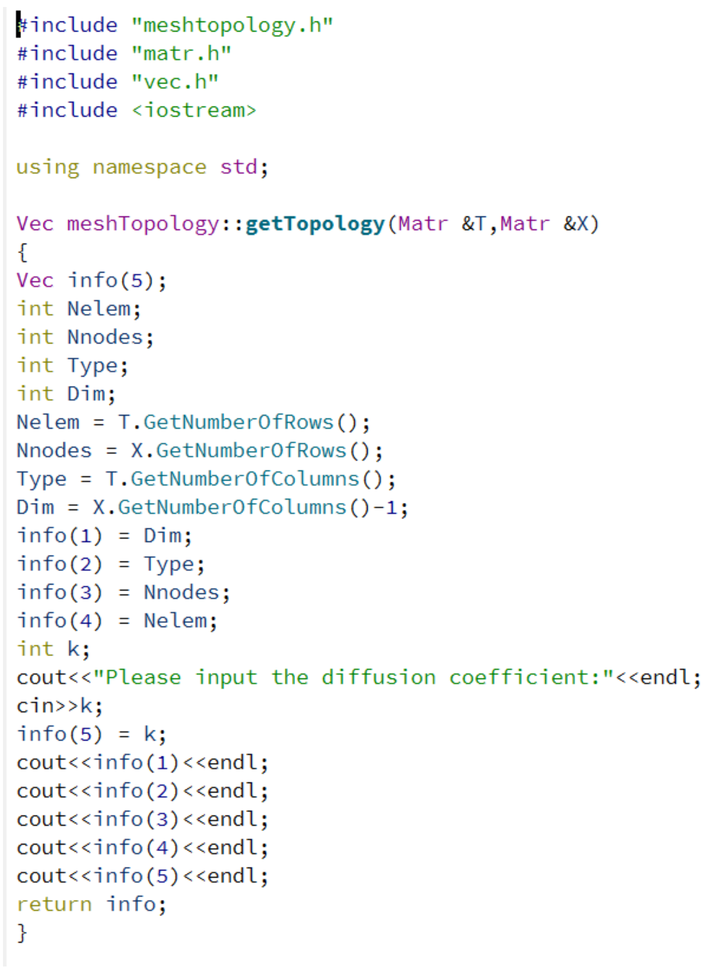
This class does the pre-processing of the problem. It collects the information coming from the mesh and gives the user the possibility to change a value for the diffusivity coefficient k. Everything, as shown below, is stored into a vector (**InfoTop**) which is the main key among all the classes in the code. The first reported is the header file:

```
#include "vec.h"
#include "matr.h"

class meshTopology
{
public:
static Vec getTopology(Matr &T,Matr &X);
};

#endif // MESHTOPOLOGY_H
```

Then the more interesting linked .cpp file:



```
#include "meshtopology.h"
#include "matr.h"
#include "vec.h"
#include <iostream>

using namespace std;

Vec meshTopology::getTopology(Matr &T,Matr &X)
{
Vec info(5);
int Nelem;
int Nnodes;
int Type;
int Dim;
Nelem = T.GetNumberOfRows();
Nnodes = X.GetNumberOfRows();
Type = T.GetNumberOfColumns();
Dim = X.GetNumberOfColumns()-1;
info(1) = Dim;
info(2) = Type;
info(3) = Nnodes;
info(4) = Nelem;
int k;
cout<<"Please input the diffusion coefficient:"<<endl;
cin>>k;
info(5) = k;
cout<<info(1)<<endl;
cout<<info(2)<<endl;
cout<<info(3)<<endl;
cout<<info(4)<<endl;
cout<<info(5)<<endl;
return info;
}
```

3. Structure of the code (presentation of the resolving procedure)

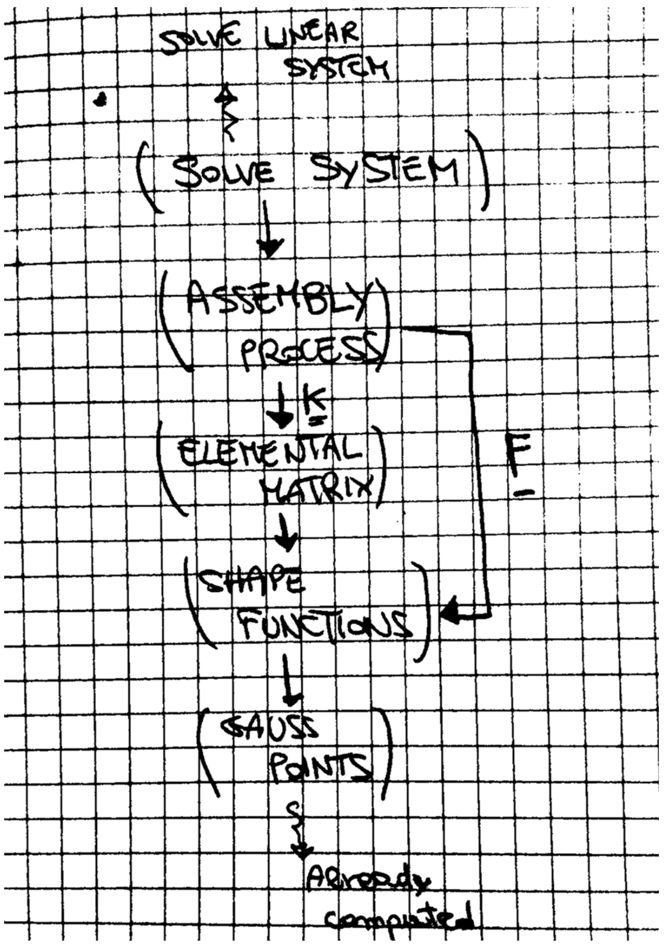
In this part of the report a short illustration of how the program works is reported. As known, the program is divided into Pre-Process, Process and Post-Process.

3.1 Pre-Process

In this step, we have following tasks to do. Firstly, we need to read the continuity matrix **T** and coordinate matrix **X** from the files T.dat and X.dat. After getting the matrix **T** and **X**, the next task is dealing with collecting the information on the geometric properties of the mesh using *getTopology* from *meshTopology* class. Then the code deals with the Neumann boundary conditions. In order to achieve this, we put the Neumann boundary conditions' information, including the continuity matrix and the coordinate matrix of Neumann boundary conditions, into the matrices **In** and **Out**, since we are supposed to divide the Neumann boundary into inflow and outflow containing the Neumann boundary conditions' values on the corresponding notes. The last task is the imposing the Dirichlet boundary conditions using the matrix *Dirich*, which will be solved by using Lagrange multiplier method in the processing part.

3.2 Process

The main part of the code uses the *SolveSystem* method from *Solve* class. The input are all the data get so far (**X, T, InfoTop, In, Out, Dirich**). The following flowchart will make more understandable how the central part of the code works (information about each class has been reported above). The output is the vector **sol**, which is shorten to the actual solution **u**.



In this flowchart it's possible to see how the problem is solved. In the chart, the information are given from the class/method below calling to the method/class above. The arrows mean "call" action.The class uses the internal information from the class below (using the incapsulation mnethod in C++).

3.3 Post-Process

In this part the code does the writing of a vtk file once known the connectivity matrix **T**, the coordinate matrix **X** and the solution vector of **u**.

4. Conclusions and Problems faced in the implementation

During the testing of the code the program aborts with no expectation during the process of solving the linear system. This made it not possible to figure out the solution, evaluate the velocity field, plot the results with ParaView and make a convergence analysis on different mashed given to us. However, this code represents a good start for a deeper development, using also 3D problems of the Poisson equation case.