Programming for Engineers and Scientists

Yuyang Wang, Carles Duno Nosas
Master on Numerical Methods in Engineering
Universitat Politcnica de Catalunya,BarcelonaTech
C++ for Finite Element Method

May 3, 2016

We are going to solve the following Poisson equation by implementing FEM and developing one single c++ code which is able to solve the problem, accounting for the following options and features:.

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

where the source term $s = 0$ and the diffusivity $\nu = 1$ are given.

- Read a general mesh and boundary condition from input files.
- Solve 2D/3D and steady state problems.
- Use triangular/tetrahedral, quad/hexahedral using linear or quadratic elements.

C++ Program

In order to implement the C++ code for Poisson equation, the Integrated Development Environment, Xcode 7.3 in Mac is used to finish the task, not Qt creator.This is just a personal preference for C++ programming.We will develop a class of vectors of *Vector*, a class of matrices called *Matrix* and a linear system class called *LinearSystem*.The vector and matrix classes will include constructors and destructors that handle memory management.These classes will overload the assignment, addition, subtraction and multiplication operators, allowing us to write code such as *"u=A*u"*, where u and v are vectors, and A is a matrix: these overloaded operators will include checks that the vectors and matrices are of the correct size. The square bracket operators will be overloaded for the vector class to provide a check that the index of the array lies within the correct range, and the round bracket operator will be overloaded to allow the entries of the vector or matrix to be accessed using MATLAB style notation, indexing from 1 rather than from zero.

1. *Class* defined for the problem.

   (a) Vector

   - Default Constructor
     The default constructor takes no arguments, and therefore this constructor has no way of knowing how many entries the vector requires. As such, it cannot allocate an appropriate size to the vector, so we ensure that a default constructor is never used by not supplying a default constructor.
   - Copy Constructor
     We want to create a new vector v according to vector u, so we can use the copy constructor. However, the automatically generated copy constructor would not allocate memory for the new copy of the data, and so it would be impossible for the entries of the vector to be copied correctly. What would actually happen is that the pointer *mData* in the original vector u would be assigned to the pointer *mData* in the new vector v.So this means that v is just a different name for u. So here we allocate a new memory for new vector v, making it has the same size and entries with the original vector u.
   - A Specialized Constructor
     We have supplied no definition for the default constructor to ensure that it is never used, and have overridden the copy constructor so that if we already have a vector we may create a copy of that vector. We also include a constructor that requires a positive integer input that represents the size of the vector. This constructor sets the member *mSize* to this value, allocates memory for the vector, and initializes all entries to zero.
   - Destructor
     In order to free the memory allocated to the instance of the class *Vector* when it goes out of scope, we construct a destructor and override the automatically generated destructor.

- Accessing the size of a Vector
  The size, or length, of a vector is accessed through the pubic method *GetSize*. This member function takes no arguments, and returns the private member *mSize*.
- Overloading the Square Bracket Operator
  We overload the square bracket operator so that, if v is a vector, the *v[i]* returns the entry of v with index i using zero-based indexing. This method first checks that the index falls within the correct range–that is, a nonnegative integer that is less than *mSize*– and then returns a reference to the value stored in this entry of the vector.
- Overloading the Round Bracket operator
  The round bracket operator is overloaded to allow us to access entries of a vector using one-based indexing. We have chosen the round bracket operator for this purpose as this allows similar notation to that employed by MATLAB.
- The Assignment Operator
  The overloaded assignment operator first checks that the vector on the left-hand side of the assignment statement is of the same size as the vector on the right-hand side. If this condition is met, the entries of the vector on the right-hand side are copied into the vector on the left-hand side.
- Unary Operators
  The overloaded unary addition and subtraction operators first declare a vector of the same size as the vector that the unary operator is applied to. The entries of the new vector are then set to the appropriate value before this vector is returned.
- Binary Operators
  The overloaded binary operators first check that the two vectors that are operated on are of the same size. If they are, a new vector of the same size is created. The entries of this new vector are assigned, and this new vector is then returned.

(b) Matrix Class *Matrix* is developed based on class *Vector*.So similarly, it includes features listed as below:
Private members *nNumRows* and *mNumCols* that are integers and store the number of rows and columns, and *mData* that is a pointer to a pointer to a double precision floating point variable which stores the address of the pointer to the first entry of the first row.Other features,constructor, destructor, assignment, unary and binary operators and so on are almost the same to class *Vector*. Here just some special members need to strength:

- Constructor with Given File
  This constructor allows us to save the node coordinates, elements connectivity, Dirichlet boundary conditions into matrix independently.

- Matrix and Vector Multiplication
  Considering that we will encounter some multiplication such a matrix multiplied by a vector or a vector multiplied by a matrix, or a matrix multiplied by a matrix, we provide an alternative method by overloading the operator "*". This also allow us to work the matrix algebra as we are in MATLAB.

(c) LinearSystem
The class *LinearSystem* is developed based previous class *Matrix* and *Vector* . Assuming the system is nonsingular, a linear system is defined by the size of the linear system, a square matrix, and vector(representing the right-hand side), with the matrix and vector being of compatible sizes. The data associated with this class is specified through an integer variable *mSize*, a pointer to a matrix *mpA*, and a pointer to the vector on the right-hand side of the linear system *mpb*.The member *mSize* is then be determined from these two members. A public method *Solve* should be written to solve this linear system by Gaussian elimination with pivoting. Finally, this method return a vector that contains the solution of the linear system.

2. Preparation for FEM

(a) In order to solve the problem,we should input the data file and save them in "T, X, In, Out" which represent connectivity matrix, node coordinates, Inlet boundary condition, Outlet boundary condition. After we have this basic data, we should define a function called *data* which can return the #of gauss point, #of coordinates, #of nodes for each element, #of total nodes, # of elements, diffusion coefficient.

(b) We also define the function called *weight*, *shape_xi*, *shape_N*, *shape_dN*. These functions read the input nodes matrix and connectivity matrix and returns weighting for gauss integration, gauss point for integration, shape function ,derivative of shape function.

3. Computation for global stiffness matrix and source vector
The global stiffness matrix and source vector is obtain by assemble the local element stiffness matrix and source vector. This process is completed in function *Stiff*, assembling local matrix, and its subroutine function *MatEl*.

4. Solving linear system through lagrange multipliers
We have Dirichlet boundary conditions which has to be applied for the inlet and outlet boundary. In order to solve it directly using our gauss elimination method, we implement the lagrange multiplier method.So until now have already solve the Poisson equation.

5. Postprocess

We are going to represent the solution in an open source software called "Paraview". The result should be written a vtk format file which can be read by Paraview. So here we create a function called *post* with argument node coordinates, connectivity matrix and solution vector. Finally the output file can be read by Paraview.

So we can see that all these work has been done is very similar to what we have did in MATLAB module for solving Poisson equation.

Solution for Poisson Equation

We solved 8 problems using the code we have developed in order to make sure our code works well, and they are:

- 2 dimensional linear quadrilateral
- 2 dimensional quadratic quadrilateral
- 2 dimensional linear triangles
- 2 dimensional quadratic triangles
- 3 dimensional linear hexahedral
- 3 dimensional quadratic hexahedral
- 3 dimensional linear tetrahedral
- 3 dimensional quadratic tetrahedral

Results and Discussions

We can represent the vtk file in Paraview and the following figure2 and figure3 show the solution result for Poisson equation from different element type and dimension.

Apparently, we can find that the maximum solution of is 1 and minimum solution is 0 for all 8 different case.This satisfy the boundary condition we have applied. The boundary condition is that

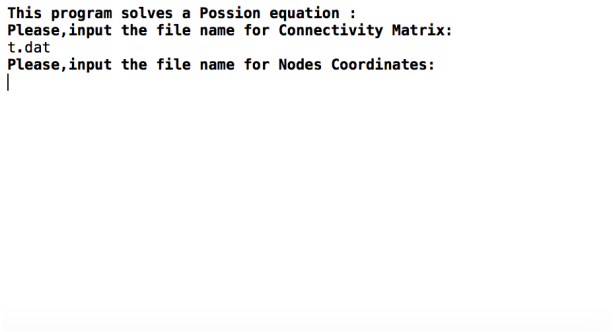
$$u = 1 \quad on \quad \Gamma_{Inlet}$$

$$u = 0 \quad on \quad \Gamma_{Outlet}$$

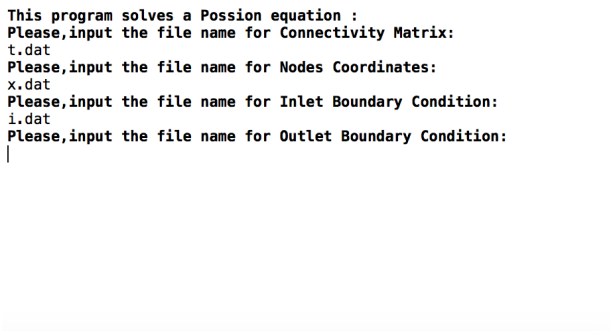
$$(\nu \nabla u) \cdot n = 0 \quad otherwise$$

So our solution verifies the boundary condition.

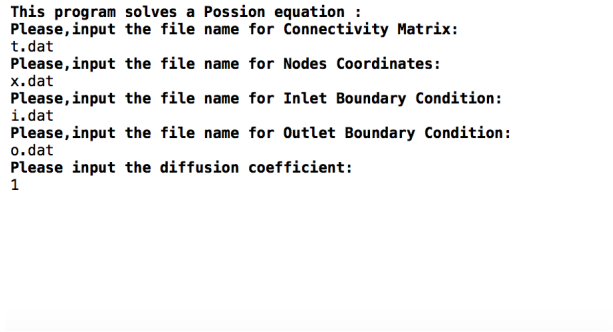
On the other hand, solution obtained from different element type represent almost the same result, so this can also be used as a verification for our code.



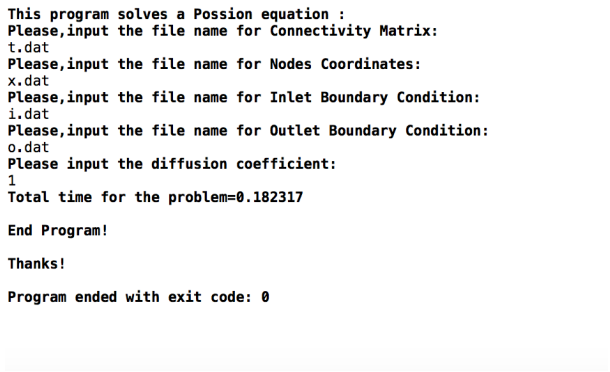
(a) Step 1



(b) Step 2



(c) Step 3



(d) Step 4

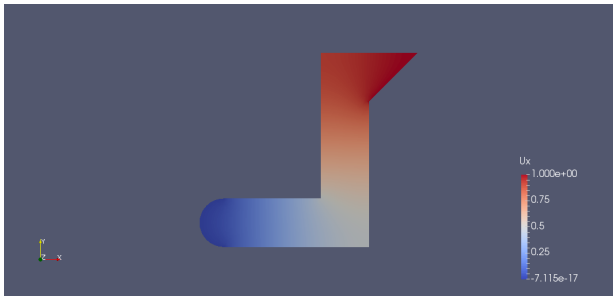
Figure 1: Explanation for the application of the code to 2D quadrilateral linear element

In order to compare the computational cost for different mesh, we run 10 times each mesh and compute the average execution time with the same computer.To be more precise, the time have been displayed in seconds, table 1.

Generally, we can see that 3D models takes more time in be executed than 2D models. That is what we would expect because it uses more data information.Besides,we can also notice that quadratic models takes around double or more time than linear ones. Even worse,since matrix size is too large ,we do not get the accurate time for hexahedral quadratic element,which is also as expected because a quadratic model will increase the number of nodes which may get a better solution but demand more computational cost.

On the other hand, we compare the computational cost with that from MATLAB. We can see that for small size matrix, Our C++ code cost less time than that of MATLAB even using the same method. This may duo to our C++ code is not fast and need to optimize in a more feasible way.

(a) Quadrilateral linear element



(b) Quadrilateral quadratic element

(c) Triangle linear element

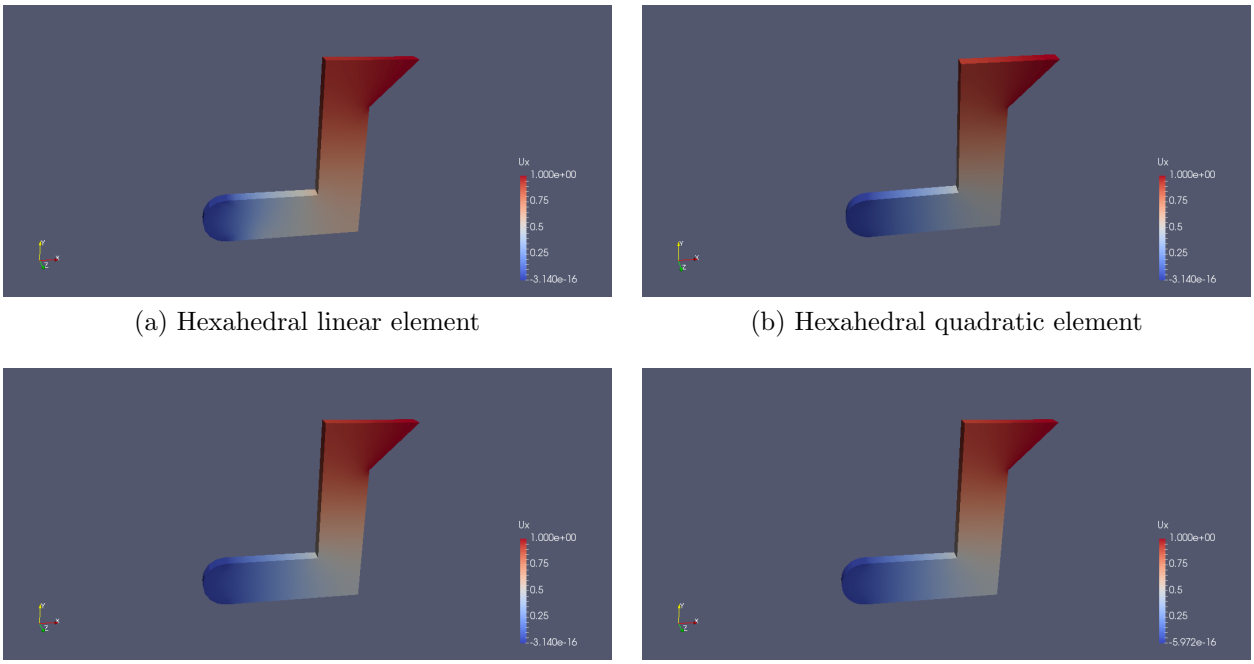
(d) Triangle quadratic element

Figure 2: Solution for 2D Poisson equation from different element type

Conclusions

We extended the given MATLAB code to a more flexible C++ code which can be used to solve not only 2D but 3D Poisson equation. The solution result verify very well to the given condition. And besides, we compared the solution result obtained from different element type, we find the color bars were almost the same which is another way to verify the correctness of our code.By comparing the computational cost for different element type, our code need to optimize in detail so that it can solve a linear system much faster.

The code is available at github: https://github.com/Yuyang01/YuyangJorge007



(a) Hexahedral linear element

(b) Hexahedral quadratic element

(c) Tetrahedral linear element

(d) Tetrahedral quadratic element

Figure 3: Solution for 3D Poisson equation from different element type

Table 1: Computational cost for each solution

Mesh type	C++ Time(s)	MATLAB Time(s)
Quadrilateral linear	0.182317	0.898
Quadrilateral quadratic	3.2176	2.064
Triangle linear	0.142054	1.122
Triangle quadratic	5.61628	1.122
Hexahedral linear	4.49461	3.630
Hexahedral quadratic	NA	7.253
Tetrahedral linear	3.67554	5.110
Tetrahedral quadratic	734.961	16.509