PROGRAMMING FOR ENGINEERS AND SCIENTISTS

Homework 1
Matlab

Submitted by
Karthik Neerala Suresh
M.Sc. Computational Mechanics
Universitat Politècnica de Catalunya, BarcelonaTECH

Submitted to
Dr. Esther Sala Lardies
Associate Professor
Universitat Politècnica de Catalunya, BarcelonaTECH

03 April 2017

Contents

List of Figures

List of Tables

1. Homework 1

1.1 Problem Description

The objective of this homework is to determine the potential flow around an object solving the following two dimensional Poisson problem

$$\begin{cases} \Delta u = 0 \quad \text{in } \Omega, \\ \boldsymbol{\nabla} u \cdot \mathbf{n} = -1 \quad \text{on } \Gamma_{\text{in}} = 0 \times (0,1), \\ \boldsymbol{\nabla} u \cdot \mathbf{n} = 1 \quad \text{on } \Gamma_{\text{out}} = 1 \times (0,1), \\ \boldsymbol{\nabla} u \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega \backslash (\Gamma_{\text{in}} \cup \Gamma_{\text{out}}), \\ u(0,0) = 0, \end{cases} \tag{1.1}$$

where Ω is the computational domain shown in Figure 1.1, $\partial\Omega$ is its boundary and $\mathbf{n}$ is the outward unit normal. Further it is required to obtain the velocity field in terms of this potential as

$$v_x = \frac{\partial u}{\partial x} \quad v_y = \frac{\partial u}{\partial y}. \tag{1.2}$$

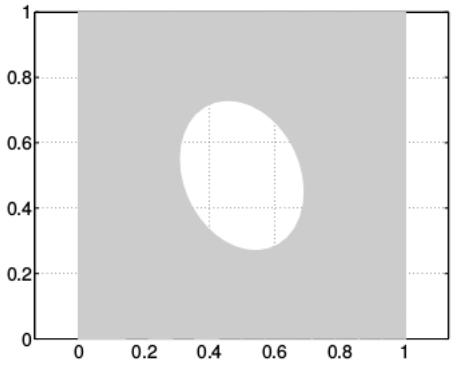


Figure 1.1: Computational domain

In order to achieve the objectives of this homework, a code must be developed for solving the two-dimensional Poisson equation, accounting for the following features:

- The computational mesh is loaded from input files,
- Triangular and quadrilateral elements of degree one and two can be used,
- A convergence analysis must be performed for the different elements, using as a reference solution the one on the finer mesh.

In order to make a comparison of the solutions, running time and accuracy for all the element's types, five different meshes are considered for each type of element, with the i-th mesh containing approximately the same number of nodes for all element types.

1.2 The Poisson's equation

The strong form of the Poisson's equation is given in Equation (1.1). In order to solve this using the standard finite element method, the first step involves determining the weak form of the problem after multiplying it with the suitable test function. The weak form can be written as

$$\int_{\Omega} \boldsymbol{\nabla} w \cdot \boldsymbol{\nabla} u \, d\Omega = \int_{\Gamma_N} w(\boldsymbol{\nabla} u \cdot \mathbf{n}) \, d\Gamma \quad \forall w \tag{1.3}$$

where Γ_N is the Neumann portion of the boundary where normal fluxes are prescribed.

The Galerkin approximation of the weak form given in (1.3) can be written as follows. First the computational domain is discretized into elements Ω_e. The approximate solution u^h and the test function w^h is written as

$$u \approx u^h(x) = \sum_j N_j(x) u_j$$
$$w \approx w^h(x) = \sum_i N_i(x) u_i$$

The discrete form of the (1.3) can be written as

$$w_j \left(\int_{\Omega} \boldsymbol{\nabla} N_j \boldsymbol{\nabla} N_i d\Omega \right) u_i = w_j \left(\int_{\Gamma_N} N_j q \, d\Gamma \right) \quad \forall w_j \tag{1.4}$$

where $\int_{\Omega} \boldsymbol{\nabla} N_j \boldsymbol{\nabla} N_i d\Omega$ is the elemental stiffness matrix $\mathbf{K}_e$.

1.3 Structure of the program

The code written in order to solve the poisson's equation has a definite structure as the finite element method is being used. In this section the structure of the code will be explained, giving emphasis to the reason behind the choice for such a structure. Also, procedures undertaken to ensure the test the routines written will also be briefed wherever applicable.

1.3.1 User Defined Data

Before beginning the process of solving the equation, there is a need to define the level of refinement of the domain, the type of element to be used and the order of polynomial. These are user defined parameters and are left to the discretion of the user within the available functionalities of the code. For example, the user can choose to use linear or quadratic finite elements to solve the problem. Also meshes with both triangles and quadrilaterals are available for both the above mentioned order of polynomials in order to solve the problem. Finally five different meshes of varied levels of mesh refinement are provided to the user for all the four different combinations of order and type of element. In the present code, the first section is devoted to obtaining data from the user in the following manner.

```
%% User defined parameters
orderOfApproximation = 1;
elementType = 0;
% Use 0 for quadrilateral element mesh
% and 1 for triangular element mesh
meshRefinement = 1;
```

The user has to manually change the parameters to switch from one option to another. There are better ways of implementing this section on user defined data. One can use the Matlab function 'input' to allow the user to choose these parameters on the command window after beginning the run of the code. This will avoid users from interfering with the source code. Yet another option that can be used to get the user input is by making use of the function 'menu'. This is ideal for a situation like that of ours where there are finite number of options from which the users need to choose one. Since this is a very small program, the time required for these operations does not really matter. However, in large programs, one needs to also consider the memory usage and time requirement before providing one of the above options for the user to input their selection.

1.3.2 Processing user defined data

Once the user has defined his choice, it become imperative to extract information that will be useful from a finite element context. The first major requirement for any finite element code is the information of the coordinates of the nodes in the mesh, called the nodal coordinates, and the connectivity table that defines the nodes constituting the elements. In the present project, mesh information is provided in data files (.dat file). These files are named using a common format that can be related to user defined parameters. The data file containing information about the connectivity matrix is named as `Element_2D_type%d_P%d_H%d.dat`, where the three `%d` correspond to the user defined input of element type (0 for quadrilaterals or 1 for triangles), order of approximation (1 for linear and 2 for quadrilaterals) and degree of mesh refinement (1 being the coarsest mesh and 5 being the finest). In a similar manner, the data files with information about nodal coordinates are named as `Node_2D_type%d_P%d_H%d.dat`.

In the present code, a cumbersome method is adopted to read information from these data files by making use of the function 'fscanf'. This function can be used after the file is opened for reading using 'fopen'. Unfortunately, fscanf stores all the information from the file in a row vector. This is not what is ideal for a finite element code. The connectivity table must be stored as a matrix with as many rows as elements with each row containing as many columns as there are nodes in the element. Also nodal coordinates must be stored as a matrix with as many rows as there are nodes in the mesh and as many columns as are the number of spatial dimensions. In order to achieve this from the information read from the files that are stored in row vectors, an additional step of rearranging is done by making use of a for loop. The entire of sequence of routine to obtain the connectivity matrix and nodal coordinates as required by a finite element code, as implemented in the code is shown below.

```
fileNameT = sprintf('Element_2D_type%d_P%d_H%d.dat',...
elementType,orderOfApproximation,meshRefinement);
fileNameX = sprintf('Node_2D_type%d_P%d_H%d.dat',...
elementType,orderOfApproximation,meshRefinement);
fileID_T = fopen(fileNameT,'r');
fileID_X = fopen(fileNameX,'r');
switch elementType
case 0
nOfNodesPerElement = orderOfApproximation*4;
nOfVertices = 4;
case 1
nOfNodesPerElement = (orderOfApproximation+1)...
*(orderOfApproximation+2)/2;
```

```
nOfVertices = 3;
otherwise
error('element type not defined');
end
Tvector = fscanf(fileID_T,'%d',[1,inf]);
Xvector = fscanf(fileID_X,'%d %f %f\n',[1,inf]);
T = zeros(length(Tvector)/nOfNodesPerElement,...
nOfNodesPerElement);
X_1 = zeros(length(Xvector)/3,3);
for indexT = 1:length(Tvector)/nOfNodesPerElement
T(indexT,:) = Tvector((indexT-1)*nOfNodesPerElement+1...
:indexT*nOfNodesPerElement);
end
for indexX = 1:length(Xvector)/3
X_1(indexX,:) = Xvector((indexX-1)*3+1:indexX*3);
end
[Y,I] = sort(X_1(:,1),1);
X = X_1(I,:);
[hMin] = computeMinElementSize(X(:,2:3), T);
```

This part of the code can be improved by converting the .dat files into .mat files and then using the Matlab function 'load'. Load saves data in the manner in which is desirable for finite element codes and thus avoids the requirement of a routine to reorganise data. It is also possible to use format specifiers in fscanf to improve the manner in which data is read and stored from the .dat file.

The final line in the routine that processes the user defined data to generate information relevant from a finite element context calls a function called `computeMinElementSize`. This function has been written to compute the minimum characteristic size of the elements in the mesh. This information is necessary when convergence study is carried out to validate the written finite element routine. This function uses distance formula to compute the edge length of every element. Thus, inside this function, a loop on elements is done and the minimum edge length of each element is computed. Finally the minimum of all the minimum elemental edge lengths is considered as the characteristic length of the given mesh.

1.3.3 Creating reference element

In every standard finite element code, every computation is carried out in a reference element. Thus it is a necessity to generate the right reference element in any standard finite element code. In the present code, four different types of elements are available, linear and quadratic triangles and quadrilaterals. Thus based on the

choice of input, the right reference element information must be accessed by the code. The fields that are present in the structure called `refElem` created by the function `referenceElement` contain information about the nodal coordinates of the reference elements, the coordinates of the nodes in each of the faces, shape functions and their derivatives computed at volume integration Gauss points and the weights corresponding to these quadrature points (in our case 2D shape functions and their derivatives). Also shape functions and derivatives are computed at quadrature points required for integration along Neumann faces (in our case 1D shape functions and their derivatives).

Since the reference element forms the crux of the finite element computations, it is mandatory to ensure the accuracy of the quadrature points, weights and the shape functions. Some functions are written to test these. In order to test the shape functions, there is a need to ensure that the shape functions satisfy the delta property at the nodes. This is tested by using a simple routine as shown below:

```
% test 1D shape functions

tol = 1e-10;
for degree = 1:3
theReferenceElement = referenceElement1D(degree);
nodes = theReferenceElement.nodeCoord;
N = shapeFunctions1D(nodes,degree);
Err = max(max(abs(eye(degree+1) - N)));
condition = sprintf('%0.16g <= %0.16g', Err, tol);
testName = sprintf('1D shape functions, degree=%d', degree);
myAssert(condition, testName, nameFunction, lineNumber);
end
```

Quadratures are tested by ensuring that they are able to integrate exactly polynomials upto the order for which they are designed. This is done by writing a simple routine that compares numerical integration to the exact value of the integral computed analytically. The routine written is as follows:

```
% test 2D quadrature

tol = 1e-10;

% Triangles
Iex_tri = [
2   -2/3   2/3   -2/5   2/5   -2/7   2/7   -2/9   2/9
-2/11   2/11
-2/3   0   -2/15   0   -2/35   0   -2/63   0   -2/99
0   -2/143
```

```
        2/3    -2/15    2/9    -2/21    2/15    -2/27    2/21    -2/33
2/27    -2/39    2/33
        -2/5    0    -2/21    0    -2/45    0    -2/77    0    -2/177
0    -2/165
        2/5    -2/35    2/15    -2/45    2/25    -2/55    2/35    -2/65
2/45    -2/75    2/55
        -2/7    0    -2/27    0    -2/55    0    -2/91    0    -2/135
0    -2/187
        2/7    -2/63    2/21    -2/77    2/35    -2/91    2/49    -2/105
2/63    -2/119    2/77
        -2/9    0    -2/33    0    -2/65    0    -2/105    0    -2/153
0    -2/209
        2/9    -2/99    2/27    -2/117    2/45    -2/135    2/63
-2/153    2/81    -2/71    2/99
        -2/11    0    -2/39    0    -2/75    0    -2/119    0    -2/171
0    -2/231
        2/11    -2/143    2/33    -2/165    2/55    -2/187    2/77
-2/209    2/99    -2/231    2/121
        ];

        Iex_qua = [
        4    0    4/3    0    4/5    0    4/7    0    4/9    0    4/11
        0    0    0    0    0    0    0    0    0    0    0
        4/3    0    4/9    0    4/15    0    4/21    0    4/27    0
4/33
        0    0    0    0    0    0    0    0    0    0    0
        4/5    0    4/15    0    4/25    0    4/35    0    4/45    0
4/55
        0    0    0    0    0    0    0    0    0    0    0
        4/7    0    4/21    0    4/35    0    4/49    0    4/63    0
4/77
        0    0    0    0    0    0    0    0    0    0    0
        4/9    0    4/27    0    4/45    0    4/63    0    4/81    0
4/99
        0    0    0    0    0    0    0    0    0    0    0
        4/11    0    4/33    0    4/55    0    4/77    0    4/99    0
4/121
        ];

        % elemType = 1;
```

```
nIP = [1 3 4 6 12 12];

for iIP = 1:length(nIP)
[z,w] = quadrature2D(1,nIP(iIP));

for powX = 0:iIP
for powY = 0:(iIP-powX)
exactInt = Iex_tri(powX+1,powY+1);
aproxInt = w*(z(:,1).^powX.*z(:,2).^powY);

if exactInt == 0
relErr = abs(aproxInt - exactInt);
else
relErr = abs(aproxInt - exactInt) / exactInt;
end
condition = sprintf('%0.16g <= %0.16g', relErr, tol);
testName = sprintf('1D quadrature points=%d,order=%d', nIP,k);
myAssert(condition, testName, nameFunction, lineNumber);
end
end
end

% elemType = 0;
for n = 1:4
nIP = n^2;
order = 2*n-1;
[z,w] = quadrature2D(0,nIP);

for powX = 0:order
for powY = 0:order
exactInt = Iex_qua(powX+1,powY+1);
aproxInt = w*(z(:,1).^powX.*z(:,2).^powY);

if exactInt == 0
relErr = abs(aproxInt - exactInt);
else
relErr = abs(aproxInt - exactInt) / exactInt;
end
condition = sprintf('%0.16g <= %0.16g', relErr, tol);
testName = sprintf('1D quadrature points=%d,order=%d', nIP,k);
```

```
        myAssert(condition, testName, nameFunction, lineNumber);
    end
    end
    end
```

Once it has been ensured that the shape functions and the quadratures have been properly computed, the code then moves onto the preprocessing, computing and postprocessing sections.

1.3.4 Preprocessing boundary data

This section of the code is useful while imposing boundary conditions. A function called `preprocess` is written which returns two matrices called `DBCmatrix` and `NBCmatrix` containing information about Dirichlet and Neumann boundary conditions. The variable `DBCmatrix`, contains as many rows as there are nodes on the boundary on which fixed displacement boundary condition is imposed. The columns of this matrix contain node number and the value of the fixed displacement at the specified node. Dirichlet boundary condition information is stored in this manner as Dirichlet boundary condition is imposed in this code by eliminating the rows and columns corresponding to dirichlet nodes from the global matrix.

The variable `NBCmatrix` is slightly different from Dirichlet counterpart. This matrix has as many rows as there are faces on the Neumann boundary. The columns of this matrix contain information about the nodes on the Neumann face considered in a counter clockwise manner, with the final column containing the value of the prescribed normal flux on the face.

1.3.5 Computation

This section of the code is the most important part where the finite element computations are carried out. In the function `elementalComputation`, a loop on elements is done to get the elemental contributions from each element. In this loop, a loop on Gauss points is done to carry out integration in the computation of the stiffness matrix and the right hand side forcing term. In the loop on elements, first the value of the shape functions and their derivatives are extracted from the structure `refElem` at the Gauss points. Following this in the loop on Gauss points the Jacobian of the transformation is computed and then the individual contribution of the gauss point to the elemental contribution is computed. Finally using the connectivity matrix, the elemental contribution is added to the global matrix.

The routine written here is not an ideal one. Since the global stiffness matrix is largely sparse, memory and time can be saved by initialising the global stiffness

matrix as a sparse matrix and then using the matrix matlab function sparse to assemble the elemental contributions. This will make the code both time and memory efficient.

Once the global stiffness matrix has been assembled, the Neumann boundary condition contribution needs to be added to the system. This is done in the function named `nuemannFaceComputation`. In this function a loop over all the Neumann faces is done and then similar to the loop in the elemental contribution computation, a loop over Gauss points for face integration is done and the contribution of each Neumann face is added to the global right hand side vector. The matrix `NBCmatrix` defined in the preprocessing part of the code is used to get information about the nuemann faces.

Once the global stiffness matrix and the right hand side vector have been computed, the Dirichlet boundary condition is imposed by making use of `DBCmatrix` defined in the preprocessing part of the code in the function `deleteRowsDBC`. In this functions, rows and columns corresponding to the nodes with imposed displacement are eliminated. In case of non zero imposed displacement, the right hand side vector is modified.

Finally displacement is computed by making us of the LU decomposition direct method. Since the system to be solved in this Poisson problem is considerably small, the direct method used for solving the system of equation does not make the code considerably slow. However, when larger problems are solved there is a need to check if this is the best way to solve the system of linear equations or whether there is a need to use some iterative solvers.

1.3.6 Post-process

Velocity field

As the velocity potential is obtained by solving the system of equations, the velocity field (1.2), which is nothing but the derivative of solution, is computed using the function as follows in the code.

```
[N,Nxi,Neta] = shapeFunctions2D(elementType,refElem.p,...
refElem.nodesCoord(:,1),refElem.nodesCoord(:,2));
for iElem = 1:size(T,1)
for jNode = 1:size(T,2)
NxiNode = Nxi(jNode,:);
NetaNode = Neta(jNode,:);
J = [NxiNode;NetaNode]*X(T(iElem,:),2:3);
res = J\[NxiNode;NetaNode];
dNdx = res(1,:);
dNdy = res(2,:);
```

```
ux = dNdx*u(T(iElem,:),:);
uy = dNdy*u(T(iElem,:),:);
velo(T(iElem,jNode),:) = [ux uy];
end
end
```

The reference element coordinates and derivatives of shape functions at these coordinates are necessary to compute the derivative of solutions. The reference element coordinates are stored in the structure refElem. The shape function at this coordinates are computed using the function `shapeFunctions2D`. Finally to get the derivatives of the solution, a loop over elements and then a loop over nodes is defined to extract nodal shape functions, Jacobian and derivatives in x and y coordinates. Elemental velocity fields are assembled using connectivity matrix and stored in the variable named as `velo`.

Error computation

In order to validate any finite element code, it is necessary to perform a convergence study to ensure the solution converges asymptotically at the desired rate. The $\mathcal{L}_2$ norm and $\mathcal{H}^1$ seminorms of the errors are computed as

$$||e||_{\mathcal{L}_2} = \left[\frac{\int_\Omega (u^h - u)^2 \, d\Omega}{\int_\Omega u^2 \, d\Omega} \right]^{(1/2)}$$

$$||e||_{\mathcal{H}^1} = \left[\frac{\int_\Omega (\boldsymbol{\nabla} u^h - \boldsymbol{\nabla} u) \cdot (\boldsymbol{\nabla} u^h - \boldsymbol{\nabla} u) \, d\Omega}{\int_\Omega \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} u \, d\Omega} \right]^{(1/2)}$$

where u^h is the solution of the current mesh and u is the analytical solution.

For the given Poisson's problem analytical solution is not available. Hence, in order to perform convergence study, the solution obtained using the finest mesh is chosen as the reference solution u. The solution computed using the finest mesh for all the four cases are stored in `.mat` as `uTriLinear`, `uTriQuadratic`, `uQuadLinear` and `uQuadQuadratic` for the four cases. Based on the type of element used and the order of approximation these are loaded at the beginning of the run of the code and the solution, nodal coordinates and the connectivity matrix of the finest mesh are stored in variables called `uForError`, `XforError` and `TforError`.

Upon computing the solution for the problem these variables, along with the computed solution and connectivities and nodal coordinates of the present mesh under consideration are sent into a function called `computeError`. In the compute error a loop on the elements in the finest mesh is done. At each Gauss point of the finest mesh, the reference solution and the computed solution are calculated and the using these values, the norms of errors are estimated.

However, finding the computed solution at the Gauss point of the fine mesh is not that straigh forward. Since isoparametric transformation is used to compute the solution at any point in the element, shape functions must be computed at these points. In order to compute shape functions at the Gauss points of the finest mesh corresponding to the course mesh, an inverse isoparamteric transformation needs to done. This is quite straight forward in the case of linear triangles as the mapping is linear. However, with quadratic triangles and quadrilateral elements, one needs to solve a non linear equation for computing the reference element position of every Gauss point. This is done using `fsolve` in the program. Also to accurate results, the tolerance of `fsolve` must be set to very small value. This makes the task of computing error extremely expensive.

Convergence test is performed to check the correctness of the code. However, the error computation is much costlier than the computation of the solution. Also, in the present problem, the error computation involves inverse mapping where nonlinear equation is to be solved to compute the coordinates of the Gauss point in the reference element. So, the accuracy of the code is checked by performing the convergence test with an example problem whose analytical solution is known. In addition to this, both 1D and 2D the quadrature and shape functions are tested.

1.4 Convergence test

The evaluation of $\mathcal{L}_2$ and $\mathcal{H}^1$ norms of the error for the present problem involves solving of non-linear equations. Also it has been observed that the finest mesh availbale is not fine enough to use its solution as analytical solution. Hence the convergence test doesn't completely tell about the correctness of the code as the reference solution is not accurate enough to compute the error. However, the convergence rate of the triangular elements with first order approximation are presented in the table 1.1. From the table and plot, it can seen that the $\mathcal{L}_2$ norm and $\mathcal{H}^1$ semi-norms error converge at the rate of 2 and 1 respectively as the characteristic element size approaches zero. Therefore, these convergence rates are in agreement with the rate of convergence of standard FEM as the element used here is of degree 1.

Table 1.1: Error and convergence associated with triangular elements of degree 1

h	Error in the norm		Convergence rate	
	$\mathcal{L}^2$	$\mathcal{H}_1$	$\mathcal{L}^2$	$\mathcal{H}_1$
3.18E-002	6.39E-003	5.29E-002	2.22	0.80
2.34E-002	3.23E-003	4.14E-002	2.05	0.76
1.42E-002	1.16E-003	2.83E-002		

Since the computation of error involves inverse isoparametric transformation of each of the Gauss point in the finest mesh, the error computation routine is highly time consuming in addition to being complex in implementation. The time taken to solve for the solution is much less than the the time required to compute the error. The time taken for triangular elements of degree 1 is compared in the table 1.2.

Table 1.2: Computation time for triangular elements of degree 1

Triangular mesh degree 1	Computation time in seconds	
	without error	with error
1	0.16	240.00
2	0.15	285.00
3	0.35	421.00
4	0.95	674.00
5	2.08	1030.00

Further it has been noticed that the meshes are not nested meshed. For accurate convergence study it is necessary that the meshes under consideration are nested meshes with the characteristic size of each refined mesh being one half of the previous course mesh. Since error computation for this problem consumes very much higher time than the computation of the solution, and since the finest mesh is not fine enough to be considered as reference solution in addition to the absence of nested meshes upon refinement,this method of checking the correctness of the code is not recommended. The correctness of the routine is tested with an example problem whose analytical solution is known.

1.4.1 Convergence test with example problem

An example problem whose analytical solution is known is solved using the same routine. The problem is described below.

$$\begin{cases} \Delta u = & 0 \quad \textbf{in } [0,1] \times [0,1] \\ u(x,0) = & \sin(\pi x) \end{cases}$$

Analytical solution of the problem is,

$$u(x,0) = \cosh(\pi y) - \coth(\pi)\sinh(\pi y)\sin(\pi x)$$

The convergence rate of the obtained solution with different types of elements is tabulated in 1.3, 1.4, 1.5 and 1.6 and plots are shown in Figures 1.2 and 1.3.

Table 1.3: Example problem: Error and convergence associated with linear triangular elements

h	Error in the norm		Convergence rate	
	$\mathcal{L}^2$	$\mathcal{H}_1$	$\mathcal{L}^2$	$\mathcal{H}_1$
1.00E-001	1.05E-002	1.45E-001	1.99	1.01
5.00E-002	2.64E-003	7.22E-002	2.00	1.00
2.50E-002	6.62E-004	3.61E-002	2.00	1.01
1.67E-002	2.94E-004	2.40E-002		

Table 1.4: Example problem: Error and convergence associated with quadratic triangular elements

h	Error in the norm		Convergence rate	
	$\mathcal{L}^2$	$\mathcal{H}_1$	$\mathcal{L}^2$	$\mathcal{H}_1$
1.00E-001	3.96E-004	6.69E-003	3.00	1.99
5.00E-002	4.96E-005	1.68E-003	3.00	1.99
2.50E-002	6.21E-006	4.22E-004	3.00	1.99
1.67E-002	1.84E-006	1.88E-004		

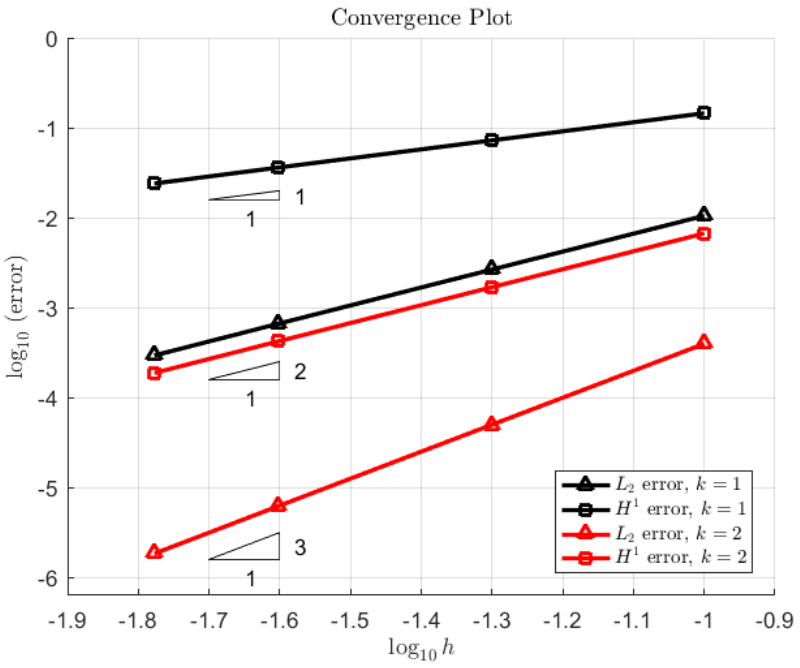


Figure 1.2: Convergence plot with triangular elements

Table 1.5: Example problem: Error and convergence associated with linear quadrilateral elements

h	Error in the norm		Convergence rate	
	$\mathcal{L}^2$	$\mathcal{H}_1$	$\mathcal{L}^2$	$\mathcal{H}_1$
1.00E-001	4.96E-003	9.07E-002	2.00	1.00
5.00E-002	1.24E-003	4.54E-002	2.00	1.00
2.50E-002	3.10E-004	2.27E-002	2.00	1.01
1.67E-002	1.38E-004	1.51E-002		

Table 1.6: Example problem: Error and convergence associated with quadratic quadrilateral elements

h	Error in the norm		Convergence rate	
	$\mathcal{L}^2$	$\mathcal{H}_1$	$\mathcal{L}^2$	$\mathcal{H}_1$
1.00E-001	2.12E-004	3.68E-003	2.99	2.00
5.00E-002	2.66E-005	9.19E-004	3.00	2.00
2.50E-002	3.33E-006	2.30E-004	3.00	2.01
1.67E-002	9.88E-007	1.02E-004		

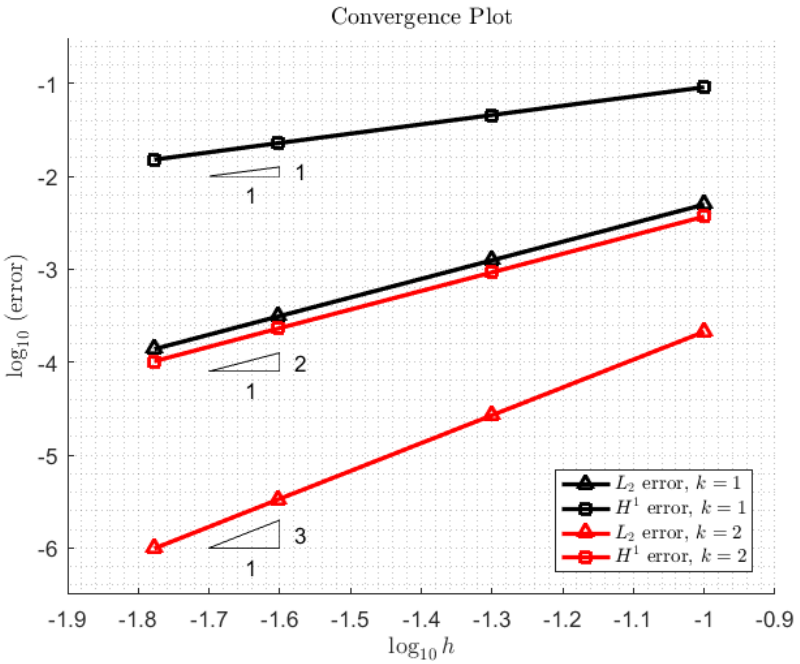


Figure 1.3: Convergence plot with quadrilateral elements of degree 2

In the case of standard FEM, for an interpolation polynomial of degree p, the norms of upper bounds of the errors is given by,

$$||e||_{\mathcal{L}_2} \leq Ch^{p+1}$$
$$||e||_{\mathcal{H}^1} \leq Ch^{p}$$

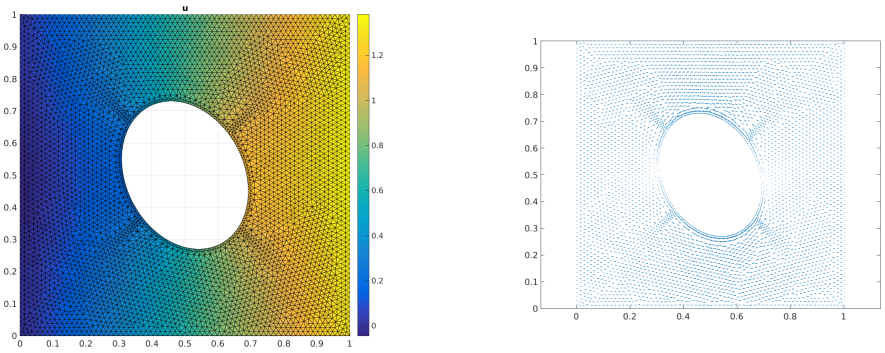
From the table and plot, it can seen that the $\mathcal{L}_2$ norm and $\mathcal{H}^1$ semi-norms error converge at the rate of $p+1$ and p respectively as the characteristic element size approaches zero. Therefore, these convergence rates are in agreement with the rate of convergence of standard FEM.

1.5 Post-processing

In this section velocity potential and velocity fields obtained upon solving the problems using different element types is presented. The post-processing is done in both matlab and paraview. Additionally, the computation time is compared for different elements.

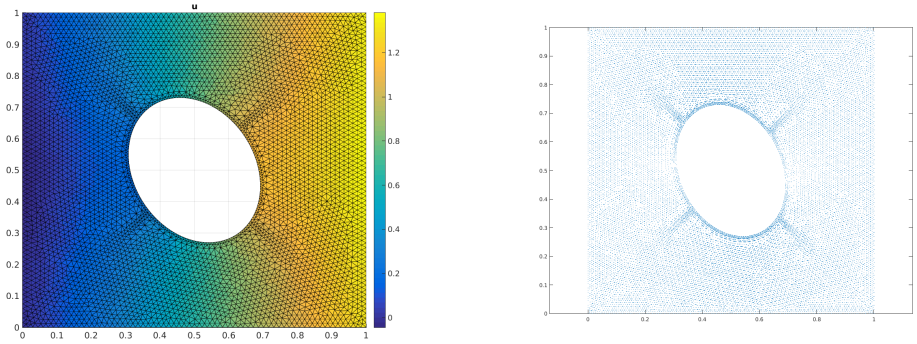
1.5.1 Matlab post-process

The velocity potential and velocity fields of the finest mesh in each element type are presented using Matlab. As the velocity potential is a scalar, `trisurf` is used to plot. Whereas the velocity field is a vector, so `quiver` is used to present the field.



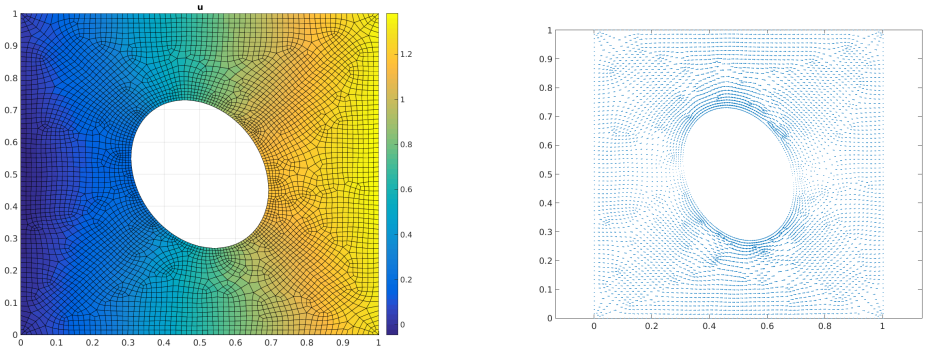
(a) Velocity potential (b) Velocity Field

Figure 1.4: Results for linear triangular elements



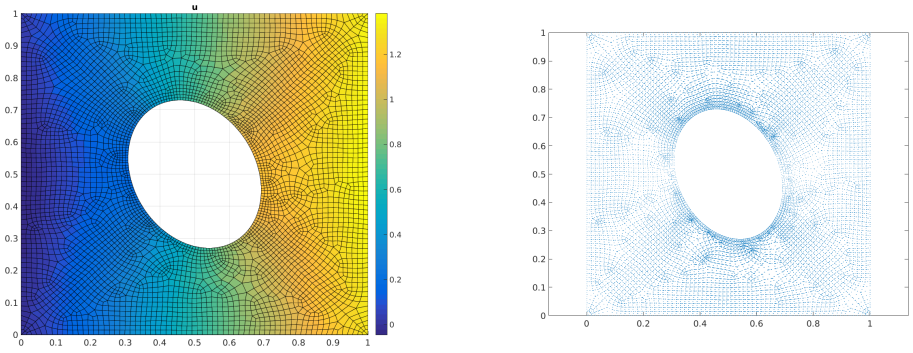
(a) Velocity potential (b) Velocity Field

Figure 1.5: Results for quadratic triangular elements



(a) Velocity potential (b) Velocity Field

Figure 1.6: Results for linear quadrilateral elements



(a) Velocity potential (b) Velocity Field

Figure 1.7: Results for quadratic quadrilateral elements

1.5.2 VTK post-process

Paraview is used to read VTK files. The VTK files are generated using Matlab program named as `writeToText.m`.It reads data from .dat files which contains the nodal coordinates, connectivity, velocity potential and velocity field. These .dat files are saved from the Poisson code `poissonFEM.m`. The figures below are the post-process results of the stated problem. The colour map represents the velocity potential whereas the arrows represent the velocity field.

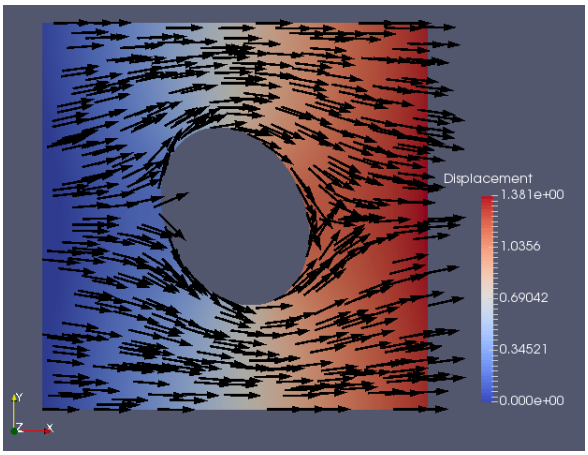


Figure 1.8: VTK results for linear triangular elements

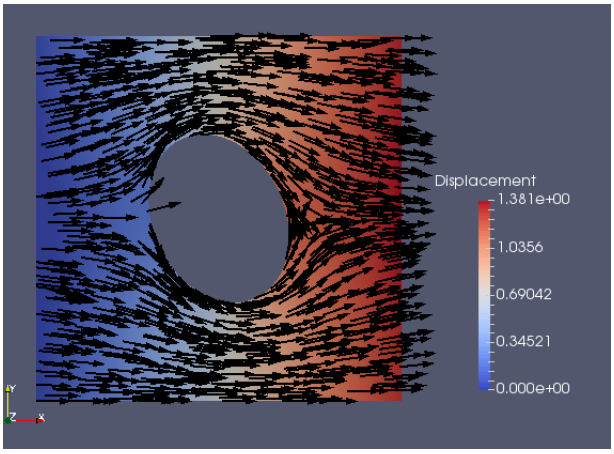


Figure 1.9: VTK results for quadratic triangular elements

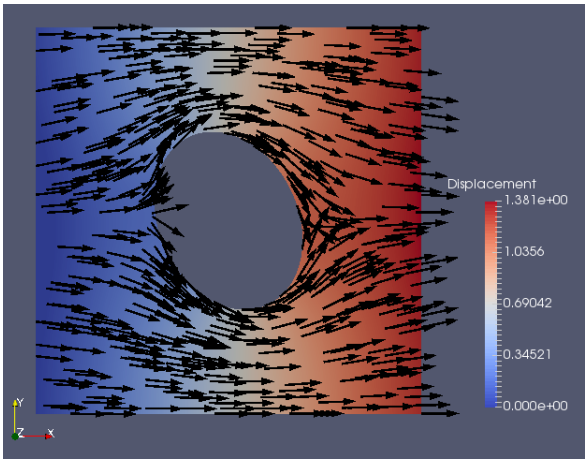


Figure 1.10: VTK results for linear quadrilateral elements

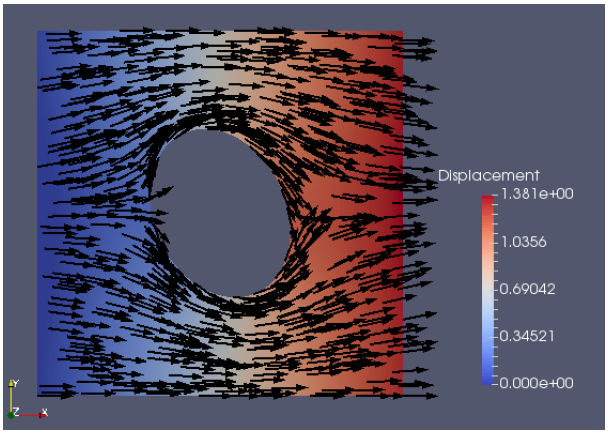


Figure 1.11: VTK results for quadratic quadrilateral elements

1.6 Comparison of computation time

The computation time for the Poisson problem using different element types is tabulated in 1.7.

Table 1.7: Description of different cases and their `CaseNum`

Description	Time in seconds	Description	Time in seconds
Linear triangular elements		Linear quadrilateral elements	
Mesh 1	0.13	Mesh 1	1.06
Mesh 2	0.17	Mesh 2	0.27
Mesh 3	0.49	Mesh 3	0.33
Mesh 4	1.17	Mesh 4	1.11
Mesh 5	2.26	Mesh 5	1.92
Quadratic triangular elements		Quadratic quadrilateral elements	
Mesh 1	0.16	Mesh 1	0.20
Mesh 2	0.31	Mesh 2	0.26
Mesh 3	1.36	Mesh 3	0.89
Mesh 4	7.67	Mesh 4	4.02
Mesh 5	71.76	Mesh 5	15.47

1.7 Conclusion

Matlab program is developed to solve the 2D Poisson problem is solved to obtain potential flow around an object using finite element method using both triangular and quadrilateral element types with linear and quadratic orders of approximation. The correctness of the code is tested using convergence study and it is found that the errors converge asymptotically to at the desired rate. Post-process results from both Matlab and paraview are presented and computation times are compared for different element types.