2D Plane Stress Analysis using Finite Element Method

MAE529 - Finite Element Structural Analysis - Course Project
Instructor - Prof. Abani Patra

University at Buffalo

Pranav Ladkat (5009-4671)

1 Introduction

In continuum mechanics, a material is said to be under plane stress if the stress vector is zero across a particular surface. When that situation occurs over an entire element of a structure, as is often the case for thin plates, the stress analysis is considerably simplified, as the stress state can be represented by a tensor of dimension 2 rather than dimension 3. Plane stress typically occurs in thin flat plates that are acted upon only by load forces that are parallel to them.

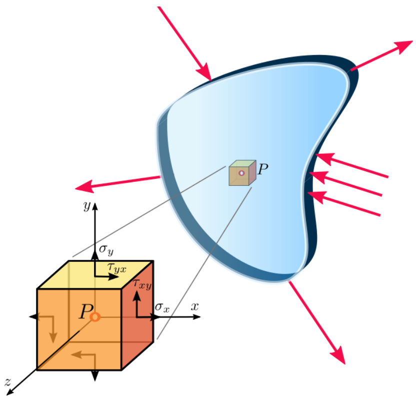

Figure 1: Plane stress state in a continuum, src: Wikipedia

i.e. in a state of plain stress, following stress field exist:

$$\begin{gathered}\sigma_{xz} = \sigma_{yz} = \sigma_{zz} = 0 \\ \sigma_{xx} = \sigma_{xx}(x,y), \sigma_{xy} = \sigma_{xy}(x,y), \sigma_{yy} = \sigma_{yy}(x,y)\end{gathered} \tag{1}$$

Assumptions: Problem is steady state and material is assumed to be Isotropic.
The strain field associated with stress field in equation (1) is

$$\begin{Bmatrix}\epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy}\end{Bmatrix} = \begin{bmatrix} s_{11} & s_{12} & 0 \\ s_{21} & s_{22} & 0 \\ 0 & 0 & s_{66}\end{bmatrix} \begin{Bmatrix}\sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy}\end{Bmatrix} \tag{2}$$

$$\epsilon_{xz} = \epsilon_{yz} = 0, \epsilon_{zz} = s_{13}\sigma_{xz} + s_{23}\sigma_{yz} \tag{3}$$

where, $s_{i,j}$ are elastic complacences

$$\begin{gathered}s_{11} = \frac{1}{E_1}, s_{22} = \frac{1}{E_2}, s_{33} = \frac{1}{E_3}, s_{66} = \frac{2(1+\nu)}{E} \\ s_{12} = -\nu_{21}s_{22} = -\nu_{12}s_{11}, s_{13} = -\nu_{31}s_{33} = -\nu_{13}s_{11}, s_{23} = -\nu_{32}s_{33} = -\nu_{23}s_{22}\end{gathered} \tag{4}$$

The stiffness matrix for plain stress and for isotropic material is found by inverting equation (2), is given by

$$\begin{Bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{Bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & 1-\nu \end{bmatrix} \begin{Bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy} \end{Bmatrix} \tag{5}$$

2 Governing Equations

The governing equation for the plane elasticity problem in expanded vector form are:

$$\begin{aligned} \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + f_x = \rho \frac{\partial^2 u_x}{\partial t^2} \\ \frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + f_y = \rho \frac{\partial^2 u_y}{\partial t^2} \end{aligned} \tag{6}$$

or,

$$\mathbf{D}^* \sigma + \mathbf{f} = \rho \ddot{\mathbf{u}} \tag{7}$$

where, f_x and f_y denote components of the body force vector in x and y direction respectively. ρ is material density, and

$$\mathbf{D}^* = \begin{bmatrix} \partial/\partial x & 0 & \partial/\partial y \\ 0 & \partial/\partial y & \partial/\partial x \end{bmatrix}, \sigma = \begin{Bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{Bmatrix}, \mathbf{f} = \begin{Bmatrix} f_x \\ f_y \end{Bmatrix}, \mathbf{u} = \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} \tag{8}$$

As per the assumption, only steady problem is taken under consideration, hence the time derivative term vanishes. Hence the governing equation becomes,

$$\mathbf{D}^* \sigma + \mathbf{f} = 0 \tag{9}$$

Strain-Displacement Relations are:

$$\epsilon_{xx} = \frac{\partial u_x}{\partial x}, \epsilon_{yy} = \frac{\partial u_y}{\partial y}, 2\epsilon_{xy} = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x}, \tag{10}$$

or,

$$\epsilon = \mathbf{D}\mathbf{u}, \ \mathbf{D} = (\mathbf{D}^*)^T \tag{11}$$

And Stress-Strain relations are:

$$\begin{Bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{Bmatrix} = \begin{bmatrix} c_{11} & c_{12} & 0 \\ c_{21} & c_{22} & 0 \\ 0 & 0 & c_{66} \end{bmatrix} \begin{Bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy} \end{Bmatrix} \tag{12}$$

where the values of c_{ij} can be found from equation (5). Therefore, the governing equation (9) becomes,

$$-\mathbf{D}^* \mathbf{C} \mathbf{D} \mathbf{u} = \mathbf{f} \tag{13}$$

3 Weak Form of the Governing Equations

The weak form of the governing equations can be obtained by multiplying the strong form by variational parameter w and integrating over the entire domain. For the first equation we get,

$$\int_{\Omega_e} w_1 \left(\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + fx \right) h_e \, dx \, dy = 0$$

Integrating by parts, (14)

$$\int_{\Omega_e} h_e \left(\frac{\partial w_1}{\partial x} \sigma_{xx} + \frac{\partial w_1}{\partial y} \sigma_{xy} - w_1 f_x \right) = \oint_{\Gamma_e} h_e \, w_1 (\sigma_{xx} n_x + \sigma_{xy} n_y) \, ds$$

Similarly, for the second equation,we get

$$\int_{\Omega_e} h_e \left(\frac{\partial w_2}{\partial x} \sigma_{xy} + \frac{\partial w_2}{\partial y} \sigma_{yy} - w_2 f_x \right) = \oint_{\Gamma_e} h_e \, w_2 (\sigma_{xy} n_x + \sigma_{yy} n_y) \, ds \tag{15}$$

where,

$$\sigma_{xx} = c_{11} \frac{\partial u_x}{\partial x} + c_{12} \frac{\partial u_y}{\partial y}, \ \sigma_{yy} = c_{12} \frac{\partial u_x}{\partial x} + c_{22} \frac{\partial u_y}{\partial y}, \ \sigma_{xy} = c_{66} \left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \right) \tag{16}$$

and h_e is the thickness of the plate and is assumed to be constant in the present study,

4 Finite Element Model

Examination of the weak form reveals that: (a) u_x and u_y are the primary variables which must be carried as primary nodal degrees of freedom and (b) only first derivative pf the u_x and u_y with respect to x and y appear. Therefore u_x and u_y must be approximated by at least bilinear interpolation. The simplest element that satisfy those requirements are the linear triangular and and linear quadrilateral elements. Although u_x and u_y are independent of each other, they are components of the displacement vector and therefore should be approximated with same type and degree of interpolation.

The Bilinear Quadrilateral Element is selected for the present study but other element can be easily incorporated as the program uses Object Oriented features of C++. The details below are for the bilinear quadrilateral element but the general case for other type of elements is similar.

Let u_x and u_y be approximated by the finite element interpolations,

$$u_x \approx \sum_{j=1}^{n} u_x^j \Psi_j(x, y), \quad u_y \approx \sum_{j=1}^{n} u_y^j \Psi_j(x, y) \tag{17}$$

or

$$\mathbf{u} = \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} = \Psi \Delta, \quad \mathbf{w} = \delta \mathbf{u} = \begin{Bmatrix} w_1 = \delta u_x \\ w_2 = \delta u_y \end{Bmatrix} = \Psi \delta \Delta \tag{18}$$

where,

$$\Psi = \begin{bmatrix} \Psi_1 & 0 & \Psi_2 & 0 & \Psi_3 & 0 & \Psi_4 & 0 \\ 0 & \Psi_1 & 0 & \Psi_2 & 0 & \Psi_3 & 0 & \Psi_4 \end{bmatrix}$$

(19)

$$\Delta = \begin{bmatrix} u_x^1 & u_y^1 & u_x^2 & u_y^2 & u_x^3 & u_y^3 & u_x^4 & u_y^4 \end{bmatrix}$$

Making the use of above approximations, the strains are

$$\epsilon = \mathbf{D}\mathbf{u} = \mathbf{D}\Psi\Delta = B\Delta \tag{20}$$

And the stresses are

$$\sigma = \mathbf{CB}\Delta \tag{21}$$

$$\mathbf{B} = \mathbf{D}\Psi = \begin{bmatrix} \frac{\partial \Psi_1}{\partial x} & 0 & \frac{\partial \Psi_2}{\partial x} & 0 & \frac{\partial \Psi_3}{\partial x} & 0 & \frac{\partial \Psi_4}{\partial x} & 0 \\ 0 & \frac{\partial \Psi_1}{\partial y} & 0 & \frac{\partial \Psi_2}{\partial y} & 0 & \frac{\partial \Psi_3}{\partial y} & 0 & \frac{\partial \Psi_4}{\partial y} \\ \frac{\partial \Psi_1}{\partial y} & \frac{\partial \Psi_1}{\partial x} & \frac{\partial \Psi_2}{\partial y} & \frac{\partial \Psi_2}{\partial x} & \frac{\partial \Psi_3}{\partial y} & \frac{\partial \Psi_3}{\partial x} & \frac{\partial \Psi_4}{\partial y} & \frac{\partial \Psi_4}{\partial x} \end{bmatrix} \tag{22}$$

Hence, the discretized governing equation in matrix form becomes,

$$[\mathbf{K}_e]\Delta = [\mathbf{F}_e] \tag{23}$$

where, $[\mathbf{K}_e]$ is the element stiffness matrix,

$$[\mathbf{K}_e] = \int_{\Omega_e} h_e \, \mathbf{B}^T \mathbf{E} \mathbf{B} \, d\Omega_e \tag{24}$$

and $[\mathbf{F}_e]$ is the node load vector which is a addition of body force and traction force,

$$[\mathbf{F}_e] = \int_{\Omega_e} \Psi^T \, \mathbf{f} \, d\Omega_e + \oint_{\Gamma_e} \Psi^T \, \mathbf{t}_s \, d\Gamma_e \tag{25}$$

5 Bilinear Quadrilateral Element

The 4 node bilinear Quadrilateral Element is used in the present work. The domain of a straight-edged quadrilateral element is defined by locations of its four nodal points $\mathbf{x}_a^e, a = 1,..,4$. The nodal points are assumed in ascending order corresponding to counterclockwise direction. To perform the numerical quadrature for computing the integral, it is useful to map this quadrilateral to a biunit square called master element. The coordinate of a point

$$\mathbf{x} = \begin{Bmatrix} x \\ y \end{Bmatrix} \tag{26}$$

in Ω_e are related to coordinates of a point

$$\boldsymbol{\xi} = \begin{Bmatrix} \xi \\ \eta \end{Bmatrix} \tag{27}$$

in the biunit square by mapping of the bilinear form:

$$x(\xi, \eta) = \alpha_0 + \alpha_1 \xi + \alpha_2 \eta + \alpha_3 \xi \eta \tag{28}$$

$$y(\xi, \eta) = \beta_0 + \beta_1 \xi + \beta_2 \eta + \beta_3 \xi \eta \tag{29}$$

where, α's and β's are found by solving the system of equation

$$[\mathbf{A}]\alpha_i = x_i \tag{30}$$

$$[\mathbf{A}]\beta_i = y_i \tag{31}$$

where,

$$[\mathbf{A}] = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \tag{32}$$

The jacobian of transformation can be found by,

$$j = det[\mathbf{J}], \quad \text{where } \mathbf{J} = \begin{bmatrix} x,_\xi & x,_\eta \\ y,_\xi & y,_\eta \end{bmatrix} \tag{33}$$

where $x,_\xi$, $x,_\eta$, $y,_\xi$, $y,_\eta$ are computed by differentiating equation (28) and (29) with respect to x and y. The inverse of the jacobian matrix is,

$$\begin{bmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{bmatrix} = \frac{1}{j} \begin{bmatrix} -y,_\eta & -x,_\eta \\ -y,_\xi & x,_\xi \end{bmatrix} \tag{34}$$

The derivative of the shape function in the master element can be found by using simple chain rule as,

$$\begin{aligned} \Psi_{a,x} &= \Psi_{a,\xi}\, \xi_{,x} + \Psi_{a,\eta}\, \eta_{,x} \\ \Psi_{a,y} &= \Psi_{a,\xi}\, \xi_{,y} + \Psi_{a,\eta}\, \eta_{,y} \end{aligned} \tag{35}$$

which is used while computation of element stiffness matrix.

6 Numerical Quadrature using Gaussian Quadrature

To compute the integrals in the stiffness matrix and node load vector, the integration must be calculated numerically. The gauss quadrature is the most common choice to do so which replaces the integration by summation of function evaluations at gauss quadrature points which are multiplied by weights.

In order to compute the integral, the function inside the integral must be transformed into the coordinates of the master element, i.e.

$$\begin{aligned} [\mathbf{K}_e] &= \int_{x_a}^{x_b} \int_{y_a}^{y_b} h_e\, \mathbf{B}^T \mathbf{E} \mathbf{B}\, dx\, dy \\ &= \int_{-1}^{1} \int_{-1}^{1} h_e \left(\mathbf{B}^T \mathbf{E} \mathbf{B}\right) \mathbf{J}\, d\xi\, d\eta \\ &= \sum_{i=1}^{p1} \sum_{j=1}^{p2} w_i w_j \ \left(\mathbf{B}^T(\xi_{i,j}, \eta_{i,j}) \mathbf{E} \mathbf{B}^T(\xi_{i,j}, \eta_{i,j})\right) \mathbf{J}\, d\xi\, d\eta \end{aligned} \tag{36}$$

In the present study, $p1 = p2 = 2$ is used and the quadrature points are $-\frac{1}{\sqrt{3}} and + \frac{1}{\sqrt{3}}$ and the both the weights are 1.

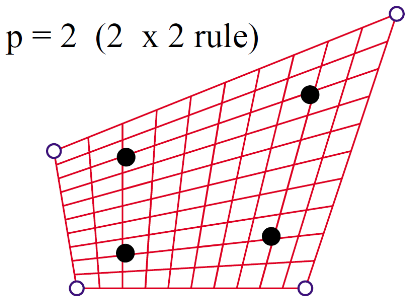

Figure 2: Quadrature points for 2 point Gaussian Quadrature

7 Numerical Results of Problems

A C++ program has been developed based on above details which computes the deflection solution parameters of the Finite element solution. The other post processing has not been included such as calculations of stresses and strains and is left as a future scope.

7.1 Problem 1

A deflection in square plate under point load is assessed. The problem description is shown in the figure (9). An essential boundary condition of zero displacement is applied on the left boundary and a point load of 1000 lbf in downward direction is applied. Plate Thickness is 0.1, modulus of Elasticity of 3e7 and Poisson ratio of 0.3.

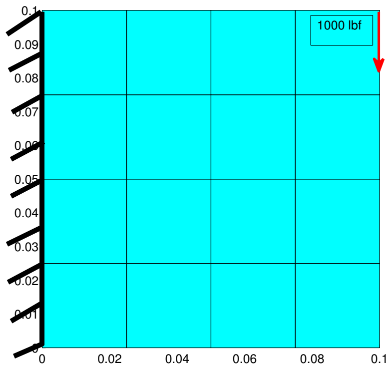

Figure 3: Problem description of plate under loading

The solution of the problem from the developed code is shown in the figure (??). Also the problem is solved using ANSYS' commercial package to validate the code.

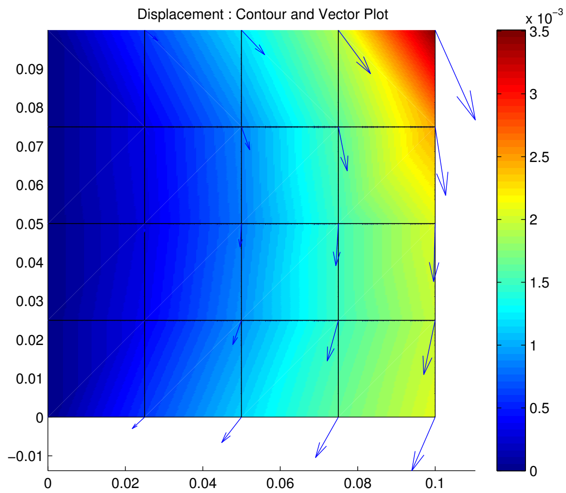

Figure 4: Solution of the problem using developed code

Comparison with ANSYS Package:

Table 1: Comparison of developed FEA code with ANSYS Package

Unknown	FEA Code	ANSYS Package and
Total Deflection	0.00350784	0.35078E-02
Maximum UX	0.00142976	0.14298E-02
Maximum UY	-0.00320324	-0.32032E-02

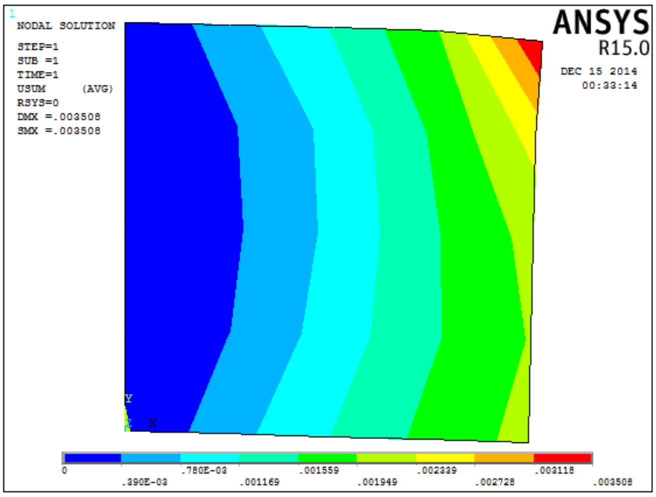

Figure 5: Solution using ANSYS package

7.2 Problem 2

Deflection of L shaped plate is under loading is analyzed in the present problem. The problem description is shown in the figure below. Plate Thickness is 0.1, modulus of Elasticity of 3e7 and Poisson ratio of 0.3.

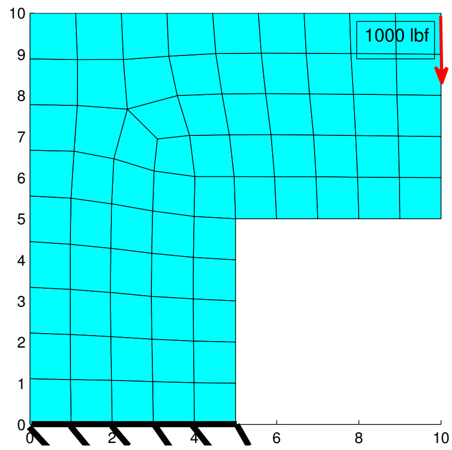

Figure 6: Problem description

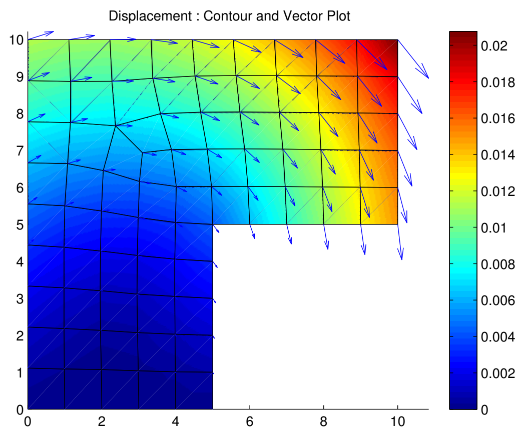

Figure 7: Solution of the problem using developed code

7.3 Problem 3

The deflection of a rectangular plate with circular hole is analyzed in the present problem.The problem description is shown in the figure below. Plate Thickness is 0.1, modulus of Elasticity of 3e7 and Poisson ratio of 0.3.

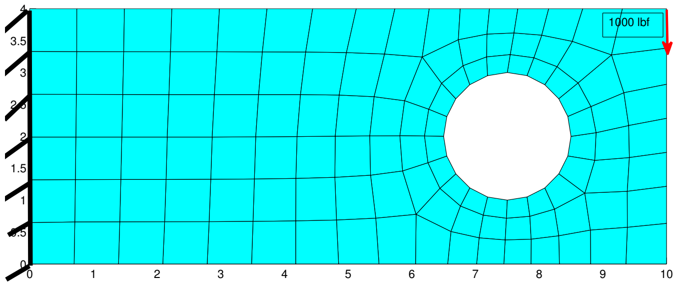

Figure 8: Problem description

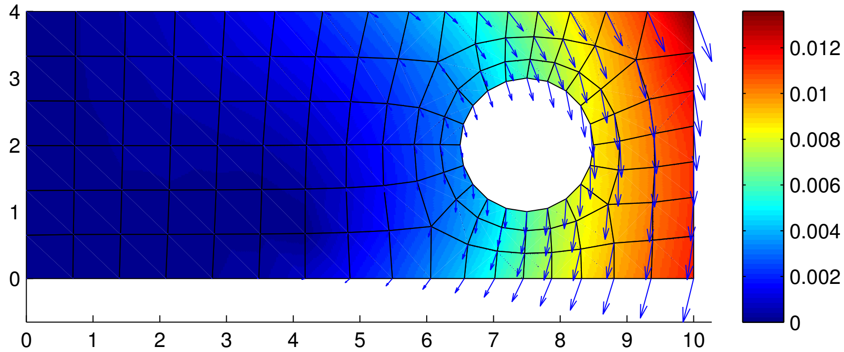

Figure 9: Solution of the problem using developed code

8 Conclusion

A program to compute plane stress using Finite Element Method is successfully developed and tested on different problems. The solutions obtained with the program matches perfectly with that of ANSYS package.

9 Future Scope

The code is written in Object Oriented C++ which takes some of the advantages of OOP. Currently only a bilinear quadrilateral element is implemented however other elements such as constant strain triangle or even higher order elements can be implemented. Also the higher quadrature rules can be added to achieve higher accuracy.

10 Appendix:

10.1 C++ Program

The code is written in C++ and consists of 9 files.

Description of the code:

1. main.cpp : includes main() which drives the program.
2. mesh.hpp : Class Mesh, which reads the mesh file
3. material.hpp : Class Material, defines the material properties
4. preprocessor.hpp : Class Pre_Processor, computes element stiffness matrix
5. solver.hpp : Class FEA_Solver, computes solution of the system of equations
6. element.hpp : Class Element, contains dereived class Quad4 which defines Bilinear Quad Element
7. quadrature.hpp : Class Quadrature, contains the quadrature information
8. stiffelement.hpp : Class Estiffness, calculates the element stiffness matrix
9. functions.h : some useful functions

main() : (main.cpp)

```
#include <iostream>
#include "mesh.hpp"
#include "material.hpp"
#include "preprocessor.hpp"
#include "solver.hpp"

using namespace std;

int main(int argc, char* argv[]){

  PetscInitialize(&argc,&argv,(char*)0,NULL);

  Mesh mesh("4x4Quad.dat");
  mesh.ReadMeshFile();
  mesh.Set_Thickness(0.1);
  mesh.ValidateMesh();
  //mesh.WriteMesh(MATLAB,"mesh");

  Material steel(3.0E+7,0.3);
  steel.Compute_Elastic_Stiffness();
  steel.Print_Elastic_Stiffness();

  PreProcessor pre(&mesh,&steel);
  pre.Set_quadrature_rule(Q2D_2point);
  pre.Create_Quadrature_Objects();

  pre.Compute_Element_properties();
  pre.Compute_Element_stiffness();
  pre.Assemble_Stiffness_Matrix();
  pre.set_pointload(-1000.0);
  pre.Apply_BC();

  FEA_Solver solver(&pre);
  solver.solve_disp();
  solver.write_sol_disp(CONTOUR,"disp_contour");
  solver.write_sol_disp(VECTOR,"disp_vector");

  cout << "Program Finished!" << endl;

  // call destructor to free PETSc objects before PetscFinalize()
  pre.~PreProcessor();
  solver.~FEA_Solver();
  PetscFinalize();

  return 0;
}
```

class Mesh : (mesh.hpp)

```
#ifndef MESH_HPP
#define MESH_HPP

#include <iostream>
#include <vector>
#include <string>
#include <fstream>
```

```cpp
#include <cassert>
#include <iomanip>

using namespace std;

typedef enum {MATLAB_MESH, MATLAB_POINTS} OUTPUT_FORMAT;

/*
 * CLASS NODE -> contains mesh coordinate details
 */
class Node{
  friend class Mesh;
  friend class PreProcessor;
  friend class Quad4;
private:
  int NodeID;
  double x,y,z;
};


/*
 * CLASS FACE -> contains mesh faces and its connected nodes
 */
class Face{
  friend class Mesh;
  friend class PreProcessor;
  friend class Quad4;
private:
  typedef enum {TRI, QUAD} FaceType;
  FaceType Ftype;
  int FaceID;
  vector<int> nodes;
};


/*
 * CLASS BOUNDARY -> identifies boundary nodes
 */
class Boundary{
  friend class Mesh;
  friend class PreProcessor;
private:
  typedef enum {NODE, ELEMENT} BoundaryType;

  string name;
  BoundaryType BType;
  vector<int> nodes;
};


/*
 * CLASS MESH -> Reads the mesh file and populates mesh data
 */
class Mesh{
  friend class PreProcessor;
private:
  vector<Node> node;
  vector<Face> face;
  vector<Boundary> boundary;
  bool set_filename;
  string filename;
  bool isQuadPresent, isTriPresent;
  double thickness;

public:

  Mesh();
  Mesh(string const&);
  void SetMeshFilename(string const&);
  void ReadMeshFile();
  void ValidateMesh();
  void WriteMesh(OUTPUT_FORMAT const&,string const&) const;
  void Set_Thickness(double const&);
  double Get_Thickness() const ;
};


/************* Function Definitions for Class Mesh *****************/

Mesh::Mesh(){
  set_filename = false;
  isQuadPresent = false;
  isTriPresent = false;
}

Mesh::Mesh(const string &a){
  SetMeshFilename(a);
  isQuadPresent = false;
  isTriPresent = false;
}

void Mesh::SetMeshFilename(const string &a){
  filename = a;
  set_filename = true;
  //cout << "Mesh Filename = " << filename << endl;
}


void Mesh::ReadMeshFile(){

  /* assert if input filename is set */
  assert(set_filename);

```

```cpp
  ifstream mfile(filename);
  assert(mfile.is_open());

  string parameter;

  while(!mfile.eof()){

    mfile >> parameter;
    //cout << parameter << endl;
    if(parameter.compare("#Nodes") == 0){
      for(;;){
        Node read_node;
        mfile >> read_node.NodeID;
        if(read_node.NodeID == -1){
          break;
        }
        mfile >> read_node.x >> read_node.y >> read_node.z;
        node.push_back(read_node);
        //cout << setw(12) << read_node.x << setw(12) << read_node.y << setw(12) << read_node.z << endl;
      }
    }

    if(parameter.compare("#Elements") == 0){

      for(;;){
        Face read_face;
        int check, temp;
        mfile >> check;
        if(check == -1){
          break;
        }
        mfile >> temp >> temp >> temp >> temp >> temp >> temp >> temp >> temp >> temp;
        mfile >> read_face.FaceID;
        mfile >> temp; read_face.nodes.push_back(temp);
        mfile >> temp; read_face.nodes.push_back(temp);
        mfile >> temp; read_face.nodes.push_back(temp);
        mfile >> temp; read_face.nodes.push_back(temp);

        assert(read_face.nodes[2] != read_face.nodes[3]);
        read_face.Ftype = Face::QUAD;
        if(read_face.Ftype == Face::QUAD && !isQuadPresent){
          isQuadPresent = true;
          cout << "Quad Face is present" << endl;
        }
        if(read_face.Ftype == Face::TRI && !isTriPresent){
          isTriPresent = true;
          cout << "Tri Face is present" << endl;
        }

        face.push_back(read_face);

        //cout << setw(12) << read_face.FaceID << setw(12) << read_face.nodes[0] << setw(12) << read_face.nodes[1]
        //     << setw(12) << read_face.nodes[2] << setw(12) << read_face.nodes[3] << endl;
      }
    }

    if(parameter.compare("#NamedSelection") == 0){

      double size;
      Boundary read_b;
      mfile >> read_b.name;
      string btype;
      mfile >> btype;

      if(btype.compare("NODE")==0){
        read_b.BType = Boundary::NODE;
      }else{
        cerr << "ERROR in Boundary "<< read_b.name << endl;
      }

      mfile >> size;
      for(int i = 0; i < size; i++){
        int buf;
        mfile >> buf;
        read_b.nodes.push_back(buf);
        //cout << read_b.nodes[i] << "\t";
      }
      //cout << endl;
      boundary.push_back(read_b);
    }

    if(parameter.compare("#End") == 0 || parameter.compare("#end") == 0){
      break;
    }

  } // end while
  mfile.close();
} // end mesh read function


void Mesh::ValidateMesh(){
  cout << "Number of nodes = " << node.size() << endl;
  cout << "Number of faces = " << face.size() << endl;
  cout << "Number of boundaries = " << boundary.size() << endl;
  cout << "size of nodes: " << sizeof(Node)*node.size() << " bytes" << endl;
  cout << "size of faces: " << sizeof(Face)*face.size() << " bytes" << endl;
  cout << endl;
}


void Mesh::WriteMesh(OUTPUT_FORMAT const& output_format, string const& name) const {

  if(output_format == MATLAB_MESH){
```

```

    ofstream mfile(name);
    assert(mfile.is_open());

    // write vertices
    mfile << "vertices = [" << endl;
    for(size_t i = 0; i < node.size(); i++){
      mfile << setw(12) << node[i].x << setw(12) << node[i].y << endl;
    }
    mfile << "];" << endl << endl;

    //write faces
    mfile << "faces = [" << endl;
    for(size_t f = 0; f < face.size(); f++){
      for(size_t n = 0; n < face[f].nodes.size(); n++){
        mfile << setw(12) << face[f].nodes[n];
      }
      mfile << endl;
    }
    mfile << "];" << endl << endl;

  }

  if(output_format == MATLAB_POINTS){

    ofstream mfile(name);
    assert(mfile.is_open());
    // write x component
    mfile << "x = [" << endl;
    for(size_t i = 0; i < node.size(); i++){
      mfile << node[i].x << endl;
    }
    mfile << "];" << endl << endl;
    // write y component
    mfile << "y = [" << endl;
    for(size_t i = 0; i < node.size(); i++){
      mfile << node[i].y << endl;
    }
    mfile << "];" << endl << endl;
    // write z component
//    mfile << "z = [" << endl;
//    for(size_t i = 0; i < node.size(); i++){
//      mfile << node[i].z << endl;
//    }
//    mfile << "];" << endl;
//    mfile.close();

  }


}  // end writemesh function


void Mesh :: Set_Thickness(double const& Thickness){
  thickness = Thickness;
}

double Mesh :: Get_Thickness() const {
  return thickness;
}


#endif // MESH_HPP
```

class Material : (material.hpp)

```
#ifndef MATERIAL_HPP
#define MATERIAL_HPP

#include <iostream>
#include <cassert>
#include <iomanip>

using namespace std;

class Material{
  friend class PreProcessor;
private:
  double E; // young's modulus
  double nu; // poisson's ratio

  double **Estiff, *Estiff_data;    // Element stiffness

public:

  Material();
  ~Material();
  Material(double const&,double const&);
  void set_YoungsModulus(double const&);
  void set_PoissonsRatio(double const&);
  void Compute_Elastic_Stiffness();
  void Print_Elastic_Stiffness();
  double** Get_Element_Stiffness() const {return Estiff;}
  void Allocate_Estiff();

};


/****************** Functions ************************/

Material :: Material(){
  E = 0.0;
```

```
  nu = 0.0;
  Allocate_Estiff();
}

Material :: Material(double const& YoungsMod, double const& PoissonRatio){
  set_YoungsModulus(YoungsMod);
  set_PoissonsRatio(PoissonRatio);
  Allocate_Estiff();
}

void Material :: Allocate_Estiff(){
  Estiff = new double* [3];
  Estiff_data = new double [9];
  for(int i = 0; i < 3; i++){
    Estiff[i] = &Estiff_data[i*3];
  }
}

void Material :: set_YoungsModulus(const double & YoungsMod){
  E = YoungsMod;
}

void Material :: set_PoissonsRatio(const double & PoissonRatio){
  nu = PoissonRatio;
}

void Material :: Compute_Elastic_Stiffness(){
  assert(E != 0 || nu != 0);

  /* Refer: Introduction to Finite Element Method, J. N. Reddy, pg. 609, Eq. 11.2.11  */
  Estiff[0][0] = E/(1-nu*nu);
  Estiff[0][1] = E*nu/(1-nu*nu);
  Estiff[0][2] = 0.0;
  Estiff[1][0] = E*nu/(1-nu*nu);
  Estiff[1][1] = E/(1-nu*nu);
  Estiff[1][2] = 0.0;
  Estiff[2][0] = 0.0;
  Estiff[2][1] = 0.0;
  Estiff[2][2] = E/(2*(1+nu));
}

void Material :: Print_Elastic_Stiffness(){
  cout << "Elastic Stiffness :"<< endl;
  cout << setw(15) << Estiff[0][0] << setw(15) << Estiff[0][1] << setw(15) << Estiff[0][2] << endl;
  cout << setw(15) << Estiff[1][0] << setw(15) << Estiff[1][1] << setw(15) << Estiff[1][2] << endl;
  cout << setw(15) << Estiff[2][0] << setw(15) << Estiff[2][1] << setw(15) << Estiff[2][2] << endl;
  cout << endl;
}



Material :: ~Material(){
  delete [] Estiff;
  delete [] Estiff_data;
}


#endif // MATERIAL_HPP
```

class Pre_Processor : (preprocessor.hpp)

```
#ifndef PREPROCESSOR_HPP
#define PREPROCESSOR_HPP

#include <iostream>
#include <vector>
#include "petscksp.h"
#include "mesh.hpp"
#include "material.hpp"
#include "element.hpp"
#include "quadrature.hpp"
#include "stiffelement.hpp"
#include "functions.h"

using namespace std;

class PreProcessor{
  friend class FEA_Solver;
private:
  const Mesh *mesh;
  const Material *material;
  Quadrature_Rule QRule;
  Quadrature *Quad_Quad, *Quad_Tri;
  vector<Element*> element;
  vector<EStiffness*> stiffness;
  Mat KMat;
  Vec RHS;
  double Point_Load;
  size_t GDof;

public:

  PreProcessor(Mesh const*, Material const*);
  ~PreProcessor();

  void Set_quadrature_rule(Quadrature_Rule const &);
  void Create_Quadrature_Objects();
  void Compute_Element_properties();
  void Compute_Element_stiffness();
  void Assemble_Stiffness_Matrix();
  void Apply_BC();
  void set_pointload(double);
```

```

};


/******************** functions **********************/

PreProcessor::PreProcessor(Mesh const* msh, Material const* matrl)
  :mesh(msh), material(matrl){
  GDof = 2*mesh->node.size();
  Quad_Quad = NULL;
  Quad_Tri = NULL;
}


void PreProcessor :: Set_quadrature_rule(const Quadrature_Rule &qrule){
  QRule = qrule;
}

PreProcessor :: ~PreProcessor(){
  if(Quad_Quad != NULL){
    delete Quad_Quad;
  }
  if(Quad_Tri != NULL){
    delete Quad_Tri;
  }
  for(size_t i = 0; i < element.size(); i++){
    delete element[i];
    delete stiffness[i];
  }
  VecDestroy(&RHS);
  MatDestroy(&KMat);
}


void PreProcessor :: Create_Quadrature_Objects(){
  if(QRule == Q2D_2point && mesh->isQuadPresent){
    Quad_Quad = new Quadrature_2PQuad4;
    Quad_Quad->Setup_Quadrature();
    Quad_Quad->Print_Quadrature_Info();
  }
  if(QRule == Q2D_3point && mesh->isQuadPresent){
    Quad_Quad = new Quadrature_3PQuad4;
    Quad_Quad->Setup_Quadrature();
    Quad_Quad->Print_Quadrature_Info();
  }
}



void PreProcessor :: Compute_Element_properties(){

  Element *elem;
  for(size_t i = 0; i < mesh->face.size(); i++){
    if(mesh->face[i].Ftype == Face::QUAD){
      elem = new Quad4(Quad_Quad,mesh->face[i]);
    }
    elem->Element_setup(mesh->node);
    element.push_back(elem);
  }

}



void PreProcessor :: Compute_Element_stiffness(){
  assert(mesh->Get_Thickness() != 0);
  EStiffness *estiff;
  for(size_t i = 0; i < mesh->face.size(); i++){
    estiff= new EStiffness(material,element[i]);
    estiff->Compute_Element_Stiffness(mesh->Get_Thickness());
    estiff->Compute_Equation_Number(mesh->face[i].nodes);
    stiffness.push_back(estiff);
  }
}



void PreProcessor :: Assemble_Stiffness_Matrix(){
  assert(stiffness.size() != 0);
  int GDof = 2*mesh->node.size();

  /* initialize K matrix */
  MatCreate(PETSC_COMM_WORLD,&KMat);
  MatSetSizes(KMat,PETSC_DECIDE,PETSC_DECIDE,GDof,GDof);
  MatSetFromOptions(KMat);
  MatSetUp(KMat);
  MatZeroEntries(KMat);
  MatAssemblyBegin(KMat,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(KMat,MAT_FINAL_ASSEMBLY);

  for(size_t e = 0; e < element.size(); e++){
    const int*  P = stiffness[e]->Get_P();
    double** K = stiffness[e]->Get_K();
    int K_size = stiffness[e]->Get_K_size();

    for(int z = 0; z < K_size; z++){
      MatSetValues(KMat,1,&P[z],K_size,P,K[z],ADD_VALUES);
    }
  }

  MatAssemblyBegin(KMat,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(KMat,MAT_FINAL_ASSEMBLY);

```

```
  //WriteMat(KMat,"KMat");

}


void PreProcessor :: Apply_BC(){

  VecCreate(PETSC_COMM_WORLD,&RHS);
  VecSetSizes(RHS,PETSC_DECIDE,GDof);
  VecSetFromOptions(RHS);
  VecSet(RHS,0.0);
  //VecDuplicate(RHS,&Solution);

  PetscReal *_RHS;
  VecGetArray(RHS,&_RHS);

  // apply point load bc
  for(size_t i = 0; i < mesh->boundary.size(); i++){
    if(mesh->boundary[i].name == "POINT_LOAD"){
      int BCnode = mesh->boundary[i].nodes[0];
      int BCP = (BCnode-1)*2 + 1;
      _RHS[BCP] = Point_Load;
    }
  }
  VecRestoreArray(RHS,&_RHS);

  // apply fixed (zero displacement bc)
  for(size_t i = 0; i < mesh->boundary.size(); i++){
    if(mesh->boundary[i].name == "FIXED"){

      int fixed_node[mesh->boundary[i].nodes.size()];
      for(size_t bc = 0; bc < mesh->boundary[i].nodes.size(); bc++){
        fixed_node[bc] = mesh->boundary[i].nodes[bc];
      }
      // find row number in global stiffness matrix to apply bc
      int rows[2*mesh->boundary[i].nodes.size()];
      for(size_t bc = 0; bc < mesh->boundary[i].nodes.size(); bc++){
        rows[bc*2] = (fixed_node[bc]-1)*2; // u displacement
        rows[bc*2+1] = rows[bc*2] + 1;     // v displacement
        //cout << fixed_node[i] << "\t" << rows[i*2] << "\t" << rows[i*2+1] << endl;
      }
      // make all entries zero and put 1 on diagonal
      MatZeroRows(KMat,2*mesh->boundary[i].nodes.size(),rows,1.0,NULL,NULL);

    }
  }

//  WriteMat(KMat,"KMat");
//  WriteVec(RHS,"RHS");

}


void PreProcessor :: set_pointload(double pl){
  Point_Load = pl;
}



#endif // PREPROCESSOR_HPP
```

class FEA_Solver : (solver.hpp)

```
#ifndef SOLVER_HPP
#define SOLVER_HPP

#include <iostream>
#include "preprocessor.hpp"

using namespace std;

typedef enum {VECTOR, CONTOUR} plot_type;

class FEA_Solver{

private:
  const PreProcessor* prep;
  Vec Solution;
  KSP ksp;
public:
  FEA_Solver(const PreProcessor* pre)
    : prep(pre)
  {
    VecDuplicate(prep->RHS,&Solution);
  }

  void solve_disp(double tol = 1e-12){

    int itn;
    KSPCreate(PETSC_COMM_WORLD,&ksp);
    KSPSetOperators(ksp,prep->KMat,prep->KMat);
    KSPSetTolerances(ksp,tol,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);
    KSPSetFromOptions(ksp);
    KSPSetUp(ksp);
    KSPSolve(ksp,prep->RHS,Solution);
    KSPGetIterationNumber(ksp,&itn);
    PetscPrintf(PETSC_COMM_WORLD,"Iterations taken by KSP: %d\n",itn);

  }

  void write_sol_disp(plot_type const& ptype, string const& name){

```

```
    if(ptype == CONTOUR){
      string fname = name + ".m";
      prep->mesh->WriteMesh(MATLAB_MESH,fname);
      ofstream disp_total(fname,ios::app);
      PetscReal *_sol;
      VecGetArray(Solution,&_sol);
      disp_total << "disp_t = [" << endl;
      for(size_t i = 0; i < prep->GDof; i+=2){
        disp_total << sqrt(pow(_sol[i],2)+pow(_sol[i+1],2)) << endl;
      }
      disp_total << "];" << endl << endl;
      disp_total << "patch('Faces',faces,'Vertices',vertices,'FaceVertexCData',disp_t,'FaceColor','interp')" << endl
    ;
      disp_total << "axis equal tight" << endl;
      disp_total << "colorbar;" << endl;
      disp_total.close();
      VecRestoreArray(Solution,&_sol);
    }

    if(ptype == VECTOR ){
      string fname = name + ".m";
      prep->mesh->WriteMesh(MATLAB_POINTS,fname);
      ofstream disp(fname,ios::app);
      PetscReal *_sol;
      VecGetArray(Solution,&_sol);
      disp << "u = [" << endl;
      for(size_t i = 0; i < prep->GDof; i+=2){
        disp << _sol[i] << endl;
      }
      disp << "];" << endl << endl;
      disp << "v = [" << endl;
      for(size_t i = 0; i < prep->GDof; i+=2){
        disp << _sol[i+1] << endl;
      }
      disp << "];" << endl << endl;
      disp << "quiver(x,y,u,v); axis equal tight;" << endl;
      disp.close();
      VecRestoreArray(Solution,&_sol);
    }

  }



  ~FEA_Solver(){
    VecDestroy(&Solution);
    KSPDestroy(&ksp);
  }

};

#endif // SOLVER_HPP
```

class Element : (element.hpp)

```
#ifndef ELEMENT_HPP
#define ELEMENT_HPP

#include <iostream>
#include <iomanip>
#include "quadrature.hpp"
#include "mesh.hpp"

// lapack routine : solves AX = B
extern "C" {
void dgesv_(int *n, int *nrhs,  double *a,  int  *lda,
            int *ipivot, double *b, int *ldb, int *info);
}

using namespace std;


class Element{
  friend class EStiffness;
protected:
  const Quadrature *Quad;          // quadrature info
  const Face& face;                // face associated with element
  double *alpha,*beta;             // mapping coeff to master element
  double **Na, *Na_data;           // shape function evaluated at quadrature points
  double *J;                       // jacobian evaluated at quadrature points
  double *dx_dxi, *dx_deta;        // derivative of x wrt xi and eta evaluated at quadrature points
  double *dy_dxi, *dy_deta;        // derivative of y wrt xi and eta evaluated at quadrature points
  double *dxi_dx, *dxi_dy;         // derivative of xi wrt x and y evaluated at quadrature points
  double *deta_dx, *deta_dy;       // derivative of eta wrt x and y evaluated at quadrature points
  double *dN1_dxi, *dN1_deta;
  double *dN2_dxi, *dN2_deta;
  double *dN3_dxi, *dN3_deta;
  double *dN4_dxi, *dN4_deta;

public:

  Element(Quadrature const* quad, const Face& f);
  virtual ~Element();

  virtual void Element_setup(vector<Node> const&) = 0;
  virtual void Compute_mapping_coeff(vector<Node> const&) = 0;
  virtual void Compute_Shape_Function() = 0;
  virtual void Compute_dX_dXI() = 0;
  virtual void Compute_Jacobian() = 0;
  virtual void Compute_dXI_dX() = 0;
  virtual void Compute_dN_dXI() = 0;

```

```
};



class Quad4 : public Element{
  friend class EStiffness;
protected:

public:
  Quad4(Quadrature const*, const Face&);
  ~Quad4();
  virtual void Element_setup(vector<Node> const&);
  virtual void Compute_mapping_coeff(vector<Node> const&);
  virtual void Compute_Shape_Function();
  virtual void Compute_dX_dXI();
  virtual void Compute_Jacobian();
  virtual void Compute_dXI_dX();
  virtual void Compute_dN_dXI();
};





// Functions

Element::Element(Quadrature const* quad, const Face& f)
  :Quad(quad), face(f)
{
  alpha = NULL;
  beta = NULL;
  Na = NULL;
  Na_data = NULL;
  J = NULL;
  dx_dxi = NULL;
  dx_deta = NULL;
  dy_dxi = NULL;
  dy_deta = NULL;
  dxi_dx = NULL;
  dxi_dy = NULL;
  deta_dx = NULL;
  deta_dy = NULL;
  dN1_dxi = NULL;
  dN1_deta = NULL;
  dN2_dxi = NULL;
  dN2_deta = NULL;
  dN3_dxi = NULL;
  dN3_deta = NULL;
  dN4_dxi = NULL;
  dN4_deta = NULL;
}


//free memory
Element :: ~Element(){
  if(alpha != NULL){
    delete [] alpha;
  }
  if(beta != NULL){
    delete [] beta;
  }
  if(Na != NULL){
    delete [] Na;
  }
  if(Na_data != NULL){
    delete [] Na_data;
  }
  if(dx_dxi != NULL){
    delete [] dx_dxi;
  }
  if(dx_deta != NULL){
    delete [] dx_deta;
  }
  if(dy_dxi != NULL){
    delete [] dy_dxi;
  }
  if(dy_deta != NULL){
    delete [] dy_deta;
  }
  if(dxi_dx != NULL){
    delete [] dxi_dx;
  }
  if(dxi_dy != NULL){
    delete [] dxi_dy;
  }
  if(deta_dx != NULL){
    delete [] deta_dx;
  }
  if(deta_dy != NULL){
    delete [] deta_dy;
  }
  if(J != NULL){
    delete [] J;
  }
  if(dN1_dxi != NULL){
    delete [] dN1_dxi;
  }
  if(dN1_deta != NULL){
    delete [] dN1_deta;
  }
  if(dN2_dxi != NULL){
    delete [] dN2_dxi;
  }
```

```
  if(dN2_deta != NULL){
    delete [] dN2_deta;
  }
  if(dN3_dxi != NULL){
    delete [] dN3_dxi;
  }
  if(dN3_deta != NULL){
    delete [] dN3_deta;
  }
  if(dN4_dxi != NULL){
    delete [] dN4_dxi;
  }
  if(dN4_deta != NULL){
    delete [] dN4_deta;
  }
}

Quad4::Quad4(Quadrature const* quad, const Face& f)
  :Element(quad,f)
{
  alpha = NULL;
  beta = NULL;
  Na = NULL;
  Na_data = NULL;
  J = NULL;
  dx_dxi = NULL;
  dx_deta = NULL;
  dy_dxi = NULL;
  dy_deta = NULL;
  dxi_dx = NULL;
  dxi_dy = NULL;
  deta_dx = NULL;
  deta_dy = NULL;
}


Quad4::~Quad4(){

}



void Quad4 :: Element_setup(vector<Node> const& node){
  assert(Quad != NULL);
  Compute_mapping_coeff(node);
  Compute_Shape_Function();
  Compute_dX_dXI();
  Compute_Jacobian();
  Compute_dXI_dX();
  Compute_dN_dXI();
}



void Quad4 :: Compute_mapping_coeff(vector<Node> const& node){

  int n = Quad->Qpoints(), nrhs = 2;
  int lda = n, ldb = n;
  int info, ipiv[n];
  double solution[2][n];
  double **Qmat = Quad->QMapping();
  double *mat = NULL; //Quad->QMapping();
  alpha = new double [n];
  beta = new double [n];

  mat = new double [n*n];
  memcpy(&mat[0],&Qmat[0][0],n*n*sizeof(double));

//  for(int i = 0; i < 4; i++){
//    for(int j = 0; j < 4; j++){
//      cout << setw(12) << mat[i*4+j];
//    }
//    cout << endl;
//  }

  // get x and y coordinates of face nodes
  for(int i = 0; i < n; i++){
    solution[0][i] = node[face.nodes[i]-1].x;
    solution[1][i] = node[face.nodes[i]-1].y;
  }

  /* solve AX = B using lapack solver */
  dgesv_(&n, &nrhs, &mat[0], &lda, ipiv, &solution[0][0], &ldb, &info);
  assert(info == 0);

  //cout << "Solution: "<< info << endl;
  //cout << setw(15) << "alpha" << setw(15) << "beta" << endl;
  for(int i = 0; i < n; i++){
    alpha[i] = solution[0][i];
    beta[i] = solution[1][i];
    //cout << setw(15) << alpha[i] << setw(15) << beta[i] << endl;
  }
  //cout << endl;

  delete [] mat;
}


void Quad4 :: Compute_Shape_Function(){

  const int n = Quad->Qpoints();
  const double* QXi  = Quad->QXipoints();
  const double* QEta = Quad->QEtapoints();
```

```
  Na = new double* [n];
  Na_data = new double [n*n];
  for(int i = 0; i < n; i++){
    Na[i] = &Na_data[i*n];
  }

  for(int i = 0; i < n; i++){
    //compute shape functions at quadrature points
    Na[0][i] = 0.25*(1.0-QXi[i])*(1.0-QEta[i]);
    Na[1][i] = 0.25*(1.0+QXi[i])*(1.0-QEta[i]);
    Na[2][i] = 0.25*(1.0+QXi[i])*(1.0+QEta[i]);
    Na[3][i] = 0.25*(1.0-QXi[i])*(1.0+QEta[i]);
  }


//  cout << "Shape Functions:" << endl;
//  for(int i = 0; i < n; i++){
//    for(int j = 0; j < n; j++){
//      cout << setw(12) << Na[i][j];
//    }
//    cout << endl;
//  }
//  cout << endl;

}


void Quad4 :: Compute_dX_dXI(){

  assert(alpha != NULL || beta != NULL);
  const int n = Quad->Qpoints();
  const double* QXi = Quad->QXipoints();
  const double* QEta = Quad->QEtapoints();
  dx_dxi = new double [n];
  dx_deta = new double [n];
  dy_dxi = new double [n];
  dy_deta = new double [n];

  //cout << setw(15) << "dx_dxi" << setw(15) << "dx_deta" << setw(15) << "dy_dxi" << setw(15) << "dy_deta" << endl;
  for(int i = 0; i < n; i++){
    dx_dxi[i] = alpha[1] + alpha[3]*QEta[i];
    dx_deta[i] = alpha[2] + alpha[3]*QXi[i];
    dy_dxi[i] = beta[1] + beta[3]*QEta[i];
    dy_deta[i] = beta[2] + beta[3]*QXi[i];
    //cout << setw(15) << dx_dxi[i] << setw(15) << dx_deta[i]
    //     << setw(15) << dy_dxi[i] << setw(15) << dy_deta[i] << endl;
  }
  //cout << endl;
}


void Quad4 :: Compute_Jacobian(){
  assert(dx_dxi != NULL || dx_deta != NULL ||dy_dxi != NULL ||dy_deta != NULL);
  const int n = Quad->Qpoints();
  J = new double [n];

  //cout << "Jacobian :" << endl;
  for(int i = 0; i < n; i++){
    J[i] = dx_dxi[i]*dy_deta[i] - dy_dxi[i]*dx_deta[i];
    //cout << setw(15) << J[i];
  }
  //cout << endl << endl;

  /* assert positive jacobian */
  assert(*J > 0);
}


void Quad4 :: Compute_dXI_dX(){
  assert(dx_dxi != NULL || dx_deta != NULL ||dy_dxi != NULL ||dy_deta != NULL);
  assert(*J < 1e8);
  const int n = 4;
  dxi_dx = new double [n];
  dxi_dy = new double [n];
  deta_dx = new double [n];
  deta_dy = new double [n];

  //cout << setw(15) << "dxi_dx" << setw(15) << "dxi_dy" << setw(15) << "deta_dx" << setw(15) << "deta_dy" << endl;
  for(int i = 0; i < n; i++){
    dxi_dx[i]  =  dy_deta[i]/J[i];
    dxi_dy[i]  = -dx_deta[i]/J[i];
    deta_dx[i] = -dy_dxi[i]/J[i];
    deta_dy[i] =  dx_dxi[i]/J[i];
    //cout << setw(15) << dxi_dx[i] << setw(15) << dxi_dy[i]
    //     << setw(15) << deta_dx[i] << setw(15) << deta_dy[i] << endl;
  }

}



void Quad4 :: Compute_dN_dXI(){
  const int n = Quad->Qpoints();
  const double* QXi = Quad->QXipoints();
  const double* QEta = Quad->QEtapoints();

  dN1_dxi = new double [n];
  dN1_deta = new double [n];
  dN2_dxi = new double [n];
  dN2_deta = new double [n];
  dN3_dxi = new double [n];
  dN3_deta = new double [n];
  dN4_dxi = new double [n];
```

```
  dN4_deta = new double [n];

  for(int i = 0; i < n; i++){
    dN1_dxi[i]  = -0.25*(1-QEta[i]);
    dN1_deta[i] = -0.25*(1-QXi[i]);
    dN2_dxi[i]  =  0.25*(1-QEta[i]);
    dN2_deta[i] = -0.25*(1+QXi[i]);
    dN3_dxi[i]  =  0.25*(1+QEta[i]);
    dN3_deta[i] =  0.25*(1+QXi[i]);
    dN4_dxi[i]  = -0.25*(1+QEta[i]);
    dN4_deta[i] =  0.25*(1-QXi[i]);
  }
}



#endif // ELEMENT_HPP
```

class Quadrature : (quadrature.hpp)

```
#ifndef QUADRATURE_HPP
#define QUADRATURE_HPP

#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

typedef enum {Q2D_2point, Q2D_3point} Quadrature_Rule;

class Quadrature{

protected:
  int Quadrature_points;            // number of quadrature points
  double *QW, *QXi, *QEta;          // quadrature weights and points
  double **mapping, *mapping_data;  // mapping matrix to compute mapping coeff for each element

public:
  virtual ~Quadrature();

  virtual void Setup_Quadrature() = 0;
  virtual void Print_Quadrature_Info() = 0;
  int Qpoints() const {return Quadrature_points;}
  double* QWeights() const {return QW;}
  double* QXipoints() const {return QXi;}
  double* QEtapoints() const {return QEta;}
  double** QMapping() const {return mapping;}

};


class Quadrature_2PQuad4 : public Quadrature{
protected:
public:
  virtual void Setup_Quadrature();
  virtual void Print_Quadrature_Info();
};


class Quadrature_3PQuad4 : public Quadrature{
public:
  virtual void Setup_Quadrature();
  virtual void Print_Quadrature_Info();
};



// functions


Quadrature :: ~Quadrature(){
  delete [] QW;
  delete [] QXi;
  delete [] QEta;
  delete [] mapping_data;
  delete [] mapping;
}


void Quadrature_2PQuad4 :: Setup_Quadrature(){
  Quadrature_points = 4;
  QW = new double [4];
  QXi = new double [4];
  QEta = new double [4];
  mapping_data = new double [16];
  mapping = new double* [4];
  for(int i = 0; i < 4; i++){
    mapping[i] = &mapping_data[i*4];
  }

  QW[0] = 1.0;
  QW[1] = 1.0;
  QW[2] = 1.0;
  QW[3] = 1.0;

  QXi[0] = -1.0/sqrt(3);
  QXi[1] =  1.0/sqrt(3);
  QXi[2] =  1.0/sqrt(3);
  QXi[3] = -1.0/sqrt(3);

  QEta[0] = -1.0/sqrt(3);
```

```
  QEta[1] = -1.0/sqrt(3);
  QEta[2] =  1.0/sqrt(3);
  QEta[3] =  1.0/sqrt(3);

  // transposed form because of lapack solver
  mapping[0][0] =  1.0;
  mapping[1][0] = -1.0;
  mapping[2][0] = -1.0;
  mapping[3][0] =  1.0;
  mapping[0][1] =  1.0;
  mapping[1][1] =  1.0;
  mapping[2][1] = -1.0;
  mapping[3][1] = -1.0;
  mapping[0][2] =  1.0;
  mapping[1][2] =  1.0;
  mapping[2][2] =  1.0;
  mapping[3][2] =  1.0;
  mapping[0][3] =  1.0;
  mapping[1][3] = -1.0;
  mapping[2][3] =  1.0;
  mapping[3][3] = -1.0;
}

void Quadrature_2PQuad4 :: Print_Quadrature_Info(){
  cout << "Quadrature points : " << Qpoints() << endl;
  cout << "Weights:" << endl;
  cout << setw(10) << QW[0] << setw(10) << QW[1] << setw(10) << QW[2] << setw(10) << QW[3] << endl;
  cout << "Xi:" << endl;
  cout << setw(10) << QXi[0] << setw(10) << QXi[1] << setw(10) << QXi[2] << setw(10) << QXi[3] << endl;
  cout << "Eta:" << endl;
  cout << setw(10) << QEta[0] << setw(10) << QEta[1] << setw(10) << QEta[2] << setw(10) << QEta[3] << endl;
  cout << "Mapping matrix(Transposed):" << endl;
  for(int i = 0; i < 4; i++){
    for(int j = 0; j < 4; j++){
      cout << setw(10) << mapping[i][j];
    }
    cout << endl;
  }
  cout << endl;
}



void Quadrature_3PQuad4 :: Setup_Quadrature(){
  Quadrature_points = 9;
  QW = new double [9];
  QXi = new double [9];
  QEta = new double [9];

  QW[0] = (5.0/9.0)*(5.0/9.0);
  QW[1] = (8.0/9.0)*(5.0/9.0);
  QW[2] = (5.0/9.0)*(5.0/9.0);
  QW[3] = (5.0/9.0)*(8.0/9.0);
  QW[4] = (8.0/9.0)*(8.0/9.0);
  QW[5] = (5.0/9.0)*(8.0/9.0);
  QW[6] = (5.0/9.0)*(5.0/9.0);
  QW[7] = (8.0/9.0)*(5.0/9.0);
  QW[8] = (5.0/9.0)*(5.0/9.0);

  QXi[0] = -sqrt(3.0/5.0);
  QXi[1] =  0.0;
  QXi[2] =  sqrt(3.0/5.0);
  QXi[3] = -sqrt(3.0/5.0);
  QXi[4] =  0.0;
  QXi[5] = -sqrt(3.0/5.0);
  QXi[6] =  sqrt(3.0/5.0);
  QXi[7] =  0.0;
  QXi[8] = -sqrt(3.0/5.0);

  QEta[0] = -sqrt(3.0/5.0);
  QEta[1] =  0.0;
  QEta[2] =  sqrt(3.0/5.0);
  QEta[3] = -sqrt(3.0/5.0);
  QEta[4] =  0.0;
  QEta[5] = -sqrt(3.0/5.0);
  QEta[6] =  sqrt(3.0/5.0);
  QEta[7] =  0.0;
  QEta[8] = -sqrt(3.0/5.0);
}

void Quadrature_3PQuad4 :: Print_Quadrature_Info(){
  cout << "Quadrature = 2D Gaussian Quadrature : 9 Points" <<endl;
  cout << "Quadrature points : " << Qpoints() << endl;
  cout << "Weights:" << endl;
  for(int i = 0; i < Qpoints(); i++)
    cout << setw(10) << QW[i];
  cout << endl;
  cout << "Xi:" << endl;
  for(int i = 0; i < Qpoints(); i++)
    cout << setw(10) << QXi[i];
  cout << endl;
  cout << "Eta:" << endl;
  for(int i = 0; i < Qpoints(); i++)
    cout << setw(10) << QEta[i];
  cout << endl;
  cout << endl;
}



#endif // QUADRATURE_HPP
```

class Estiffness : (stiffelement.hpp)

```
#ifndef STIFFELEMENT_HPP
#define STIFFELEMENT_HPP

#include <iostream>
#include "element.hpp"
#include "mesh.hpp"
#include "material.hpp"
#include "cblas.h"

using namespace std;


class EStiffness{
private:
  const Element* element;
  const Material* material;
  const size_t K_size;            // stiffness matrix dimension
  double **K, *K_data;            // element stiffness matrix
  int *P;                    // global equation number
public:
  EStiffness(const Material*,const Element*);
  ~EStiffness();
  void Compute_Element_Stiffness(double const&);
  void Compute_Equation_Number(const vector<int>&);
  int Get_K_size() const {return K_size;}
  int* Get_P() const {return P;}
  double** Get_K() const {return K;}

};



// Functions


EStiffness :: EStiffness(const Material* m,const Element* e)
  : element(e),material(m), K_size(2*e->Quad->Qpoints())
{
  assert(K_size != 0);
  P = new int [K_size];
  K = new double* [K_size];
  K_data = new double [K_size*K_size];
  for(size_t i = 0; i < K_size; i++){
    K[i] = &K_data[i*K_size];
  }
}


EStiffness :: ~EStiffness(){
  delete [] K_data;
  delete [] K;
  delete [] P;
}



void EStiffness :: Compute_Element_Stiffness(double const& thickness){
  double B[3][K_size];
  double** C = material->Get_Element_Stiffness();
  double* QW = element->Quad->QWeights();
  double result[8][3];
  double beta;

  assert(K_size == 8);
  for(size_t z = 0; z < K_size/2; z++){
    // compute B
    B[0][0] = element->dN1_dxi[z]*element->dxi_dx[z] + element->dN1_deta[z]*element->deta_dx[z];
    B[0][1] = 0.0;
    B[1][0] = 0.0;
    B[1][1] = element->dN1_dxi[z]*element->dxi_dy[z] + element->dN1_deta[z]*element->deta_dy[z];
    B[2][0] = B[1][1];
    B[2][1] = B[0][0];

    B[0][2] = element->dN2_dxi[z]*element->dxi_dx[z] + element->dN2_deta[z]*element->deta_dx[z];
    B[0][3] = 0.0;
    B[1][2] = 0.0;
    B[1][3] = element->dN2_dxi[z]*element->dxi_dy[z] + element->dN2_deta[z]*element->deta_dy[z];
    B[2][2] = B[1][3];
    B[2][3] = B[0][2];

    B[0][4] = element->dN3_dxi[z]*element->dxi_dx[z] + element->dN3_deta[z]*element->deta_dx[z];
    B[0][5] = 0.0;
    B[1][4] = 0.0;
    B[1][5] = element->dN3_dxi[z]*element->dxi_dy[z] + element->dN3_deta[z]*element->deta_dy[z];
    B[2][4] = B[1][5];
    B[2][5] = B[0][4];

    B[0][6] = element->dN4_dxi[z]*element->dxi_dx[z] + element->dN4_deta[z]*element->deta_dx[z];
    B[0][7] = 0.0;
    B[1][6] = 0.0;
    B[1][7] = element->dN4_dxi[z]*element->dxi_dy[z] + element->dN4_deta[z]*element->deta_dy[z];
    B[2][6] = B[1][7];
    B[2][7] = B[0][6];

//    for(int i = 0; i < 3; i++){
//      for(int j = 0; j < 8; j++){
//        cout << setw(12) << B[i][j];
//      }
//      cout << endl;
//    }
//    cout << endl;
```

```
    double alpha = element->J[z]*thickness*QW[z];
    // matrix multiplication --> result = B(transpose)*C*alpha
    cblas_dgemm(CblasRowMajor,CblasTrans,CblasNoTrans,K_size,3,3,alpha,&B[0][0],K_size,&C[0][0],3,0.0,&result
     [0][0],3);

//      for(int i = 0; i < 8; i++){
//        for(int j = 0; j < 3; j++){
//          cout << setw(15) << result[i][j];
//        }
//        cout << endl;
//      }

    if(z == 0){
      beta = 0.0;
    }else if(z > 0){
      beta = 1.0;
    }
    // matrix multiplication --> K = beta*K + result*B
    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,K_size,K_size,3,1.0,&result[0][0],3,&B[0][0],K_size,beta,&K
     [0][0],K_size);
  }

//  for(int i = 0; i < 8; i++){
//    for(int j = 0; j < 8; j++){
//      cout << setw(15) << K[i][j];
//    }
//    cout << endl;
//  }
//  cout << endl;

}


void EStiffness :: Compute_Equation_Number(const vector<int>& node){
  for(size_t i = 0; i < node.size(); i++){
    P[i*2] = (node[i]-1)*2;
    P[i*2+1] = P[i*2] + 1;
  }

//  cout << "Equation Number: " << endl;
//  for(size_t i = 0; i < K_size; i++){
//    cout << setw(12) << P[i];
//  }
//  cout << endl << endl;

}





#endif // STIFFELEMENT_HPP
```

functions : (functions.h)

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
#include <petsc.h>

PetscErrorCode WriteMat(Mat mat,char const *name){

    PetscViewer viewer;
    PetscErrorCode ierr;

    char filename[64] = "Mat_"; char pfix[12] = ".m";
    strcat(filename,name); strcat(filename,pfix);
    ierr = PetscObjectSetName((PetscObject)mat,name); CHKERRQ(ierr);
    ierr = PetscViewerASCIIOpen(PETSC_COMM_WORLD, filename, &viewer); CHKERRQ(ierr);
    ierr = PetscViewerSetFormat(viewer, PETSC_VIEWER_ASCII_MATLAB); CHKERRQ(ierr);
    ierr = MatView(mat,viewer);CHKERRQ(ierr);
    ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
    PetscFunctionReturn(0);
}


PetscErrorCode WriteVec(Vec vec, char const *name){
    PetscViewer viewer;
    PetscErrorCode ierr;

    char filename[64] = "Vec_"; char pfix[12] = ".m";
    strcat(filename,name); strcat(filename,pfix);
    ierr = PetscObjectSetName((PetscObject)vec,name); CHKERRQ(ierr);
    ierr = PetscViewerASCIIOpen(PETSC_COMM_WORLD, filename, &viewer); CHKERRQ(ierr);
    ierr = PetscViewerSetFormat(viewer, PETSC_VIEWER_ASCII_MATLAB); CHKERRQ(ierr);
    ierr = VecView(vec,viewer);CHKERRQ(ierr);
    ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
    PetscFunctionReturn(0);
}


#endif // FUNCTIONS_H
```

10.2 Sample Input Mesh file

The mesh file read in the program is written using ANSYS Workbench. The unnecessary information is deleted from the file and only required information is kept which is organized in the following way.

Sample Mesh File used for Program 2.

```
#Nodes
    1   0.000000000E+000   1.000000000E+001   0.000000000E+000
    2   0.000000000E+000   0.000000000E+000   0.000000000E+000
    3   0.000000000E+000   8.888888889E+000   0.000000000E+000
    4   0.000000000E+000   7.777777778E+000   0.000000000E+000
    5   0.000000000E+000   6.666666667E+000   0.000000000E+000
    6   0.000000000E+000   5.555555556E+000   0.000000000E+000
    7   0.000000000E+000   4.444444444E+000   0.000000000E+000
    8   0.000000000E+000   3.333333333E+000   0.000000000E+000
    9   0.000000000E+000   2.222222222E+000   0.000000000E+000
   10   0.000000000E+000   1.111111111E+000   0.000000000E+000
   11   5.000000000E+000   0.000000000E+000   0.000000000E+000
   12   1.000000000E+000   0.000000000E+000   0.000000000E+000
   13   2.000000000E+000   0.000000000E+000   0.000000000E+000
   14   3.000000000E+000   0.000000000E+000   0.000000000E+000
   15   4.000000000E+000   0.000000000E+000   0.000000000E+000
   16   5.000000000E+000   5.000000000E+000   0.000000000E+000
   17   5.000000000E+000   1.000000000E+000   0.000000000E+000
   18   5.000000000E+000   2.000000000E+000   0.000000000E+000
   19   5.000000000E+000   3.000000000E+000   0.000000000E+000
   20   5.000000000E+000   4.000000000E+000   0.000000000E+000
   21   1.000000000E+001   5.000000000E+000   0.000000000E+000
   22   6.000000000E+000   5.000000000E+000   0.000000000E+000
   23   7.000000000E+000   5.000000000E+000   0.000000000E+000
   24   8.000000000E+000   5.000000000E+000   0.000000000E+000
   25   9.000000000E+000   5.000000000E+000   0.000000000E+000
   26   1.000000000E+001   1.000000000E+001   0.000000000E+000
   27   1.000000000E+001   6.000000000E+000   0.000000000E+000
   28   1.000000000E+001   7.000000000E+000   0.000000000E+000
   29   1.000000000E+001   8.000000000E+000   0.000000000E+000
   30   1.000000000E+001   9.000000000E+000   0.000000000E+000
   31   8.888888889E+000   1.000000000E+001   0.000000000E+000
   32   7.777777778E+000   1.000000000E+001   0.000000000E+000
   33   6.666666667E+000   1.000000000E+001   0.000000000E+000
   34   5.555555556E+000   1.000000000E+001   0.000000000E+000
   35   4.444444444E+000   1.000000000E+001   0.000000000E+000
   36   3.333333333E+000   1.000000000E+001   0.000000000E+000
   37   2.222222222E+000   1.000000000E+001   0.000000000E+000
   38   1.111111111E+000   1.000000000E+001   0.000000000E+000
   39   3.100885795E+000   6.940821089E+000   0.000000000E+000
   40   3.883906237E+000   7.027860768E+000   0.000000000E+000
   41   3.017869238E+000   6.161626033E+000   0.000000000E+000
   42   4.850074589E+000   7.040770750E+000   0.000000000E+000
   43   5.864808320E+000   7.043474312E+000   0.000000000E+000
   44   6.897973722E+000   7.037079226E+000   0.000000000E+000
   45   4.014156390E+000   6.025411191E+000   0.000000000E+000
   46   2.984856810E+000   5.187710894E+000   0.000000000E+000
   47   2.963924128E+000   4.159491948E+000   0.000000000E+000
   48   2.954305198E+000   3.113634563E+000   0.000000000E+000
   49   2.966833201E+000   2.070582688E+000   0.000000000E+000
   50   1.954872141E+000   3.206756202E+000   0.000000000E+000
   51   1.964133590E+000   4.280955357E+000   0.000000000E+000
   52   1.987442885E+000   5.363330639E+000   0.000000000E+000
   53   2.037966141E+000   6.460608169E+000   0.000000000E+000
   54   2.369722555E+000   7.667980533E+000   0.000000000E+000
   55   3.585407332E+000   7.997051341E+000   0.000000000E+000
   56   4.672581651E+000   8.037841747E+000   0.000000000E+000
   57   5.733421173E+000   8.044844080E+000   0.000000000E+000
   58   6.801325531E+000   8.036846126E+000   0.000000000E+000
   59   7.868898928E+000   8.028595372E+000   0.000000000E+000
   60   7.925684523E+000   7.023684703E+000   0.000000000E+000
   61   1.963418650E+000   2.132912786E+000   0.000000000E+000
   62   4.968222423E+000   6.024805920E+000   0.000000000E+000
   63   5.950275027E+000   6.023627968E+000   0.000000000E+000
   64   6.960337601E+000   6.022422539E+000   0.000000000E+000
   65   7.971188476E+000   6.011697947E+000   0.000000000E+000
   66   8.927457869E+000   8.014515675E+000   0.000000000E+000
   67   8.961487394E+000   7.010908471E+000   0.000000000E+000
   68   2.983632029E+000   1.037919977E+000   0.000000000E+000
   69   1.984671847E+000   1.070964476E+000   0.000000000E+000
   70   9.864873052E−001   1.087871232E+000   0.000000000E+000
   71   9.783216106E−001   2.183930277E+000   0.000000000E+000
   72   9.782477164E−001   3.284728024E+000   0.000000000E+000
   73   9.781132470E−001   4.382433521E+000   0.000000000E+000
   74   9.974305756E−001   5.503140644E+000   0.000000000E+000
   75   1.072256462E+000   6.644678490E+000   0.000000000E+000
   76   1.150918303E+000   7.757754861E+000   0.000000000E+000
   77   1.153760333E+000   8.870140771E+000   0.000000000E+000
   78   2.272715412E+000   8.874785514E+000   0.000000000E+000
   79   3.378616377E+000   8.955660205E+000   0.000000000E+000
   80   4.521905029E+000   9.017703001E+000   0.000000000E+000
   81   5.627472506E+000   9.027073068E+000   0.000000000E+000
   82   6.720590452E+000   9.025172424E+000   0.000000000E+000
   83   7.808554241E+000   9.018864371E+000   0.000000000E+000
   84   8.908424832E+000   9.011734730E+000   0.000000000E+000
   85   8.986541286E+000   6.005180504E+000   0.000000000E+000
   86   3.993993695E+000   5.054644401E+000   0.000000000E+000
   87   3.975665129E+000   4.060554879E+000   0.000000000E+000
   88   3.974355784E+000   3.051025159E+000   0.000000000E+000
   89   3.980336212E+000   2.026422574E+000   0.000000000E+000
   90   3.990475831E+000   1.010320637E+000   0.000000000E+000
−1
```

```

#Elements
    1    1    1    1    0    0    0    0    4    0    1   39   54   53   41
    1    1    1    1    0    0    0    0    4    0    2   39   40   55   54
    1    1    1    1    0    0    0    0    4    0    3   41   45   40   39
    1    1    1    1    0    0    0    0    4    0    4   45   62   42   40
    1    1    1    1    0    0    0    0    4    0    5   41   46   86   45
    1    1    1    1    0    0    0    0    4    0    6   40   42   56   55
    1    1    1    1    0    0    0    0    4    0    7   53   52   46   41
    1    1    1    1    0    0    0    0    4    0    8   42   43   57   56
    1    1    1    1    0    0    0    0    4    0    9   42   62   63   43
    1    1    1    1    0    0    0    0    4    0   10   43   44   58   57
    1    1    1    1    0    0    0    0    4    0   11   43   63   64   44
    1    1    1    1    0    0    0    0    4    0   12   44   60   59   58
    1    1    1    1    0    0    0    0    4    0   13   44   64   65   60
    1    1    1    1    0    0    0    0    4    0   14   86   16   62   45
    1    1    1    1    0    0    0    0    4    0   15   52   51   47   46
    1    1    1    1    0    0    0    0    4    0   16   47   87   86   46
    1    1    1    1    0    0    0    0    4    0   17   51   50   48   47
    1    1    1    1    0    0    0    0    4    0   18   48   88   87   47
    1    1    1    1    0    0    0    0    4    0   19   50   61   49   48
    1    1    1    1    0    0    0    0    4    0   20   49   89   88   48
    1    1    1    1    0    0    0    0    4    0   21   49   68   90   89
    1    1    1    1    0    0    0    0    4    0   22   61   69   68   49
    1    1    1    1    0    0    0    0    4    0   23   50   72   71   61
    1    1    1    1    0    0    0    0    4    0   24   51   73   72   50
    1    1    1    1    0    0    0    0    4    0   25   52   74   73   51
    1    1    1    1    0    0    0    0    4    0   26   53   75   74   52
    1    1    1    1    0    0    0    0    4    0   27   54   76   75   53
    1    1    1    1    0    0    0    0    4    0   28   54   78   77   76
    1    1    1    1    0    0    0    0    4    0   29   55   79   78   54
    1    1    1    1    0    0    0    0    4    0   30   56   80   79   55
    1    1    1    1    0    0    0    0    4    0   31   57   81   80   56
    1    1    1    1    0    0    0    0    4    0   32   58   82   81   57
    1    1    1    1    0    0    0    0    4    0   33   59   83   82   58
    1    1    1    1    0    0    0    0    4    0   34   59   66   84   83
    1    1    1    1    0    0    0    0    4    0   35   60   67   66   59
    1    1    1    1    0    0    0    0    4    0   36   65   85   67   60
    1    1    1    1    0    0    0    0    4    0   37   71   70   69   61
    1    1    1    1    0    0    0    0    4    0   38   62   16   22   63
    1    1    1    1    0    0    0    0    4    0   39   63   22   23   64
    1    1    1    1    0    0    0    0    4    0   40   64   23   24   65
    1    1    1    1    0    0    0    0    4    0   41   65   24   25   85
    1    1    1    1    0    0    0    0    4    0   42   66   29   30   84
    1    1    1    1    0    0    0    0    4    0   43   67   28   29   66
    1    1    1    1    0    0    0    0    4    0   44   85   27   28   67
    1    1    1    1    0    0    0    0    4    0   45   68   14   15   90
    1    1    1    1    0    0    0    0    4    0   46   69   13   14   68
    1    1    1    1    0    0    0    0    4    0   47   70   12   13   69
    1    1    1    1    0    0    0    0    4    0   48   70   10    2   12
    1    1    1    1    0    0    0    0    4    0   49   71    9   10   70
    1    1    1    1    0    0    0    0    4    0   50   72    8    9   71
    1    1    1    1    0    0    0    0    4    0   51   73    7    8   72
    1    1    1    1    0    0    0    0    4    0   52   74    6    7   73
    1    1    1    1    0    0    0    0    4    0   53   75    5    6   74
    1    1    1    1    0    0    0    0    4    0   54   76    4    5   75
    1    1    1    1    0    0    0    0    4    0   55   77    3    4   76
    1    1    1    1    0    0    0    0    4    0   56   77   38    1    3
    1    1    1    1    0    0    0    0    4    0   57   78   37   38   77
    1    1    1    1    0    0    0    0    4    0   58   79   36   37   78
    1    1    1    1    0    0    0    0    4    0   59   80   35   36   79
    1    1    1    1    0    0    0    0    4    0   60   81   34   35   80
    1    1    1    1    0    0    0    0    4    0   61   82   33   34   81
    1    1    1    1    0    0    0    0    4    0   62   83   32   33   82
    1    1    1    1    0    0    0    0    4    0   63   84   31   32   83
    1    1    1    1    0    0    0    0    4    0   64   84   30   26   31
    1    1    1    1    0    0    0    0    4    0   65   85   25   21   27
    1    1    1    1    0    0    0    0    4    0   66   87   20   16   86
    1    1    1    1    0    0    0    0    4    0   67   88   19   20   87
    1    1    1    1    0    0    0    0    4    0   68   89   18   19   88
    1    1    1    1    0    0    0    0    4    0   69   90   17   18   89
    1    1    1    1    0    0    0    0    4    0   70   90   15   11   17
-1

#NamedSelection
FIXED    NODE         6
        2       11       12       13       14       15

#NamedSelection
POINT_LOAD    NODE         1
       26

#End
```