Concordia University

Mechanical forming of metals

MECH 6511

Implementation Project

Student Name:
David Libera

Student ID:
26635075

April 19th 2017

Introduction

This report will outline the theory behind the finite element method (FEM) and describe its implementation into a MFC application using Visual C++. Finally, an analysis of various stress problems will validate the implementation and identify future works.

Theory

The finite element method is a numerical method used to solve elasticity problems over complicated domains and boundary conditions. The body is divided into sub-domains known as elements. The solution is quantified in terms of nodal quantities which are solved as a linear system of equations.

Formulation of isotropic elasticity problem

An isotropic elasticity problem is formulated as a partial differential equation with the appropriate boundary conditions.

$$\mathbf{div}(\sigma) + \vec{f_b} = 0$$

There exists two assumptions that simplify stress problems, *plane strain assumption* and *plane stress assumption*. In this implementation, the problem is assumed to be under plane stress because the thickness of the geometry is considered much smaller than the other two dimensions (width and length). Consequently, the problem is two-dimensional. It is assumed the variation of stress and strain throughout the thickness is negligible.

Our reduced constitutive relations are,

$$\vec{\epsilon} = \mathbf{C}\vec{\sigma} = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_{xy} \end{bmatrix} = \frac{1}{E}\begin{bmatrix} 1 & -v & 0 \\ -v & 1 & 0 \\ 0 & 0 & 2(1+v) \end{bmatrix}\begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} \tag{1}$$

Consequently, stress can be solved for by inverting matrix B and obtaining matrix D.

$$\mathbf{D} = \mathbf{C^{-1}} = \frac{E}{1-v^2}\begin{bmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & 1-v \end{bmatrix} \tag{2}$$

where E is the Young's modulus and v is poisson's ratio.

Numerical Method

The T3 element was used in this implementation (Boeraeve,2010). The triangular element consists of 3 nodes wher the ith node has 2 degrees of freedom, horizontal displacement u_i and vertical displacment, v_i. In total, T3 element is fully defined by 6 displacements.The displacements are approximated in terms of shape functions and these nodal quantities as shown below. These elements have constant stress and strain on their domain.

$$u(x,y) = N_1(x,y)u_1 + N_2(x,y)u_2 + N_3(x,y)u_3 \tag{3}$$
$$v(x,y) = N_1(x,y)v_1 + N_2(x,y)v_2 + N_3(x,y)v_3 \tag{4}$$

where,

$$N_1(x,y) = \frac{a_1 + b_1x + c_1y}{2A} \qquad N_2(x,y) = \frac{a_2 + b_2x + c_2y}{2A} \qquad N_3(x,y) = \frac{a_3 + b_3x + c_3y}{2A}$$

In matrix form,

$$\vec{u} = \mathbf{N}\vec{d} = \begin{bmatrix} u(x,y) \\ v(x,y) \end{bmatrix} = \begin{bmatrix} N1 & 0 & N2 & 0 & N3 & 0 \\ 0 & N1 & 0 & N2 & 0 & N3 \end{bmatrix}\begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{bmatrix} \tag{5}$$

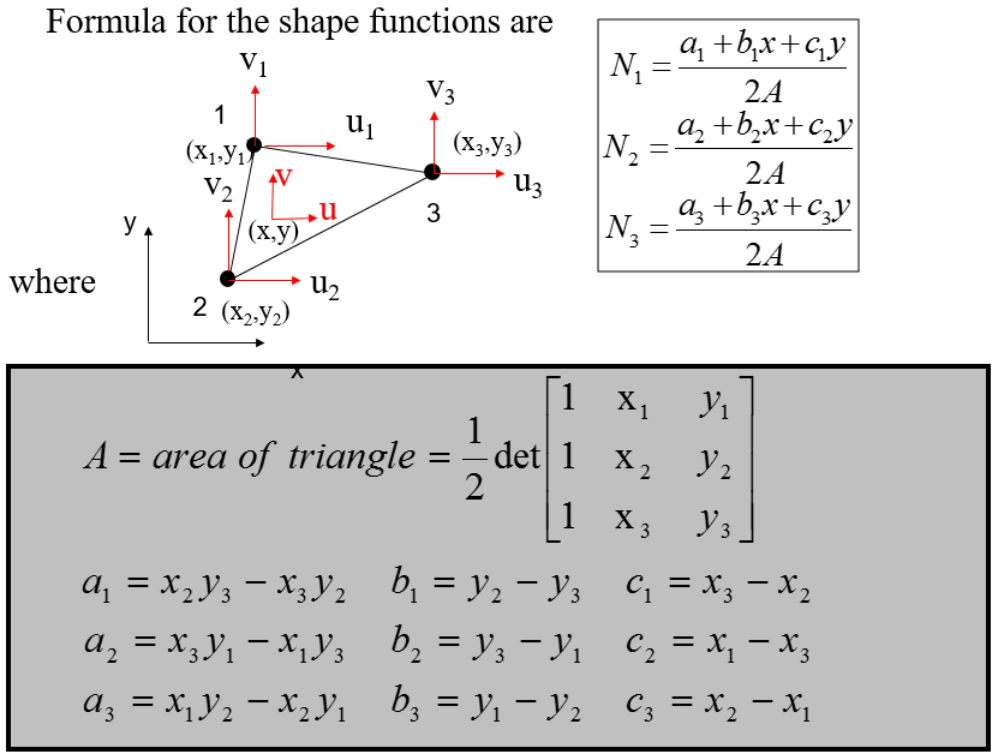


Figure 1: T3-Element (De,P)

Based on (De,P) and as shown in figure 1 variables, the relation between ϵ and d can be expressed using matrix **B** as shown below.

$$\vec{\epsilon} = \mathbf{B}\vec{d} = \begin{bmatrix} N1_{,x} & 0 & N2_{,x} & 0 & N3_{,x} & 0 \\ 0 & N1_{,y} & 0 & N2_{,y} & 0 & N3_{,y} \\ N1_{,y} & N1_{,x} & N2_{,y} & N2_{,x} & N3_{,y} & N3_{,x} \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{bmatrix} \tag{6}$$

$$\mathbf{B} = \frac{1}{2A} \begin{bmatrix} b1 & 0 & b2 & 0 & b3 & 0 \\ 0 & c1 & 0 & c2 & 0 & c3 \\ c1 & b1 & c2 & b2 & c3 & b3 \end{bmatrix} \tag{7}$$

Finally, stresses can be computed (De,P),

$$\vec{\sigma} = \mathbf{DB}\vec{d} \tag{8}$$

Numerical methods basics

The finite-element method is the solution of a system of linear equation of the form,

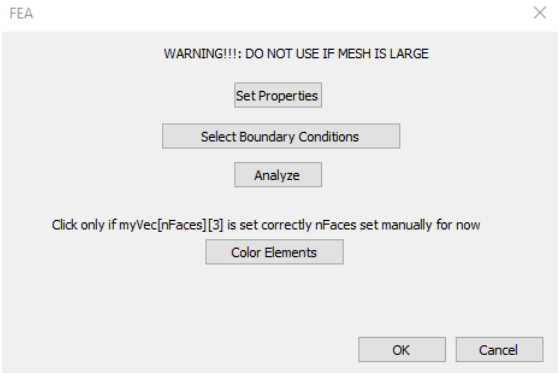
$$\mathbf{K}\vec{d} = \vec{f}$$

The global matrix K is not invertible and therefore the system of equations must be modified by eliminating certain rows and columns corresponding to zero boundary conditions. The remaining displacements are called *free displacements*. A solver is used to solve the following system of equations.

$$\mathbf{K_{mod}}\vec{d_{mod}} = \vec{f_{mod}}$$

Software implementation

The stress analysis functionality is added to existing MFC application using Windows programming. To run successfully, one must first have a mesh uploaded before going into "Mesh Opertion > FEA" from the main menu. Upon clicking FEA, the following dialog box will appear. Upon appearing, class FEM is created in memory, meaning its constructor is called as shown in listing below and all memory required is created using the *FEM::Create()* function.



Several things should be noted. First, the dialog box is not fully functional. Further work is required to allow the user to click on "Set Properties" to set values to E, v and thickness t. Similarly, the user will have the ability to select boundary conditions using the "Select Boundary Conditions" and clicking on the nodes of interest. These button controls do nothing at the moment and all properties and boundary conditions are hard-coded. The only functional button "Analyze" invokes *FEM::MainFunction()* which runs the finite-element method. This function will be described in great detail.

Activating a function using a button

Briefly, it can be seen that the dialog box creates an instance of the class FEM in dynamic memory which is used to invoke functions. It should be noted that in order to unallocate memory for class FEM the dialog box must be closed to invoke the class destructor. This is a short coming of the implementation. Future work is required to ensure that if the user exits the application without closing the dialog box, the memory is still unallocated properly.

Listing 1: CFEADlg's constructor

```
1 CFEADlg::CFEADlg() : CDialog(CFEADlg::IDD)
2 {
3     std::cout << "CONSTRUCTOR: Dyn Alloc Space for class FEM" << std::endl;
4     pFEM = new FEM();
5 }
```

Listing 2: CFEADlg's functions

```
1 void CFEADlg::OnBnClickedAnalyze()
2 {
3     pFEM->MainFunction();
4 }
5
6 void CFEADlg::OnBnClickedButton4()
7 {
8     pFEM->colorFaces();
9 }
```

FEM Methodology

The finite-element method is described along with the supporting code starting with the **MainFunction**, which is run when the user clicks "Analyze" in the dialog box. Post-processing is omitted in listing 3.

Outline of MainFunction

1. Set the values for E and v using **setProps**
2. Extract node i's x and y coordinates into columns of **matNodes** using **setNodeMatrix**
3. Extract face j's nodes into columns of **matConn** using **setConnMatrix**
4. Set **D** matrix according to equation 2 using **setDMatrix**, used later in computing stresses from strains
5. Function **computeKMatrix** computes the elemental stiffness matrix and "scatters" each entry into the global stiffness matrix **K**
6. Function **setBCs** contains the data which modifies the global matrix such that is is invertible using matrix **Isol**
7. Based on the information in **Isol** certain rows and columns are eliminated from the system of equations. Functions **scatterKmod** and **scatterfmod** are used to populate a matrix with the appropriate modified size.
8. Finally, **GaussSeidelSolver** from class **GLKMatrixLib** is used to solve the system of equations
9. Function **scatterBackDisplacements** rearranged the results into the original vector
10. Function **computeStress** uses displacement results to solve for stresses in each element
11. Post-processing using **ColorFaces** based on values in array **vonMisVec**

Listing 3: Main Function

```
void FEM::MainFunction() {
    // Do computations of matrix
    setProps();
    setNodeMatrix();
    setConnMatrix();
    setDMatrix();
    //Compute K matrix
    computeKMatrix();
    //Set BCS
    setBCs();
    //Modify System of Equations based on BCs
    scatterKmod(K, Kmod);
    scatterfmod(f, fmod);
    //Solve system of equations
    GLKMatrixLib::GaussSeidelSolver(Kmod,fmod, vars, dmod,1e-6);
    scatterBackDisplacements(dmod, d);
    //Compute Stresses
    computeStress();
    // Post Processing {...}
    std::cout << "END OF FEM" << std::endl;
}
```

Functions

Setting properties

Based on the values set in setProps(), equation 2 shows how matrix D values are computed. Special attention must be paid to units.

Listing 4: setProps()

```
void FEM::setProps() {
    t = 0.5; // thickness
    E = 30e6; // modulus
    v = 0.25; // poissons
}
```

Listing 5: setDMatrix()

```
void FEM::setDMatrix() {
    // stress-strain matrix
    D[0][0] = 1; D[0][1] = v; D[0][2] = 0;
    D[1][0] = v; D[1][1] = 1; D[1][2] = 0;
    D[2][0] = 0; D[2][1] = 0; D[2][2] = (1-v)/2;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            D[i][j] = E / (1 - v*v)*D[i][j];
        }
    }
}
```

Setting matNodes and connMatrix

This section was based on (Chessa, 2002). Figure 2 illustrated the matrices containing information on the mesh geometry used for the computation of matrix B of equation 7. Note, in the implementation, matrix nodes is called **matNodes** and matrix elements is called **connNodes**.

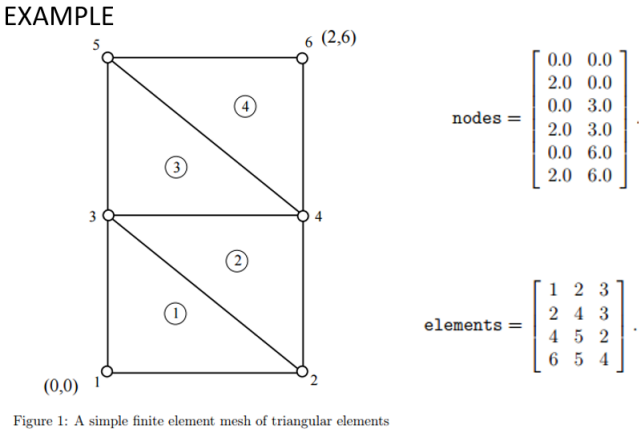


Figure 1: A simple finite element mesh of triangular elements

Figure 2: Setting matrices describing mesh geometry (Chessa,2002)

Listing 6: setNodeMatrix()

```
void FEM::setNodeMatrix() {


    CMainFrame *pWnd = (CMainFrame *)(AfxGetMainWnd());
    CMeshWorksDoc *pDoc = (CMeshWorksDoc *)(pWnd->GetActiveDocument());
    CGLKernelView *cView = pWnd->GetMainView()->GetGLKernelView();

    QBody * body = (QBody*)(pDoc->m_meshList).GetHead();
    QMeshNode* pNode = (QMeshNode*)body->GetTrglNodeList().GetHead();

    int c = 0; // counter
    for (GLKPOSITION Pos = body->GetTrglNodeList().GetHeadPosition(); Pos != NULL; ) {
        pNode = (QMeshNode*)body->GetTrglNodeList().GetNext(Pos);
        double xx, yy, zz;
        pNode->GetCoord3D(xx, yy, zz);
        matNodes[c][0] = xx;
        matNodes[c][1] = yy;

        c = c + 1;
    }
}
```

Listing 7: setConnMatrix()

```

void FEM::setConnMatrix() {

    CMainFrame *pWnd = (CMainFrame *)(AfxGetMainWnd());
    CMeshWorksDoc *pDoc = (CMeshWorksDoc *)(pWnd->GetActiveDocument());
    CGLKernelView *cView = pWnd->GetMainView()->GetGLKernelView();

    QBody * body = (QBody*)(pDoc->m_meshList).GetHead();
    QMeshFace* pFace = (QMeshFace*)body->GetTrglFaceList().GetHead();

    int c = 0; // counter
    for (GLKPOSITION Pos = body->GetTrglFaceList().GetHeadPosition(); Pos != NULL; ) {
        pFace = (QMeshFace*)body->GetTrglFaceList().GetNext(Pos);
        matConn[c][0] = (pFace->GetNodeRecordPtr(1))->GetIndexNo();
        matConn[c][1] = (pFace->GetNodeRecordPtr(2))->GetIndexNo();
        matConn[c][2] = (pFace->GetNodeRecordPtr(3))->GetIndexNo();
        c = c + 1;
    }
}
```

Computing and assembling global stiffness matrix

The advantages of storing the geometry information in matNodes and matConn will be apparent in the construction of the global stiffness matrix.

As shown in equation 7, the matrix B must be computed individually for each element. Listing 8 lines 9 to 26 show geometrical data extracted and used to create B. This matrix in turn is used to compute the elemental stiffness matrix using equation 9. This computation is done in lines 28 to 35.

$$\mathbf{K_e} = At\mathbf{B^T}\mathbf{DB} \tag{9}$$

Finally, once K_e is computed for each element the values are scattered into the much larger global stiffness matrix as shown in line 37.

Listing 8: computeKMatrix()

```
1  void FEM::computeKMatrix() {
2      // Declaring local variables
3      int n1, n2, n3;
4      double x1, y1, z1, x2, y2, z2, x3, y3, z3;
5      double b1, b2, b3, c1, c2, c3, A;
6      // Iterating through each face/element
7      for (int element = 0; element < nFaces; element++) {
8          // Extracting node number from matConn matrix
9          n1 = matConn[element][0];
10         n2 = matConn[element][1];
11         n3 = matConn[element][2];
12         // Using node number from above to access x,y coordinates
13         // (Note: index starts is 1 less for matrices, hence the -1)
14         x1 = matNodes[n1 - 1][0]; y1 = matNodes[n1 - 1][1];
15         x2 = matNodes[n2 - 1][0]; y2 = matNodes[n2 - 1][1];
16         x3 = matNodes[n3 - 1][0]; y3 = matNodes[n3 - 1][1];
17         // Computing area of element
18         A = 0.5*((x2*y3 - x3*y2) + (x3*y1 - x1*y3) + (x1*y2 - x2*y1));
19         // Computing elements of matrix B
20         b1 = y2 - y3; b2 = y3 - y1; b3 = y1 - y2;
21         c1 = x3 - x2; c2 = x1 - x3; c3 = x2 - x1;
22         // Setting matrix B
23         B[0][0] = b1 / (2 * A); B[0][2] = b2 / (2 * A); B[0][4] = b3 / (2 * A);
24         B[1][1] = c1 / (2 * A); B[1][3] = c2 / (2 * A); B[1][5] = c3 / (2 * A);
25         B[2][1] = b1 / (2 * A); B[2][3] = b2 / (2 * A); B[2][5] = b3 / (2 * A);
26         B[2][0] = c1 / (2 * A); B[2][2] = c2 / (2 * A); B[2][4] = c3 / (2 * A);
27         // Performing Ke computation (Ke = A*t*BT*D*B )
28         GLKMatrixLib::Transpose(B, Brow, Bcol, BT);
29         GLKMatrixLib::Mul(BT, D, 6, 3, 3, temp);
30         GLKMatrixLib::Mul(temp, B, 6, 3, 6, Ke);
31         for (int i = 0; i < Kerow; i++) {
32             for (int j = 0; j < Kecol; j++) {
33                 Ke[i][j] = t*A*Ke[i][j];
34             }
35         }
36         // scattering results into global matrix
37         scatter(Ke, K, n1, n2, n3);
38
39     }
40 }
```

By storing the index of each node in matConn. The node index for each vertex of each element can be scattered into the appropriate row and column of the global stiffness matrix as shown in listing 9.

Listing 9: scatter()

```
void FEM::scatter(double** &Ke, double** &K, int n1, int n2, int n3) {
    int row = 0, col =0;
    for (int i = 0; i < Kerow; i++) {
        switch (i) {
        case 0: row = 2 * n1 - 2; break;
        case 1: row = 2 * n1 - 1; break;
        case 2: row = 2 * n2 - 2; break;
        case 3: row = 2 * n2 - 1; break;
        case 4: row = 2 * n3 - 2; break;
        case 5: row = 2 * n3 - 1; break;
        }
        for (int j = 0; j < Kecol; j++) {
            switch (j) {
            case 0: col = 2 * n1 - 2; break;
            case 1: col = 2 * n1 - 1; break;
            case 2: col = 2 * n2 - 2; break;
            case 3: col = 2 * n2 - 1; break;
            case 4: col = 2 * n3 - 2; break;
            case 5: col = 2 * n3 - 1; break;
            }
            K[row][col] = K[row][col] + Ke[i][j];
        }
    }
}
```

Setting Boundary Conditions

Boundary conditions could be written and commented out in order to use the appropriate node index for the corresponding mesh. As shown in listing 10, four cases can be used. However, only cases 1-3 will be presented in this report. The objective of setBCs() is to populate matrix **Isol** which contains the indices corresponding to the degree of freedom (u,v) to be excluded from the system of equations. This function also sets the values of the force in the appropriate index of array **f**. **The number of degrees of freedom that are fixed will have an effect on the size of the Isol and Kmod, matrices among others. This size called vars in FEM.h is declared in the constructor, it must be modified properly BEFORE running the program**.

Listing 10: setBCs()

```
void FEM::setBCs() {
    // CASE 1: FOR SQUARE MESH {...}

    // CASE 2: FOR PLATE WITH HOLE {...}

    // CASE 3: THIN BEAM IN TENSILE {...}

    // CASE 4: Euler Bernouilli {...}
}
```

In the simplest case, the boundary conditions can be set as shown in listing 11. This boundary condition will consistently set the same corners to zero displacement, taking advantage of the ordering of the nodes generated using *MeshLab*. Figure 3 shows the boundary conditions corresponding to listing 11.

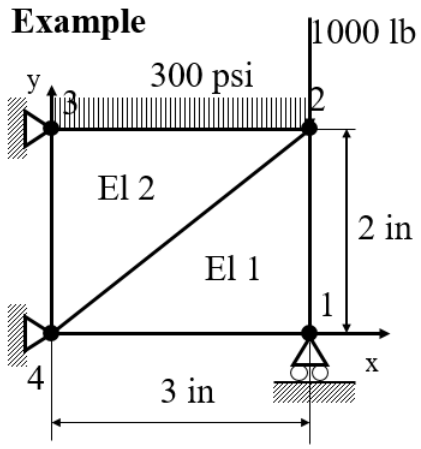


Figure 3: Case 1 Square Mesh Boundary Conditions

Listing 11: Case 1 Boundary Conditions

```
    // CASE 1: FOR SQUARE MESH
    //(ONLY FOR SQUARE MESH 2by2,3by3,...10by10,...)
    int DOF1 = sqrt(nNodes) * 2;
    int DOF3 = (nNodes - sqrt(nNodes) + 1) * 2;
    int DOF4 = (nNodes - sqrt(nNodes) + 1) * 2 - 1;
    int DOF5 = 1;
    int DOF6 = 2;
    int FORCEDOF = nNodes * 2;

    int c = 0;
    for (int i = 1; i <= nDOF; i++) {
        if (i != DOF1 && i != DOF3 && i != DOF4 && i != DOF5 && i != DOF6) {
            Isol[c] = i;
            c = c + 1;
        }
    }
    f[FORCEDOF-1] = 1225; // Fy at node 3 is 1225
    //f[FORCEDOF - 2] = -1225; // Fx can also be set
```

A more complicated geometry requires more manual setting of boundary conditions. To compensate for this tedious task, a vector of integers called **listNodesExclude** is adjusted such that all zero displacement nodes are set here. To establish which nodes are to be inserted in this vector and which nodes correspond to the horizontal force, boundary nodes of interest were clicked using MeshWorks' "Mesh Operations > Select Node", which was modified to output node index in addition to its default x,y,z coordinates by MessageBox.

Listing 12: Case 2 Boundary Conditions

```
    // Isol: an array that mimics vector Isol (required form for FEM)
    std::vector<int> listNodesExclude = { 2, 5, 7, 3, 6, 4, 8, 1 };
    std::vector<int> listDOFInclude; //contains all degrees of freedom
    // setting listDOFInclude
    for (int i = 0; i < nDOF; i++) {
        listDOFInclude.push_back(i + 1);
    }
    // eliminating values from listDOFInclude ...
    // from list listNodes Include (both x and y components)
    int my_int;
```

```
    for (int i = 0; i < listNodesExclude.size(); i++) {
        for (int j = 0; j < 2; j++) {
            if (j == 0)
                my_int = listNodesExclude[i] * 2;
            else
                my_int = listNodesExclude[i] * 2 - 1;

            auto it = std::find(listDOFInclude.begin(), listDOFInclude.end(), my_int);
            if (it != listDOFInclude.end()) {
                listDOFInclude.erase(it);
            }
        }
    }

    // Set Isol to vector
    for (int i = 0; i < listDOFInclude.size(); i++) {
        Isol[i] = listDOFInclude[i];
    }
    // Set force vector
    f[2 * 251 - 2] = 1500;
    f[2 * 243 - 2] = 1500;
    f[2 * 245 - 2] = 1500;
    f[2 * 246 - 2] = 1500;
    f[2 * 248 - 2] = 1500;
    f[2 * 249 - 2] = 1500;
    f[2 * 247 - 2] = 1500;
    f[2 * 244 - 2] = 1500;
    f[2 * 242 - 2] = 1500;
    f[2 * 250 - 2] = 1500;
```

Scattering global matrix values into modified matrix

The finite-element method is the solution of a system of linear equation which must be modified by applying boundary conditions.

$$\mathbf{K}\vec{d} = \vec{f} \qquad \mathbf{K_{mod}}\vec{d_{mod}} = \vec{f_{mod}}$$

Listing 13: scatterKmod()

```
void FEM::scatterKmod(double** &K, double** &Kmod) {
    // Entries of Isol represent rows and cols that will be extracted
    int n = vars;
    int row, col;
    //Creating Kmod
    for (int i = 0; i < n; i++) {
        row = Isol[i] - 1;
        for (int j = 0; j < n; j++) {
            col = Isol[j] - 1;
            Kmod[i][j] = K[row][col];
        }
    }
}
```

Listing 14: scatterfmod()

```
void FEM::scatterfmod(double* &f, double* &fmod) {
    int n = vars;
    int row;
    // Creating fmod
    for (int i = 0; i < n; i++) {
        row = Isol[i] - 1;
        fmod[i] = f[row];
    }
}
```

Solving system of equations

Listing 15: In MainFunction...

```
    //Solve system of equations
    bool doesSystemSolve = GLKMatrixLib::GaussJordanElimination(Kmod, vars, fmod);
    scatterBackDisplacements(fmod, d);
    // Alternatively
    //GLKMatrixLib::GaussSeidelSolver(Kmod, fmod, vars, dmod,1e-6);
    //scatterBackDisplacements(dmod, d);
```

The reverse scattering must be done. When computing stress of elements, even zero displacement nodes must be included.

Listing 16: scatterBackDiplacements()

```
void FEM::scatterBackDisplacements(double* &dmod, double* &d) {
    int n = vars; // n is the number of columns in Isol
    int row;
    for (int i = 0; i < n; i++) {
        row = Isol[i] - 1;
        d[row] = dmod[i];
    }
}
```

Solving for stresses

With the displacements solved, the stresses can be computed as shown in equation 8. For each element, matrix B must be recomputed and matConn and matNodes are conveniently used again. Additionally, listing 17 shows how the values are outputted to a text file (for later visualizing in MATLAB). Also, one can see an array called **vonMisVec**, storing the Von Mises stress for each element. This is used for post-processing and visualization in MeshWorks. Although, colors can be displayed in MeshWorks ,further work is needed and MATLAB was preferred for visualizing results due to built in functions.

Listing 17: computeStress()

```
void FEM::computeStress()
{
    int n1, n2, n3;
    double b1, b2, b3, c1, c2, c3, x1, x2, x3, y1, y2, y3, A;
    //Create stream for writing data to text file for MATLAB
    std::ofstream myfile;
    myfile.open("stresses.txt");
    // For each element
```

```
    for (int element = 0; element < nFaces; element++) {
        // Extract node index
        n1 = matConn[element][0];
        n2 = matConn[element][1];
        n3 = matConn[element][2];
        // Extract elemental displacement values
        de[0] = d[2 * n1 - 2];
        de[1] = d[2 * n1 - 1];
        de[2] = d[2 * n2 - 2];
        de[3] = d[2 * n2 - 1];
        de[4] = d[2 * n3 - 2];
        de[5] = d[2 * n3 - 1];
        // RECOMPUTE B MATRIX
        // Using node number from above to access x,y coordinates
        // (Note: index starts is 1 less for matrices, hence the -1)
        x1 = matNodes[n1 - 1][0]; y1 = matNodes[n1 - 1][1];
        x2 = matNodes[n2 - 1][0]; y2 = matNodes[n2 - 1][1];
        x3 = matNodes[n3 - 1][0]; y3 = matNodes[n3 - 1][1];
        // Computing area of element
        A = abs(0.5*((x2*y3 - x3*y2) + (x3*y1 - x1*y3) + (x1*y2 - x2*y1)));
        // Computing elements of matrix B
        b1 = y2 - y3; b2 = y3 - y1; b3 = y1 - y2;
        c1 = x3 - x2; c2 = x1 - x3; c3 = x2 - x1;
        // Setting matrix B
        B[0][0] = b1 / (2 * A); B[0][2] = b2 / (2 * A); B[0][4] = b3 / (2 * A);
        B[1][1] = c1 / (2 * A); B[1][3] = c2 / (2 * A); B[1][5] = c3 / (2 * A);
        B[2][1] = b1 / (2 * A); B[2][3] = b2 / (2 * A); B[2][5] = b3 / (2 * A);
        B[2][0] = c1 / (2 * A); B[2][2] = c2 / (2 * A); B[2][4] = c3 / (2 * A);
        // Computing Stress
        GLKMatrixLib::Mul(D, B, 3, 3, 6, temp2);
        GLKMatrixLib::Mul(temp2, de, 3, 6, sige);
        // Store vonMises for drawShade function
        double vonMises = computeVonMises();
        vonMisVec[element] = vonMises;
        // Element center (where values are assumed)
        double xavg = (x1 + x2 + x3) / 3;
        double yavg = (y1 + y2 + y3) / 3;
        // Function toFile writes values to text file
        toFile(element, xavg,yavg,sige[0], sige[1], sige[2], vonMises,myfile);
    }

}
```

Post-processing

Array **vonMisVec** contains the von Mises stress for each element as computed in **computeStress**.

Listing 18: In MainFunction...

```
    //Normalize vonMisVec (first find max value, then divide all by it)
    double maxVM = 0.0;
    int maxVMindex = 0;
    for (int i = 0; i < nFaces; i++) {
        if (vonMisVec[i] > maxVM) {
            maxVM = vonMisVec[i];
            maxVMindex = i;
        }

    }
    // Reverse order (i.e largest stress = red )
    for (int i = 0; i < nFaces; i++) {
        vonMisVec[i] = 1.0 - (vonMisVec[i]/ maxVM);
    }
```

Listing 19: In MainFunction...

```
void CFEADlg::OnBnClickedButton4()
{
    pFEM->colorFaces();
}
```

Listing 20: In FEM...

```
void FEM::colorFaces() {
    std::cout << "colorFaces" << std::endl;

    AfxGetApp()->BeginWaitCursor();
    CMainFrame *pWnd = (CMainFrame *)(AfxGetMainWnd());

    pWnd->ChangeColor(vonMisVec, vonMisVecrow);

    AfxGetApp()->EndWaitCursor();

}
```

I added an array called **myVec** in class **QBody** and **m FEAON** in **GLKernelView**. The array contains the information about the magnitude of the colors required for each element. It gets populated in listing 21. Also, the boolean gets turned to true.

Listing 21: In CMainFrame...

```
// FOR FEA
void CMainFrame::ChangeColor(double* &vec, int vecRow)
{
    CMeshWorksDoc *pDoc = (CMeshWorksDoc *)GetActiveDocument();
    GetMainView()->GetGLKernelView()->m_FEAON = true;

    // Acquiring max, min for color map using ChangeValueToColor
    double vecMax=0.0, vecMin = vec[0];
    for (int i = 0; i < vecRow; i++) {
```

```
10          if (vec[i] > vecMax) { vecMax = vec[i]; }
11          if (vec[i] < vecMin) { vecMin = vec[i]; }
12      }
13
14      // Need access to QBody through pointer
15      float red, green, blue;
16      for (POSITION Pos = (pDoc->m_meshList).GetHeadPosition(); Pos != NULL;) {
17          QBody *patch = (QBody *)((pDoc->m_meshList).GetNext(Pos));
18
19          for (int i = 0; i < vecRow; i++) {
20              ChangeValueToColor2(vecMax, vecMin, vec[i],red, green, blue);
21              patch->myVec[i][0] = red;//vec[i]; // simplest and ok
22              patch->myVec[i][1] = green;//vec[i];// vec[i]; //green/ 255;
23          }
24      }
25      GetMainView()->GetGLKernelView()->refresh();
26  }
```

The solution was implemented where two draw functions exist. The original draw function is called **drawShade()**. When the switch gets activated **drawShade2** is used as shown in one instance in listing 22. Having access to QBody's member variables, it can draw itself based on the values stored in **myVec**. A more elegant solution should exist. Also, after clicking "Color Element" the body does not change colors immediately. One must go into "Mesh Operations > Select Node" to make the colors change.

Listing 22: Effects...

```
1                   if (m_FEAON) {
2                       entity->drawShade2();
3                   }
4                   else {
5                       entity->drawShade();
6                   }
```

Results and Discussion

Case 1: Square Mesh

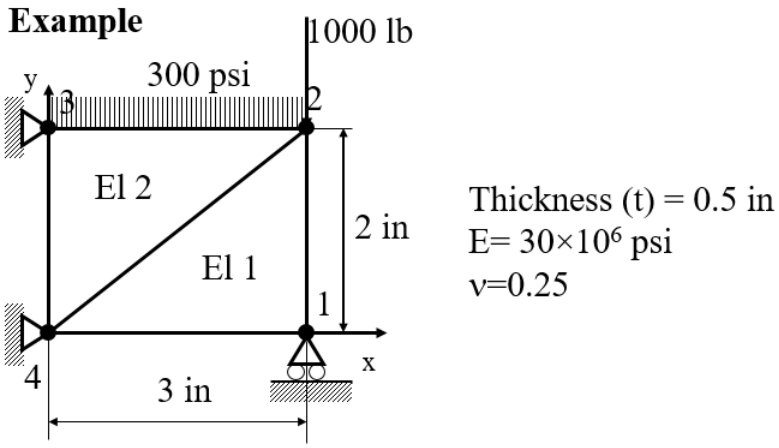
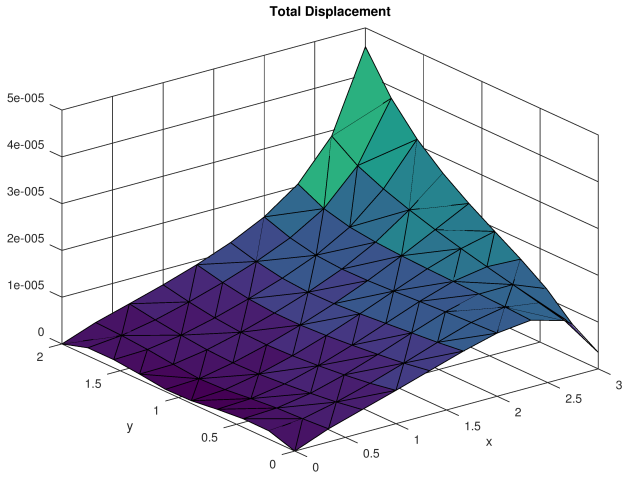
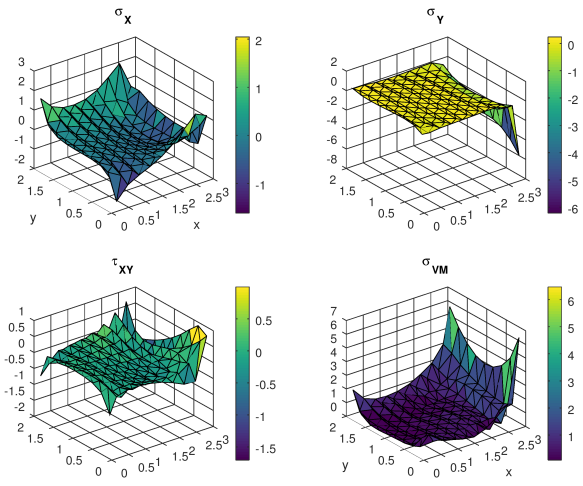


Figure 4: Total displacement for 10 by 10 mesh

Figure 3 showed the boundary conditions and force applied. The full problem is shown in figure 4 and solved in (De,P). The only modification to the problem was a simplification which saw the 300 psi surface force be replaced by the equivalent 225 lb force in the corner such that a traction force of 1225 lb was applied. This is shown in (De,P). This case was convenient as a starting point because of the results could be compared. The values were successful obtained for the 2 by 2 mesh.



(a) Total displacements



(b) Stresses

Figure 5: Results for 10 by 10 mesh

As expected the largest total displacement was at the corner where the force was applied. Next, convergence was studied by increasing the number of elements. The results are shown in table 1 and figure 6.

	Total Nodes	Total Faces	MAXσ_{VM}	..at index	MAX d	..at index	node #
1	4	2	1344.2	0	-	-	-
2	25	32	5770.8	6	3.3509E-04	49	25
3	36	50	7241.6	8	3.9625E-04	71	36
4	100	162	12995	16	5.7155E-04	199	100
5	144	242	15820	20	6.3301E-04	287	144
6	196	338	18626	24	6.8426E-04	391	196
7	256	450	-	-	7.2820E-04	511	256
8	400	722	-	-	8.0090E-04	799	400
9	484	795	-	-	8.3166E-04	967	484

Table 1: Case 1 maximum values for refined mesh

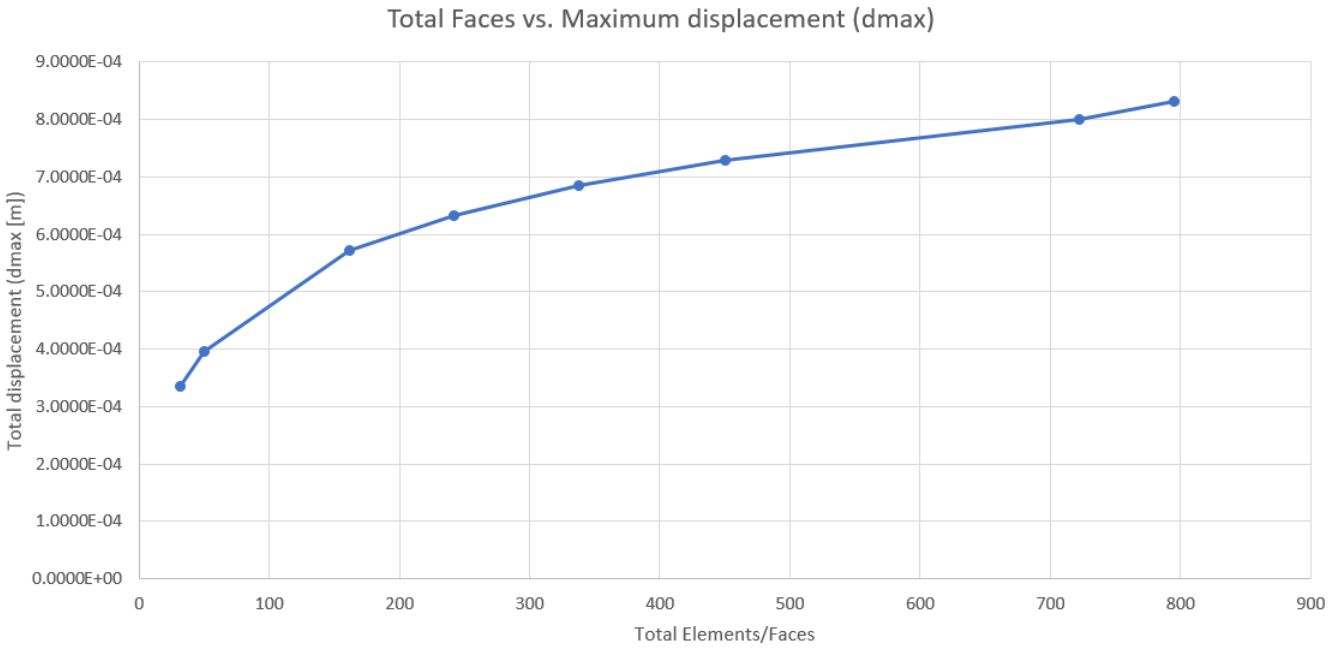


Figure 6: Convergence graph

Case 2: Plate with Hole

A mesh was generated using DistMesh - a simple mesh generator in MATLAB (Burkardt, 2011). A problem described in figure 8 was studied with results displayed in figure 9 which were not validated. The objective was to implement the program on a geometry with higher complexity. Further work will mimic the problem defined in Cornell University's ANSYS module (confluence.cornell.edu) where stresses and displacements can be compared. The appropriate meshes will have to be selected as shown in 7.

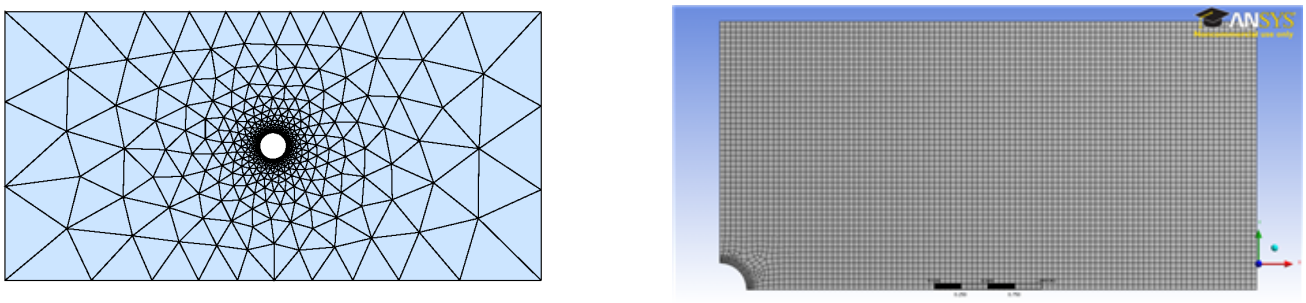


Figure 7: Meshes for plate with hole problem: MATLAB generated (left) and ANSYS Cornell (right)

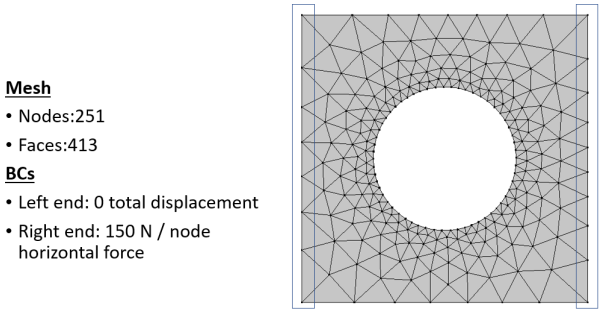
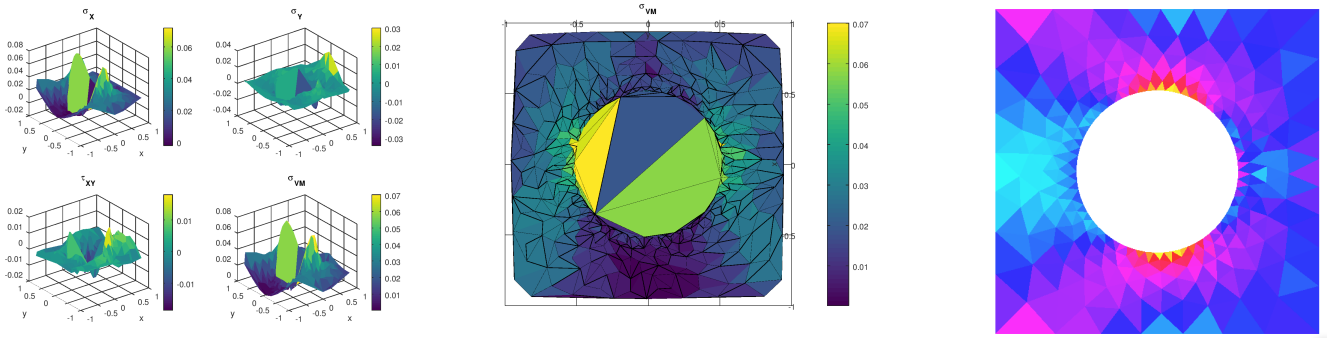


Figure 8: Case 2 Plate with hole



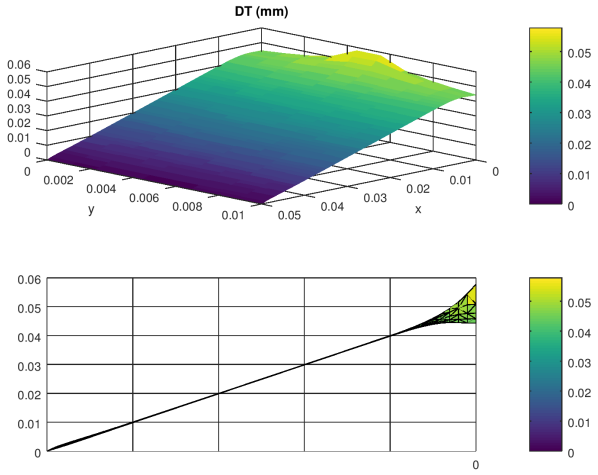
(a) Stresses (b) MATLAB σ_{VM} (c) MeshWorks σ_{VM}

Figure 9: Meshes for plate with hole problem: MATLAB generated (left) and ANSYS Cornell (right)

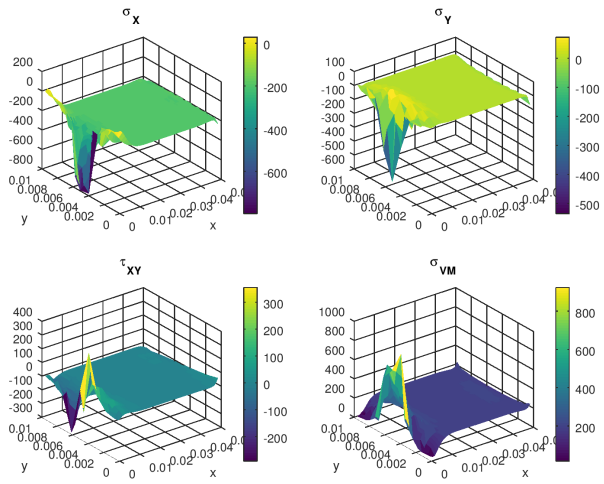
Case 3: Beam under tensile stress

Case 3 seeks to validate the finite element functionality. The beam under uni-axial loading problem could be readily compared with the results from Cornell University's ANSYS Module (confluence.cornell.edu). The same material properties and loading forces were considered. The results displayed in figure 10a validate the computation of displacements. However, the stresses can be seen to be exaggerated at the point where the force is applied. Figure 11 confirms that σ_x is approximated 200 MPa along most of the beam's cross section as was shown in (confluence.cornell.edu). Figure 11 is three plots of σ_x at the lower edge, middle and upper edge of the beam. There is a curious fluctuation at the lower boundary which is not present at the upper boundary, requiring further investigation. Aside from that, large fluctuations at the point at which the force is applied in the middle which is

expected.



(a) Total displacements



(b) Stresses

Figure 10: Meshes for plate with hole problem: MATLAB generated (left) and ANSYS Cornell (right)

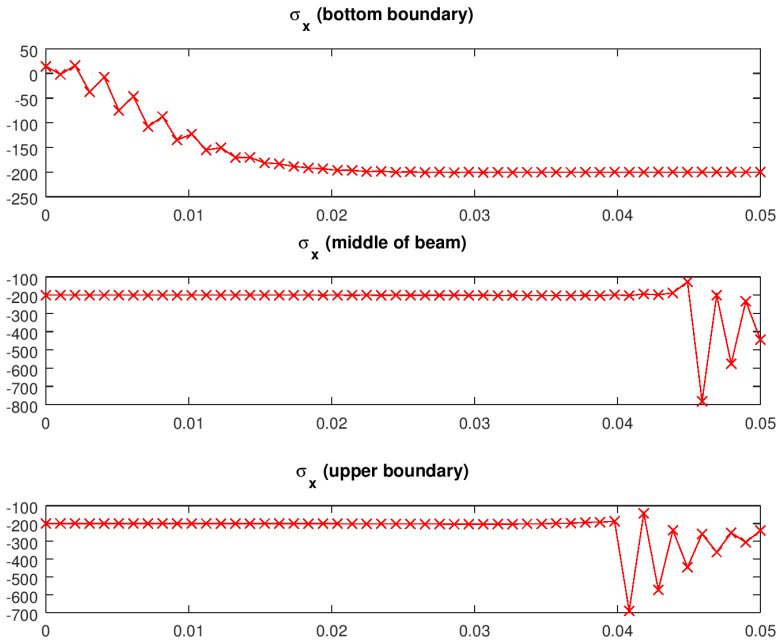


Figure 11: σ_x along beam at three heights

References

- Boeraeve, Ir P. "Introduction To The Finite Element Method (FEM)." Gramme Institut. Liege, Blgica (2010): 68.
- De, P. S. "MANE 4240/CIVL 4240: Introduction to Finite Elements." Abaqus Handout.
- Burkardt, J. "DISTMESH A Simple Mesh Generator in MATLAB." (2011).
- https://confluence.cornell.edu/display/SIMULATION/ANSYS+-+Plate+With+a+Hole
- https://confluence.cornell.edu/pages/viewpage.action?pageId=124958703

Appendix

Listing 23: class FEM

```
#pragma once
#include <iostream>

class FEM {

public:
    // Members
    float t, E, v; // thickness, modulus, poisson

    // Functions
    FEM() { ... };

    ~FEM() { ... };

    // printing
    void printNodeCoord();
    void PrintMatrix(double** &a, int row, int col);
    void PrintVector(double* &a, int row);

    // Setting
    void setProps();
    void setNodeMatrix();
    void setConnMatrix();
    void setDMatrix(); // strain-stress matrix
    void setBCs();

    // Computing
    void computeKMatrix();
    void scatter(double** &Ke, double** &K, int n1, int n2, int n3);
    void scatterKmod(double** &K, double** &Kmod);
    void scatterfmod(double* &f, double* &fmod);
    void scatterBackDisplacements(double* &dmod, double* &d);
    void computeStress();
    double computeVonMises();

    // Getting functions
    int getNumberOfNodes();
    int getNumberOfFaces();

    // Setting memory
    void Create();
    void Destroy();
    void CreateVector(double* &ptr, int ptrsize);
    void DeleteVector(double* &ptr);
    void CreateVectorIsol(int* &ptr, int ptrsize);
    void DeleteVectorIsol(int* &ptr);

    // PostProcessing
    void colorFaces();
```

```
50      void openFile();
51      void toFile(int node, double xavg, double yavg,
52                  double sigx, double sigy, double tauxy,
53                  double vonMis, std::ofstream& myfile);
54
55      // Main function
56      void MainFunction();
57
58  public:
59      // Sizes
60      int nNodes;
61      int nFaces;
62      int nDOF;
63
64      // Matrices/Vectors
65      double** matNodes; int matNodesRow,matNodesCol;
66      double** matConn;  int matConnRow,matConnCol;
67      double** D;  int Drow, Dcol;
68      double** B;  int Brow, Bcol;
69      double* f;   int frow;
70      double* d;   int drow;
71      double** K;  int Krow, Kcol;
72      double** Ke; int Kerow, Kecol;
73
74      // Matrices/Vectors for solving systems
75
76      int vars;
77      int* Isol;      int Isolrow;
78      double** Kmod; int Kmodrow, Kmodcol;
79      double* fmod;  int fmodrow;
80      double* dmod;  int dmodrow;
81
82      //Matrices/Vectors for stress strain elemental
83      double* sige; int sigerow;
84      double* de; int derow;
85
86      //miscallenous matrices
87      double** BT;            int BTrow,BTcol;
88      double**temp;           int trow,tcol;
89      double**temp2;          int t2row, t2col;
90      double* vonMisVec;      int vonMisVecrow;
91
92  };
```

Listing 24: FEM Constructor

```
1       FEM() { std::cout << "FEM constructor called" << std::endl;
2               // Initializing Sizes
3               nNodes = getNumberOfNodes();
4               nFaces = getNumberOfFaces();
5               nDOF = nNodes * 2;
6               //Initializing Matrices/Vectors
7               matNodesRow = nNodes; matNodesCol = 2;
8               matConnRow = nFaces; matConnCol = 3;
```

```
9           Drow = 3; Dcol = 3;
10          Brow = 3; Bcol = 6;
11          frow = nDOF;
12          drow = nDOF;
13          Krow = nDOF; Kcol = nDOF;
14          Kerow = 6; Kecol = 6;
15          //Initializing for solver
16          vars = nDOF - 5; //**********************
17          Isolrow = vars;
18          Kmodrow = vars; Kmodcol = vars;
19          fmodrow = vars;
20          dmodrow = vars;
21          //Initialize for each element
22          sigerow = 3;
23          derow = 6;
24          //Initialize
25          BTrow = Bcol; BTcol = Brow;
26          trow = 6; tcol = 3;
27          t2row = 3; t2col = 6;
28          vonMisVecrow = nFaces;
29          //Allocate memory
30          Create();
31          };
```