Designing the 3-D FEM algorithm with Python

Hongsup Oh

PhD student

Department of the mechanical engineering

The university of Utah

ME EN 6510

Contents

Contents

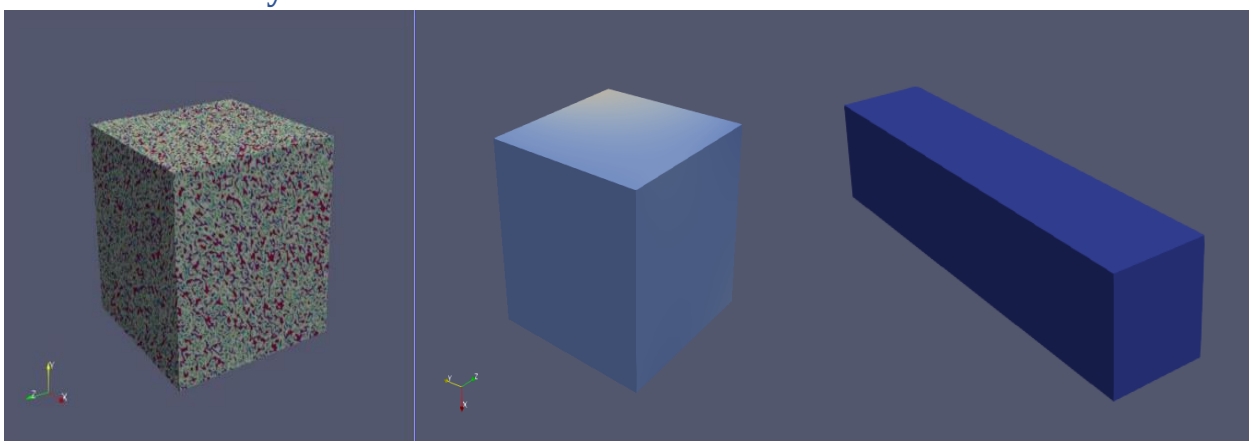
1 Introduction

The main goal of the project is designing the FEM algorithm by Python based on the learning of the lecture. Its solution is expected to be same with that of commercial software such as ANSYS or Abaqus. Input of the algorithm is known displacement (Dirichlet boundary condition) and force (Neumann boundary condition). And output is unknown displacement vector and force vector as well as stress vectors. Based on designed algorithm, a convergence study for displacement and stress will be introduced. The result of displacement and stress will be visualized with Paraview. Its answers will be verified with mathematical equation and validated with experimental data. Finally, future work will be discussed.

2 Geometry

It is cubic and rectangle mortar comprising of cement, graded sand and water. The cubic mortar is for compression situation, and the rectangle mortar is for bending situation. In the case of compression, dimension of the material is 2 by 2 by 2 inches. In the case of bending, dimension of the material is 10 by 2 by 2 inches. Since it is simple cubic/rectangle, everything will be included. In Python, 3D array is designed. Python numpy library has function for generating mesh with the length of the material. In addition, cubic/rectangle mortar can be expressed by three numerical value. Three different number will be used to express cement, graded sand, and water. For example, cement is 1, graded sand is 50, and water is 100. Then it is possible to express cubic mortar with three materials.

2.1 3-D Geometry



<Figure 1. 3-D Geometries>

Figure 1 is generated by Python, and expressed by Paraview. At most left of the figure 1, it is assumed that three materials such as cement, graded sand and water are homogeneously distributed in cubic mortar. Three different numerical value will be assigned to each material. They are homogeneously distributed, because random library of Python is used to generate it. At middle of the figure 1, the material is assumed as homogeneous material. Both two figures are for compression test. At most right of the figure 1, it is a geometry for the bending test, and it is assumed as homogeneous material.

2.2 Code for Geometry

```
DeformX = np.zeros((zNode,xNode,yNode))
DeformY = np.zeros((zNode,xNode,yNode))
DeformZ = np.zeros((zNode,xNode,yNode))
```

<Figure 2. Generate deformation grid>

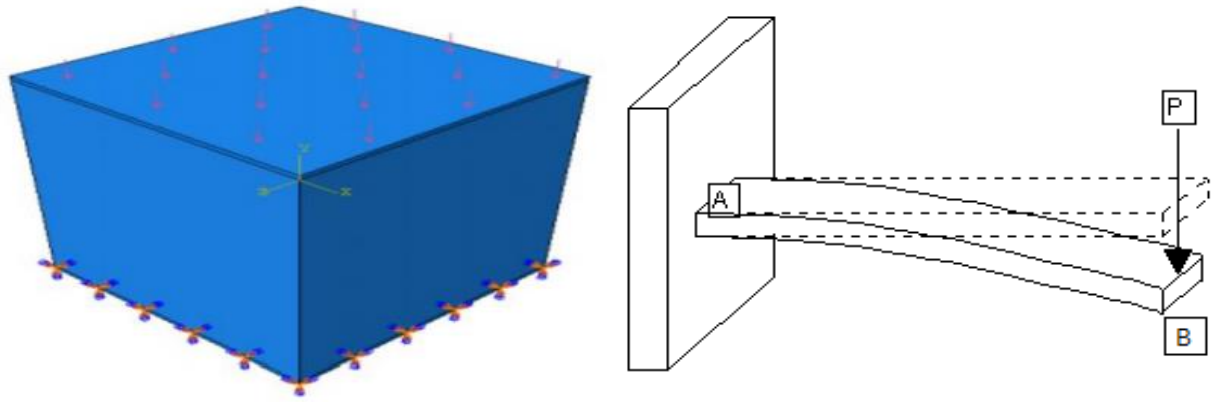
Since it is Python algorithm, it does not have any specific way to design a geometry. Current algorithm only works for cubic/hexahedron material. But its code is very simple. In figure 2, numpy zeros function create shape of the cubic material.

```
matProperties = {}
matProperties[ele] = (np.random.choice(Es),np.random.choice(nus))
```

<Figure 3. Generate composite material>

In order to generate composite material, hash map and random algorithm is used. For example, empty hash map named matProperties is generated, then random value of Young's modulus and Poisson's ratio selected from Es and nus array. Where Es has various values of Young's modulus and nus has various value of Poisson's ratio.

3 Boundary conditions



<Figure 4. Boundary conditions>

Two cases of engineering situations will be tested such as compression and bending. Figure 4 is an example of boundary conditions. There are two types of boundary conditions such as Neumann boundary conditions (force) and Dirichlet boundary conditions (displacement). Left figure of figure 4 is compression case. Compression load will be applied on the top of the material such as $f_x = f_y = 0 \; and \; f_z = -p$, and the other side is totally fixed such as $u_x = u_y = u_z = 0$. Right figure is bending case. A point load will be applied on the end of free end such as $f_x = f_y = 0 \; and \; f_z = -p$, and fixed end point is totally fixed such as $u_x = u_y = u_z = 0$.

3.1 Code for boundary conditions

```
def nodeVal(self):
    self.generate_mesh()
    self.coor = defaultdict(int)
    glb = 0
    for x in self.xl:
        for y in self.yl:
            self.nodeLocation[glb] = [x,y]
            self.coor[(x,y)] = glb
            glb+=1
```

<Figure 5. The function for assigning global nodes>

Figure 5 is 2-D example of the function for assigning global node number for each node. For example, global node number of [0, 0] will be 0, [0, 1] will be 1 and [0, 2] will be 2. Its relation will be saved in hash map named self.coor and self.nodeLocation. Then, self.nodeLocation = {0:[0, 0], 1:[0, 1], 2:[0, 2],…} and self.coor = {[0, 0]:0, [0, 1]:1, [0, 2]:2, …}.

```
def defineBoundaryConditions(x,y,z,xNode,yNode,zNode):
    coor,nodeLocation = nodeVal(x,y,z,xNode,yNode,zNode)
    DR_BC = defaultdict(int)
    NN_BC = defaultdict(int)
    totalNodes = xNode*yNode*zNode
    for globalnode in range(totalNodes):
        #Dirichlet BC
        if nodeLocation[globalnode][2] == 0:
            DR_BC[3*globalnode+0] = 0
            DR_BC[3*globalnode+1] = 0
            DR_BC[3*globalnode+2] = 0
        #Neumann BC
        if nodeLocation[globalnode][2] == 2:
            NN_BC[3*globalnode+0] = 0
            NN_BC[3*globalnode+1] = 0
            NN_BC[3*globalnode+2] = -5000

        elif nodeLocation[globalnode][2] != 0 and nodeLocation[globalnode][2]!= 2:
            NN_BC[3*globalnode+0] = 0
            NN_BC[3*globalnode+1] = 0
            NN_BC[3*globalnode+2] = 0

    return DR_BC,NN_BC
```

```
def defineBoundaryConditions(x,y,z,xNode,yNode,zNode):
    coor,nodeLocation = nodeVal(x,y,z,xNode,yNode,zNode)
    DR_BC = defaultdict(int)
    NN_BC = defaultdict(int)
    totalNodes = xNode*yNode*zNode
    for globalnode in range(totalNodes):
        #Dirichlet BC
        if nodeLocation[globalnode][0] == 0:
            DR_BC[3*globalnode+0] = 0
            DR_BC[3*globalnode+1] = 0
            DR_BC[3*globalnode+2] = 0

        #Neumann BC
        if nodeLocation[globalnode][0] == 10 and nodeLocation[globalnode][2] == 2:
            NN_BC[3*globalnode+0] = 0
            NN_BC[3*globalnode+1] = 0
            NN_BC[3*globalnode+2] = -5000

        elif not(nodeLocation[globalnode][0] == 10 and nodeLocation[globalnode][2] == 2) and nodeLocation[globalnode][0]!= 0:
            NN_BC[3*globalnode+0] = 0
            NN_BC[3*globalnode+1] = 0
            NN_BC[3*globalnode+2] = 0

    return DR_BC,NN_BC
```

<Figure 6. The function for boundary conditions>

Figure 6 is the function of defining boundary conditions. In hash map named self.nodeLocation has an information of global node number of each locations. The hash map named self.boundary_values defines known deformation information. The hash map named self.NN_BC defines known force values. Left figure is the boundary condition for compression. Since bottom surface is fixed, all deformation values are zero, where z-axis value is zero. Since compression load is applied at top surface, z-axis force is defined as -5000, and x-axis and y-axis forces are defined as 0, where z-axis value is 2. Right figure is the boundary condition for bending. Since left surface is fixed, all deformation values are zero, where x-axis value is zero. Since z direction point load is applied at free end surface, z-axis force is defined as -5000, and x-axis and y-axis forces are defined as 0, where x-axis value is 10 and z-axis value is 2.

4 Mesh

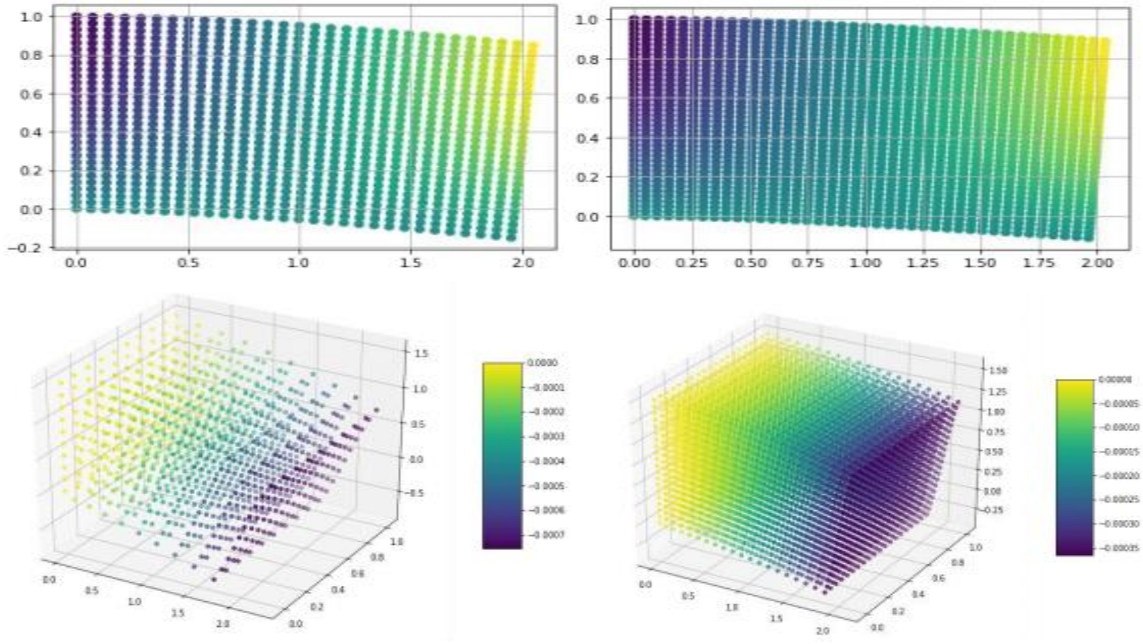
In order to make mesh, one function named generate_mesh is designed.

```
def generate_mesh(x,y,xNode,yNode):
    xmin,xmax = 0,x
    ymin,ymax = 0,y
    xl = np.linspace(xmin,xmax,xNode)
    yl = np.linspace(ymin,ymax,yNode)
    xm, ym = np.meshgrid(xl,yl)
    return xl,yl,xm,ym
```

<Figure 7. The function for making mesh>

It is possible to make mesh, with generate_mesh function, Numpy library's linspace and meshgrid function is utilized to make it. And it only makes rectangle/hexahedron mesh. In order to make triangle/tetrahedron mesh, more advanced knowledges are required. It would be one of the next jobs to upgrade the FEM algorithm.

4.1 Figures of mesh



<Figure 8. 2-D and 3-D mesh>

Rectangle mesh is used for 2D model and hexahedron mesh is used for 3D model. Since it is the result of self-made FEM algorithm with Python, it is difficult to use different type of element such as triangular or tetrahedron mesh. In addition, it is impossible for me to use different meshing strategies such as partitioning and local refinement. Above figures are example result of FEM algorithm. Left edge/surface is fixed and right edge/surface is applied down direction loads. In 2D, 4 nodes (points) are one element. In 3D, 8 nodes (points) are one element.

4.2 Mesh convergence

It is possible to use two refinement methods. The one is h-refinement increasing the number of elements, the other is p-refinement increasing the complexity of the equation. In the project, only h-refinement is used for convergence study. For the mesh convergence, maximum and minimum values of displacement, and maximum and minimum values of stress value will be used. Its result will be discussed in result section.

5 Quantities of interest

Deformation, strain, and stress are the quantities of interest in the model. Since, goal of the FEM analysis is getting displacement and stress, they are important. First, unknow deformation must be calculated:

$$d_f = K_{ff}^{-1}\left(f_f - K_{fs}d_s\right)$$

$$d_f: Unknown\ displacement\ and\ d_s: Known\ displacement$$
$$f_f: Known\ force$$
$$f\ is\ free\ index\ and\ s\ is\ supported\ index$$

Then, unknown deformation vector is combined with known deformation vector. Now we have whole deformation vector named {d}. Second, strain vector must be calculated:

$$\begin{Bmatrix}\varepsilon_x\\ \varepsilon_y\\ \varepsilon_z\end{Bmatrix} = [B]\{d\}$$

$$[B]: Derivative\ of\ the\ shape\ function$$

Then, stress vector will be calculated with strain vector:

$$\begin{Bmatrix}\sigma_x\\ \sigma_y\\ \sigma_z\end{Bmatrix} = [D]\begin{Bmatrix}\varepsilon_x\\ \varepsilon_y\\ \varepsilon_z\end{Bmatrix}$$

$$[D]: Elasticity\ matrix$$

6 Verification and validation

Displacements and stresses will be checked with a simplified verification and validation. That is, results of FEM algorithm such as displacement and stresses will be compared with analytical solution of displacements and stresses for the verification and with experiment result for the validation.

6.1 Verification

In the case of compression, maximum displacement of the simulation result will be checked with following equation:

$$\delta = \frac{pL}{EA}$$
$$where, \delta: displacement, E: Young's\ modulus, A: Area, p: pressure, L: length$$

Second, maximum stress will be compared with following equation:

$$\sigma = \frac{p}{A}$$

$$where, \sigma: \text{stress}, A: Area, p: pressure$$

In the case of bending, maximum displacement of the simulation result will be checked with following equation:

$$\delta = \frac{pL^3}{3EI}\ or\ \delta = \frac{wL^2}{8EI}$$

Where, I: moment of inertial around the neutral axis p: point load, w: distributed load

Second, maximum stress will be checked with following equation:

$$\sigma = \frac{My}{I}$$

Where, I: moment of inertial around the neutral axis M: calculated bending moment, y: vertical distance from neutral axis

6.2 Validation

Real model is cubic/rectangle mortar with cement, graded sand and water. Experiment data are provided by Al muatasim al nadabi. Through the experiment, cubic mortar's Young's modulus is achieved. But Poisson's ratio is still unknown. Material properties and data values will be discussed in result section. The project algorithm will be validated by the given data.

7 Uncertainty quantification

Material parameters such as Young's modulus and Poisson ratio strongly affect the result of FEM algorithm. I will do simulations about a composite material consisting of cement, graded sand and water. It will be very difficult to find correct effective material properties. Thus, I expect there would be a difference between experimental and computational result. In addition, boundary condition also affects the computed result. For example, I will define no displacements at bottom surface for compression test, but very small displacements can be measured in experiment because of various reasons such as lab environment and individual mistakes.

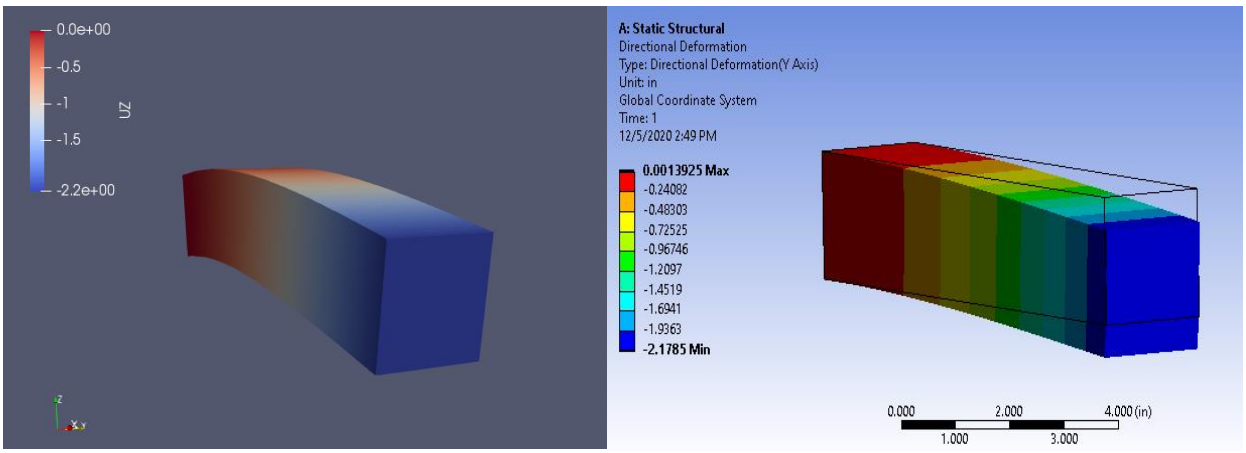
8 Simulation result

8.1 Compare solutions between Python FEM and ANSYS

Since it is Python algorithm, its solution must be compared with commercial software such as ANSYS, before doing verification and validation.

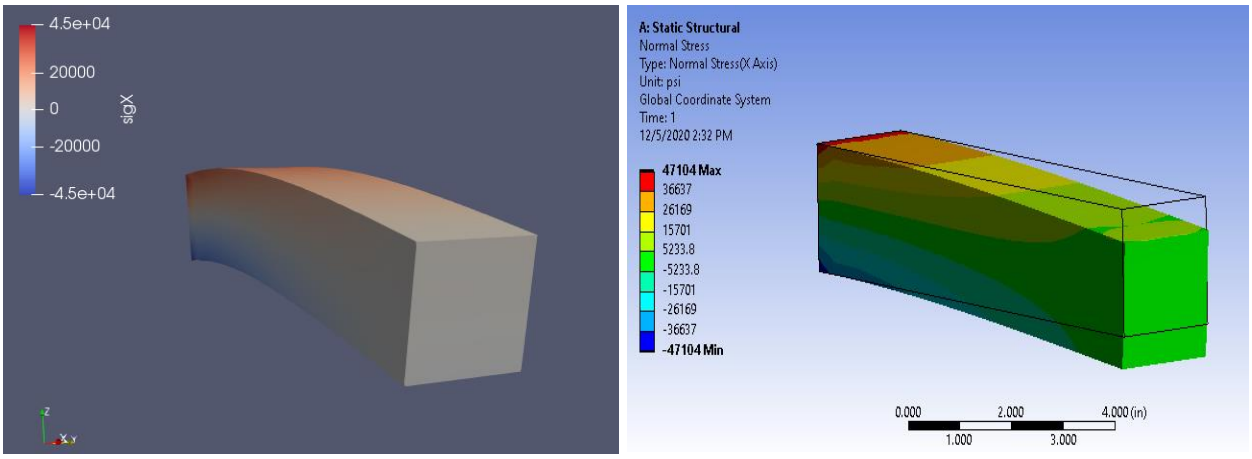
8.1.1 Bending at beam

In this example, material properties are defined such as Young's modulus is 588989.63 psi and Poison's ratio is 0.3. Dimension of model is 10 inches for x-axis, and 2 inches for y and z-axis.



<Figure 9. Result of the deformation at bending of the beam>

Left figure of figure 9 is the result of Python algorithm visualized by Paraview. And right figure is the result of ANSYS. Result of the algorithm is -2.2 inches and that of ANSYS is -2.1785 inches. It is possible to check their answers are almost same.

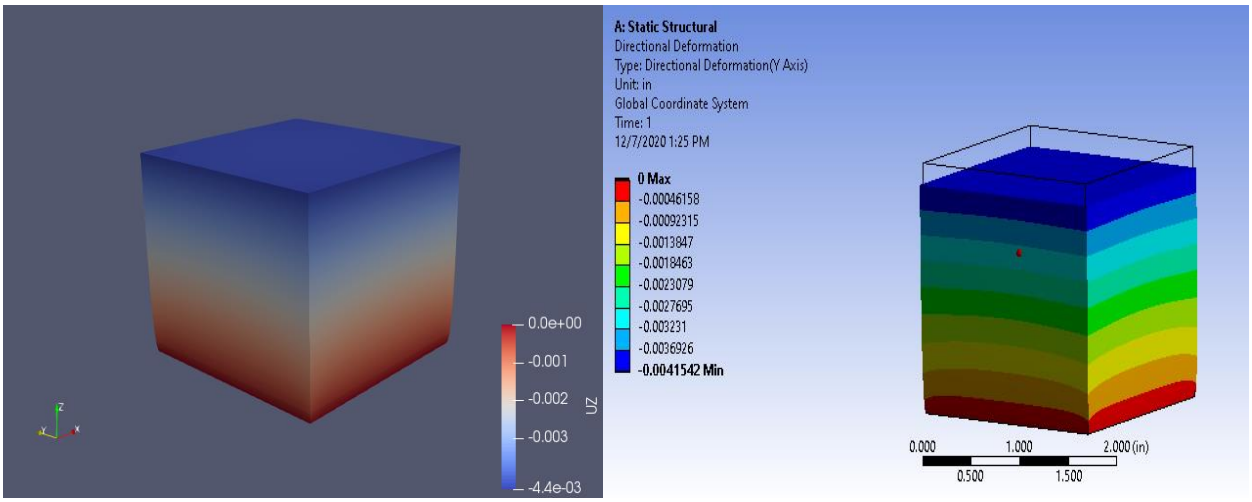


<Figure 10. Result of the normal stress at bending of the beam>

In figure 10, Python algorithms maximum and minimum stress is 45000 and -45000 psi, and that of ANSYS is 47107 and -47104 psi. It is possible to check their answers are almost similar each other, even though its difference is bigger than that of displacement.

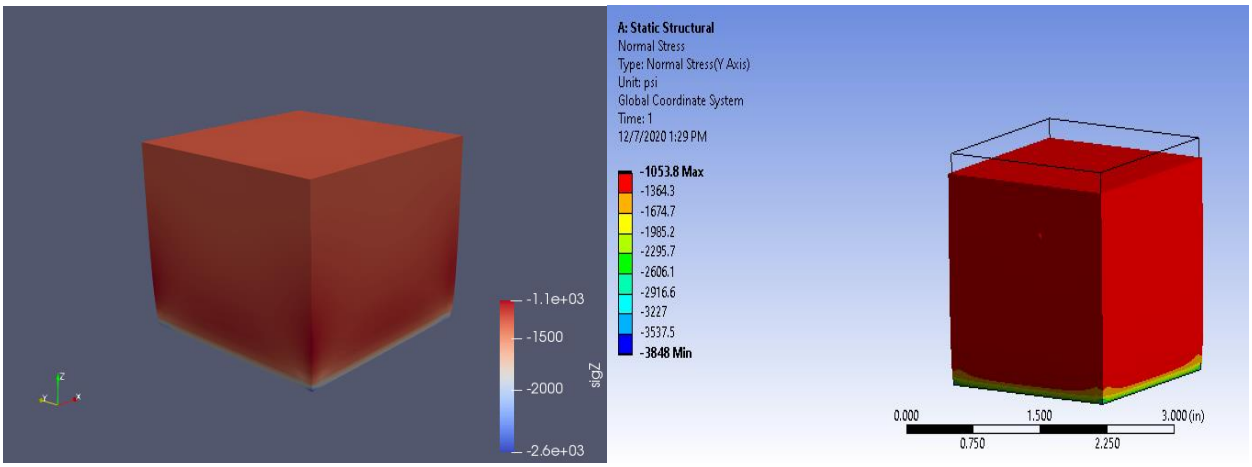
8.1.2 Compression at cubic

In this example, material properties are defined such as Young's modulus is 588989.63 psi and Poison's ratio is 0.3. Dimension of model is 10 inches for x-axis, and 2 inches for y and z-axis.



<Figure 11. Result of the deformation at compressed cubic>

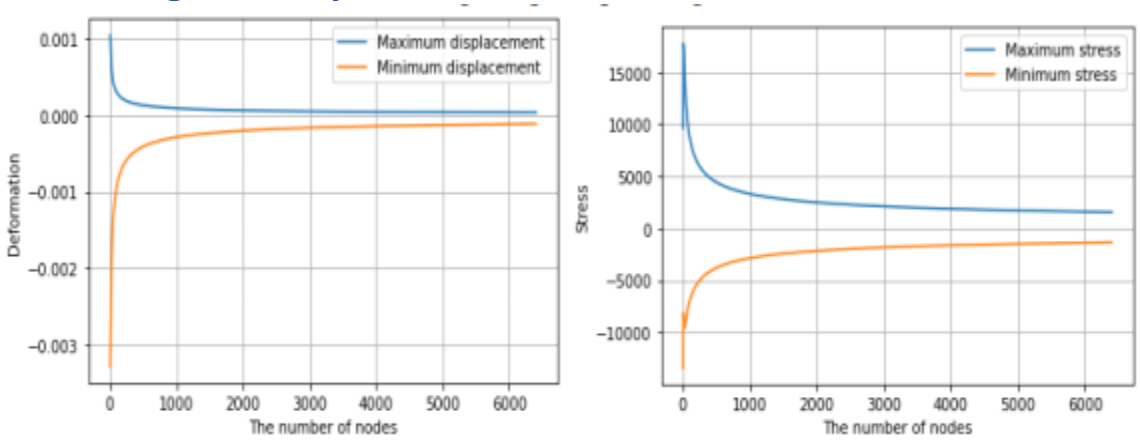
Left figure of figure 11 is the result of Python algorithm visualized by Paraview. Right figure is the result of ANSYS. Python algorithm's result is -0.0044 inches and ANSYS's result is -0.0041 inches. Python algorithm's result is slightly higher than that of ANSYS. But It is possible to check their answers are almost same.



<Figure 12. Result of the normal stress at compressed cubic>

Left figure of figure 12 is the result of Python algorithm. Right figure is the result of ANSYS. Minimum and maximum stress of Python algorithm is respectively -2600 and -1100 psi and those of ANSYS's is respectively -3848 and -1053.8 psi. Minimum stress value of the algorithm is higher than that of ANSYS, and maximum stress value of the algorithm is lower than that of ANSYS. But it is possible to check their answers are almost similar each other, even though its difference is bigger than that of displacement.

8.2 Convergence study



<Figure 13. Convergence of deformation and stress>

Based on figure 13, as the number of nodes increase, difference between maximum and minimum displacement/stress decrease. That is, maximum value decreases and minimum value increases, as the number of node increases. Left figure of figure 13 is convergence study of displacement. And right figure of figure 13 is convergence study of stress. In displacement case, maximum displacement's convergence speed is faster than that of minimum one. And they are all converged to specific value at 4000 nodes. In stress case, they are almost converged to specific value at 5000 nodes, but its convergence speed is slower than that of displacement.

8.3 Verification

Previous section's displacement and stress solutions will be compared with analytical solution of bending and compaction situation to verify them.

8.3.1 Bending at beam

$$\delta = \frac{pL^3}{3EI} = \frac{-5000 \cdot 10^3}{3 \cdot 588989.63 \cdot 1.333} = -2.12\ in$$

$$where, I = \frac{bh^3}{12} = \frac{2^4}{12} = 1.333\ in^4$$

Result of Python algorithm is -2.2 in and that of ANSYS is -2.18 inches. Analytical solution is -2.12 inches. All answers are similar each other.

$$\sigma = \frac{My}{I} = \pm\frac{-5000 \cdot 10 \cdot 1}{1.333} = \pm 37500\ psi$$

Result of Python algorithm is $\pm 45000\ psi$ in and that of ANSYS is $\pm 47107\ psi$. Analytical solution is $\pm 37500\ psi$ in. There is a difference between analytical and Python algorithm solution. This is because the solution of Python algorithm might not be converged yet.

8.3.2 Compression at cubic

$$\delta = \frac{pL}{EA} = \frac{-5000 \cdot 2}{588989.63 \cdot 4} = -0.004245\ in$$

Result of Python algorithm is -0.0044 inches and that of ANSYS is -0.0041 inches. Analytical solution is -0.004245 inches. All answers are similar each other.

$$\sigma = \frac{p}{A} = \frac{-5000}{4} = -1250\ psi$$

Result of Python algorithm is -1100 psi and that of ANSYS is -1053.8 psi. Analytical solution is $-1250\ psi$ in. There is a difference between analytical and Python algorithm solution. The solution of Python algorithm is expected not to be converged yet. But its difference is much smaller than that of bending situation.

8.4 Validation

Experimental data about the compression at a cubic are provided by Al muatasim al nadabi. In his original data, strain-stress graph is not linear elasticity. At the beginning and end of the strain-stress graph, its behavior is nonlinear. Since current FEM algorithm is designed by linear elasticity theory, there are huge difference between real data and simulation. So, linear strain-stress graph is generated based on typical Young's modulus of the data. Youngs modulus is 429717 psi. Poisson's ratio data are not given. So, it is assumed as 0.3.

8.4.1 Displacement (z-axis)

	Python FEM	ANSYS	Original data	Linearized data
1036 lbf	0.0012	0.00118	0.012	0.00121

3364 lbf	0.004	0.00383	0.0203	0.00391
5560 lbf	0.0067	0.00633	0.0247	0.00647
8610 lbf	0.01	0.0098	0.0297	0.0100
11059 lbf	0.013	0.0126	0.0335	0.0129
13110 lbf	0.016	0.0150	0.0368	0.0153
16708 lbf	0.02	0.019	0.0442	0.0194
17557 lbf	0.021	0.020	0.044	0.0204

<Table1. Displacement data>

In table 1, Python FEM is the algorithm designed system at the project. The displacements of Python FEM, ANSYS, and linearized data are almost same. But original data's values are three to ten times higher than them. This trend is observed, because original behavior is not totally linear.

8.4.2 Stress (z-axis)

	Python algorithm	ANSYS	Original data	Linearized data
1036 lbf	230	216.93	329	259
3364 lbf	950	704.38	1070.3	841
5560 lbf	1300	1164.2	1769.8	1390
8610 lbf	1900	1802.8	2740.642	2152.5
11059 lbf	2500	2315.6	3520	2764.75
13110 lbf	3000	2745.1	4173.033	3277.5
16708 lbf	3800	3498.5	5318.309	4177
17557 lbf	4000	3676.2	5588.554	4389.25

<Table2. Stress data>

In table 2, Python FEM is the algorithm designed system at the project. Original data's stress value is larger than linearized data. It is very obvious, because original data's displacement is larger than that of linearized data. The stress value of both Python FEM and ANSYS are smaller than linearized data. Even, there are about 10% difference between them. It would be caused by different number of elements.

9 Future work

First, it is very difficult for users to define boundary conditions on the system. Users are required to have both mechanical and computer science knowledge to define them. So, it must be improved. Second, computational speed is not very good. Parallel algorithm and numerical linear algebra knowledge will be useful to improve it. Finally, the system causes memory problem due to the large size of stiffness matrix. In the case of 20 by 20 by 20 system, the size of stiffness matrix is 24000 by 24000. Sparse matrix solving method will be applied to reduce the size of the matrix.

10 Conclusion

Designed Python FEM works very well about bending, compression and torsion. Its solutions show same behavior/trend with those of commercial software such as ANSYS. At both bending and compression situations, Python algorithm shows very close answers with analytical

solutions. But its results about displacement and stresses show differences with original experiment data due to they are not linear elasticity. In order to solve the situation, the data is linearized with typical Young's modulus. The results of the algorithm show very similar solutions with linearized data. Therefore, it is possible to say results of both verification and validation are very reasonable. But it is not perfect FEM system yet. For example, it is not easy to define boundary conditions. Unlike commercial software, it is required for users to have both mechanical and computer science knowledge to defined boundary conditions on the system. Its computation speed is much slower than that of commercial software. In addition, it causes memory problem, if the number of nodes for each axis are over thirty. They will be solved with parallel algorithm and numerical linear algebra knowledge.

11. Reference

https://www.brown.edu/Departments/Engineering/Courses/En123/Exams/SGquizes/SG_QUIZ_11.htm

https://physics.stackexchange.com/questions/357082/euler-bernoulli-equation-for-a-cantilever-strained-by-a-force

Experimental data from Al muatasim al nadabi

Appendix A Python Code

```
import numpy as np
from collections import defaultdict
import matplotlib.pyplot as plt
import itertools
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as tri
```

```
def elasticity_matrix(E,nu):
    D = (E/((1+nu)*(1-2*nu)))*np.array([[1-nu,nu,nu,0,0,0],
                                        [nu,1-nu,nu,0,0,0],
                                        [nu,nu,1-nu,0,0,0],
                                        [0,0,0,(1-2*nu)/2,0,0],
                                        [0,0,0,0,(1-2*nu)/2,0],
                                        [0,0,0,0,0,(1-2*nu)/2]])

    return D
```

```
def shape_function(xi):
    N = np.zeros((1,8))

    N[0,0] = (1/8)*(1-xi[0])*(1-xi[1])*(1-xi[2])
    N[0,1] = (1/8)*(1+xi[0])*(1-xi[1])*(1-xi[2])
    N[0,2] = (1/8)*(1+xi[0])*(1+xi[1])*(1-xi[2])
    N[0,3] = (1/8)*(1-xi[0])*(1+xi[1])*(1-xi[2])
    N[0,4] = (1/8)*(1-xi[0])*(1-xi[1])*(1+xi[2])
    N[0,5] = (1/8)*(1+xi[0])*(1-xi[1])*(1+xi[2])
    N[0,6] = (1/8)*(1+xi[0])*(1+xi[1])*(1+xi[2])
    N[0,7] = (1/8)*(1-xi[0])*(1+xi[1])*(1+xi[2])

    Nq = np.array([[N[0,0],0,0,N[0,1],0,0,N[0,2],0,0,N[0,3],0,0,N[0,4],0,0,N[0,5],0,0,N[0,6],0,0,N[0,7],0,0],
                   [0,N[0,0],0,0,N[0,1],0,0,N[0,2],0,0,N[0,3],0,0,N[0,4],0,0,N[0,5],0,0,N[0,6],0,0,N[0,7],0],
                   [0,0,N[0,0],0,0,N[0,1],0,0,N[0,2],0,0,N[0,3],0,0,N[0,4],0,0,N[0,5],0,0,N[0,6],0,0,N[0,7]]])

    return N,Nq
```

```
def shape_gradient(xi):
    N_der = np.zeros((3,8))

    N_der[0,0] = -(1/8)*(1-xi[1])*(1-xi[2])
    N_der[0,1] = (1/8)*(1-xi[1])*(1-xi[2])
    N_der[0,2] = (1/8)*(1+xi[1])*(1-xi[2])
    N_der[0,3] = -(1/8)*(1+xi[1])*(1-xi[2])
    N_der[0,4] = -(1/8)*(1-xi[1])*(1+xi[2])
    N_der[0,5] = (1/8)*(1-xi[1])*(1+xi[2])
    N_der[0,6] = (1/8)*(1+xi[1])*(1+xi[2])
    N_der[0,7] = -(1/8)*(1+xi[1])*(1+xi[2])

    N_der[1,0] = -(1/8)*(1-xi[0])*(1-xi[2])
    N_der[1,1] = -(1/8)*(1+xi[0])*(1-xi[2])
    N_der[1,2] = (1/8)*(1+xi[0])*(1-xi[2])
    N_der[1,3] = (1/8)*(1-xi[0])*(1-xi[2])
    N_der[1,4] = -(1/8)*(1-xi[0])*(1+xi[2])
    N_der[1,5] = -(1/8)*(1+xi[0])*(1+xi[2])
    N_der[1,6] = (1/8)*(1+xi[0])*(1+xi[2])
    N_der[1,7] = (1/8)*(1-xi[0])*(1+xi[2])

    N_der[2,0] = -(1/8)*(1-xi[0])*(1-xi[1])
    N_der[2,1] = -(1/8)*(1+xi[0])*(1-xi[1])
    N_der[2,2] = -(1/8)*(1+xi[0])*(1+xi[1])
    N_der[2,3] = -(1/8)*(1-xi[0])*(1+xi[1])
    N_der[2,4] = (1/8)*(1-xi[0])*(1-xi[1])
    N_der[2,5] = (1/8)*(1+xi[0])*(1-xi[1])
    N_der[2,6] = (1/8)*(1+xi[0])*(1+xi[1])
    N_der[2,7] = (1/8)*(1-xi[0])*(1+xi[1])

    return N_der
```

```python
def Jocobian(node_coors,xi):
    N_der = shape_gradient(xi)
    J = N_der.dot(node_coors)
    J_inv = np.linalg.inv(J)
    detJ = np.linalg.det(J)
    return J, J_inv, detJ
```

```python
def quad_B_matrix(node_coors,xi):
    N_der = shape_gradient(xi)
    J,J_inv,detJ = Jocobian(node_coors,xi)

    B_vals = J_inv.dot(N_der)

    B = np.array([[B_vals[0,0],0,0,B_vals[0,1],0,0,B_vals[0,2],0,0,B_vals[0,3],0,0,
                   B_vals[0,4],0,0,B_vals[0,5],0,0,B_vals[0,6],0,0,B_vals[0,7],0,0],
                  [0,B_vals[1,0],0,0,B_vals[1,1],0,0,B_vals[1,2],0,0,B_vals[1,3],0,0,
                   B_vals[1,4],0,0,B_vals[1,5],0,0,B_vals[1,6],0,0,B_vals[1,7],0],
                  [0,0,B_vals[2,0],0,0,B_vals[2,1],0,0,B_vals[2,2],0,0,B_vals[2,3],0,0,
                   B_vals[2,4],0,0,B_vals[2,5],0,0,B_vals[2,6],0,0,B_vals[2,7]],
                  [B_vals[1,0],B_vals[0,0],0,B_vals[1,1],B_vals[0,1],0,B_vals[1,2],
                   B_vals[0,2],0,B_vals[1,3],B_vals[0,3],0,B_vals[1,4],B_vals[0,4],
                   0,B_vals[1,5],B_vals[0,5],0,B_vals[1,6],B_vals[0,6],0,B_vals[1,7],B_vals[0,7],0],
                  [0,B_vals[2,0],B_vals[1,0],0,B_vals[2,1],B_vals[1,1],
                   0,B_vals[2,2],B_vals[1,2],0,B_vals[2,3],B_vals[1,3],
                   0,B_vals[2,4],B_vals[1,4],0,B_vals[2,5],B_vals[1,5],0,B_vals[2,6],B_vals[1,6],0,B_vals[2,7],B_vals[1,7]],
                  [B_vals[2,0],0,B_vals[0,0],B_vals[2,1],0,B_vals[0,1],B_vals[2,2],0,B_vals[0,2]
                   ,B_vals[2,3],0,B_vals[0,3],B_vals[2,4],0,B_vals[0,4],B_vals[2,5],0,B_vals[0,5],
                   B_vals[2,6],0,B_vals[0,6],B_vals[2,7],0,B_vals[0,7]]])
    return B
```

```python
def gauss(gauss_order):
    if gauss_order==1:
        gp = np.array([0])
        gw = np.array([2])
    elif gauss_order==2:
        gp = np.array([-np.sqrt(1/3),np.sqrt(1/3)])
        gw = np.array([1,1])
    elif gauss_order==3:
        gp = np.array([0,-np.sqrt(3/5),np.sqrt(3/5)])
        gw = np.array([8/9,5/9,5/9])
    return gp,gw
```

```python
def quad_stiffness(node_coors,E,nu,gauss_order):
    #Init k
    k = np.zeros((24,24))
    #Define D
    D = elasticity_matrix(E,nu)

    #Define gauss points (gp) and gauss weights (gw)
    gp,gw = gauss(gauss_order)

    #Get k matrix
    for i in range(len(gp)):
        for j in range(len(gp)):
            for l in range(len(gp)):
                xi = [gp[i],gp[j],gp[l]]
                J, J_inv, detJ = Jocobian(node_coors,xi)
                B = quad_B_matrix(node_coors,xi)
                k += gw[i]*gw[j]*gw[l]*detJ*(B.transpose().dot(D.dot(B)))
    return k
```

```python
def quad_force(gauss_order,node_coors,q):
    #Init f vector
    f = np.zeros(24)

    #Define gauss points (gp) and gauss weights (gw)
    gp,gw = gauss(gauss_order)

    #Geet f vector
    for i in range(len(gp)):
        for j in range(len(gp)):
            for l in range(len(gp)):
                xi = [gp[i],gp[j],gp[l]]
                J, J_inv, detJ = Jocobian(node_coors,xi)
                N,Nq = shape_function(xi)
                f+= gw[i]*gw[j]*gw[l]*Nq.transpose().dot(q)*detJ
    return f
```

```python
def solve(K,f,s,f_f,d_s):
    f_ext = f_f - np.dot(K[np.ix_(f,s)],d_s)
    d_f = np.linalg.solve(K[np.ix_(f,f)],f_ext)
    f_s = np.dot(K[np.ix_(s,f)],d_f) + np.dot(K[np.ix_(s,s)],d_s)
    return d_f,f_s
```