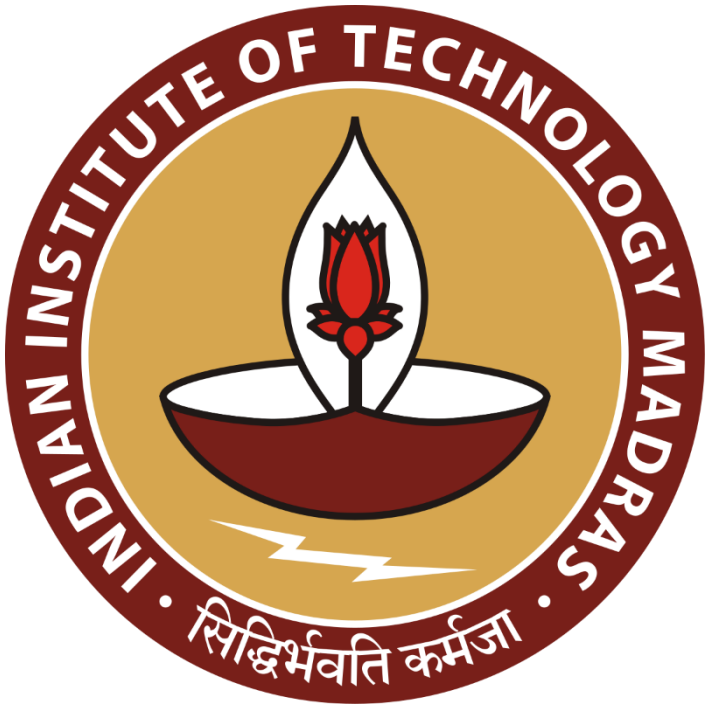
Report on Finite Element Methods



Thacker Setu Rameshbhai (ME23S027)

ME5204: Finite Element Methods

Faculty: Prof. Raju Sethuraman

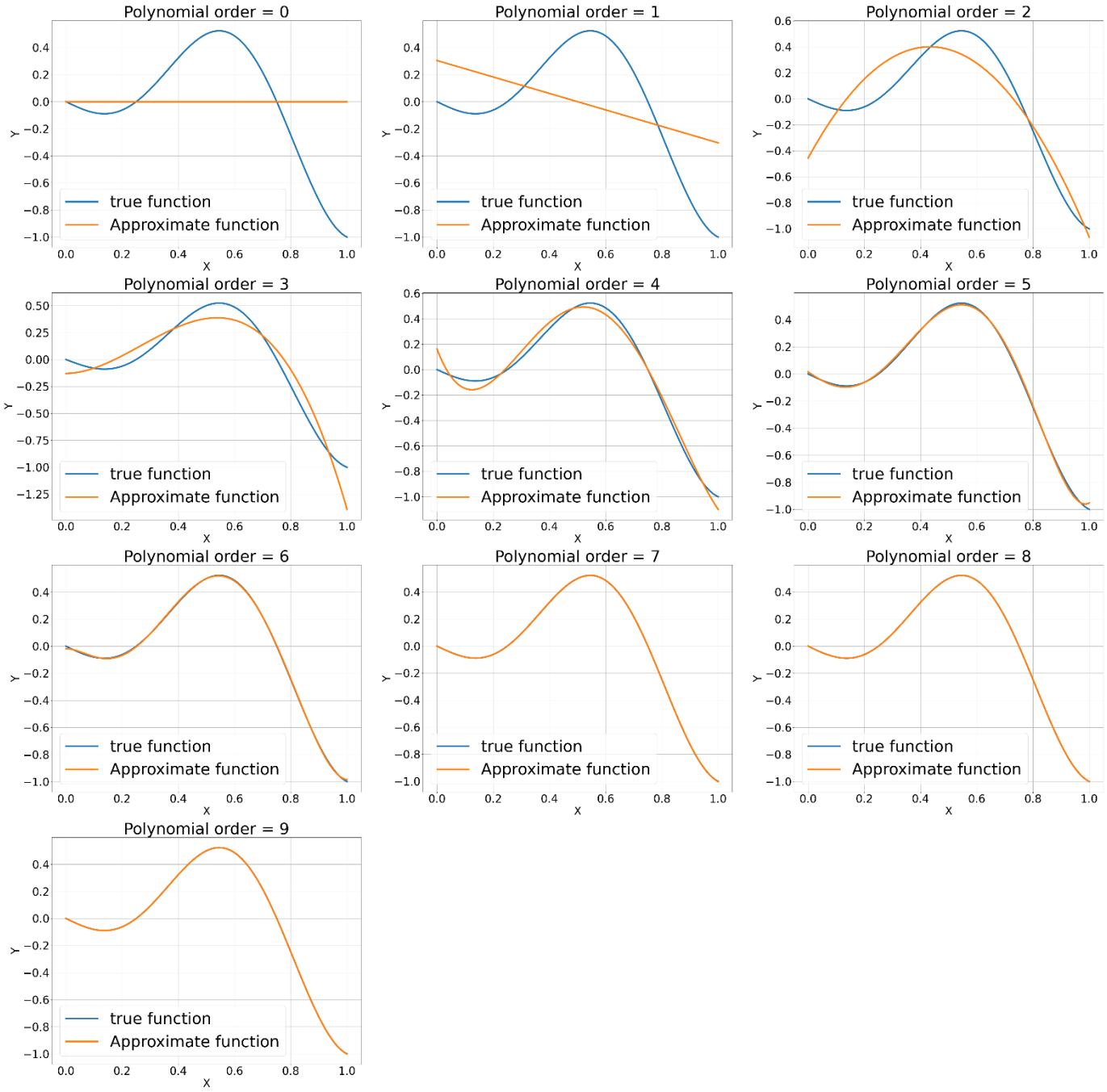
Table of Contents

Assignment 01

Code

```
#Import libraries
import numpy as np
from scipy import integrate
import matplotlib.pyplot as plt

#define the variables
l1 = 0
l2 = 1
omega = 2
poly_orders = np.arange(1,11)
graph_per_row = 3 #number of plots per row define by user
row = round((len(poly_orders)/graph_per_row))+1 #total numbers
of rows
col = graph_per_row #total numbers of columns
fig = plt.figure(figsize=(60,60))

for poly_order in poly_orders:
    #define the functions
    u = lambda x:-x*np.cos(omega*np.pi*x)
    basis = np.array([lambda x,n=i:x**n for i in
range(poly_order)])

    #define the required variables to store values
    k = np.empty((poly_order,poly_order))
    c = np.empty((poly_order))
    f = np.empty((poly_order))

    #filling of the required variables
    for i in range(poly_order):
        function_f = lambda x,i=i:u(x)*basis[i](x)
        f[i] = integrate.quad(function_f,l1,l2)[0]
        for j in range(poly_order):
            function_k = lambda
x,i=i,j=j:basis[i](x)*basis[j](x)
            k[i,j] = integrate.quad(function_k,l1,l2)[0]

    #solve the governing equation > {c} = inverse([K])@{f}
    c = np.linalg.solve(k,f)

    #define the apporximate function having polynomical basis
    w_star = lambda x:sum(c[i]*basis[i](x) for i in
range(poly_order))

    #range of X
    x = np.linspace(l1,l2,500)

    plt.subplot(row,col,poly_order)
    plt.grid()
    plt.plot(x,u(x),linewidth = 5)
    plt.plot(x,w_star(x),linewidth = 5)
```

```
    plt.title("Polynomial order = {}".format(poly_order-
1),fontsize=40)
    plt.xlabel("X",fontsize=35)
    plt.ylabel("Y",fontsize=35)
    plt.xticks(fontsize=25)
    plt.yticks(fontsize=25)
    plt.legend(["true function","Approximate
function"],fontsize=30)
plt.savefig('./Assignment01.png')
plt.show()
```

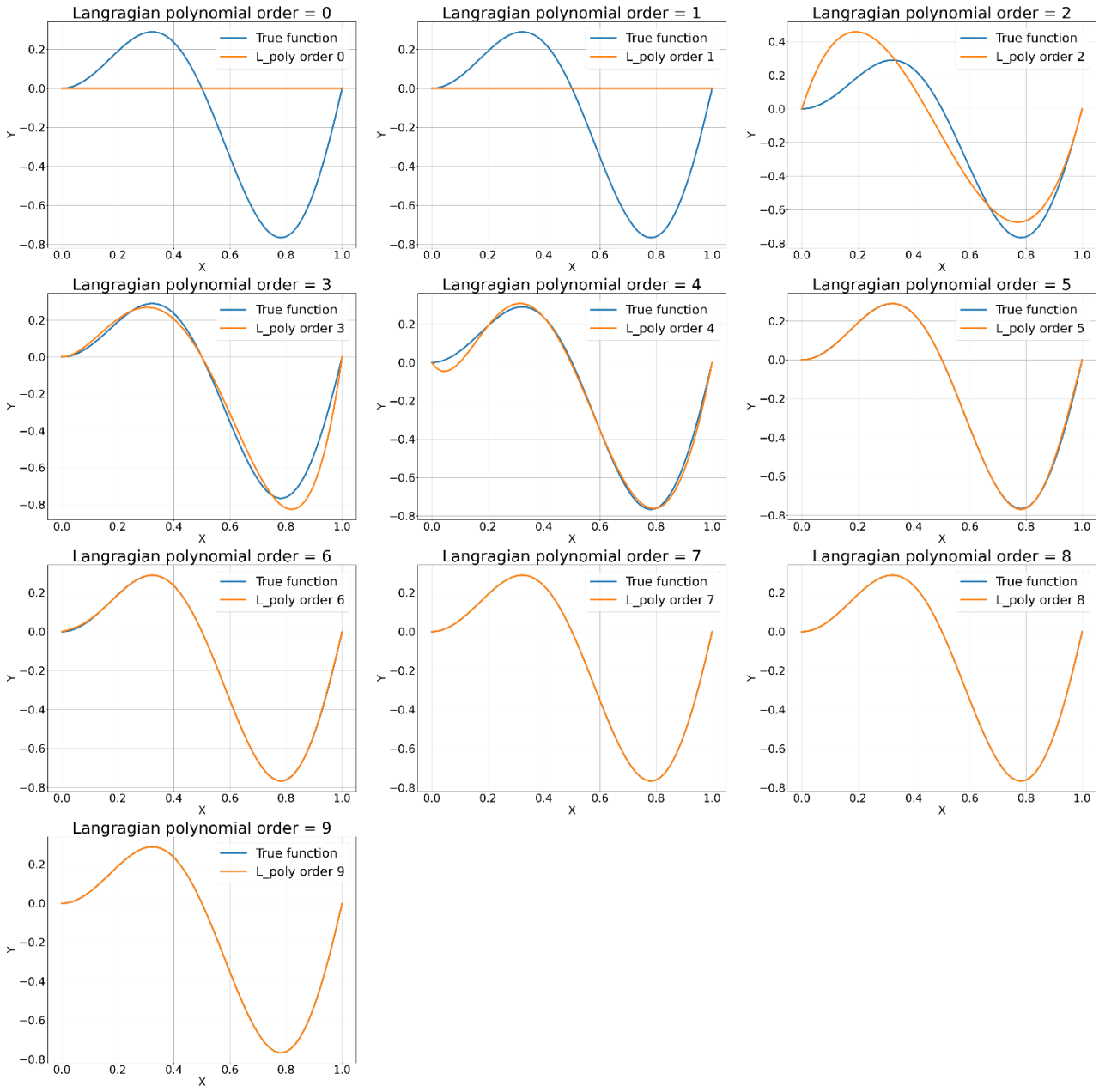


Assignment 02

Code

```
#Import libraries
import numpy as np
from scipy import integrate
import matplotlib.pyplot as plt

#define the variables
l1 = 0
l2 = 1
omega = 2

#poly_order = 4 #if polynomial order is 1 then total number of
nodes required is 1+1=2
poly_orders = np.arange(1,11)
graph_per_row = 3 #number of plots per row define by user
row = round((len(poly_orders)/graph_per_row))+1 #total numbers
of rows
col = graph_per_row #total numbers of columns
fig = plt.figure(figsize=(60,60))

for poly_order in poly_orders:
    #define the original function
    u = lambda x:x*np.sin(np.pi*omega*x)
    nodes = np.linspace(l1,l2,poly_order+1)
    x = np.linspace(l1,l2,100)

    #define the generation of langragian basis
    def langragian_basis(x,nodes,i):
        result1 = np.ones_like(x)
        for j in range(len(nodes)):
            if j!=i:
                result1*=(x-nodes[j])/(nodes[i]-nodes[j])
        return result1

    #define the Summation of product of langragian basis and
displacement for all the nodes are present
    def langragian_poly(x,nodes):
        result2 = np.zeros_like(x)
        for i in range(poly_order):
            result2 += u(nodes[i])*langragian_basis(x,nodes,i)
        return result2

    #approximate langragian function
    w_star = langragian_poly(x,nodes)

    plt.subplot(row,col,poly_order)
    plt.grid()
    plt.plot(x,u(x),linewidth = 5)
    plt.plot(x,w_star,linewidth = 5)
    plt.title("Langragian polynomial order =
{}".format(poly_order-1),fontsize=40)
    plt.xlabel("X",fontsize=35)
```

```
    plt.ylabel("Y",fontsize=35)
    plt.xticks(fontsize=25)
    plt.yticks(fontsize=25)
    plt.legend(["True function","Langragian polynomical order
{}".format(poly_order-1)],fontsize=30)
plt.savefig('./Assignment02.png')
plt.show()
```



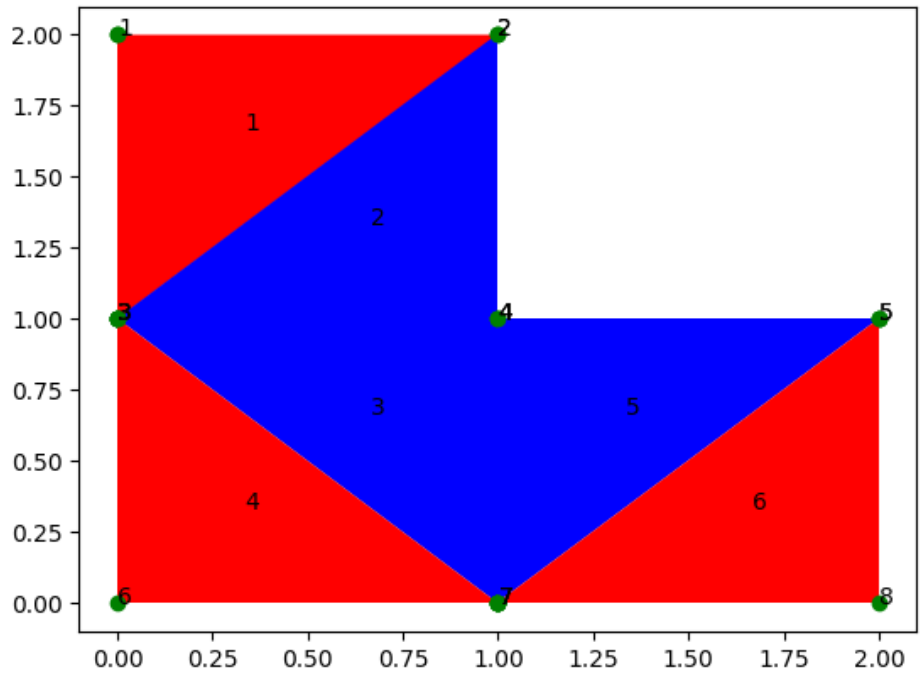
Assignment 03

Code

```
import numpy as np
import matplotlib.pyplot as plt
import copy

#load the required data
coord = np.loadtxt("./Coord.txt", dtype="float32", comments="#",
ndmin=2)
NCA = np.loadtxt("./NCA.txt", dtype="int16", comments="#",
ndmin=2)
mat_data = np.loadtxt("./MAT_DATA.txt", dtype="float32",
comments="#", ndmin=2)
print(coord)
print(NCA)

#define problem type
#problem_type = input("Press 11 > \n press 12 > \n press 13 \n
press 21 > plane stress \n press 22 > plane strain n press 3 >
for 3-D")
problem_type = 21

thickness = 1

#verify the given data
Num_nodes = coord.shape[0]-1
Num_elements = NCA.shape[0]-1
Num_mats = mat_data.shape[0]-1
print("Number of node = {}, Number of elements = {} and Number
of different material =
{}".format(Num_nodes,Num_elements,Num_mats))

for elem_idx in range(1,Num_elements+1):
    N1 = NCA[elem_idx][1]
    N2 = NCA[elem_idx][2]
    N3 = NCA[elem_idx][3]

    X1N1 = coord[N1][1]
    X1N2 = coord[N2][1]
    X1N3 = coord[N3][1]
    X2N1 = coord[N1][2]
    X2N2 = coord[N2][2]
    X2N3 = coord[N3][2]

    X1 = [X1N1,X1N2,X1N3,X1N1]
    X2 = [X2N1,X2N2,X2N3,X2N1]

    if NCA[elem_idx][4]== 1:
        plt.fill(X1,X2,facecolor = "red")
    elif NCA[elem_idx][4] == 2:
        plt.fill(X1,X2,facecolor = "blue")
    X1CG = (X1N1+X1N2+X1N3)/3.0
    X2CG = (X2N1+X2N2+X2N3)/3.0
```

```
    plt.scatter(X1,X2,color = "green")
    plt.text(X1CG,X2CG,elem_idx)
    plt.text(X1N1,X2N1,N1)
    plt.text(X1N2,X2N2,N2)
    plt.text(X1N3,X2N3,N3)
mat_data
```

```
[[0. 0. 0.]
 [1. 0. 2.]
 [2. 1. 2.]
 [3. 0. 1.]
 [4. 1. 1.]
 [5. 2. 1.]
 [6. 0. 0.]
 [7. 1. 0.]
 [8. 2. 0.]]
[[0 0 0 0 0]
 [1 1 3 2 1]
 [2 3 4 2 2]
 [3 7 4 3 2]
 [4 3 6 7 1]
 [5 5 4 7 2]
 [6 7 8 5 1]]
Number of node = 8, Number of elements = 6 and Number of different material = 2
array([[0.e+00, 0.e+00, 0.e+00],
       [1.e+00, 1.e+11, 3.e-01],
       [2.e+00, 2.e+11, 2.e-01]], dtype=float32)
```




```
#D-martix
D_store = []
for elem_idx in range(1,Num_elements+1):
    mat_num = NCA[elem_idx][4]
    E = mat_data[mat_num][1]
    PR = mat_data[mat_num][2]

    if problem_type == 21:
        #print("For plane stress condition, D matrix = ")
        const1 = E/(1-PR**2)
        D = np.multiply([[1,PR,0],[PR,1,0],[0,0,(1-
PR)/2]],const1)
    elif problem_type == 22:
        #print("For plane strain condition, D martix = ")
        const2 = E/((1+PR)*(1-2*PR))
        D = np.multiply([[1-PR,PR,0],[PR,1-PR,0],[0,0,1-
PR]],const2)
    else:
        print("Problem type is not defined. Please, check once
again.")
    print("D matrix for element {} = \n{}\n".format(elem_idx,D))
    D_store.append(D)
print(np.shape(D_store))
#D matrix > There are only 2 types.
```

D matrix for element 1 =

[[1.09890109e+11 3.29670339e+10 0.00000000e+00]

[3.29670339e+10 1.09890109e+11 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 3.84615373e+10]]

D matrix for element 2 =

[[2.08333329e+11 4.16666665e+10 0.00000000e+00]

[4.16666665e+10 2.08333329e+11 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 8.33333314e+10]]

D matrix for element 3 =

[[2.08333329e+11 4.16666665e+10 0.00000000e+00]

[4.16666665e+10 2.08333329e+11 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 8.33333314e+10]]

D matrix for element 4 =

[[1.09890109e+11 3.29670339e+10 0.00000000e+00]

[3.29670339e+10 1.09890109e+11 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 3.84615373e+10]]

D matrix for element 5 =

[[2.08333329e+11 4.16666665e+10 0.00000000e+00]

[4.16666665e+10 2.08333329e+11 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 8.33333314e+10]]

D matrix for element 6 =

[[1.09890109e+11 3.29670339e+10 0.00000000e+00]

[3.29670339e+10 1.09890109e+11 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 3.84615373e+10]]

(6, 3, 3)

```
DOF_PN = 2 #DOF per node = 2; X1 and X2
Num_nodes=8
Total_DOF = DOF_PN*Num_nodes
#print("Total degrees of freedom = {}".format(Total_DOF))
GSTIFF = np.zeros([Total_DOF,Total_DOF])
```

```
#print("Global stifness matrix = \n{}\n".format(GSTIFF))
#print("Force vector = {}".format(F))

#B-matrix
B_store = []
for elem_idx in range(1,Num_elements+1):
    N1 = NCA[elem_idx,1]
    N2 = NCA[elem_idx,2]
    N3 = NCA[elem_idx,3]

    X1N1 = coord[N1,1]
    X1N2 = coord[N2,1]
    X1N3 = coord[N3,1]
    X2N1 = coord[N1,2]
    X2N2 = coord[N2,2]
    X2N3 = coord[N3,2]

    two_delta_matrix =
[[1,X1N1,X2N1],[1,X1N2,X2N2],[1,X1N3,X2N3]]
    two_delta = np.linalg.det(two_delta_matrix)
    #print(two_delta)

    b1 = X2N2-X2N3
    b2 = X2N3-X2N1
    b3 = X2N1-X2N2
    v1 = X1N3-X1N2
    v2 = X1N1-X1N3
    v3 = X1N2-X1N1

    B = [[b1,0,b2,0,b3,0],[0,v1,0,v2,0,v3],[v1,b1,v2,b2,v3,b3]]
    B_store.append(B)
    #print(B)
    ESTIFF =
np.matmul(np.matmul(np.transpose(B),D_store[elem_idx-
1]),B)*(two_delta/2)*thickness

    col_idx = [[2*N1-2,2*N1-1,2*N2-2,2*N2-1,2*N3-2,2*N3-1]]
    row_idx = np.transpose(col_idx)

    GSTIFF[row_idx,col_idx] = GSTIFF[row_idx,col_idx]+ESTIFF
    #print(GSTIFF)
    #print(ESTIFF)
#print(D)
print(B)
#print(np.shape(GSTIFF),np.linalg.det(GSTIFF))
#print(np.shape(B_store))
```

[[-1.0, 0, 1.0, 0, 0.0, 0], [0, 0.0, 0, -1.0, 0, 1.0], [0.0, -1.0, -1.0, 1.0, 1.0, 0.0]]

```
load_BC = np.loadtxt("./Load_BC.txt", dtype="float32",
comments="#", ndmin=2)
#print(load_BC)
num_force_bc = load_BC.shape[0]

F = np.zeros([Total_DOF])
```

```
for idx in range(1,num_force_bc):
    node_num = int(load_BC[idx,0])
    load_type = load_BC[idx,1]
    #print(node_num)

    match load_type:
        case 1:
            load_X1 = load_BC[idx,2]
            F[2*node_num-2] = load_X1
            #print("369")
        case 2:
            load_X2 = load_BC[idx,3]
            F[2*node_num-1] = load_X2
            #print("Hariprabodham")
        case 12:
            load_X1 = load_BC[idx,2]
            F[2*node_num-2] = load_X1
            load_X2 = load_BC[idx,3]
            F[2*node_num-1] = load_X2
            #print("I am Akshar")
print("Nodal force vector = {}".format(F))
```

Nodal force vector = [0. 0. 0. 0. 0. 0. 0. 0. 0. -1000. 0. 0. 0. 0. 0. -1000.]

```
GSTIFF_mod = copy.deepcopy(GSTIFF)
Disp_BC_data = np.loadtxt("./Disp_BC_data.txt", dtype="float32",
comments="#", ndmin=2)
Num_disp_BC = Disp_BC_data.shape[0]
P = 10e16
for idx in range(Num_disp_BC):
    node_num = int(Disp_BC_data[idx,0])
    data_type = int(Disp_BC_data[idx,1])
    match data_type:
        case 1:
            X1_disp = Disp_BC_data[idx,2]
            GSTIFF_mod[2*node_num-2,2*node_num-2] = P
            F[2*node_num-2] = X1_disp*P
            #print("369")
        case 1:
            X2_disp = Disp_BC_data[idx,3]
            GSTIFF_mod[2*node_num-1,2*node_num-1] = P
            F[2*node_num-1] = X2_disp*P
            #print("Hariiii")
        case 12:
            X1_disp = Disp_BC_data[idx,2]
            X2_disp = Disp_BC_data[idx,3]
            GSTIFF_mod[2*node_num-2,2*node_num-2] = P
            GSTIFF_mod[2*node_num-1,2*node_num-1] = P
            F[2*node_num-2] = X1_disp*P
            F[2*node_num-1] = X2_disp*P
            #print("Prabodham")

#Solution
u = np.linalg.solve(GSTIFF_mod,F)
```

```
print(np.shape(B),np.shape(D),np.shape(u))
#strain = np.matmul(B,u)
#stress = np.matmul(D,strain)
F_new = np.matmul(GSTIFF,u)
print("\nCorrected nodal force vector F = \n{}\n".format(F_new))
print("Sum of all the above elements  =
{}\n".format(np.sum(F_new)))
#plt.plot(F_new)
print("Deformation matrix =  \n{} \n".format(u))
#print("Nodal force vector =  \n{} \n".format(F))

#u_2d = u.reshape(-1,2)
```

(3, 6) (3, 3) (16,)

Corrected nodal force vector F =

[1.07843128e+03 -2.00000000e+03 -1.07843128e+03 4.00000000e+03

  4.54747351e-13 0.00000000e+00 1.81898940e-12 1.81898940e-12

  4.54747351e-13 -1.00000000e+03 0.00000000e+00 -4.54747351e-13

  1.81898940e-12 -1.42108547e-12 2.27373675e-12 -1.00000000e+03]

Sum of all the above elements = 3.410605131648481e-12

Deformation matrix =

[-1.07843208e-14 2.00000148e-14 1.07843232e-14 -4.00000494e-14

 -3.55407601e-08 2.39607859e-08 -4.03598887e-08 -3.28025085e-08

 -3.63948342e-08 -1.62182017e-07 -1.27563909e-07 3.04299926e-08

 -1.34033115e-07 -3.75646735e-08 -1.47204452e-07 -1.75010682e-07]

```
#for getting strain and stress values. It looks wrong :(
strain_store,stress_store,von_mises_store = [],[],[]
for elem_idx in range(1,Num_elements+1):
    u_store = []
    #print(u)
    N1 = NCA[elem_idx,1]
    N2 = NCA[elem_idx,2]
    N3 = NCA[elem_idx,3]
    #print(np.round(u[2*N1-2],7),np.round(u[2*N1-
1],7),np.round(u[2*N2-2],7),np.round(u[2*N2-
1],7),np.round(u[2*N3-2],7),np.round(u[2*N3-1],7))
    #print(N1,N2,N3)
    u_store.append(u[2*N1-2])
    u_store.append(u[2*N1-1])
```

```
    u_store.append(u[2*N2-2])
    u_store.append(u[2*N2-1])
    u_store.append(u[2*N3-2])
    u_store.append(u[2*N3-1])
    #print(np.shape(B_store[elem_idx]),np.shape(u_store))
    strain = np.matmul(B_store[elem_idx-1],u_store)
    stress = np.matmul(D_store[elem_idx-1],strain)
    strain_store.append(strain)
    stress_store.append(stress)

    print("Strain for element {} = {}".format(elem_idx,strain))
    print("Stress for element {} = {}
Pa".format(elem_idx,stress))
    #print("Strain matrices:")
    #print(strain)
    #print("Stress matrices:")
    #print(stress)

    sigma_xx = stress[0]
    sigma_yy = stress[1]
    sigma_xy = stress[2]

    von_mises = np.sqrt((sigma_xx**2+sigma_yy**2-
sigma_xx*sigma_yy+3*sigma_xy**2))
    von_mises_store.append(von_mises)
    print("Von-mises stress for element {} = {} Pa
\n".format(elem_idx,von_mises))

#print(np.round(u,10))
#print(np.shape(strain_store),np.shape(stress_store))

plt.grid()
plt.scatter(x=range(1,Num_elements+1), y=von_mises_store,
c=von_mises_store)
plt.colorbar(label="Von-mises stress", orientation="vertical")
plt.xlabel("Elements")
plt.ylabel("Von-mises stress")
plt.title("Von-mises Stress (Pa) Vs. Elements")
plt.show()
```

Strain for element 1 = [2.15686440e-14 -2.39607659e-08 3.55406893e-08]

Stress for element 1 = [-789.91301035 -2633.05045274 1366.94954726] Pa

Von-mises stress for element 1 = 3329.0673822155422 Pa

Strain for element 2 = [-4.81912862e-09 3.28024685e-08 -1.64033949e-08]

Stress for element 2 = [362.78440556 6633.05045274 -1366.94954726] Pa

Von-mises stress for element 2 = 6879.553529282195 Pa

Strain for element 3 = [-4.81912862e-09 4.76216493e-09 3.69099323e-08]

Stress for element 3 = [-805.56157241 791.32064998 3075.82761959] Pa

Von-mises stress for element 3 = 5504.064131297851 Pa

Strain for element 4 = [-6.46920669e-09 -6.46920669e-09 2.40284826e-08]

Stress for element 4 = [-924.17238041 -924.17238041 924.17238041] Pa

Von-mises stress for element 4 = 1848.3447608145686 Pa

Strain for element 5 = [3.96505453e-09 4.76216493e-09 -3.57062822e-08]

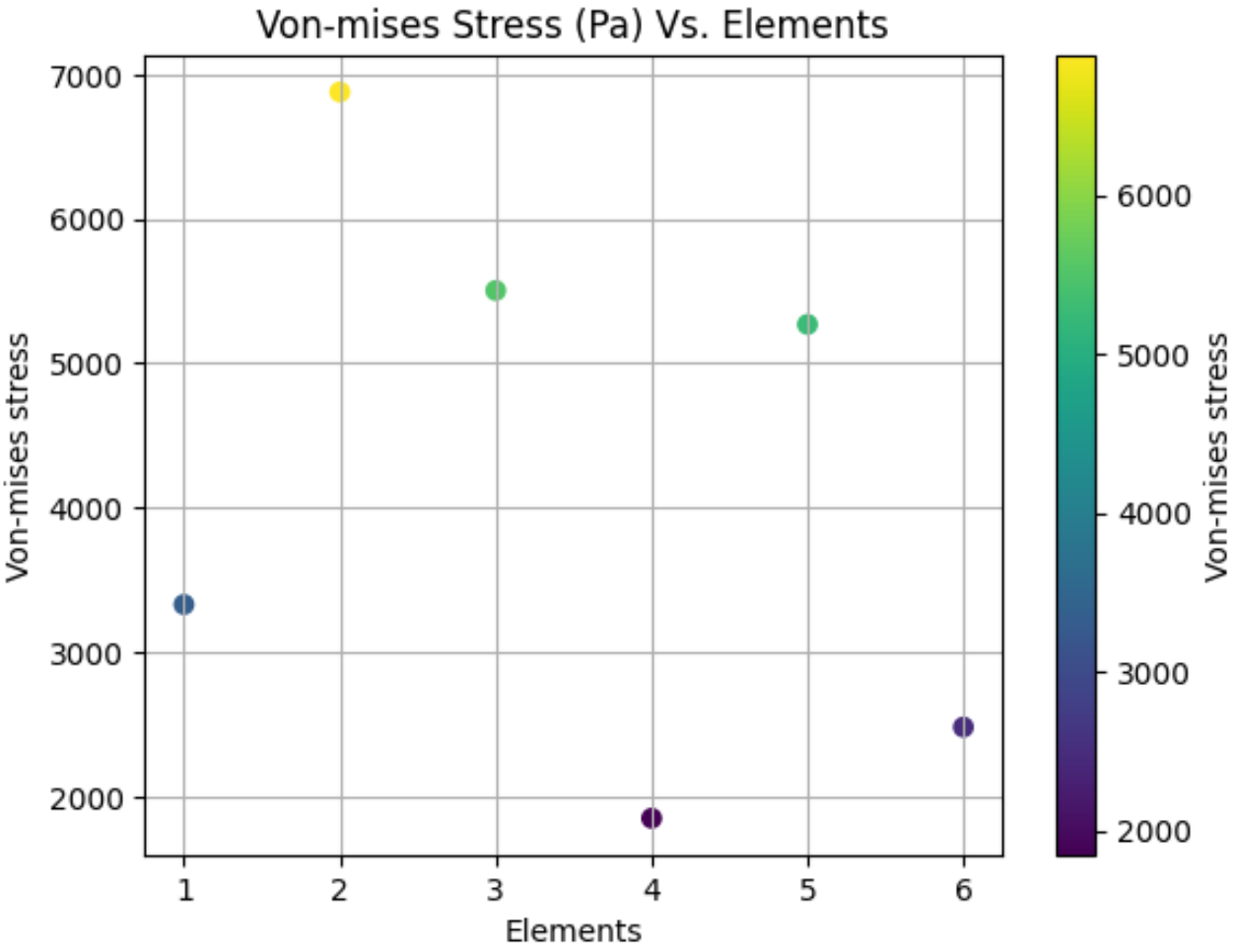
Stress for element 5 = [1024.47654935 1157.32827978 -2975.52345065] Pa

Von-mises stress for element 5 = 5269.205317778571 Pa

Strain for element 6 = [-1.31713362e-08 1.28286645e-08 -2.66363911e-08]

Stress for element 6 = [-1024.47654935   975.52345065 -1024.47654935] Pa

Von-mises stress for element 6 = 2479.769284021626 Pa


Von-mises Stress (Pa) Vs. Elements
Von-mises stress
Elements
7000
6000
5000
4000
3000
2000
1
2
3
4
5
6
Von-mises stress

Abaqus

Here, I have performed 2 FEA Simulations to replicate our original problem. Here I am going to share the procedure to do run FEA simulation in Abaqus.

1. Create a part of given dimensions
2. Assign property to part after creating sections in part (fig. 1)
3. Create a mesh in part according to problem statement. In this case, it is static FEA analysis and plane stress problem (fig. 3 and 4)

Note: In Abaqus, we couldn't provide too much coarse mesh size which we have used for our coding problem. So, I have use mesh size of 0.5 m and 0.25 m respectively for two FEA simulations.

4. Apply loading condition at nodes (fig. 2)
5. Define boundary conditions at the node for fixed joint (fig. 2)
6. Start the job and export the desired result.

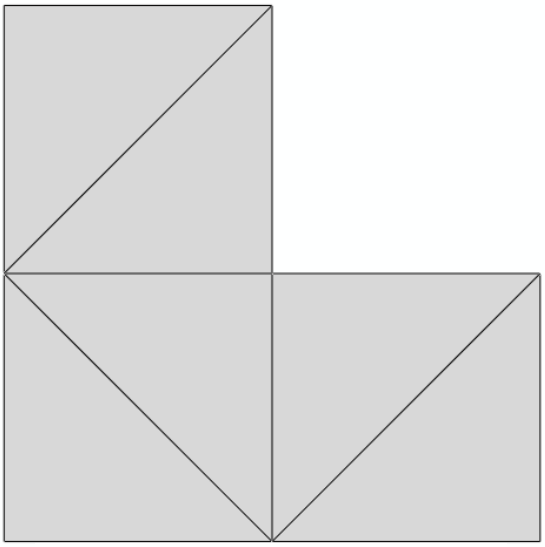


Figure 1: Part having sections for assigning different property

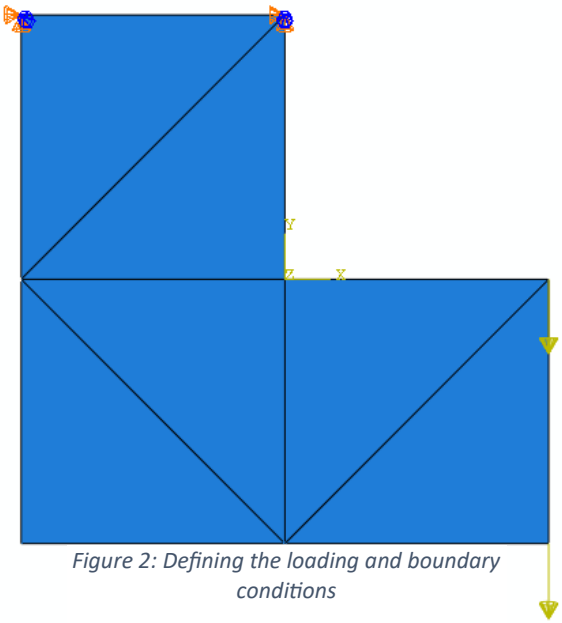


Figure 2: Defining the loading and boundary conditions

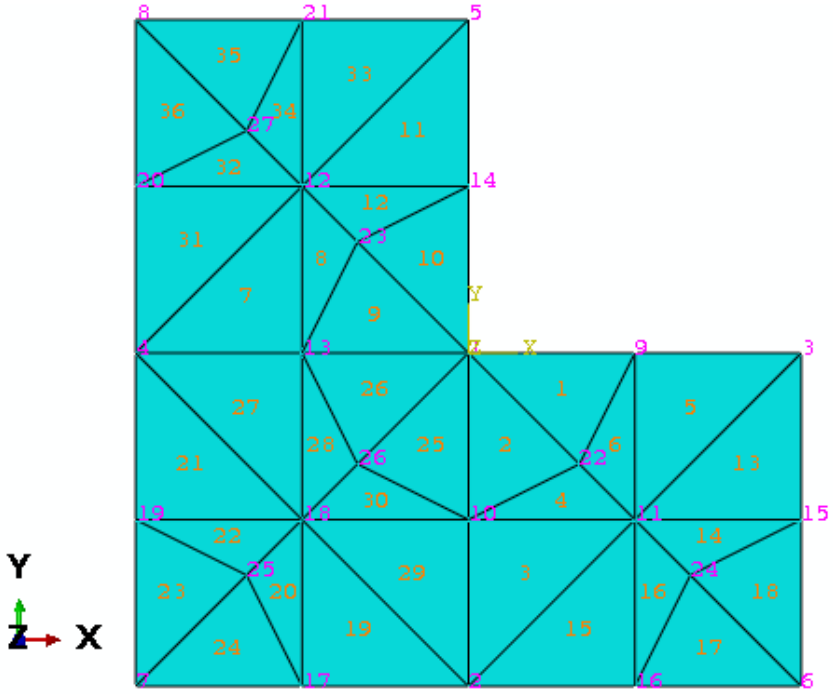


Figure 3: Node and element order for mesh size = 0.5 m

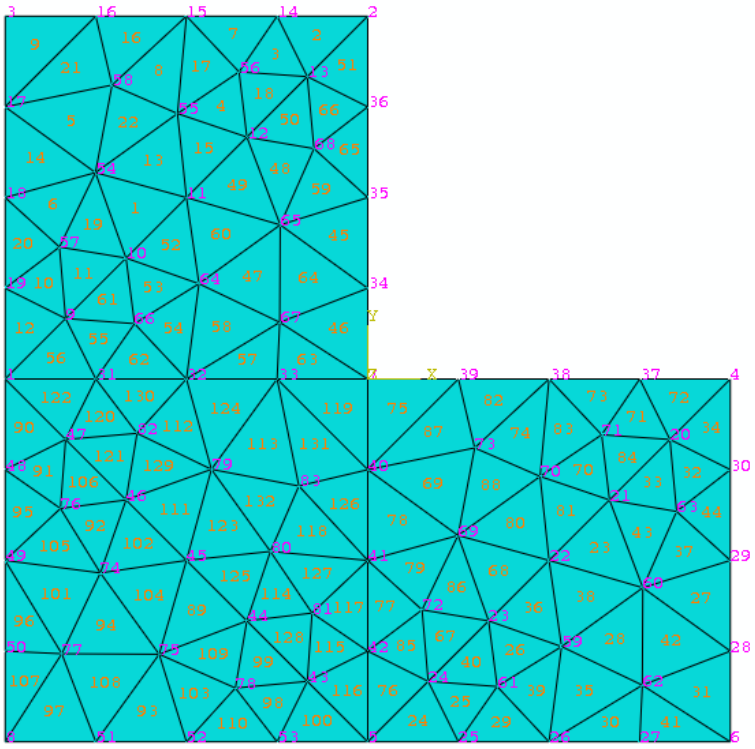


Figure 4: Node and element order for mesh size = 0.25 m

Deformation study

Comparison of deformation between code Vs. Abaqus (Mesh size = 0.5 m)

Node Label	Abaqus-U1	Code-U1	Error-U1	Abaqus-U2	Code-U2	Error-U2
1	0.00E+00	0.00E+00	0%	0.00E+00	0.00E+00	0%
2	0.00E+00	0.00E+00	0%	0.00E+00	0.00E+00	0%
3	-9.43E-08	-1.34E-07	42%	8.45E-08	-3.76E-08	144%
4	-8.16E-08	-4.04E-08	51%	-8.66E-08	-3.28E-08	62%
5	-5.65E-08	-3.64E-08	36%	-4.24E-07	-1.62E-07	62%
6	-3.12E-07	-1.28E-07	59%	1.11E-07	3.04E-08	73%
7	-3.37E-07	-3.55E-08	89%	-1.07E-07	2.40E-08	122%
8	-3.71E-07	-1.47E-07	60%	-4.32E-07	-1.75E-07	60%
		Total error	337%		Total error	523%

Comparison of deformation between code Vs. Abaqus (Mesh size = 0.25 m)

Node Label	Abaqus-U1	Code-U1	Error	Abaqus-U2	Code-U2	Error2
1	0.00E+00	0.00E+00	0%	0.00E+00	0.00E+00	0%
2	0.00E+00	0.00E+00	0%	0.00E+00	0.00E+00	0%
3	-1.34E-07	-1.34E-07	0%	1.18E-07	-3.76E-08	132%
4	-1.18E-07	-4.04E-08	66%	-1.42E-07	-3.28E-08	77%
5	-7.84E-08	-3.64E-08	54%	-6.08E-07	-1.62E-07	73%
6	-4.48E-07	-1.28E-07	72%	1.49E-07	3.04E-08	80%
7	-4.74E-07	-3.55E-08	93%	-1.64E-07	2.40E-08	115%
8	-5.19E-07	-1.47E-07	72%	-6.12E-07	-1.75E-07	71%
			355%			-547%

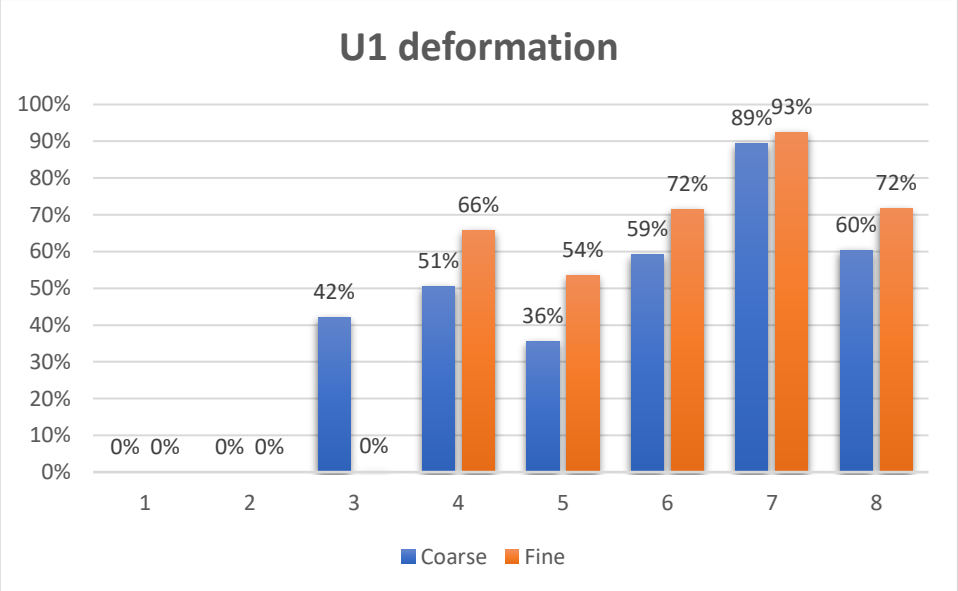


Figure 5: Deformation U1 for coarse and fine mesh

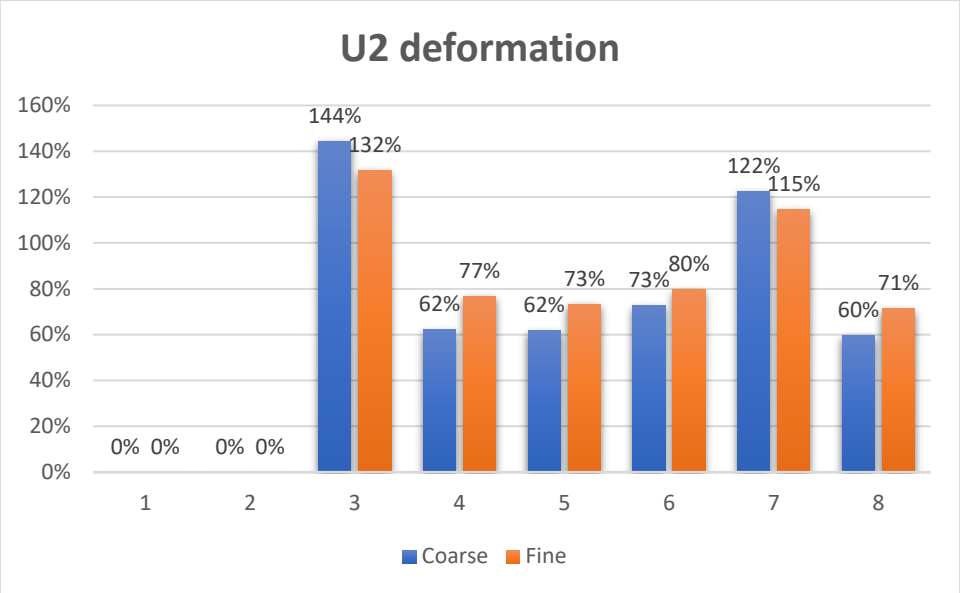


Figure 6: Deformation U2 for coarse and fine mesh

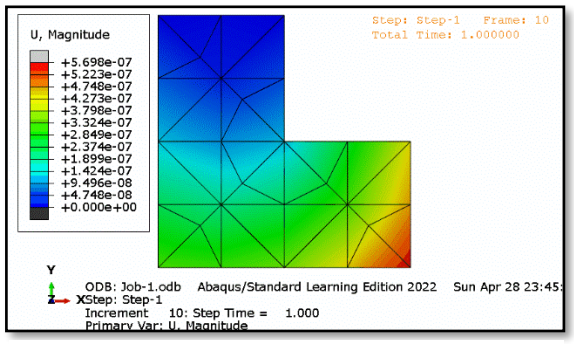


Figure 7: Resultant deformation on mesh size = 0.5 m

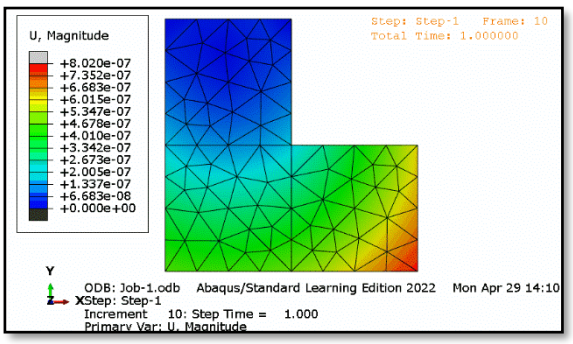


Figure 8: Resultant deformation on mesh size = 0.25 m

From fig 3 and 4, we can conclude that answer which we derived from the coding is way far from the real answer. So, as we increase mesh size from 0.5 m to 0.25 m. Our error will get increased for majority of case. Where error is defined as, | ((deformation from Abaqus – deformation from code)/deformation from code) |*100 %.

From fig 5 and 6, those are the results of Abaqus simulations with different mesh size. We couldn't compare the result directly due to mismatching of element numbers. But we could comment that our code is working very fine by observing the nature of deformation of the or direction of deformation. From fig 5 and 6, it looks like red area is the one where highest deformation is happening due to initial and boundary conditions. This area has node no. 6 as per Abaqus notation and node no. 8 as per coding problem.

Reaction forces study

Comparison of reaction forces between code Vs. Abaqus (Mesh size = 0.5 m)

Node Label	RF-RF1	Code-RF1	RF-RF2	Code-RF2
1	151.370361	1078	-2.00E+03	-2000
2	-151.370361	-1078	4.00E+03	4000
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0

Comparison of reaction forces between code Vs. Abaqus (Mesh size = 0.25 m)

Node Label	RF-RF1	Code-RF1	RF-RF2	Code-RF2
1	-92.889	1078	-2.00E+03	-2000
2	92.8891	-1078	4.00E+03	4000
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0

Von-mises stress analysis

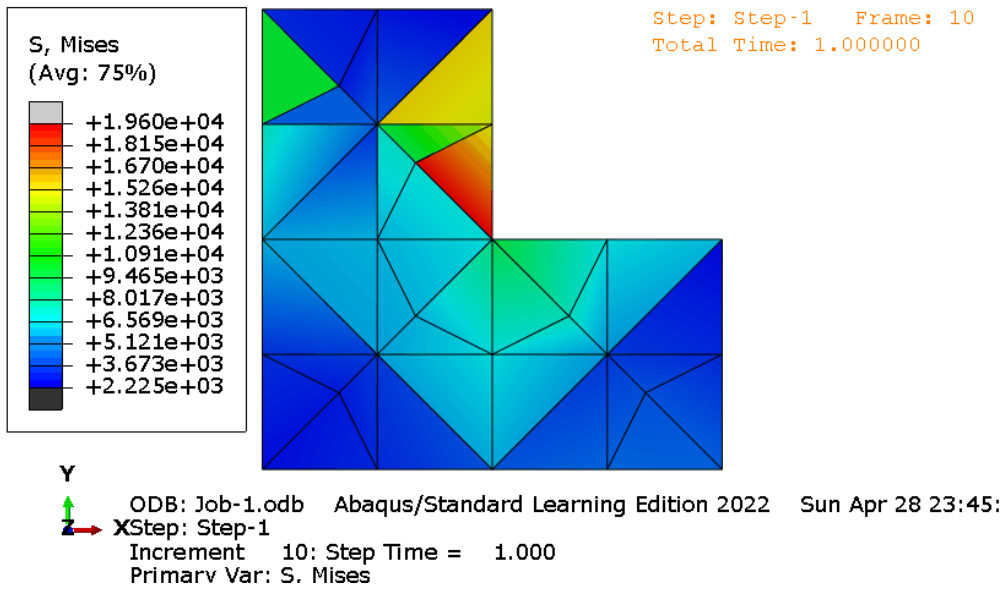


Figure 9: Von-mises stress distribution (mesh size = 0.5m)

Element Label	Node Label	X	Y	S-Mises
15	24	-3.33E-01	3.33E-01	6.81E+03
16	7	0	0	1.96E+04
16	16	0	5.00E-01	1.49E+04
16	24	-3.33E-01	3.33E-01	1.96E+04
17	2	0	1	1.57E+04
17	10	-5.00E-01	5.00E-01	1.57E+04

50_cm_mesh | 25 cm mesh | Code results | +

Figure 10: Von-mises result from mesh size = 0.5 m

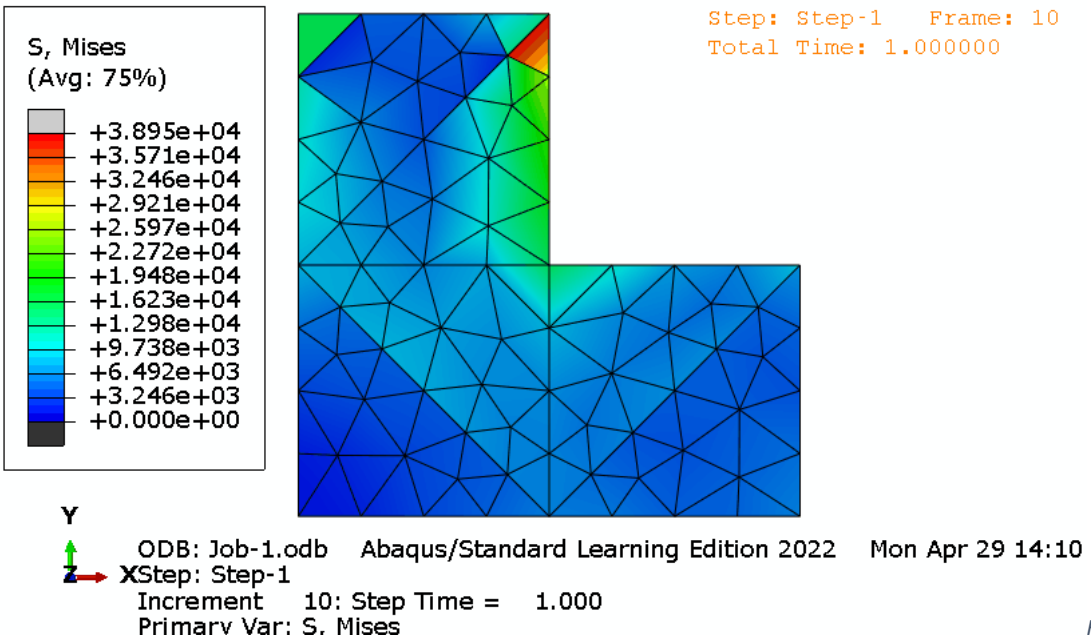


Figure 11: Von-mises stress distribution (mesh size = 0.25m)

Element Label	Node Label	X	Y	S-Mises
50	12	-3.33E-01	6.67E-01	8.46E+03
51	2	0	1	3.90E+04
51	13	-1.67E-01	8.33E-01	3.90E+04
51	36	0	7.50E-01	2.73E+04
52	10	-6.67E-01	3.33E-01	5.36E+03
52	64	-4.66E-01	2.63E-01	5.10E+03

50_cm_mesh | 25_cm_mesh | Code_results | +

Figure 12: Von-mises result from mesh size = 0.25 m

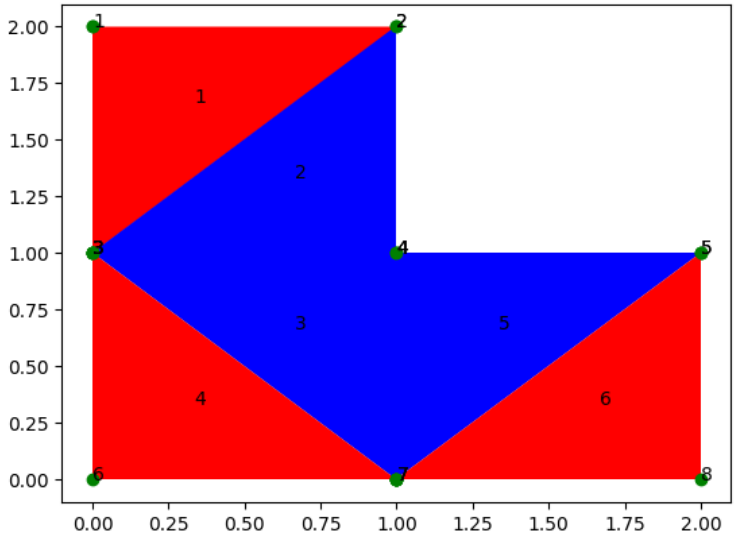


Figure 13: Node and element notation as per code

Element	Von-mises stress
1	3329.067382
2	6879.553529
3	5504.064131
4	1848.344761
5	5269.205318
6	2479.769284

50_cm_mesh | 25_cm_mesh | Code_results | +

Figure 13: Von-mises result from code results

Fig 9 and 10 are the result generated from abaqus with mesh size 0.5m and 0.25m respectively. After simulating the FEA analysis, I've exported the necessary data to excel for post processing analysis. In fig 11, it's shows that element 16 have highest von-misses stress. Similarly in fig 12, it's shows that element 51 have highest von-misses stress compared to other elements. In code result, element 2 have highest von-misses stress compared to other elements. These result which generates from the abaqus is having same nature of vonmises stress which generates from code results. Element 2 in code result, element 16 in mesh size of 0.5 m simulation and element 51 in mesh size 0.25 m simulation are excepted to high stress as per the concept of strength of material. As we increase the mesh size in analysis we approch towards the correct area where high stress is being generated.

Conclusion

So, we can conclude that finite element analysis using python is being use as backend in all the FEM solver like Abaqus and Ansys. As we can see in fig.5 and fig.6, the result derived from codes are away from the actual result. As increasing mesh size from 0.5m to 0.25m we are getting the results which are more realistically correct. So, finite element methods are one of the way to find the deformation. From deformation we can derived all the required quantities like stress, strain, von-mises stress for each element. This process is called post-processiong which we have done in our code also.

Click here https://github.com/SetuThacker/FEM_Assignments to get a pdf of the report.