Two-Dimensional Finite Element Analysis of Solid Elastic Structures

COE 321K Computational Methods for Structural Analysis

May 1, 2025

Vaidehi Joshi ***vj3993***

Contents

1 Introduction **3**

2 Geometry and FE Mesh **3**

2.1 Input and Output Features 3

2.2 Normalization Scheme 5

3 Computational Solution **6**

3.1 Plate Deformation 6

3.2 Stresses 7

4 Conclusion **11**

5 Appendices **12**

A Mesh Visualization 12

B Axial Stress Plots 14

C Finite Element Software 17

1 Introduction

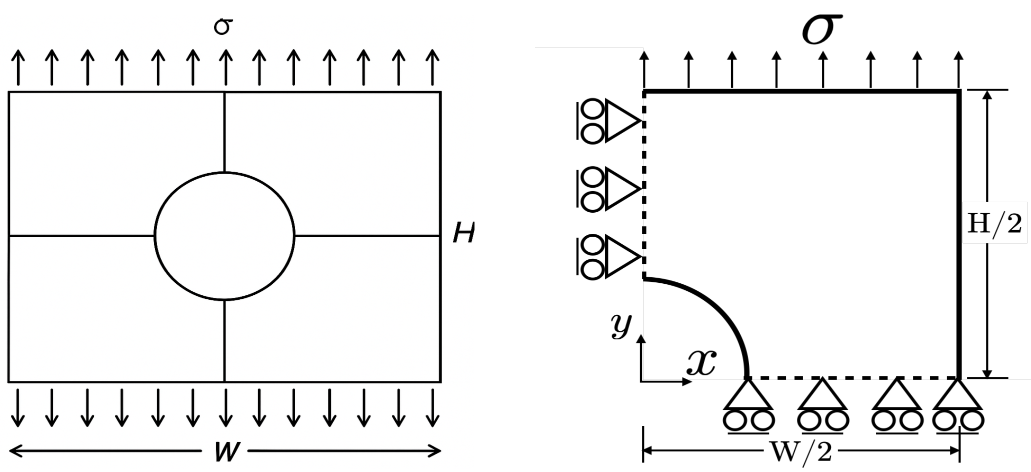
This work presents a finite element (FE) elastic stress analysis of a rectangular plate ($W \times H \times t$) with a central circular hole (radius R) subject to applied stress on the edges. The thickness t of the plate is considered to be much smaller than height H and width W, reducing the problem to two dimensions. Thus, the FE analysis adopts a plane stress approximation with the following elasticity:

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

where E is the Young's modulus and ν is the Poisson's ratio.

2 Geometry and FE Mesh

Figure 1a shows the schematic of the plate with applied loading and Figure 1b shows the FE domain modeled with symmetry boundary conditions.



(a) Geometry and loading

(b) FE domain

Figure 1: Rectangular plate with a circular hole subjected to uniform traction

Due to the symmetry of the structure and loading, in this work only the first quadrant is analyzed.

The domain in Figure 1b is discretized using three-noded constant strain triangular finite elements. Each triangular element has six degrees of freedom (DOFs) with two displacements at each node – u_1 along the x-direction and u_2 along the y-direction.

2.1 Input and Output Features

A Python-based FE solver is implemented, which reads input files specifying element nodal coordinates, element connectivity, and displacement, as well as nodal force boundary conditions. These files are provided to the user.

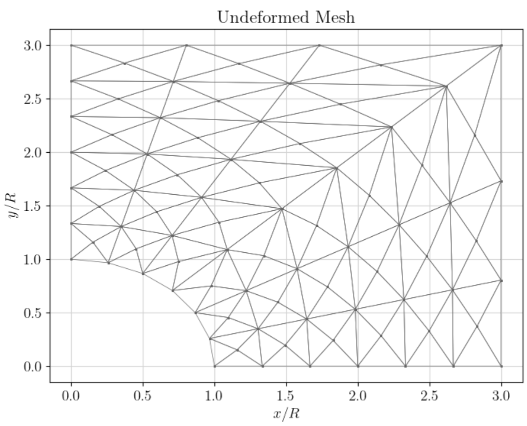
The plate geometry is specified as follows. The ratios H/R and W/R are both 3, with $R = 1$. The displacement (u_1, u_2) and traction (t_y) boundary conditions are as follows:

1. At $y = 0$, $u_2(x) = 0$

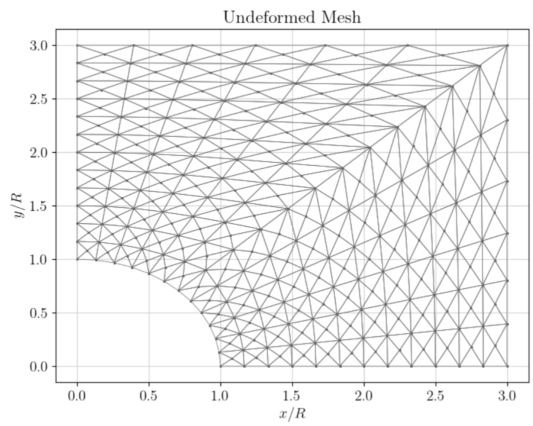
2. At $x = 0$, $u_1(y) = 0$

3. At $y = H/2$, $t_y = \sigma$

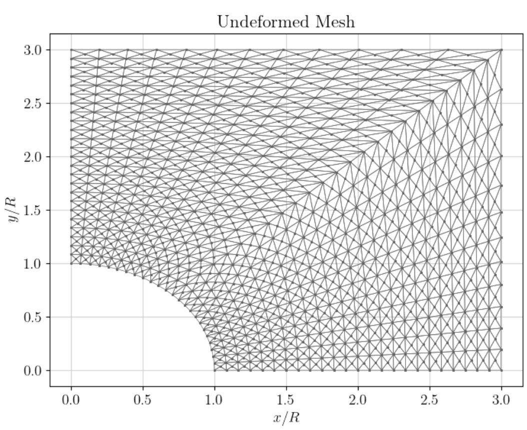
The traction stress σ per unit thickness is applied as equivalent consistent nodal forces whose magnitudes depend on the mesh density along the edge. The different FE meshes (shown in Figure 2) are referred to as M_6 (6 divisions per axis), M_{12} (12 divisions per axis), M_{24} (24 divisions per axis), and M_R (non-uniform grid, refined near the hole and coarser towards the edges).



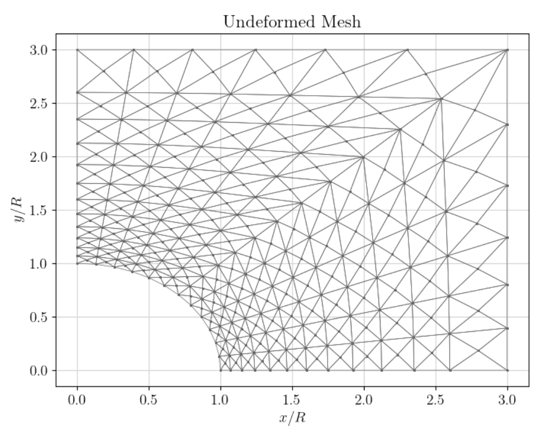
(a) Undeformed mesh M_6



(b) Undeformed mesh M_{12}



(c) Undeformed mesh M_{24}



(d) Undeformed mesh M_R

Figure 2: Undeformed mesh geometry for each level of refinement

Table 1 lists the total number of nodes and elements for each FE mesh.

Mesh	Number of Nodes	Number of Elements
M_6	85	144
M_{12}	313	576
M_{24}	1201	2304
M_R	313	576

Table 1: FE meshes

While M_{12} and M_R contain the same total number of nodes and elements (i.e., same number of DOFs), M_R is a more accurate mesh as it is designed to capture high stress variation near the hole.

2.2 Normalization Scheme

To simplify the analysis and improve generality, all quantities are normalized with respect to reference parameters, i.e., the applied stress σ (per unit thickness), plate dimensions H and W, hole radius R, Young's modulus E, and Poisson's ratio ν.

Position and Geometry

Spatial coordinates are normalized by the hole radius:

$$\tilde{x} = \frac{x}{R}, \quad \tilde{y} = \frac{y}{R}$$

Plate dimensions are similarly expressed as ratios:

$$\frac{H}{R}, \quad \frac{W}{R}$$

Stress and Strain Fields

Since the governing equations are linear in σ, all field quantities scale proportionally with σ. Normalized stress is defined as:

$$\tilde{\sigma}_{ij} = \frac{\sigma_{ij}}{\sigma}$$

Poisson's ratio ν is already dimensionless and remains unchanged.

Strain and Displacement

Assuming linear elasticity $\sigma_{ij} \sim E\varepsilon_{ij}$, normalized strain is defined as:

$$\tilde{\varepsilon}_{ij} = \frac{E\varepsilon_{ij}}{\sigma}$$

Given that $\varepsilon_{ij} \sim \partial u_i/\partial x_j$ and $E\varepsilon_{ij}\ \sigma \sim E/\sigma R\, \partial u_i/\partial \left(x_j/R\right)$, the normalized displacement can be written as:

$$\tilde{u}_i = \frac{Eu_i}{\sigma R}$$

A summary of these normalizations is shown in Table 2.

Quantity	Normalized Form
Spatial coordinates $(x,\, y)$	$\tilde{x} = \dfrac{x}{R}, \quad \tilde{y} = \dfrac{y}{R}$
Plate in-plane dimensions $(H,\, W)$	$\dfrac{H}{R}, \quad \dfrac{W}{R}$
Stress (σ_{ij})	$\tilde{\sigma}_{ij} = \dfrac{\sigma_{ij}}{\sigma}$
Strain (ε_{ij})	$\tilde{\varepsilon}_{ij} = \dfrac{E\varepsilon_{ij}}{\sigma}$
Displacement (u_i)	$\tilde{u}_i = \dfrac{Eu_i}{\sigma R}$
Poisson's ratio (ν)	Dimensionless

Table 2: Applied normalizations

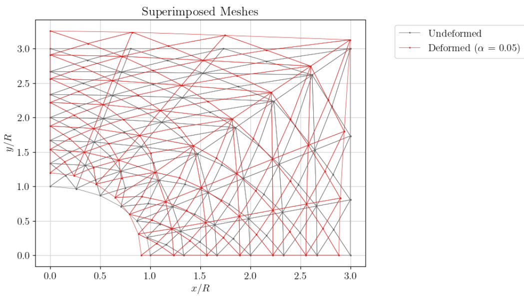
3 Computational Solution

Given the prescribed input, the FE code produces nodal displacements, element strains, and element stresses, which provide the basis for deformation visualization, comparison across meshes, and extrapolation near critical regions.

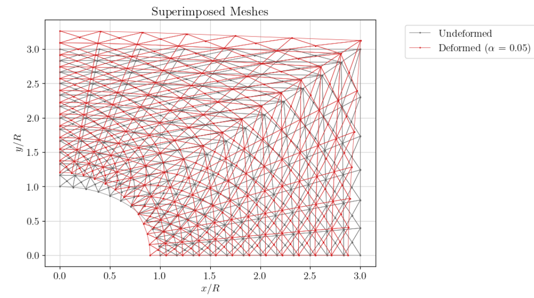
3.1 Plate Deformation

The deformed configuration of each mesh is generated by applying the computed displacements – as well as a scaling factor α of 0.05 – to the initial nodal coordinates (Figure 2). Thus, if the original nodal positions are $(\tilde{x}_i, \tilde{y}_i)$, then the deformed coordinates become $(\tilde{x}_i + \alpha\tilde{u}_i, \tilde{y}_i + \alpha\tilde{v}_i)$.

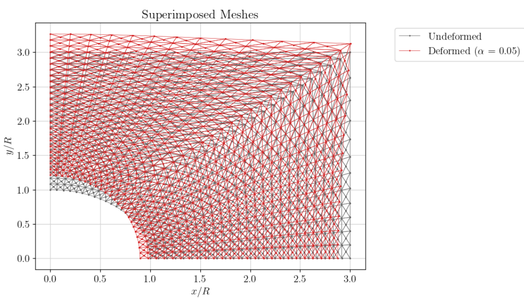
Figure 3 shows the original and deformed meshes for each level of refinement.



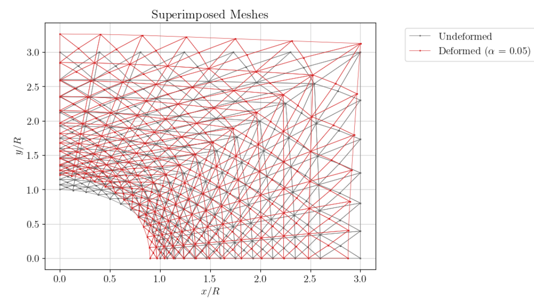
(a) Mesh M_6: Coarse mesh



(b) Mesh M_{12}: Moderate refinement



(c) Mesh M_{24}: Fine mesh



(d) Mesh M_R: Refined near hole

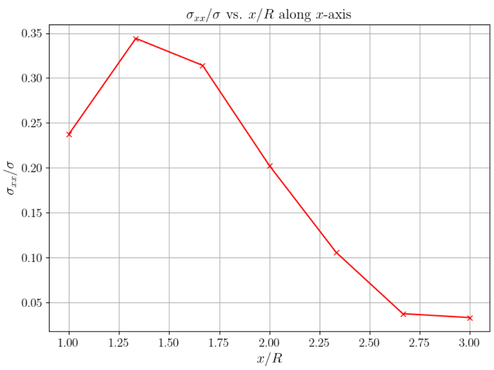
Figure 3: Superimposed undeformed (gray) and scaled deformed (red) meshes for each level of refinement

It is evident that mesh refinement better captures the deformed shape of the geometry. In particular, the deformed shape of the hole is much smoother for finer meshes.

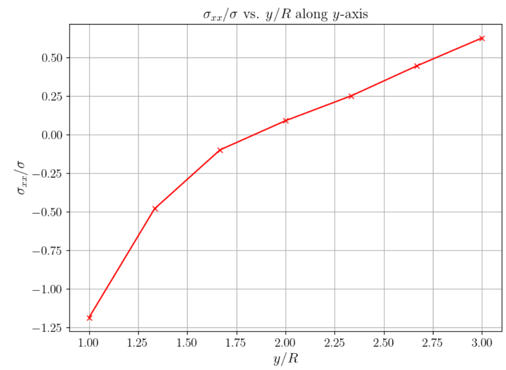
3.2 Stresses

While the code computes stresses at the centroid of each triangular finite element, it is important to more accurately estimate stresses along the edges of the plate, where critical values may occur. To this end, the fields σ_{xx}, σ_{yy} are interpolated to identify the peak stresses and quantify the stress gradients along the axes. Due to the symmetry of the plate geometry and loading conditions, these results can be extended to the full domain by mirroring them about the symmetry lines. Note that the shear stresses σ_{xy} are omitted from the analysis, as they are negligible and cancel out due to symmetry.

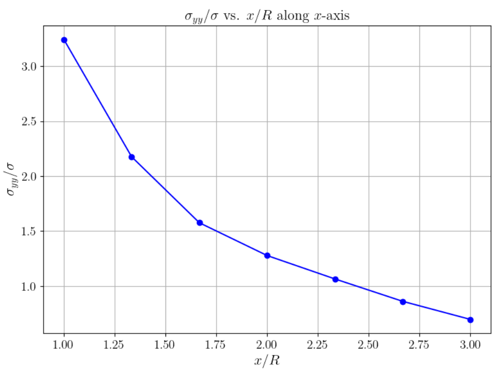
The interpolations of the axial stresses σ_{xx}, σ_{yy} along the x and y axes for the different meshes are shown in Figure 4–7. With increasing mesh refinement from M_6 to M_{24}, all the edge stresses become increasingly smoother. This comes with increasing computational cost. In comparison, the M_R mesh, which is non-uniform shows stress distributions that are comparable to the M_{24} mesh while its computational expense is the same as the M_{12} mesh. In general, increasing the mesh density results in higher values of critical stress values due to higher density of element centroids and hence more accurate resolution of stress concentrations.



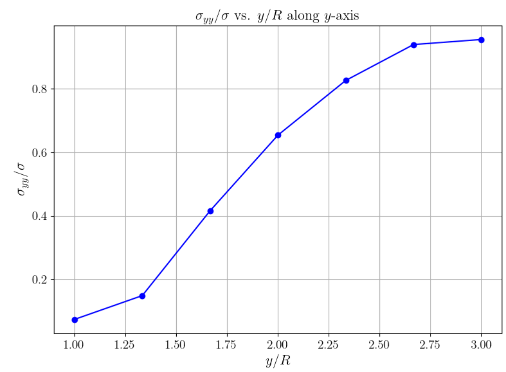
(a) σ_{xx}/σ vs. x/R



(b) σ_{xx}/σ vs. y/R

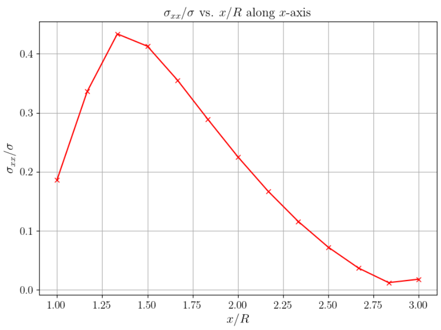


(c) σ_{yy}/σ vs. x/R

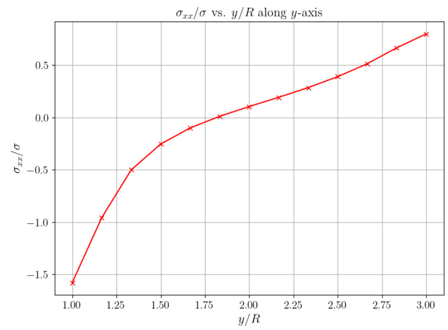


(d) σ_{yy}/σ vs. y/R

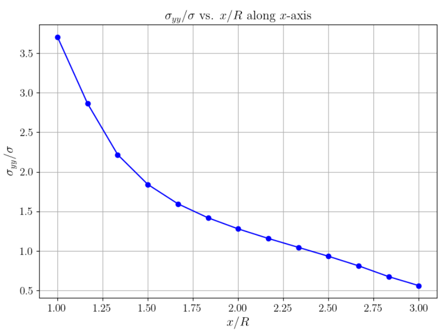
Figure 4: Interpolated axial stresses along the x and y axes for mesh M_6



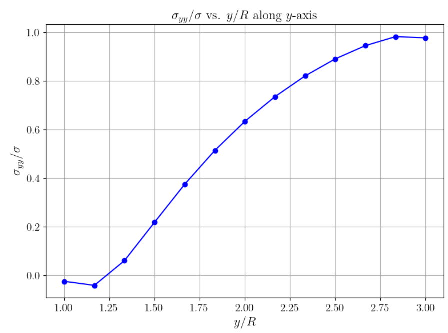
(a) σ_{xx}/σ vs. x/R



(b) σ_{xx}/σ vs. y/R

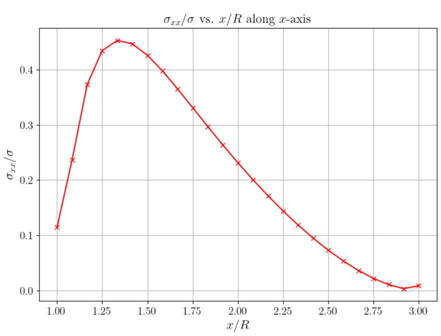


(c) σ_{yy}/σ vs. x/R

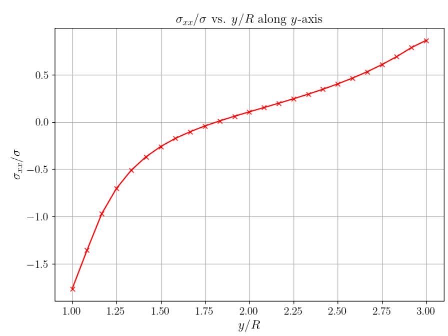


(d) σ_{yy}/σ vs. y/R

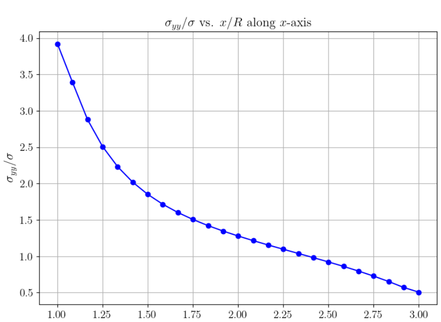
Figure 5: Interpolated axial stresses along the x and y axes for mesh M_{12}



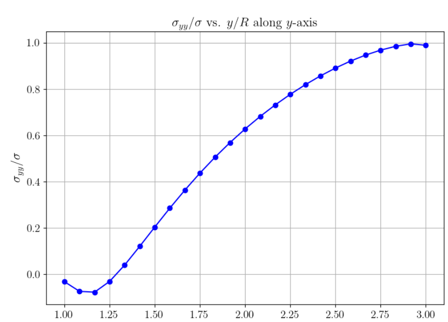
(a) σ_{xx}/σ vs. x/R



(b) σ_{xx}/σ vs. y/R

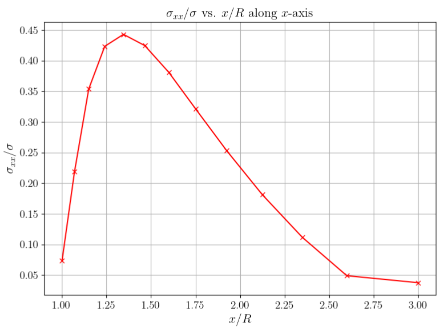


(c) σ_{yy}/σ vs. x/R

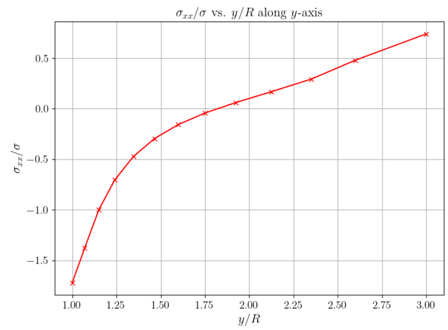


(d) σ_{yy}/σ vs. y/R

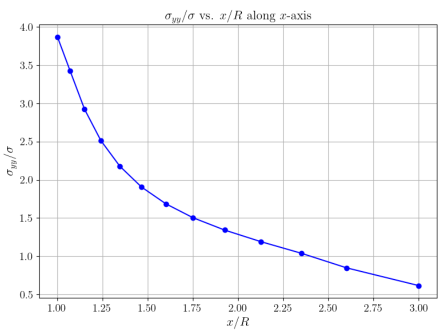
Figure 6: Interpolated axial stresses along the x and y axes for mesh M_{24}



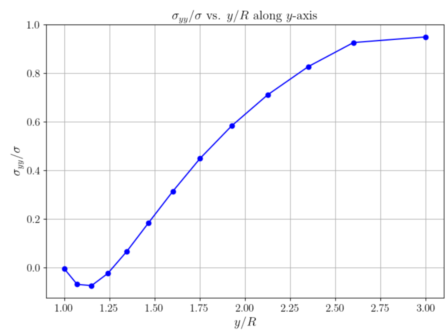
(a) σ_{xx}/σ vs. x/R



(b) σ_{xx}/σ vs. y/R



(c) σ_{yy}/σ vs. x/R



(d) σ_{yy}/σ vs. y/R

Figure 7: Interpolated axial stresses along the x and y axes for mesh M_R

Regardless of the mesh type, some salient features are notable. First, σ_{yy} increases rapidly as x/R approaches 1. The calculations here show that the stress concentration ($\sigma_{yy}/\sigma \approx 3.8$), which is somewhat higher than the theoretical value of 3 in the case of an infinite rectangular plate with a hole subjected to tensile stress. Since the geometry analyzed in this case is a finite – rather than an infinite – plate, the results are expected to be similar to the analytical value of 3 but somewhat higher due to boundary effects. The analytical solution serves to verify the computationally determined finite element approximation. In fact, the finite element result seems to agree with the Kirsch solution (per the ASE 324L Materials course), which states that near a hole in a finite plate of the given $H/R = 3$ should be approximately 3.46.

Second, the variation of σ_{xx}/σ along the x-axis shows a peak at $x/R \approx 1.40$. At $x/R = 1$, σ_{xx}/σ values are sensitive to the mesh density. With increasing mesh refinement, this value drops from $\sigma_{xx}/R \sim 0.24$ (M_6) to $\sigma_{xx}/\sigma \sim 0.11$ (M_{24}). For the M_R mesh, $\sigma_{xx}/\sigma \sim 0.075$, which is much smaller than the M_{12} case (which has the same DOFs as M_R) where $\sigma_{xx}/\sigma \sim 0.19$. This dependence of σ_{xx}/σ highlights the importance of design the FE mesh to capture strong variations in the stress near a discontinuity such as a hole.

Variations of σ_{xx}/σ and σ_{yy}/σ along the y-axis also exhibit a strong dependence on the mesh density. In general, σ_{xx}/σ transitions from being compressive at $y/R = 1$ (at the hole) to being tensile at $y/R = 3$. The transition occurs at $y/R \sim 1.8$ for all four meshes considered here. However, the largest compressive and tensile values depend on the mesh size. For the coarsest mesh (M_6) $\sigma_{xx}/\sigma \approx -1.2$ and for the finest mesh (M_{24}, we obtain $\sigma_{xx}/\sigma \approx -1.75$. For M_R, $\sigma_{xx}/\sigma \approx -1.75$, similar to the M_{24} case. The largest tensile stresses are: $\sigma_{xx}/\sigma \approx 0.67$ (M_6), $\sigma_{xx}/\sigma \approx 0.75$ (M_{12}), $\sigma_{xx}/\sigma \approx 0.75$ (M_{24}, and $\sigma_{xx}/\sigma \approx 0.75$ (M_R).

The variation of σ_{yy}/σ also transitions from compressive to tensile along y/R. The transition occurs at $y/R = 1.25$ for the three fine meshes; for M_6, on the other hand, the transition is observed at $y/R \approx 1.37$. This is likely because of the lower node density along the y-direction, which results in poor sampling of the transition. As expected, $\sigma_{yy}/\sigma = 1$ at $y/R = 3$ where the traction boundary condition is applied. This is well captured by mesh M_{24} and to some extent by mesh M_{12}. Meshes M_6 and M_R show a slightly lower value, which again suggests the importance of the mesh refinement even in the case of non-uniform finite element mesh.

4 Conclusion

This work demonstrates a two-dimensional finite element (FE) solution for elastic stress analysis of a plate with a central hole under uniaxial tensile loading. The FE solver incorporates three-noded constant strain triangular elements, allowing the computation of nodal displacements, element strains, and stresses for varying mesh resolutions.

Normalized results show that coarse meshes may be insufficient to fully capture high stress gradients near geometric discontinuities (e.g., holes), while fine and non-uniform meshes perform significantly better. In particular, the refined mesh M_R - which maintains a higher node density near the hole — achieves a better accuracy while requiring fewer elements. This makes it the most efficient option for resolving local stress concentrations.

The interpolated and extrapolated stress fields confirm the presence of stress concentrations consistent with the analytical Kirsch solution. Peak stresses near the hole approach 3σ, verifying numerically obtained results against theoretical expectations, and validating the finite element approximation.

5 Appendices

A Mesh Visualization

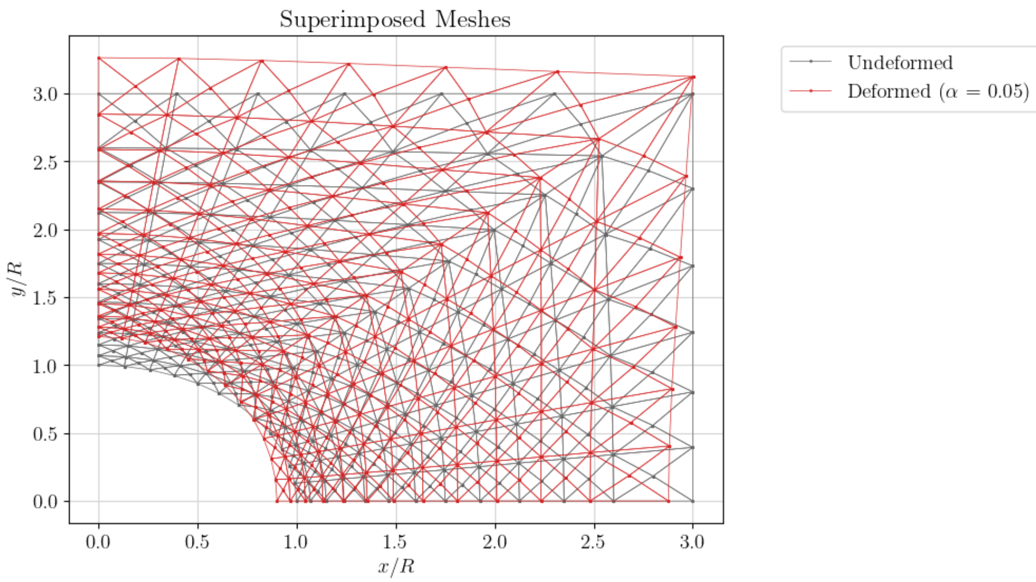


Figure 8: Undeformed and deformed mesh M_R

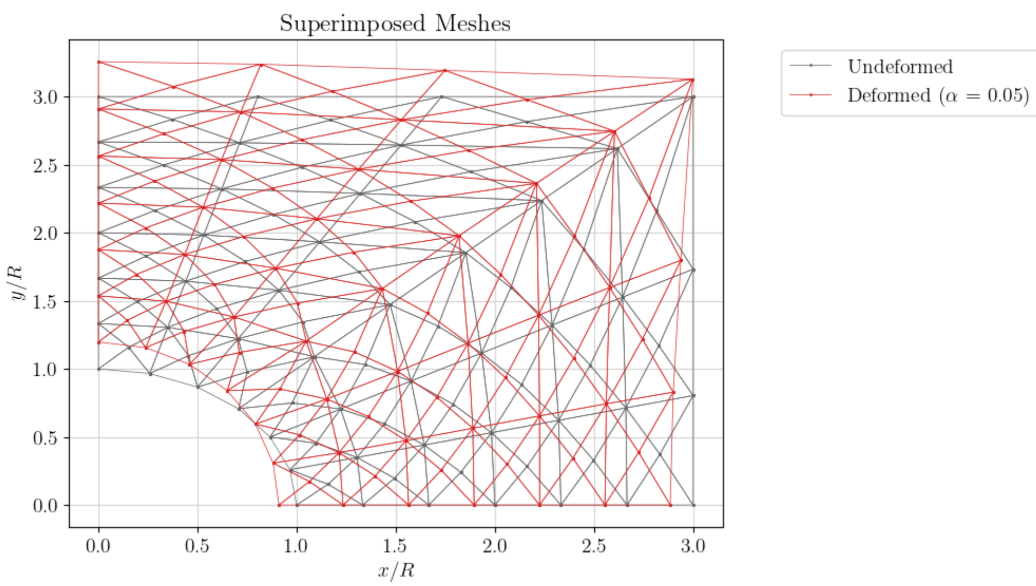


Figure 9: Undeformed and deformed mesh M_6

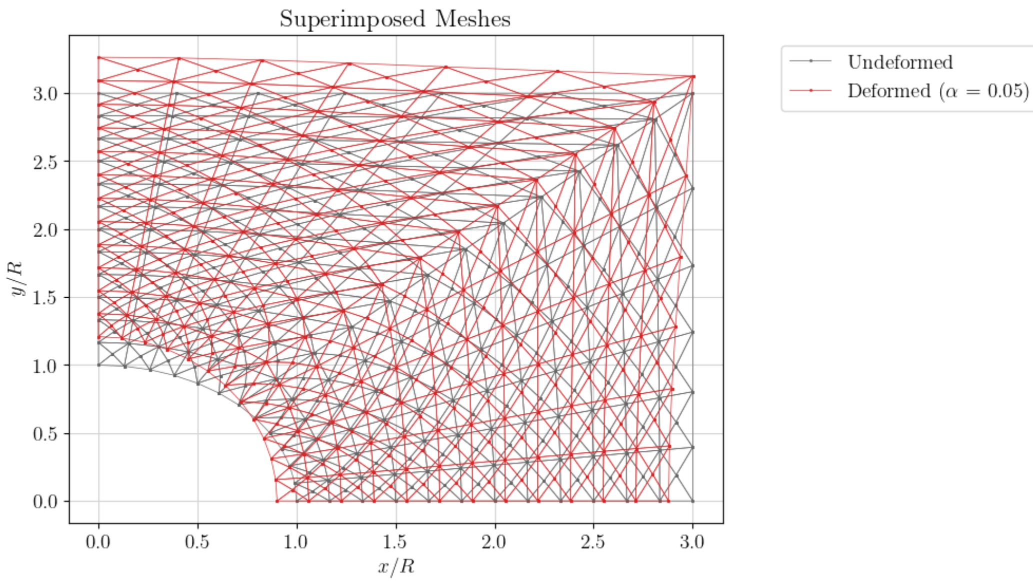


Figure 10: Undeformed and deformed mesh M_{12}

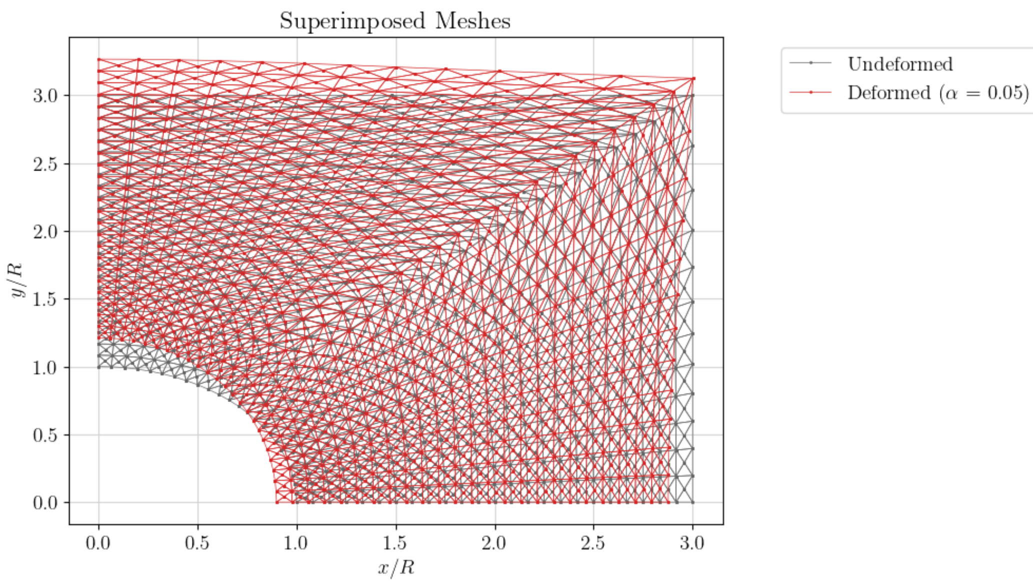


Figure 11: Undeformed and deformed mesh M_{24}

B Axial Stress Plots

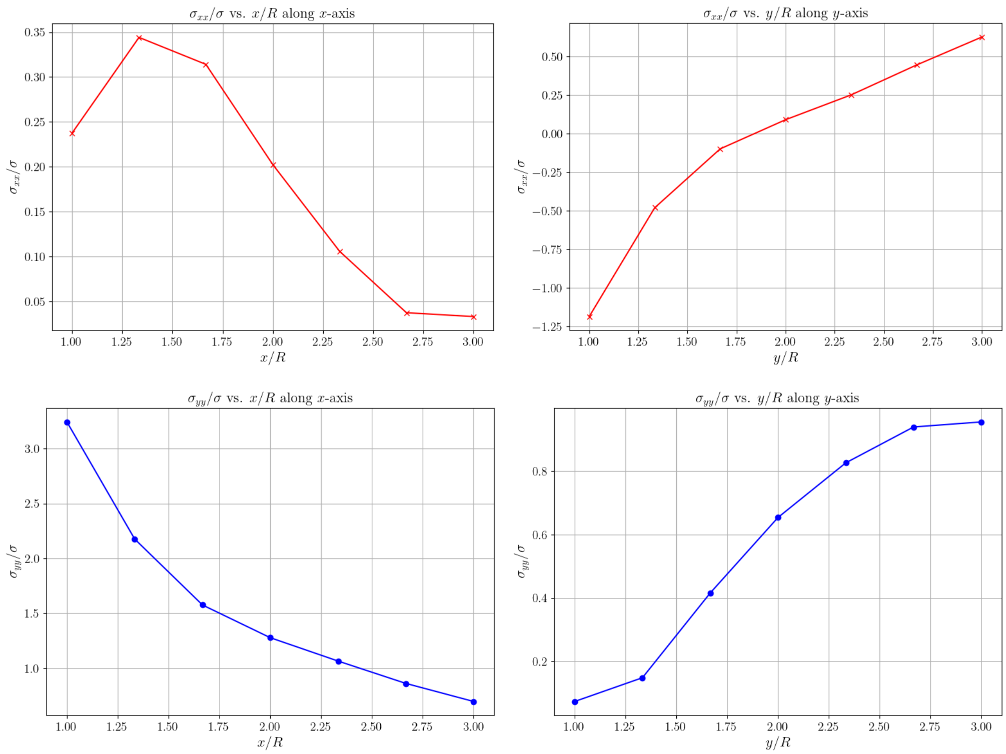


Figure 12: Interpolated stress components along x and y for mesh M_6

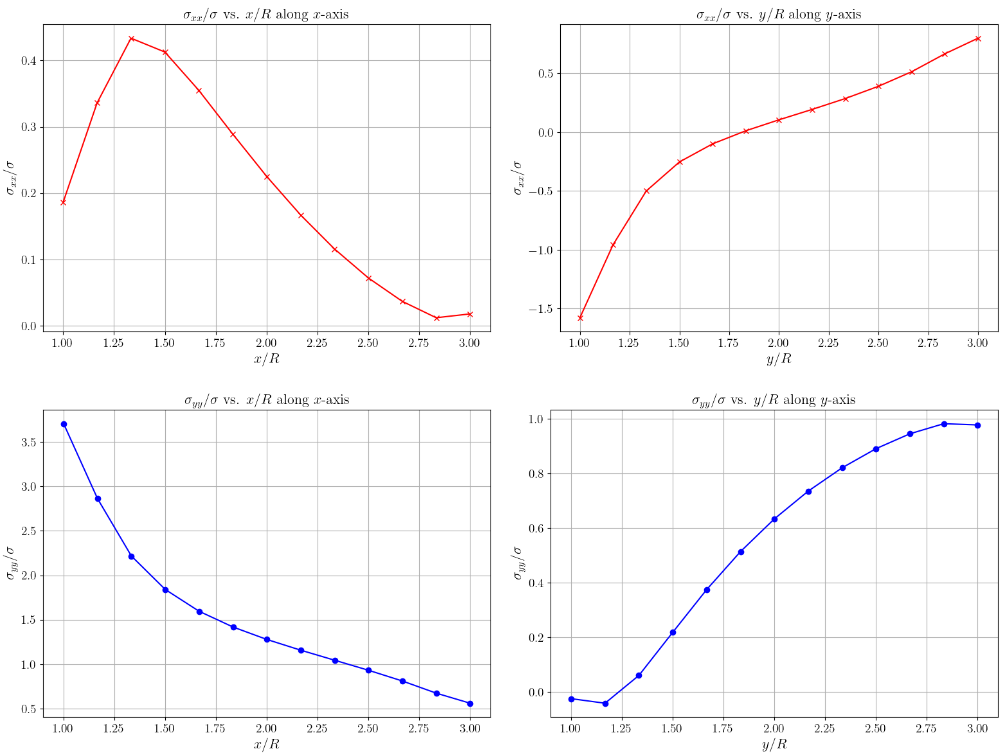


Figure 13: Interpolated stress components along x and y for mesh M_{12}

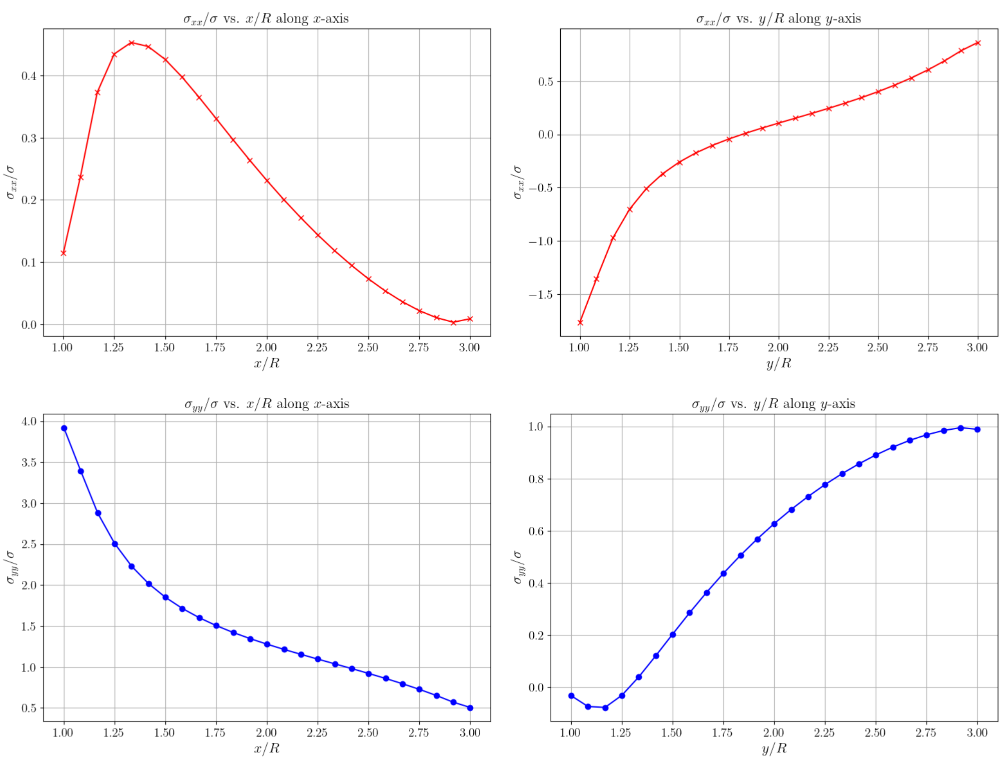


Figure 14: Interpolated stress components along x and y for mesh M_{24}

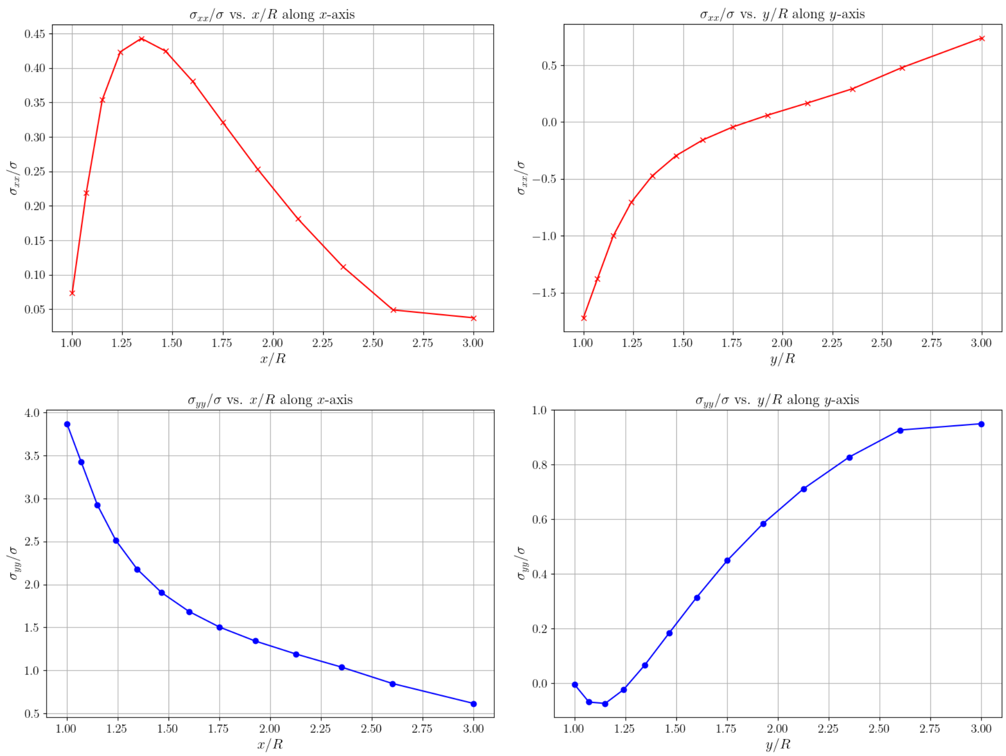


Figure 15: Interpolated stress components along x and y for mesh M_R

C Finite Element Software

Continued on next page.

Solid 2D Elasticity Analysis

COE 321K Final Report

In [ ]:
```
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt

# LaTeX plot formatting
plt.rcParams['text.usetex'] = True

plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = ['Computer Modern Roman']

plt.rcParams.update({
    'text.usetex': True,
    'font.family': 'serif',
})

def load_data(file_path):
    file_name = os.path.basename(file_path)

    with open(file_path, 'r') as file:
        first_line = file.readline().strip().split()

        if file_name == 'elements.txt' or file_name == 'elementsR.txt' or file_name == 'elements6.txt' or file_name == 'elements12.txt' or file_name == 'elements24.txt':
            num_elements = int(first_line[0])
            E = float(first_line[1])
            nu = float(first_line[2])
            data = np.loadtxt(file).astype(float)
            return num_elements, E, nu, data
        else:
            num_constraints = int(first_line[0])
            data = np.loadtxt(file).astype(float)
            return num_constraints, data

def convert_to_txt(file_path):
    output_path = file_path + '.txt'
    with open(file_path, 'r', encoding='utf-8', errors='ignore') as infile, open(output_path, 'w', encoding='utf-8') as outfile:
        outfile.write(infile.read())
    print(f'Converted: {file_path} → {output_path}')

def print_mdof(name, array):
    print(f'{name}:')
    for i, row in enumerate(array):
        dof_values = ' '.join(f'DOF {j+1}: {val:.5f};' for j, val in enumerate(row))
        print(f'  # {i+1}: {dof_values}')
    print()

def print_1dof(name, array):
    print(f'{name}:')
    if array.ndim == 1:
        for i, val in enumerate(array):
            print(f'  # {i+1}: {val:.5f}')
    elif array.ndim == 2:
        for i, row in enumerate(array):
            dof_values = ' '.join(f'{val:.5f}' for j, val in enumerate(row))
            print(f'  # {i+1}: {dof_values}')
    print()
```

Preprocessing

In [ ]:
```
path_displacements = 'displacementsR.txt'
path_elements = 'elementsR.txt'
path_nodes = 'nodesR.txt'
path_forces = 'forcesR.txt'

# path_displacements = 'displacements24.txt'
# path_elements = 'elements24.txt'
# path_nodes = 'nodes24.txt'
# path_forces = 'forces24.txt'

# path_displacements = 'displacements12.txt'
# path_elements = 'elements12.txt'
# path_nodes = 'nodes12.txt'
# path_forces = 'forces12.txt'

# path_displacements = 'displacements6.txt'
# path_elements = 'elements6.txt'
# path_nodes = 'nodes6.txt'
# path_forces = 'forces6.txt'

# path_displacements = 'displacements.txt'
# path_elements = 'elements.txt'
# path_nodes = 'nodes.txt'
# path_forces = 'forces.txt'

num_displacement_BC, displacement_data = load_data(path_displacements)
num_elements, E, nu, elements_data = load_data(path_elements)
num_nodes, nodes_data = load_data(path_nodes)
num_force_BC, forces_data = load_data(path_forces)

print(f'Number of displacement BCs: {num_displacement_BC}\n')
print(f'Number of elements: {num_elements}\n')
print(f'Number of nodes: {num_nodes}\n')
print(f'Number of external force BCs: {num_force_BC}\n')

# print(f'Displacement input:\n {displacement_data}\n')
# print(f'Elements input:\n {elements_data}\n')
# print(f'Nodes input:\n {nodes_data}\n')
# print(f'Forces input:\n {forces_data}')
```

Nodes Input

Exctract the number of dimensions, number of nodes, and individual node number. Also determine the DOFs for each node and the entire structure.

In [ ]:
```
num_dimensions = nodes_data.shape[1] - 1

# nodes_data = nodes_data.astype(int)

node = np.zeros((num_nodes, num_dimensions)) # x_j position of node 'i'

for i in range(num_nodes):
    node_number = nodes_data[i][0].astype(int)
    for j in range(num_dimensions):
        node[i][j] = nodes_data[node_number-1][j+1] # Local DOF 'j' of node 'i'

dof_per_node = num_dimensions
num_dofs = num_nodes * dof_per_node # Total number of DOFS in the structure
g_con = (np.zeros((num_nodes, dof_per_node))).astype(int) # Global Connectivity matrix

for i in range(num_nodes):
    for j in range(dof_per_node):
        g_con[i][j] = dof_per_node*i + (j+1) # g_con[i][j] is the global DOF of node 'i' with local DOF 'j'

# CHECK
node_numbers = nodes_data[:num_nodes, 0].astype(int)
node_df = pd.DataFrame(
    node,
    index=node_numbers,
    columns=[f'Node x{j+1}' for j in range(num_dimensions)]
)
node_df.index.name = 'Node number'
# display(node_df.head(len(node_df)))
print(f'Number of nodes: {num_nodes}\n')
print(f'Number of dimensions: {num_dimensions}\n')
# print(f'Global connectivity: \n {g_con}\n')
```

Elements Input

Extract the number of elements in the structure, Young's Modulus, and Poisson's Ratio. Also calculate the area and shape function derivative matrix (B) for each element.

In [ ]:
```
nodes_per_ele = 3

A = np.zeros(num_elements)
B = np.zeros((num_elements, 3, nodes_per_ele*num_dimensions))

element_nodes = (np.zeros((num_elements, nodes_per_ele))).astype(int)

for i in range(num_elements):
    for j in range(nodes_per_ele):
        element_nodes[i][j] = elements_data[i][j+1]

# Calculate Triangular Element Areas & Construct B Matrix (per element)
for i in range(num_elements):

    node_indices = [element_nodes[i][j]-1 for j in range(nodes_per_ele)]

    x1, y1 = nodes_data[node_indices[0]][1], nodes_data[node_indices[0]][2]
    x2, y2 = nodes_data[node_indices[1]][1], nodes_data[node_indices[1]][2]
    x3, y3 = nodes_data[node_indices[2]][1], nodes_data[node_indices[2]][2]

    A[i] = 0.5 * np.abs((x2 - x1)*(y3 - y1) - (x3 - x1)*(y2 - y1))  # Determinant formula for element area

    # Shape function derivatives
    N1x = (y2-y3) / (2*A[i])
    N1y = (x3-x2) / (2*A[i])

    N2x = (y3-y1) / (2*A[i])
    N2y = (x1-x3) / (2*A[i])

    N3x = (y1-y2) / (2*A[i])
    N3y = (x2-x1) / (2*A[i])

    B[i] = np.array([[N1x, 0, N2x, 0, N3x, 0],
                     [0, N1y, 0, N2y, 0, N3y],
                     [N1y, N1x, N2y, N2x, N3y, N3x]
                     ]) # Unique B for each element

    # print_mdof(f'B-Matrix for element {i+1}', B[i])

print(f'Number of elements: {num_elements}\n')
print(f'Young\'s Modulus: {E}\n')
print(f'Poisson\'s ratio: {nu}\n')
# print_1dof('Triangular element areas', A)
# print_mdof('Element global node numbers', element_nodes)
```

Forces Input

Extract the node, DOF, and magnitude of any external forces acting on the structure.

In [ ]:
```
force_node = np.array([])
force_dof = np.array([])
force_value = np.array([])

for i in range(num_force_BC):
    force_node = (np.append(force_node, forces_data[i, 0])).astype(int)
    force_dof = (np.append(force_dof, forces_data[i, 1])).astype(int)
    force_value = (np.append(force_value, forces_data[i, 2])).astype(float)

# print(force_dof)
```

Displacements Input

Extract the node, DOF, and magnitude of any external displacements of the structure.

In [ ]:
```
displacement_node = np.array([])
displacement_dof = np.array([])
displacement_value = np.array([])

for i in range(num_displacement_BC):
```

```
    displacement_node = (np.append(displacement_node, displacement_data[i][0])).astype(int)
    displacement_dof = (np.append(displacement_dof, displacement_data[i][1])).astype(int)
    displacement_value = (np.append(displacement_value, displacement_data[i][2])).astype(float)
```

DOF Bookkeeping

Rearrange the Global Connectivity matrix to shift active DOFs to the top.

In [ ]:
```
for i in range(num_displacement_BC):
    dof_BC = g_con[displacement_node[i]-1][displacement_dof[i]-1]
    for j in range(num_nodes):
        for k in range(dof_per_node):
            if g_con[j][k] > dof_BC:
                g_con[j][k] -= 1
    g_con[displacement_node[i]-1][displacement_dof[i]-1] = num_nodes * dof_per_node
    num_dofs -= 1

# print(g_con)
```

Force and Stiffness Assembly

In [ ]:
```
K = np.zeros((num_dofs, num_dofs)) # Global Stiffness Matrix
F = np.zeros((num_dofs, 1))
u = np.zeros((num_nodes, dof_per_node))

for i in range(num_force_BC):
    dof = g_con[force_node[i]-1][force_dof[i]-1]
    F[dof-1] += force_value[i]

for i in range(num_displacement_BC):
    u[displacement_node[i]-1][displacement_dof[i]-1] = displacement_value[i]

# print_1dof('Force vector', F)
# print_1dof('Displacmement vector', u)
```

Reduced Element-by-Element Assembly

In [ ]:
```
K_elements = np.zeros((nodes_per_ele*dof_per_node, nodes_per_ele*dof_per_node)) # Global Stiffness Matrix (reduced)

C = (E / (1-nu**2)) * np.array([[1, nu, 0],
                                [nu, 1, 0],
                                [0, 0, (1-nu)/2]
                                ])

for i_ele in range(num_elements):

    K_elements = A[i_ele] * (B[i_ele].T @ C @ B[i_ele])

    for i_node in range(nodes_per_ele):
        for i_dof in range(dof_per_node):
            i_dof_local = ((i_node) * dof_per_node + i_dof)
            i_dof_global = g_con[element_nodes[i_ele][i_node]-1][i_dof] - 1

            if i_dof_global < num_dofs:
                for j_node in range(nodes_per_ele):
                    for j_dof in range(dof_per_node):
                        j_dof_local = ((j_node) * dof_per_node + j_dof)
                        j_dof_global = g_con[element_nodes[i_ele][j_node]-1][j_dof] - 1

                        if (j_dof_global < num_dofs):
                            K[i_dof_global][j_dof_global] += K_elements[i_dof_local][j_dof_local]
                        else:
                            F[i_dof_global] -= K_elements[i_dof_local][j_dof_local] * u[element_nodes[i_ele][j_node]-1][j_dof]

# print(f'Global Stiffness Matrix: \n {K}\n')
```

Displacement Solution

In [ ]:
```
soln = np.linalg.solve(K, F)
```

Postprocessing

Manipulate computational results to derive nodal displacements, element stresses and strains, and external forces.

In [ ]:
```
for i in range(num_nodes):
    for j in range(dof_per_node):
        dof = g_con[i][j]
        if dof <= num_dofs:
            u[i][j] = soln[dof-1]

# print_mdof('Global node displacements:', u)
```

In [ ]:
```
u_local = np.zeros((nodes_per_ele*dof_per_node, 1))
epsilon_elements = np.zeros((num_elements, 3)) # Element strains
sigma_elements = np.zeros((num_elements, 3))  # Element stresses
F_ext = np.zeros((num_nodes, num_dimensions))

for i_ele in range(num_elements):
    for local_node in range(nodes_per_ele):
        for local_dof in range(dof_per_node):
            u_local[dof_per_node * local_node + local_dof] = u[element_nodes[i_ele][local_node]-1][local_dof]

        strain = B[i_ele] @ u_local
        epsilon_elements[i_ele, :] = strain.flatten()

        stress = C @ strain
        sigma_elements[i_ele, :] = stress.flatten()

        ext_virtual_work = A[i_ele] * (B[i_ele].T @ stress)

    # print_mdof(f'External Forces for Element {i_ele+1}', ext_virtual_work)

    for local_node in range(nodes_per_ele):
        for local_dof in range(dof_per_node):
```

```
            global_node = element_nodes[i_ele][local_node] - 1

            F_ext[global_node][local_dof] += ext_virtual_work[dof_per_node * local_node + local_dof]

# print_mdof('Nodal Displacements', u)
# print_mdof('Element Strains', epsilon_elements)
# print_mdof('Element Stresses', sigma_elements)
# print_mdof('External Forces', F_ext)
```

Visualizations

Undeformed/Deformed Mesh Plots

In [ ]:
```
# Undeformed Mesh
fig1 = plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 14})

for idx in range(num_elements):
    nodes = np.array(element_nodes[idx]) - 1  # force as array
    nodes = np.append(nodes, nodes[0])

    x = node[nodes, 0]
    y = node[nodes, 1]

    plt.plot(x, y, color='0.4', linewidth=0.5, marker='o', markersize=1)

plt.title('Undeformed Mesh')
plt.xlabel('$x/R$')
plt.ylabel('$y/R$')
plt.grid(color='lightgray')

# Deformed Mesh
alpha = 0.05
deformed_coord = node.copy()

deformed_coord[:, 0] += alpha * u[:, 0] # u-displacement
deformed_coord[:, 1] += alpha * u[:, 1] # v-displacement

fig2 = plt.figure(figsize=(8, 6))

for i in element_nodes:
    node_ele = np.append(i, i[0]) -1

    deformed_x = deformed_coord[node_ele, 0]
    deformed_y = deformed_coord[node_ele, 1]

    plt.plot(deformed_x, deformed_y, color='tab:red', linewidth=0.5, marker='o', markersize=1)

plt.title(rf'Deformed Mesh ($\alpha$ = {alpha})')
plt.xlabel('$x/R$')
plt.ylabel('$y/R$')
plt.grid()
plt.grid(color='lightgray')

plt.tight_layout()

# Superimposed Plots
fig, ax = plt.subplots(figsize=(8, 6))

for idx, i in enumerate(element_nodes):
    node_ele = np.append(i, i[0]) -1

    x = node[node_ele, 0]
    y = node[node_ele, 1]

    ax.plot(x, y, color='0.4', linewidth=0.5, marker='o', markersize=1, label='Undeformed' if idx == 0 else '')

for idx, i in enumerate(element_nodes):
    node_ele = np.append(i, i[0]) -1

    deformed_x = deformed_coord[node_ele, 0]
    deformed_y = deformed_coord[node_ele, 1]

    ax.plot(deformed_x, deformed_y, color='tab:red', linewidth=0.5, marker='o', markersize=1, label=rf'Deformed ($\alpha$ = {alpha})' if idx == 0 else '')

ax.set_title(rf'Superimposed Meshes')
ax.set_xlabel('$x/R$')
ax.set_ylabel('$y/R$')
ax.legend(loc='upper right', bbox_to_anchor=(1.5, 1))
ax.grid(color='lightgray')

plt.show()
```

Axial Stress Interpolation

In [ ]:
```
# Interpolate nodal stresses to centroid stresses
# Constant strain triangles

stress_per_node = np.zeros((num_nodes, 3))
node_counter = np.zeros(num_nodes)

for i_ele in range(num_elements):
    nodes_per_trianle = element_nodes[i_ele] - 1
    stress_per_triangle = sigma_elements[i_ele]

    for n in nodes_per_trianle:
        stress_per_node[n, :] += stress_per_triangle # Interpolating about centroid
        node_counter[n] += 1

for i_node in range(num_nodes):
    if node_counter[i_node] > 0:
        stress_per_node[i_node, :] /= node_counter[i_node] # Average steress for shared nodes

# Extend to three more meshes
nodes_OG = node.copy()
stresses_OG = stress_per_node.copy()

# Mirror across x = 0 (y-axis)
nodes_mirror_x = node.copy()
```

```
nodes_mirror_x[:, 0] *= -1
stress_mirror_x = stress_per_node.copy()
stress_mirror_x[:,2] *= -1  # Opposite sigma-xy

# Mirror across y = 0 (x-axis)
nodes_mirror_y = node.copy()
nodes_mirror_y[:, 1] *= -1
stress_mirror_y = stress_per_node.copy()
stress_mirror_y[:,2] *= -1  # Opposite sigma-xy

# Mirror across x = 0 and y = 0 ('Quadrant III')
nodes_mirror_xy = node.copy()
nodes_mirror_xy[:,0] *= -1
nodes_mirror_xy[:,1] *= -1
stress_mirror_xy = stress_per_node.copy()

# Full Plate Mapping
nodes_plate = np.vstack((nodes_OG, nodes_mirror_x, nodes_mirror_y, nodes_mirror_xy))
stresses_plate = np.vstack((stresses_OG, stress_mirror_x, stress_mirror_y, stress_mirror_xy))

# Generate four meshes
# Mesh 1 is original, Mesh 2 is mirrored about x = 0, Mesh 3 is mirrored about y = 0, Mesh 4 is mirrored about both x = 0 and y = 0

num_nodes_mesh = node.shape[0]
nodes_meshes = [nodes_plate[0*num_nodes_mesh:1*num_nodes_mesh],  # Original
                nodes_plate[1*num_nodes_mesh:2*num_nodes_mesh],  # Mirror X
                nodes_plate[2*num_nodes_mesh:3*num_nodes_mesh],  # Mirror Y
                nodes_plate[3*num_nodes_mesh:4*num_nodes_mesh]   # Mirror XY
                ]

stresses_meshes = [stresses_plate[0*num_nodes_mesh:1*num_nodes_mesh],
                   stresses_plate[1*num_nodes_mesh:2*num_nodes_mesh],
                   stresses_plate[2*num_nodes_mesh:3*num_nodes_mesh],
                   stresses_plate[3*num_nodes_mesh:4*num_nodes_mesh]
                   ]

mesh_labels = ['Mesh 1', 'Mesh 2', 'Mesh 3', 'Mesh 4']

num_meshes = 1

for i in range(num_meshes):
    nodes_m = nodes_meshes[i]
    stresses_m = stresses_meshes[i]

    tol = 1e-10
    x_cut = np.abs(nodes_m[:, 1]) < tol  # approx. y = 0
    y_cut = np.abs(nodes_m[:, 0]) < tol  # approx. x = 0

    sort_x = np.argsort(nodes_m[x_cut, 0])
    sort_y = np.argsort(nodes_m[y_cut, 1])

    # Plot 1: sigma_xx along x-axis
    plt.figure(figsize=(8, 6))
    plt.plot(nodes_m[x_cut, 0][sort_x], stresses_m[x_cut, 0][sort_x], 'rx-')
    plt.title(r'$\sigma_{xx}/\sigma$ vs. $x/R$ along $x$-axis', fontsize=16)
    plt.xlabel('$x/R$', fontsize=16)
    plt.ylabel(r'$\sigma_{xx}/\sigma$', fontsize=16)
    plt.grid()
    plt.tight_layout()
    plt.show()

    # Plot 2: sigma_yy along x-axis
    plt.figure(figsize=(8, 6))
    plt.plot(nodes_m[x_cut, 0][sort_x], stresses_m[x_cut, 1][sort_x], 'bo-')
    plt.title(r'$\sigma_{yy}/\sigma$ vs. $x/R$ along $x$-axis', fontsize=16)
    plt.xlabel('$x/R$', fontsize=16)
    plt.ylabel(r'$\sigma_{yy}/\sigma$', fontsize=16)
    plt.grid()
    plt.tight_layout()
    plt.show()

    # Plot 3: sigma_xx along y-axis
    plt.figure(figsize=(8, 6))
    plt.plot(nodes_m[y_cut, 1][sort_y], stresses_m[y_cut, 0][sort_y], 'rx-')
    plt.title(r'$\sigma_{xx}/\sigma$ vs. $y/R$ along $y$-axis', fontsize=16)
    plt.xlabel('$y/R$', fontsize=16)
    plt.ylabel(r'$\sigma_{xx}/\sigma$', fontsize=16)
    plt.grid()
    plt.tight_layout()
    plt.show()

    # Plot 4: sigma_yy along y-axis
    plt.figure(figsize=(8, 6))
    plt.plot(nodes_m[y_cut, 1][sort_y], stresses_m[y_cut, 1][sort_y], 'bo-')
    plt.title(r'$\sigma_{yy}/\sigma$ vs. $y/R$ along $y$-axis', fontsize=16)
    plt.xlabel('$y/R$', fontsize=16)
    plt.ylabel(r'$\sigma_{yy}/\sigma$', fontsize=16)
    plt.grid()
    plt.tight_layout()
    plt.show()
```

Edge Stress Determination

In [ ]:

```
tol = 1e-10

for i in range(num_meshes):
    nodes_m = nodes_meshes[i]
    stresses_m = stresses_meshes[i]

    print(f'MESH {i+1} \n')

    # (x = +R, y = 0)
    xR_y0 = np.where((np.abs(nodes_m[:,0] - 1) < tol) & (np.abs(nodes_m[:,1]) < tol))[0]

    # (x = 0, y = +R)
    x0_yR = np.where((np.abs(nodes_m[:,0]) < tol) & (np.abs(nodes_m[:,1] - 1) < tol))[0]

    # (x = -R, y = 0)
    xmR_y0 = np.where((np.abs(nodes_m[:,0] + 1) < tol) & (np.abs(nodes_m[:,1]) < tol) )[0]

    # (x = 0, y = -R)
```

```python
    x0_ymR = np.where((np.abs(nodes_m[:,0]) < tol) & (np.abs(nodes_m[:,1] + 1) < tol))[0]

    for label, i in zip(['(x=+R, y=0)', '(x=0, y=+R)', '(x=-R, y=0)', '(x=0, y=-R)'], [xR_y0, x0_yR, xmR_y0, x0_ymR]):

        if len(i) != 0:
            print(f'Nodes near {label}:')

            for index in i:
                print(f'Node {i}: x = {nodes_m[index, 0]:.5f}, y = {nodes_m[index, 1]:.5f}')
                print(f'sigma_xx = {stresses_m[index, 0]:.5f}, sigma_yy = {stresses_m[index, 1]:.5f}, sigma_xy = {stresses_m[index, 2]:.5f}\n')

        else:
            print(f'No node found near {label}\n')
```