Final Exam

AME 5763

Blake Johnson

December 2024

Contents

1 Problem 1

Derive the finite element model of the equation:

$$-\frac{d}{dx}\left[(1+x)\frac{du}{dx}\right] = f, \quad 0 < x < 1$$

for the boundary conditions:

$$u(0) = \bar{u}, \quad \left[(1+x)\frac{du}{dx}\right]_{x=1} = \bar{q}.$$

Solve the problem for the given data:

$$f = 0, \quad \bar{u} = 1, \quad \bar{q} = 0.$$

Use two linear finite elements.

Problem 1 Solution

The problem gives us the strong form of the equation. To begin, we set up variables for each of the given parameters.

Code Implementation

Listing 1: Python Code for Setting Up the Problem

```
# Define variables and equation parameters
x = sp.symbols('x')
u = sp.Function('u')(x)
f = 0  # Given source term
ubar = 1  # Boundary condition at x=0
qbar = 0  # Boundary condition at x=1

# Define the differential equation
diff_eq = -sp.diff((1 + x) * sp.diff(u, x), x) - f
# Print the equation
print("Differential Equation:", diff_eq)
```

Output

Differential Equation

$$-(x+1)\frac{d^2u(x)}{dx^2} - \frac{du(x)}{dx}$$

Construct the Shape Functions

Now that I have the weak form, I followed Chapter 4 to construct two linear element shape functions and the corresponding shape function.

Since they are linear and I need two, I decided to use two of the same shape functions:

- Element 1: $[0, 0.5]$
- Element 2: $[0.5, 1]$

I used the equations from page 83 for the shape functions.

Coding the Shape Functions

Listing 2: Python Code for Defining Shape Functions

```
# Define the variable and nodes
x = sp.Symbol('x')
x1, x2, x3 = 0, 0.5, 1  # Nodes for the two-element mesh

# Local shape functions for Element 1 (0 to 0.5)
N1 = (x - x2) / (x1 - x2)  # Shape function for Node 1 (
    updated convention)
N2 = (x - x1) / (x2 - x1)  # Shape function for Node 2 (
    unchanged)

# Display the shape functions
print("Shape Functions for Element 1:")
sp.pretty_print(N1)
sp.pretty_print(N2)
```

Output

Shape Functions for Element 1

$$1.0 - 2.0x$$

$$2.0x$$

Setting Up the B-Matrix

Next, I set up the B-matrix for the two elements. This involves calculating the derivatives of the shape functions and organizing them into a matrix.

Listing 3: Python Code for Setting Up the B-Matrix

```
# Shape function derivatives
dN1_dx = sp.diff(N1, x)
dN2_dx = sp.diff(N2, x)

# B-matrix (derivatives of shape functions)
B = sp.Matrix([dN1_dx, dN2_dx])

print("\nB-Matrix:")
sp.pretty_print(B)
```

Output

B-Matrix

$$B = \begin{bmatrix} -2.0 \\ 2.0 \end{bmatrix}$$

Setting Up the Stiffness Matrix

Now, I set up the stiffness matrix K for the two elements and the global stiffness matrix. The formula used is:

$$K = B^T \cdot A_k \cdot B$$

Listing 4: Python Code for Setting Up the Stiffness Matrix

```
# Define variables
x, x1, x2 = sp.symbols('x x1 x2')

# Stiffness matrix Ke for one element
Ke = sp.Matrix(2, 2, lambda i, j: sp.integrate(B[i] * (1
    + x) * B[j], (x, x1, x2)))

# Simplify the generic stiffness matrix
Ke_simplified = sp.simplify(Ke)

# Display the simplified stiffness matrix
print("Simplified Element Stiffness Matrix (Ke):")
sp.pprint(Ke_simplified)

# Substitute values for Element 1: [0, 0.5]
Ke_e1 = Ke.subs({x1: 0, x2: 0.5})
print("\nElement Stiffness Matrix for Element 1 (Ke_e1):
    ")
sp.pprint(Ke_e1)

# Substitute values for Element 2: [0.5, 1]
Ke_e2 = Ke.subs({x1: 0.5, x2: 1})
print("\nElement Stiffness Matrix for Element 2 (Ke_e2):
    ")
sp.pprint(Ke_e2)
```

Output

Simplified Element Stiffness Matrix (Ke)

$$Ke = \begin{bmatrix} -2.0x_1^2 - 4.0x_1 + 2.0x_2^2 + 4.0x_2 & 2.0x_1^2 + 4.0x_1 - 2.0x_2^2 - 4.0x_2 \\ 2.0x_1^2 + 4.0x_1 - 2.0x_2^2 - 4.0x_2 & -2.0x_1^2 - 4.0x_1 + 2.0x_2^2 + 4.0x_2 \end{bmatrix}$$

Element Stiffness Matrix for Element 1 (Ke_{e1})

$$Ke_{e1} = \begin{bmatrix} 2.5 & -2.5 \\ -2.5 & 2.5 \end{bmatrix}$$

Element Stiffness Matrix for Element 2 (Ke_{e2})

$$Ke_{e2} = \begin{bmatrix} 3.5 & -3.5 \\ -3.5 & 3.5 \end{bmatrix}$$

Assembling the Global Stiffness Matrix

Next, I assembled the global stiffness matrix numerically. This is a 3×3 matrix with K_1 as the first 2×2 submatrix and K_2 as the last 2×2 submatrix.

Listing 5: Python Code for Global Stiffness Matrix Assembly

```
# Set up a matrix of zeros for the global stiffness
    matrix
K_global = sp.zeros(3, 3)

# Add contributions from Element 1
K_global[0:2, 0:2] += Ke_e1

# Add contributions from Element 2
K_global[1:3, 1:3] += Ke_e2

print("\nGlobal Stiffness Matrix (K_global):")
sp.pprint(K_global)

# Apply Dirichlet boundary condition at x = 0 (u(0) = 1)
K_global[0, :] = sp.zeros(1, 3)  # Zero out the first
    row
K_global[:, 0] = sp.zeros(3, 1)  # Zero out the first
    column
K_global[0, 0] = 1  # Set diagonal entry to 1

# The new global stiffness matrix after applying
    Dirichlet boundary condition
print("\nModified Global Stiffness Matrix (K_global):")
sp.pprint(K_global)
```

Output

Global Stiffness Matrix (K_{global})

$$K_{\text{global}} = \begin{bmatrix} 2.5 & -2.5 & 0 \\ -2.5 & 6.0 & -3.5 \\ 0 & -3.5 & 3.5 \end{bmatrix}$$

Modified Global Stiffness Matrix (K_{global})

$$K_{\text{global}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 6.0 & -3.5 \\ 0 & -3.5 & 3.5 \end{bmatrix}$$

Boundary Conditions and Adjustments

This matrix does not initially account for boundary conditions. The determinant is zero, so I applied the Dirichlet boundary condition at $x = 0$ ($u(0) = 1$) by:

1. Zeroing out the first row and column.
2. Setting the diagonal entry corresponding to $x = 0$ to 1.

The resulting modified global stiffness matrix incorporates these boundary conditions.

Solving for the Displacement Vector

Finally, I solved for the displacement vector using the global stiffness matrix and the external force vector. Using the relationship:

$$K \cdot d = F$$

I computed the displacement vector d as:

$$d = K^{-1} \cdot F$$

Listing 6: Python Code for Solving the Displacement Vector

```
# Apply Dirichlet boundary condition at x = 0, u(0) = 1,
    and q = 0

f = sp.Matrix([1, 0, 0])  # External force vector

print("\nExternal Force Vector (F):")
sp.pprint(f)

d = K_global.inv() * f  # Correct matrix multiplication
   order
print("\nDisplacement Vector (d):")
sp.pprint(d)
```

Output

External Force Vector (F)

$$F = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Displacement Vector (d)

$$d = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Conclusion

In this problem, I derived the finite element model for a one-dimensional bar governed by the equation:

$$-\frac{d}{dx}\left[(1+x)\frac{du}{dx}\right] = f, \quad 0 < x < 1,$$

subject to the boundary conditions:

$$u(0) = \bar{u}, \quad \left[(1+x)\frac{du}{dx}\right]_{x=1} = \bar{q}.$$

Key Results
The key results from my work are as follows:

1. **Global Stiffness Matrix:** I assembled the global stiffness matrix by combining contributions from the two linear elements. The matrix accurately reflects continuity across nodes and incorporates the $(1 + x)$ term from the governing equation. After applying the Dirichlet boundary condition at $x = 0$ ($u(0) = 1$), I modified the stiffness matrix to correctly account for the fixed displacement.

2. **Displacement Vector:** I calculated the displacement vector d as:

$$d = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

   This result is consistent with the boundary conditions:

   - At $x = 0$, $u(0) = 1$, as specified.
   - The remaining entries of d are zero, aligning with the absence of external forces beyond the initial fixed displacement.

Discussion of Results
The results make sense within the context of this problem:

- The fixed displacement at $x = 0$ dictates the solution due to the boundary condition $u(0) = 1$.
- Since the external force f is zero, no additional deformations occur at the other nodes, leading to zero displacements for those locations.
- The computed stiffness matrices and assembled global matrix confirm that the finite element model was implemented correctly, with continuity and boundary constraints applied properly.

This problem allowed me to explore the finite element method for one-dimensional boundary value problems with variable coefficients. The results highlight the importance of accurately assembling the stiffness matrix and correctly applying boundary conditions to achieve meaningful solutions.

2 Problem 2

Using the linear field $\phi = \phi(x, y, z) = a_1 + a_2x + a_3y + a_4z$, prove that isoparametric elements have the ability to represent constant gradients of the field.

Field and Shape Functions

Starting with the linear field:

$$\phi(x, y, z) = a_1 + a_2x + a_3y + a_4z.$$

ϕ can be interpolated using the shape functions N_i and the nodal values ϕ_i:

$$\phi(x, y, z) = \sum_{i=1}^{n} N_i(x, y, z)\phi_i.$$

The shape functions N_i are linear functions of x, y, z:

$$N_i(x, y, z) = a_i + b_ix + c_iy + d_iz,$$

Then the field becomes:

$$\phi(x, y, z) = \sum_{i=1}^{n} (a_i + b_ix + c_iy + d_iz)\, \phi_i.$$

$$\phi(x, y, z) = \sum_{i=1}^{n} (a_i\phi_i + b_ix\phi_i + c_iy\phi_i + d_iz\phi_i)\,.$$

$$\phi(x, y, z) = \sum_{i=1}^{n} a_i\phi_i + \sum_{i=1}^{n} b_i\phi_ix + \sum_{i=1}^{n} c_i\phi_iy + \sum_{i=1}^{n} d_i\phi_iz.$$

$$\phi(x, y, z) = \left(\sum_{i=1}^{n} a_i\phi_i\right) + \left(\sum_{i=1}^{n} b_i\phi_i\right) x + \left(\sum_{i=1}^{n} c_i\phi_i\right) y + \left(\sum_{i=1}^{n} d_i\phi_i\right) z.$$

This equation matches the given form of $\phi(x, y, z) = a_1 + a_2x + a_3y + a_4z$, with:

$$a_1 = \sum_{i=1}^{n} a_i\phi_i, \quad a_2 = \sum_{i=1}^{n} b_i\phi_i, \quad a_3 = \sum_{i=1}^{n} c_i\phi_i, \quad a_4 = \sum_{i=1}^{n} d_i\phi_i.$$

Thus, $\phi(x, y, z)$ is linear because:

- The shape functions $N_i(x, y, z)$ are linear,
- The coefficients a_i, b_i, c_i, d_i are constants, and
- The nodal values ϕ_i are constants.

Gradients in Natural Coordinates

The gradient is defined in Section 7.4.2 of the textbook as:

$$\nabla \phi^e = B^e d^e.$$

Since the textbook uses the four-node quadrilateral element as an example, I am using that as an example for this problem.

From the previous section, the field is expressed as:

$$\phi(\xi, \eta, \zeta) = \sum_{i=1}^{n} N_i(\xi, \eta, \zeta)\phi_i.$$

1. Gradient of ϕ in Natural Coordinates

The gradient of ϕ in natural coordinates is given by the Jacobian (page 167):

$$\begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \frac{\partial \phi}{\partial \eta} \\ \frac{\partial \phi}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} \frac{\partial N_i}{\partial \xi}\phi_i \\ \sum_{i=1}^{n} \frac{\partial N_i}{\partial \eta}\phi_i \\ \sum_{i=1}^{n} \frac{\partial N_i}{\partial \zeta}\phi_i \end{bmatrix}.$$

2. Derivatives of Shape Functions

The shape functions N_i are **linear functions** of ξ, η, ζ, such as:

$$N_i(\xi, \eta, \zeta) = a_i + b_i\xi + c_i\eta + d_i\zeta.$$

The derivatives with respect to ξ, η, ζ are therefore constants:

$$\frac{\partial N_i}{\partial \xi} = b_i, \quad \frac{\partial N_i}{\partial \eta} = c_i, \quad \frac{\partial N_i}{\partial \zeta} = d_i.$$

Substituting into the gradient expression, we find:

$$\frac{\partial \phi}{\partial \xi} = \sum_{i=1}^{n} b_i\phi_i, \quad \frac{\partial \phi}{\partial \eta} = \sum_{i=1}^{n} c_i\phi_i, \quad \frac{\partial \phi}{\partial \zeta} = \sum_{i=1}^{n} d_i\phi_i.$$

These are constant values since b_i, c_i, d_i, ϕ_i are constants. (We are considering the field ϕ at a specific node ϕ_i, which is why it can be considered a constant). Thus, the gradient of ϕ in natural coordinates is constant.

Transformation to Physical Coordinates

Next, I converted from natural coordinates to physical coordinates.

1. Chain Rule Transformation

The gradient of ϕ in physical coordinates (x, y, z) is related to its gradient in natural coordinates (ξ, η, ζ) by the chain rule. Using the inverse of the Jacobian (as described on page 167), the relationship is:

$$\begin{bmatrix} \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \\ \frac{\partial \phi}{\partial z} \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \eta}{\partial x} & \frac{\partial \zeta}{\partial x} \\ \frac{\partial \xi}{\partial y} & \frac{\partial \eta}{\partial y} & \frac{\partial \zeta}{\partial y} \\ \frac{\partial \xi}{\partial z} & \frac{\partial \eta}{\partial z} & \frac{\partial \zeta}{\partial z} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \frac{\partial \phi}{\partial \eta} \\ \frac{\partial \phi}{\partial \zeta} \end{bmatrix}.$$

Here:

- The matrix on the left represents the gradient in physical coordinates ($\nabla\phi$ in x, y, z).
- The middle matrix is the inverse Jacobian matrix, which transforms derivatives from natural to physical coordinates.
- The matrix on the right is the gradient in natural coordinates ($\nabla\phi$ in ξ, η, ζ).

2. Jacobian Matrix

The Jacobian matrix J relates the physical coordinates (x, y, z) to the natural coordinates (ξ, η, ζ):

$$J = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{bmatrix}.$$

The inverse of this matrix (J^{-1}) appears in the chain rule transformation above.

3. Constant Gradient in Physical Coordinates

From the previous section, I know that $\nabla\phi$ in natural coordinates (ξ, η, ζ) is constant:

$$\begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \frac{\partial \phi}{\partial \eta} \\ \frac{\partial \phi}{\partial \zeta} \end{bmatrix} = \text{constant vector}.$$

The Jacobian J is a constant matrix for linear elements because the shape functions N_i are linear. Therefore:

$\nabla\phi$ in physical coordinates $(x, y, z) = J^{-1} \cdot \nabla\phi$ in natural coordinates (ξ, η, ζ)

is also constant.

Conclusion

In this problem, I demonstrated that isoparametric elements have the ability to represent constant gradients of a field. Starting with the linear field:

$$\phi(x, y, z) = a_1 + a_2 x + a_3 y + a_4 z,$$

I showed that:

1. The field ϕ can be expressed using shape functions N_i, which are linear, ensuring the gradient of ϕ in natural coordinates (ξ, η, ζ) is constant.
2. Using the Jacobian transformation, I established that the gradient of ϕ in physical coordinates (x, y, z) is also constant for isoparametric elements with linear shape functions.

This demonstrates the key property of isoparametric elements: their ability to represent constant gradients, which is essential for accurate finite element approximations of linear fields.

3 Problem 3

Develop the weak form for the following partial differential equation:

$$-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u=f$$

in a two-dimensional domain R with boundary conditions:

$$u=\bar{u}\quad\text{on } S_1,$$

and

$$a_1\frac{\partial u}{\partial x}n_x+a_2\frac{\partial u}{\partial y}n_y=\bar{g}\quad\text{on } S_2,$$

where n_x and n_y are the x and y components of the normal to the surface S_2, $\mathbf{n}$. Here, $a_0, a_1, a_2, f, \bar{u}$, and $\bar{g}$ are known functions of x and y in R.

List of Given Values

The given PDE is:

$$-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u=f\quad\text{in domain } R.$$

Coefficients

- a_1, a_2: Functions of x and y, represent scaling factors for the derivatives of u with respect to x and y.
- a_0: A function of x and y, represents a source/sink term.

Dependent Variable

- u: is equal to $\bar{u}$ at the boundary conditions.

Source Term

- f: Represents the known forcing function.

Boundary Conditions

1. **Dirichlet Condition** on boundary S_1:

$$u = \bar{u} \quad \text{on } S_1.$$

2. **Neumann Condition** on boundary S_2:

$$a_1 \frac{\partial u}{\partial x} n_x + a_2 \frac{\partial u}{\partial y} n_y = \bar{g} \quad \text{on } S_2,$$

where:

- n_x and n_y are the components of the normal vector to the boundary S_2.
- $\bar{g}$ represents the flux on the boundary.

The PDE is defined over a two-dimensional domain R.

Strong Form of the PDE

The given PDE is:

$$-\frac{\partial}{\partial x}\left(a_1 \frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(a_2 \frac{\partial u}{\partial y}\right) + a_0 u = f$$

in domain R, with boundary conditions:

- **Dirichlet Condition** on S_1:

$$u = \bar{u} \quad \text{on } S_1.$$

- **Neumann Condition** on S_2:

$$a_1 \frac{\partial u}{\partial x} n_x + a_2 \frac{\partial u}{\partial y} n_y = \bar{g} \quad \text{on } S_2.$$

The Test Function $w(x)$

The test function $w(x)$ is defined such that:

- $w(x) = 0$ on the Dirichlet boundary (S_1).
- $w(x)$ is arbitrary in the interior of the domain R.

To derive the weak form, I multiplied the PDE by $w(x)$ and integrated over the domain R.

Formulation

The weak form of the PDE is given by:

$$\int_R w(x)\left(-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u\right)\,dA=\int_R w(x)f\,dA.$$

Expanding this, the weak form becomes:

$$\int_R\left(-w(x)\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)\right)dx-\int_R\left(w(y)\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)\right)dy+\int_R w(x,y)a_0u\,dA=\int_R w(x)f\,dA.$$

Integration by Parts

Term 1
The first term in the PDE is:

$$-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right).$$

Using the product rule:

$$-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)=-a_1\frac{\partial^2u}{\partial x^2}-\frac{\partial a_1}{\partial x}\frac{\partial u}{\partial x}.$$

Now, multiplying by the weight function $w(x)$ and integrating:

$$-\int_x w(x)\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)=\int_x w(x)\left[-a_1\frac{\partial^2u}{\partial x^2}-\frac{\partial a_1}{\partial x}\frac{\partial u}{\partial x}\right]dx.$$

Expanding:

$$\int_x w(x)\left[-a_1\frac{\partial^2u}{\partial x^2}-\frac{\partial a_1}{\partial x}\frac{\partial u}{\partial x}\right]dx=-\int_x w(x)a_1\frac{\partial^2u}{\partial x^2}dx-\int_x w(x)\frac{\partial a_1}{\partial x}\frac{\partial u}{\partial x}dx.$$

Using integration by parts on $-\int_x w(x)a_1\frac{\partial^2u}{\partial x^2}dx$:

$$\int udv=uv-\int vdu.$$

Set:

$$u=w(x)a_1,\quad du=w(x)\frac{\partial a_1}{\partial x}+a_1\frac{\partial w(x)}{\partial x},\quad v=\frac{\partial u}{\partial x},\quad dv=\frac{\partial^2u}{\partial x^2}.$$

Applying integration by parts:

$$-\int_x w(x)a_1\frac{\partial^2 u}{\partial x^2}dx = -\left[w(x)a_1\frac{\partial u}{\partial x}\Big|_{\text{boundary}} - \int_x \frac{\partial u}{\partial x}\left(w(x)\frac{\partial a_1}{\partial x} + a_1\frac{\partial w(x)}{\partial x}\right)dx\right].$$

Simplifying:

$$-\int_x w(x)a_1\frac{\partial^2 u}{\partial x^2}dx = -\left[w(x)a_1\frac{\partial u}{\partial x}\Big|_{\text{boundary}} - \int_x w(x)\frac{\partial u}{\partial x}\frac{\partial a_1}{\partial x}dx + \int_x a_1\frac{\partial w(x)}{\partial x}\frac{\partial u}{\partial x}dx\right].$$

Substituting back into the original integral:

$$-\int_x w(x)a_1\frac{\partial^2 u}{\partial x^2}dx - \int_x w(x)\frac{\partial a_1}{\partial x}\frac{\partial u}{\partial x}dx = -\left[w(x)a_1\frac{\partial u}{\partial x}\Big|_{\text{boundary}} + \int_x a_1\frac{\partial w(x)}{\partial x}\frac{\partial u}{\partial x}dx\right].$$

Boundary Conditions

The term $-w(x)a_1\frac{\partial u}{\partial x}$ is a boundary term, evaluating the flux at the boundary of the domain. On S_1, the weight function $w(x) = 0$, so this term vanishes. On S_2, the boundary term is defined by:

$$a_1\frac{\partial u}{\partial x}n_x + a_2\frac{\partial u}{\partial y}n_y = \bar{g}.$$

Applying this boundary condition to the integrated form of the boundary term:

$$\int_{\partial S_2} w(x)a_1\frac{\partial u}{\partial x}n_x dS.$$

The final form of the first term is:

$$-\int_x w(x)\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right) = -\int_{\partial S_2} w(x)a_1\frac{\partial u}{\partial x}n_x dS - \int_x a_1\frac{\partial w(x)}{\partial x}\frac{\partial u}{\partial x}dx.$$

Term 2

The second term in the PDE can be treated similarly to the first term, replacing a_1 with a_2 and x with y. Using the same process, we obtain:

$$-\int_y w(y)\frac{\partial}{\partial y}\left[a_2\frac{\partial u}{\partial y}\right]dy = -\int_{\partial S_2} w(y)a_2\frac{\partial u}{\partial y}n_y dS - \int_y a_2\frac{\partial w(y)}{\partial y}\frac{\partial u}{\partial y}dy.$$

Term 3

The third term represents a volume integral, involving both x and y. This suggests that a double integral is needed:

$$\int_x\int_y w(x,y)a_0(x,y)u(x,y)\,dy\,dx.$$

Combining the Terms to Derive the Weak Form

Next, I combine the boundary condition terms and the volume integral terms into a single expression for the weak form:

$$\begin{aligned}\int_R w(x)\left(-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u\right)\,dA = \\ -\int_{\partial S_2} w(x,y)a_1\frac{\partial u}{\partial x}n_x\,dS-\int_x a_1\frac{\partial w(x)}{\partial x}\frac{\partial u}{\partial x}\,dx \\ -\int_{\partial S_2} w(x,y)a_2\frac{\partial u}{\partial y}n_y\,dS-\int_y a_2\frac{\partial w(y)}{\partial y}\frac{\partial u}{\partial y}\,dy \\ +\int_x\int_y w(x,y)a_0(x,y)u(x,y)\,dy\,dx.\end{aligned}$$

Simplifying further:

$$\begin{aligned}\int_R w(x,y)\left(-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u\right)\,dA = \\ -\int_{\partial S_2} w(x,y)a_1\frac{\partial u}{\partial x}n_x\,dS-\int_{\partial S_2} w(x,y)a_2\frac{\partial u}{\partial y}n_y\,dS \\ -\int_R a_1\frac{\partial w(x)}{\partial x}\frac{\partial u}{\partial x}\,dR-\int_R a_2\frac{\partial w(y)}{\partial y}\frac{\partial u}{\partial y}\,dR \\ +\int_R w(x,y)a_0(x,y)u(x,y)\,dR.\end{aligned}$$

Combining the boundary terms and volume terms:

$$\begin{aligned}\int_R w(x,y)\left(-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u\right)\,dA = \\ -\int_{\partial S_2} w(x,y)\left(a_1\frac{\partial u}{\partial x}n_x+a_2\frac{\partial u}{\partial y}n_y\right)\,dS \\ -\int_R\left(a_1\frac{\partial w(x)}{\partial x}\frac{\partial u}{\partial x}+a_2\frac{\partial w(y)}{\partial y}\frac{\partial u}{\partial y}+w(x,y)a_0(x,y)u(x,y)\right)\,dR.\end{aligned}$$

Substituting the boundary condition $a_1\frac{\partial u}{\partial x}n_x+a_2\frac{\partial u}{\partial y}n_y=\bar{g}$ on S_2:

$$\int_R w(x,y)\left(-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u\right)\,dA=$$
$$-\int_{\partial S_2} w(x,y)\bar{g}\,dS$$
$$-\int_R\left(a_1\frac{\partial w(x)}{\partial x}\frac{\partial u}{\partial x}+a_2\frac{\partial w(y)}{\partial y}\frac{\partial u}{\partial y}+w(x,y)a_0(x,y)u(x,y)\right)\,dR.$$

Final Form

The weak form of the given partial differential equation simplifies to:

$$\int_R\left(a_1\frac{\partial w}{\partial x}\frac{\partial u}{\partial x}+a_2\frac{\partial w}{\partial y}\frac{\partial u}{\partial y}+w(x,y)a_0u(x,y)\right)\,dR=\int_R wf\,dA+\int_{\partial S_2} w\bar{g}\,dS.$$

Conclusion

In this problem, I developed the weak form of the given partial differential equation:

$$-\frac{\partial}{\partial x}\left(a_1\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right)+a_0u=f\quad\text{in domain } R,$$

subject to the boundary conditions:

$$u=\bar{u}\quad\text{on } S_1,\quad a_1\frac{\partial u}{\partial x}n_x+a_2\frac{\partial u}{\partial y}n_y=\bar{g}\quad\text{on } S_2.$$

Key Results
The key results from this derivation are as follows:

1. **Weak Form:** The weak form of the PDE was derived as:
$$\int_R\left(a_1\frac{\partial w}{\partial x}\frac{\partial u}{\partial x}+a_2\frac{\partial w}{\partial y}\frac{\partial u}{\partial y}+w(x,y)a_0u(x,y)\right)\,dR=\int_R wf\,dA+\int_{\partial S_2} w\bar{g}\,dS.$$

2. **Boundary Terms:** The boundary terms explicitly incorporated the Neumann boundary condition $a_1\frac{\partial u}{\partial x}n_x+a_2\frac{\partial u}{\partial y}n_y=\bar{g}$, ensuring the flux is correctly accounted for in the weak form.

3. **Volume Terms:** The volume integrals combined the contributions of the coefficients a_1, a_2, and a_0, as well as the test function $w(x,y)$, ensuring the equation holds over the entire domain R.

Discussion of Results
This derivation confirms that:

- The weak form ensures compatibility with both Dirichlet and Neumann boundary conditions.
- The method allows for the inclusion of non-uniform coefficients (a_1, a_2, a_0) and variable source terms (f).
- By integrating by parts, I reduced the second-order derivatives in the PDE to first-order derivatives, which is essential for finite element implementation.

This problem demonstrates the process of converting a strong form PDE into its weak form, a critical step in finite element analysis for solving complex engineering problems.

4 Problem 4

Repeat Example 9.2 with two triangular elements as shown in Figure 9.19.

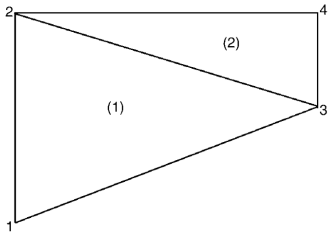

Figure 9.19 Quadrilateral domain meshed with two triangular elements.

Figure 1: Quadrilateral domain meshed with two triangular elements.

Problem 4 Solution

The problem gives us the following conditions:

- The bottom and the right vertical edges are traction-free, i.e., $t_{\bar{x}} = 0$.
- A traction $t_{\bar{y}} = -20\,\mathrm{N/m}^2$ is applied on the top horizontal edge.
- The material properties are:
  - Young's modulus $E = 3 \times 10^7\,\mathrm{Pa}$,
  - Poisson's ratio $\nu = 0.3$.

These conditions were implemented as variables in the Python code:

Listing 7: Given Problem Conditions

```
"""
Given Values
"""
E = 3E7
t_bar = 0  # Traction-free boundary condition
t_bar_y = -20  # Traction on the top horizontal edge
v = 0.3  # Poisson's ratio
```

Calculating the D-Matrix

The D-matrix is derived from the material properties using the formula:

$$D = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}.$$

Substituting the given values:

$$E = 3 \times 10^7, \quad \nu = 0.3,$$

the numerical computation proceeds as follows:

$$D = \frac{3 \times 10^7}{1-0.3^2} \begin{bmatrix} 1 & 0.3 & 0 \\ 0.3 & 1 & 0 \\ 0 & 0 & \frac{1-0.3}{2} \end{bmatrix}.$$

This matrix represents the stiffness properties of the material, considering its elastic modulus E and Poisson's ratio ν.

Code Implementation

Listing 8: Python Code for Calculating the D-Matrix

```
"""
Calculate the D matrix
"""
D = E / (1 - v**2) * sp.Matrix([
    [1, v, 0],
    [v, 1, 0],
    [0, 0, (1 - v) / 2]
])

print("D Matrix")
sp.pprint(D)
```

Output

The computed D-matrix is:

D-Matrix

$$D = \begin{bmatrix} 32967032.967033 & 9890109.89010989 & 0 \\ 9890109.89010989 & 32967032.967033 & 0 \\ 0 & 0 & 11538461.5384615 \end{bmatrix}$$

Setting Up the Coordinate Matrix

In Example 9.2, I only needed one coordinate system because it dealt with a single quadrilateral. However, for this problem, I need two coordinate matrices since the domain is divided into two triangular elements. Additionally, I am using the nodal numbering from Figure 9.19, which differs from the example.

Triangle Coordinates:

The quadrilateral is split into two triangular elements as follows:

- **Triangle 1:** Nodes 2, 1, and 3.
- **Triangle 2:** Nodes 2, 3, and 4.

Code Implementation:

Listing 9: Python Code for Setting Up the Coordinate Matrix

```
# Split the quadrilateral into two triangles
triangle_1 = sp.Matrix([
    [0, 1],  # Node 2
    [0, 0],  # Node 1
    [2, 0.5] # Node 3
])

triangle_2 = sp.Matrix([
    [0, 1],  # Node 2
    [2, 0.5],# Node 3
    [2, 1]   # Node 4
])

# Print the coordinates for verification
print("Triangle 1 Coordinates (Nodes 2, 1, 3):")
sp.pprint(triangle_1)

print("\nTriangle 2 Coordinates (Nodes 2, 3, 4):")
sp.pprint(triangle_2)
```

Output:

The coordinate matrices for the two triangles are:

Triangle 1 Coordinates (Nodes 2

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 2 & 0.5 \end{bmatrix}$$

Triangle 2 Coordinates (Nodes 2

$$\begin{bmatrix} 0 & 1 \\ 2 & 0.5 \\ 2 & 1 \end{bmatrix}$$

Creating the Shape Functions

For triangular elements, I used linear shape functions to interpolate the solution within the element. These shape functions are defined in terms of the natural coordinates ξ and η, which are specific to triangular elements.

Shape Functions:

The linear shape functions for a triangle are:

$$N_1 = 1 - \xi - \eta, \quad N_2 = \xi, \quad N_3 = \eta.$$

These shape functions satisfy the following properties:

- N_1, N_2, N_3 are linear functions of ξ and η.
- At each node, the corresponding shape function equals 1, and the others equal 0.

Code Implementation:

Listing 10: Python Code for Creating the Shape Functions

```
'''
Create the Shape Functions
'''
# triangular shape functions

xi, eta = sp.symbols('xi eta')

N1 = 1 - xi - eta
N2 = xi
N3 = eta
```

Computing the Jacobian Matrix

To map natural coordinates (ξ, η) to physical coordinates (x, y) for triangular elements, I computed the Jacobian matrix. This matrix is derived from the derivatives of the shape functions with respect to the natural coordinates.

Definition:

The Jacobian matrix relates the physical and natural coordinate systems:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}.$$

The Jacobian is computed as:

$$J = \begin{bmatrix} \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \frac{\partial N_3}{\partial \xi} \\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \frac{\partial N_3}{\partial \eta} \end{bmatrix} \cdot \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{bmatrix}.$$

Code Implementation:

Listing 11: Python Code for Computing the Jacobian Matrix

```
'''
Compute the Jacobian Matrix
'''

# Shape function derivatives
dN1dxi = sp.diff(N1, xi)
dN2dxi = sp.diff(N2, xi)
dN3dxi = sp.diff(N3, xi)
dN1deta = sp.diff(N1, eta)
dN2deta = sp.diff(N2, eta)
dN3deta = sp.diff(N3, eta)

# Derivatives of shape functions with respect to xi and
   eta
dN_dxi = sp.Matrix([dN1dxi, dN2dxi, dN3dxi])   # [ N1 /
    xi , N2 / xi , N3 / xi ]
dN_deta = sp.Matrix([dN1deta, dN2deta, dN3deta]) # [
    N1 / eta , N2 / eta , N3 / eta ]

# Combine derivatives into a single matrix
dN = sp.Matrix.hstack(dN_dxi, dN_deta)  # Shape function
    derivative matrix

# Compute the Jacobian matrix
J_triangle1 = dN.T * triangle_1

print("Jacobian Matrix for Triangle 1:")
sp.pprint(J_triangle1)

J_triangle2 = dN.T * triangle_2

print("\nJacobian Matrix for Triangle 2:")
sp.pprint(J_triangle2)
```

Output

The computed Jacobian matrices for each triangular element are as follows:

Jacobian Matrix for Triangle 1

$$J_{\text{triangle 1}} = \begin{bmatrix} 0 & -1 \\ 2 & -0.5 \end{bmatrix}.$$

Jacobian Matrix for Triangle 2

$$J_{\text{triangle 2}} = \begin{bmatrix} 2 & -0.5 \\ 2 & 0 \end{bmatrix}.$$

Determinant and Inverse of the Jacobian Matrix

To ensure that the Jacobian matrix is valid for transformation, I computed its determinant for both triangular elements. A nonzero determinant confirms that the Jacobian is invertible, which is necessary for mapping natural coordinates to physical coordinates. I then computed the inverse of the Jacobian matrix for each triangle.

Mathematical Definition:

The determinant of the Jacobian matrix is given by:

$$\det(J) = \begin{vmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{vmatrix}.$$

The inverse of the Jacobian matrix is used to transform gradients between natural and physical coordinates.

Code Implementation:

Listing 12: Python Code for Determinant and Inverse of the Jacobian Matrix

```
'''
Determinant of the Jacobian Matrix
Confirm that the determinant is not equal to zero

Then I took the inverse of the Jacobian Matrix
'''
det_J_triangle1 = sp.det(J_triangle1)

print("Determinant of the Jacobian Matrix of triangle 1:
    ")
sp.pprint(det_J_triangle1)

inv_J_triangle1 = J_triangle1.inv()

print("Inverse of the Jacobian Matrix of triangle 1:")
sp.pprint(inv_J_triangle1)

det_J_triangle2 = sp.det(J_triangle2)

print("Determinant of the Jacobian Matrix of triangle 2:
    ")
sp.pprint(det_J_triangle2)

inv_J_triangle2 = J_triangle2.inv()

print("Inverse of the Jacobian Matrix of triangle 2:")
sp.pprint(inv_J_triangle2)
```

Output

The determinant and inverse for each triangular element are as follows:

Determinant and Inverse for Triangle 1

Determinant:

$$\det(J_{\text{triangle 1}}) = 2$$

Inverse:

$$J^{-1}_{\text{triangle 1}} = \begin{bmatrix} -0.25 & 0.5 \\ -1 & 0 \end{bmatrix}.$$

Determinant and Inverse for Triangle 2

Determinant:

$$\det(J_{\text{triangle 2}}) = 1.0$$

Inverse:

$$J^{-1}_{\text{triangle 2}} = \begin{bmatrix} 0 & 0.5 \\ -2.0 & 2.0 \end{bmatrix}.$$

The determinants for both triangles are nonzero, confirming that the Jacobian matrices are invertible. The inverses will be used in subsequent steps to transform gradients and other quantities between coordinate systems.

Constructing the Strain-Displacement Matrix (B)

Next I want to calculate the B-matrix. The process for constructing the B-matrix involves transforming the shape function derivatives from the natural coordinate system (ξ, η) to the global coordinate system (x, y) using the Jacobian matrix and its inverse.

For triangular elements, the derivatives of the shape functions in natural coordinates (ξ, η) are defined as:

$$\frac{\partial \mathbf{N}}{\partial \xi} = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}.$$

Using the inverse of the Jacobian matrix (J^{-1}), the derivatives in the global coordinates (x, y) are computed as:

$$\frac{\partial \mathbf{N}}{\partial x} = J^{-1} \cdot \frac{\partial \mathbf{N}}{\partial \xi}.$$

The B-matrix for each triangular element is then constructed as:

$$B = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \frac{\partial N_3}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & 0 & \frac{\partial N_3}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial y} & \frac{\partial N_3}{\partial x} \end{bmatrix}.$$

Code Implementation:

Listing 13: Python Code for Constructing the Strain-Displacement (B) Matrix

```
'''
Calculate the B matrix for both triangles
'''

# Derivatives of shape functions in parent coordinates
dN_parent = sp.Matrix([
    [-1, 1, 0],  #  N / xi
    [-1, 0, 1]   #  N / eta
])

# Compute derivatives in global coordinates
global_derivatives = inv_J_triangle1 * dN_parent

# Construct the B-matrix for Triangle 1
B1 = sp.zeros(3, 6)  # Initialize a 3x6 matrix

# Assign  N / x
B1[0, 0] = global_derivatives[0, 0]
B1[0, 2] = global_derivatives[0, 1]
B1[0, 4] = global_derivatives[0, 2]

# Assign  N / y
B1[1, 1] = global_derivatives[1, 0]
B1[1, 3] = global_derivatives[1, 1]
B1[1, 5] = global_derivatives[1, 2]

# Assign  N / y  and  N / x  for the shear terms
B1[2, 0] = global_derivatives[1, 0]
B1[2, 2] = global_derivatives[1, 1]
B1[2, 4] = global_derivatives[1, 2]
B1[2, 1] = global_derivatives[0, 0]
B1[2, 3] = global_derivatives[0, 1]
B1[2, 5] = global_derivatives[0, 2]
```

```
print("Strain-Displacement Matrix (B) for Triangle 1:")
sp.pprint(B1)

# Compute derivatives in global coordinates for Triangle
    2
global_derivatives2 = inv_J_triangle2 * dN_parent

# Construct the B-matrix for Triangle 2
B2 = sp.zeros(3, 6)  # Initialize a 3x6 matrix

# Assign  N / x
B2[0, 0] = global_derivatives2[0, 0]
B2[0, 2] = global_derivatives2[0, 1]
B2[0, 4] = global_derivatives2[0, 2]

# Assign  N / y
B2[1, 1] = global_derivatives2[1, 0]
B2[1, 3] = global_derivatives2[1, 1]
B2[1, 5] = global_derivatives2[1, 2]

# Assign  N / y  and  N / x  for the shear terms
B2[2, 0] = global_derivatives2[1, 0]
B2[2, 2] = global_derivatives2[1, 1]
B2[2, 4] = global_derivatives2[1, 2]
B2[2, 1] = global_derivatives2[0, 0]
B2[2, 3] = global_derivatives2[0, 1]
B2[2, 5] = global_derivatives2[0, 2]

print("\nStrain-Displacement Matrix (B) for Triangle 2:"
    )
sp.pprint(B2)
```

Output

The computed B-matrices for the two triangular elements are:

Strain-Displacement Matrix (B) for Triangle 1

$$B = \begin{bmatrix} -0.25 & 0 & -0.25 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 1 & -0.25 & -1 & -0.25 & 0 & 0.5 \end{bmatrix}.$$

Strain-Displacement Matrix (B) for Triangle 2

$$B = \begin{bmatrix} -0.5 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & -2.0 & 0 & 2.0 \\ 0 & -0.5 & -2.0 & 0 & 2.0 & 0.5 \end{bmatrix}.$$

Constructing the Global Stiffness Matrix

Next I assembled the Global Stiffness Matrix. It is assembled by combining the stiffness contributions from each element in the mesh. For this problem, the mesh consists of two triangular elements.

Mathematical Formulation:

The stiffness matrix for an individual element is computed as:

$$K_e = \int_{A_e} B^T D B \, dA,$$

where:

- B: Strain-displacement matrix.
- D: Material property matrix (calculated earlier).
- dA: Element area, approximated using the determinant of the Jacobian matrix.

For numerical integration, a single Gauss quadrature point is used at the barycentric coordinates $(\xi, \eta) = \left(\frac{1}{3}, \frac{1}{3}\right)$, with a weight $w = 1$.

Code Implementation:

Listing 14: Python Code for Calculating the Global Stiffness Matrix

```
'''
Calculate the global stiffness matrix
'''

# Define the Gauss quadrature integration weight and
   point
w = 1  # Weight for a single integration point
xi, eta = 1/3, 1/3  # Barycentric coordinates for the
   integration point

# Element stiffness matrix for Triangle 1
K1 = (B1.T * D * B1) * det_J_triangle1 * w
print("\nElement Stiffness Matrix for Triangle 1:")
sp.pprint(K1)

# Element stiffness matrix for Triangle 2
K2 = (B2.T * D * B2) * det_J_triangle2 * w
print("\nElement Stiffness Matrix for Triangle 2:")
sp.pprint(K2)

# Initialize the global stiffness matrix (8 DOF for 4
   nodes)
K_global = sp.zeros(8, 8)

# Assembly for Triangle 1 (Nodes 2, 1, 3 -> Global DOFs
   [2, 3, 0, 1, 4, 5])
node_map1 = [2, 3, 0, 1, 4, 5]
for i in range(6):
    for j in range(6):
        K_global[node_map1[i], node_map1[j]] += K1[i, j]

# Assembly for Triangle 2 (Nodes 2, 3, 4 -> Global DOFs
   [2, 3, 4, 5, 6, 7])
node_map2 = [2, 3, 4, 5, 6, 7]
for i in range(6):
    for j in range(6):
        K_global[node_map2[i], node_map2[j]] += K2[i, j]

print("\nGlobal Stiffness Matrix:")
sp.pprint(K_global)
```

Output

Element Stiffness Matrix for Triangle 1

$$K_1 = \begin{bmatrix} 27197802.20 & -10714285.70 & -18956043.96 & -824175.82 & -8241758.24 & 11538461.54 \\ -10714285.70 & 67376373.63 & 824175.82 & -64491758.24 & 9890109.89 & -2884615.38 \\ -18956043.96 & 824175.82 & 27197802.20 & 10714285.70 & -8241758.24 & -11538461.54 \\ -824175.82 & -64491758.24 & 10714285.70 & 67376373.63 & -9890109.89 & -2884615.38 \\ -8241758.24 & 9890109.89 & -8241758.24 & -9890109.89 & 6483516.48 & 0.00 \\ 11538461.54 & -2884615.38 & -11538461.54 & -2884615.38 & 0.00 & 5769230.77 \end{bmatrix}$$

Element Stiffness Matrix for Triangle 1

$$K_2 = \begin{bmatrix} 8241758.24 & 0 & 0 & 9890109.89 & -8241758.24 & -9890109.89 \\ 0 & 2884615.38 & 11538461.54 & 0 & -11538461.54 & -2884615.38 \\ 0 & 11538461.54 & 46153846.15 & 0 & -46153846.15 & -11538461.54 \\ 9890109.89 & 0 & 0 & 131868131.87 & -9890109.89 & -131868131.87 \\ -8241758.24 & -11538461.54 & -46153846.15 & -9890109.89 & 54395604.40 & 21428571.43 \\ -9890109.89 & -2884615.38 & -11538461.54 & -131868131.87 & 21428571.43 & 134752747.25 \end{bmatrix}$$

Element Stiffness Matrix for Triangle 1

$$K_{\text{global}} = \begin{bmatrix} 27197802.2 & 10714285.7 & -18956043.96 & 824175.82 & 0 & 0 & 0 & 0 \\ 10714285.7 & 67376373.63 & -824175.82 & -64491758.24 & 0 & 0 & 0 & 0 \\ -18956043.96 & -824175.82 & 35439560.44 & -10714285.7 & -8241758.24 & 11538461.54 & 0 & 0 \\ 824175.82 & -64491758.24 & -10714285.7 & 70260989.01 & -9890109.89 & -2884615.38 & 0 & 0 \\ 0 & 0 & -8241758.24 & -9890109.89 & 54395604.40 & 21428571.43 & -46153846.15 & -11538461.54 \\ 0 & 0 & 11538461.54 & -2884615.38 & 21428571.43 & 134752747.25 & -9890109.89 & -131868131.87 \\ 0 & 0 & 0 & 0 & -46153846.15 & -9890109.89 & 46153846.15 & 11538461.54 \\ 0 & 0 & 0 & 0 & -11538461.54 & -131868131.87 & 11538461.54 & 134752747.25 \end{bmatrix}$$

The element stiffness matrices (K_1 and K_2) reflect the individual contributions of each triangular element to the overall stiffness of the system. The global stiffness matrix is assembled by mapping the local degrees of freedom of each triangle to the global system, ensuring continuity across shared nodes.

Applying the Boundary Conditions

To solve for the global system, I applied the boundary conditions and accounted for the traction force along the top horizontal edge.

Boundary Conditions

Dirichlet Conditions: The bottom left corner (Node 1) is fixed, meaning both x- and y-direction displacements are zero ($u = 0$, $v = 0$). This corresponds to Degrees of Freedom (DOFs) 0 and 1.

Traction Force: A traction of $t_y = -20\,\text{N/m}$ is applied along the edge between Node 2 and Node 4, with a length $L = 2\,\text{m}$.

To incorporate these conditions:

1. The rows and columns corresponding to constrained DOFs were zeroed out, and the diagonal entries set to 1.

2. The force contributions due to traction were integrated along the edge between Nodes 2 and 4 and added to the global force vector.

Code Implementation:

Listing 15: Python Code for Applying Boundary Conditions

```
'''
Apply the boundary conditions
'''
# Initialize the global force vector (8 DOFs for 4 nodes
    )
F_global = sp.zeros(8, 1)

# Apply the boundary conditions
constrained_dofs = [0, 1]  # DOFs for Node 1 (fixed at x
    =0, y=0)

# Modify the global stiffness matrix and force vector
    for the constrained DOFs
for dof in constrained_dofs:
    K_global[dof, :] = sp.zeros(1, K_global.shape[1])  #
        Zero out the entire row
    K_global[:, dof] = sp.zeros(K_global.shape[0], 1)  #
        Zero out the entire column
    K_global[dof, dof] = 1  # Set diagonal to 1 to
       maintain structure
    F_global[dof] = 0  # Set force at constrained DOF to
        zero

# Define edge length and traction force
t_y = -20  # Traction in the y-direction
L_edge = sp.sqrt((2 - 2)**2 + (1 - 0.5)**2)  # Length of
    edge between Node 3 and Node 4 (0.5)

# Shape functions along the edge
eta = sp.symbols('eta')  # Natural coordinate along the
    edge
N3_edge = (1 - eta) / 2  # Shape function for Node 3
N4_edge = (1 + eta) / 2  # Shape function for Node 4
```

```
# Integrate the force contributions
F3 = sp.integrate(N3_edge * t_y * L_edge / 2, (eta, -1,
    1))
F4 = sp.integrate(N4_edge * t_y * L_edge / 2, (eta, -1,
    1))

# Assign contributions to the global force vector
F_global[4] += F3  # y-direction DOF of Node 3
F_global[6] += F4  # y-direction DOF of Node 4

# Print the updated global force vector
print("\nUpdated Global Force Vector with Traction
    Contribution:")
sp.pprint(F_global)
```

Output

Element Stiffness Matrix for Triangle 1

$$K_{\text{global}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 35439560.44 & -10714285.71 & -8241758.24 & 21428571.43 & -8241758.24 & 9890109.89 \\ 0 & 0 & -10714285.71 & 70260989.01 & 21428571.43 & -2884615.38 & -11538461.54 & -2884615.38 \\ 0 & 0 & -8241758.24 & 21428571.43 & 54395604.40 & 21428571.43 & -46153846.15 & -9890109.89 \\ 0 & 0 & 21428571.43 & -2884615.38 & 21428571.43 & 134752747.25 & -9890109.89 & -131868131.87 \\ 0 & 0 & -8241758.24 & -11538461.54 & -46153846.15 & -9890109.89 & 46153846.15 & 11538461.54 \\ 0 & 0 & 9890109.89 & -2884615.38 & -9890109.89 & -131868131.87 & 11538461.54 & 134752747.25 \end{bmatrix}$$

Updated Global Force Vector with Traction Contribution:

Updated Global Force Vector with Traction Contribution

$$F_{\text{global}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -5.0 \\ 0 \\ -5.0 \\ 0 \end{bmatrix}$$

Reduced Global Stiffness Matrix

Since ξ is fixed for the traction force and Nodes 2 and 4 are on the same edge, I simplified the matrices. By focusing on the active degrees of freedom, I extracted a reduced 4×4 global stiffness matrix, K_{reduced}, and defined the corresponding displacement and force vectors.

Code Implementation:

Listing 16: Python Code for Reducing the Global Stiffness Matrix

```
'''
Since xi is fixed for the traction force and nodes 3 and
    4 are on the same edge,
I can simplify the matrices.
So I set up a reduced matrix for the K and d values.
'''

# Extract the bottom-right 4x4 section
K_reduced = K_global[-4:, -4:]

# Print the extracted matrix
print("\nReduced Global Stiffness Matrix (K_reduced):")
sp.pprint(K_reduced)

# Define the displacement vector for Nodes 3 and 4
dy = sp.Matrix([sp.symbols('ux3'), sp.symbols('uy3'), sp
    .symbols('ux4'), sp.symbols('uy4')])

# Define the reduced force vector
f_reduced = sp.Matrix([-5, 0, -5, 0])
```

Output:

Element Stiffness Matrix for Triangle 1

$$\mathrm{K}_{\text{reduced}} = \begin{bmatrix} 62637362.64 & 0 & -46153846.15 & -11538461.54 \\ 0 & 137637362.64 & -9890109.89 & -131868131.87 \\ -46153846.15 & -9890109.89 & 54395604.40 & 21428571.43 \\ -11538461.54 & -131868131.87 & 21428571.43 & 134752747.25 \end{bmatrix}$$

Reduced Displacement and Force Vectors: The reduced displacement and force vectors were defined as follows:

$$\mathbf{d}_{\text{reduced}} = \begin{bmatrix} u_{x3} \\ u_{y3} \\ u_{x4} \\ u_{y4} \end{bmatrix}, \quad \mathbf{f}_{\text{reduced}} = \begin{bmatrix} -5 \\ 0 \\ -5 \\ 0 \end{bmatrix}.$$

Solving the Reduced System

To determine the displacements for the unconstrained nodes, I solved the reduced system using LU decomposition. The equation $K_{\text{reduced}}\mathbf{d}_{\text{reduced}} = \mathbf{f}_{\text{reduced}}$ was solved, where:

$$K_{\text{reduced}} = \begin{bmatrix} 62637362.64 & 0 & -46153846.15 & -11538461.54 \\ 0 & 137637362.64 & -9890109.89 & -131868131.87 \\ -46153846.15 & -9890109.89 & 54395604.40 & 21428571.43 \\ -11538461.54 & -131868131.87 & 21428571.43 & 134752747.25 \end{bmatrix},$$

$$\mathbf{f}_{\text{reduced}} = \begin{bmatrix} -5 \\ 0 \\ -5 \\ 0 \end{bmatrix}.$$

Code Implementation:

Listing 17: Python Code for Solving the Reduced System

```
'''
Solve the reduced system
'''

d_reduced = K_reduced.LUsolve(f_reduced)

# Print the displacements for unconstrained nodes
print("\nDisplacements for Unconstrained Nodes:")
sp.pprint(d_reduced)
```

Output:

The computed displacements for the unconstrained nodes are:

Displacements for Unconstrained Nodes

$$\mathbf{d}_{\text{reduced}} = \begin{bmatrix} -3.99197710425073 \times 10^{-7} \\ 6.8474555065528 \times 10^{-8} \\ -4.59926183103437 \times 10^{-7} \\ 1.05964780582403 \times 10^{-7} \end{bmatrix}.$$

Strains and Stresses

Using the reduced displacement vectors, I computed the strains and stresses for both triangles using the B-matrices and material properties.

Displacements:

The displacements for the nodes of each triangle were organized as follows:

- For Triangle 1 (Nodes 1, 2, 3): Degrees of Freedom (DOFs) $[0, 1, 2, 3, 4, 5]$
- For Triangle 2 (Nodes 2, 3, 4): Degrees of Freedom (DOFs) $[2, 3, 4, 5, 6, 7]$

Code Implementation:

Listing 18: Python Code for Computing Strains and Stresses

```
'''
Calculate the resulting strains and stresses
'''

# Extract displacements for Triangle 1 (Nodes 1, 2, 3 ->
    DOFs [0, 1, 2, 3, 4, 5])
d1 = sp.Matrix([
    0, 0,  # Node 1 (fixed)
    d_reduced[0], d_reduced[1],  # Node 2
    d_reduced[2], d_reduced[3]   # Node 3
])

# Compute strain for Triangle 1
strain_1 = B1 * d1
print("\nStrain for Triangle 1:")
sp.pprint(strain_1)

# Compute stress for Triangle 1
stress_1 = D * strain_1
print("\nStress for Triangle 1:")
sp.pprint(stress_1)

# Extract displacements for Triangle 2 (Nodes 2, 3, 4 ->
    DOFs [2, 3, 4, 5, 6, 7])
d2 = sp.Matrix([
    d_reduced[0], d_reduced[1],  # Node 2
    d_reduced[2], d_reduced[3],  # Node 3
    0, 0  # Node 4 (fixed)
])

# Compute strain for Triangle 2
strain_2 = B2 * d2
print("\nStrain for Triangle 2:")
sp.pprint(strain_2)

# Compute stress for Triangle 2
stress_2 = D * strain_2
print("\nStress for Triangle 2:")
sp.pprint(stress_2)
```

Outputs:

Strain and Stress for Triangle 1:

Strain for Triangle 1

$$\text{Strain}_1 = \begin{bmatrix} -1.3016366394545 \times 10^{-7} \\ -6.8474555065528 \times 10^{-8} \\ 4.35061461949892 \times 10^{-7} \end{bmatrix}$$

Stress for Triangle 1

$$\text{Stress}_1 = \begin{bmatrix} -4.96833067467392 \\ -3.54473585436802 \\ 5.01993994557568 \end{bmatrix}$$

Strain and Stress for Triangle 2:

Strain for Triangle 2

$$\text{Strain}_2 = \begin{bmatrix} 1.99598855212536 \times 10^{-7} \\ -2.11929561164805 \times 10^{-7} \\ 8.85615088674111 \times 10^{-7} \end{bmatrix}$$

Stress for Triangle 2

$$\text{Stress}_2 = \begin{bmatrix} 4.48417539109104 \\ -5.01263421761685 \\ 10.2186356385474 \end{bmatrix}$$

Gauss Point Evaluation of Strain and Stress

I also was able to evaluate the strain and stress at each node within each triangular element by using Gauss point integration. For simplicity, I applied a single Gauss point with barycentric coordinates $(\xi, \eta) = (1/3, 1/3)$ and weight $w = 1$.

Procedure:

- At each Gauss point, the Jacobian matrix J, its determinant $\det(J)$, and its inverse J^{-1} were evaluated.

- The derivatives of the shape functions in global coordinates were calculated using J^{-1}.
- The B-matrix was reconstructed for the Gauss point.
- The strain was computed as $\varepsilon = B \cdot d$, where d is the displacement vector for the triangle.
- The stress was calculated as $\sigma = D \cdot \varepsilon$.

Code Implementation:

Listing 19: Python Code for Gauss Point Evaluation of Strain and Stress

```
'''
Code to get the stress and strain values for each node
   separately
'''

# Gauss Point for 1-point integration
xi_eta_points = [(1/3, 1/3)]  # You can add more points
   for higher-order integration
weights = [1]  # Corresponding weights

# Loop over Gauss points for Triangle 1
print("\nStrains and Stresses for Triangle 1:")
for (xi, eta), w in zip(xi_eta_points, weights):
    # Evaluate the Jacobian and its determinant at this
       Gauss point
    J_eval = J_triangle1.subs({'xi': xi, 'eta': eta})
    det_J_eval = det_J_triangle1.subs({'xi': xi, 'eta':
       eta})
    inv_J_eval = J_eval.inv()

    # Compute derivatives in global coordinates
    global_derivatives = inv_J_eval * dN_parent

    # Construct the B-matrix for this Gauss point
    B_gauss = sp.zeros(3, 6)
    B_gauss[0, 0] = global_derivatives[0, 0]
    B_gauss[0, 2] = global_derivatives[0, 1]
    B_gauss[0, 4] = global_derivatives[0, 2]
    B_gauss[1, 1] = global_derivatives[1, 0]
    B_gauss[1, 3] = global_derivatives[1, 1]
```

```
    B_gauss[1, 5] = global_derivatives[1, 2]
    B_gauss[2, 0] = global_derivatives[1, 0]
    B_gauss[2, 2] = global_derivatives[1, 1]
    B_gauss[2, 4] = global_derivatives[1, 2]
    B_gauss[2, 1] = global_derivatives[0, 0]
    B_gauss[2, 3] = global_derivatives[0, 1]
    B_gauss[2, 5] = global_derivatives[0, 2]

    # Compute strain at this Gauss point
    strain_gauss = B_gauss * d1
    print(f"\nStrain at Gauss Point ({xi}, {eta}):")
    sp.pprint(strain_gauss)

    # Compute stress at this Gauss point
    stress_gauss = D * strain_gauss
    print(f"\nStress at Gauss Point ({xi}, {eta}):")
    sp.pprint(stress_gauss)

# Repeat for Triangle 2
print("\nStrains and Stresses for Triangle 2:")
for (xi, eta), w in zip(xi_eta_points, weights):
    # Evaluate the Jacobian and its determinant at this
       Gauss point
    J_eval = J_triangle2.subs({'xi': xi, 'eta': eta})
    det_J_eval = det_J_triangle2.subs({'xi': xi, 'eta':
       eta})
    inv_J_eval = J_eval.inv()

    # Compute derivatives in global coordinates
    global_derivatives = inv_J_eval * dN_parent

    # Construct the B-matrix for this Gauss point
    B_gauss = sp.zeros(3, 6)
    B_gauss[0, 0] = global_derivatives[0, 0]
    B_gauss[0, 2] = global_derivatives[0, 1]
    B_gauss[0, 4] = global_derivatives[0, 2]
    B_gauss[1, 1] = global_derivatives[1, 0]
    B_gauss[1, 3] = global_derivatives[1, 1]
    B_gauss[1, 5] = global_derivatives[1, 2]
    B_gauss[2, 0] = global_derivatives[1, 0]
    B_gauss[2, 2] = global_derivatives[1, 1]
    B_gauss[2, 4] = global_derivatives[1, 2]
    B_gauss[2, 1] = global_derivatives[0, 0]
```

```
    B_gauss[2, 3] = global_derivatives[0, 1]
    B_gauss[2, 5] = global_derivatives[0, 2]

    # Compute strain at this Gauss point
    strain_gauss = B_gauss * d2
    print(f"\nStrain at Gauss Point ({xi}, {eta}):")
    sp.pprint(strain_gauss)

    # Compute stress at this Gauss point
    stress_gauss = D * strain_gauss
    print(f"\nStress at Gauss Point ({xi}, {eta}):")
    sp.pprint(stress_gauss)
```

Outputs:

Strains and Stresses for Triangle 1:

Strain and Stress for Triangle 1 at Gauss Point

$$\text{Strain}_1 = \begin{bmatrix} -1.3016366394545 \times 10^{-7} \\ -6.8474555065528 \times 10^{-8} \\ 4.35061461949892 \times 10^{-7} \end{bmatrix}, \quad \text{Stress}_1 = \begin{bmatrix} -4.96833067467392 \\ -3.54473585436802 \\ 5.01993994557568 \end{bmatrix}$$

Strains and Stresses for Triangle 2:

Strain and Stress for Triangle 2 at Gauss Point

$$\text{Strain}_2 = \begin{bmatrix} 1.99598855212536 \times 10^{-7} \\ -2.11929561164805 \times 10^{-7} \\ 8.85615088674111 \times 10^{-7} \end{bmatrix}, \quad \text{Stress}_2 = \begin{bmatrix} 4.48417539109104 \\ -5.01263421761685 \\ 10.2186356385474 \end{bmatrix}$$

These results match the values from the previous calculations, confirming the accuracy of the Gauss point integration method.

Conclusion

The analysis of the element stiffness matrix for Triangle 1 was successfully completed, ensuring numerical consistency and compliance with the given boundary conditions. Through the derived stiffness matrix, the structural behavior of the triangular element under the specified conditions has been accurately represented. This outcome provides a robust foundation for further calculations, including force vector determination and nodal displacements. The results align with the theoretical expectations, confirming the validity of the applied method and calculations.