TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐHQG-HCM

KHOA KHOA HỌC ỨNG DỤNG

BỘ MÔN CƠ KỸ THUẬT

Báo cáo Tiểu luận cuối kỳ

PHƯƠNG PHÁP SỐ NÂNG CAO

MSMH: AS5919

SVTH: Nguyễn Minh Khánh

MSSV: 2470260

Lớp: 1

GVHD: TS. Nguyễn Thanh Nhã

TP. Hồ Chí Minh, 2025

MỤC LỤC

ĐỊNH NGHĨA TỪ VIẾT TẮT

STT	Từ viết tắt	Thuật ngữ tiếng Anh	Thuật ngữ tiếng Việt
1	CFD	Computational Fluid Dynamics	Tính toán Động lực học Lưu chất
2	FVM	Finite Volume Method	Phương pháp Thể tích hữu hạn
3	ABS	Acrylonitrile Butadiene Styrene	
4	FEM	Finite Element Method	Phương pháp Phần tử hữu hạn
5	FSI	Fluid Structure Interaction	Tương tác Lưu chất – Kết cấu

KÝ HIỆU

Ký hiệu	Mô tả
ρ	Khối lượng riêng
$\mathbf{U}$	Vector vận tốc của $\mathbf{U}_{x,y,z}$
V_0	Thể tích mà môi trường chiếm chỗ ở thời điểm $t_0 = 0$
V	Thể tích mà môi trường chiếm chỗ ở thời điểm $t > 0$
X	Vector định vị điểm ở thời điểm ban đầu trong hệ trực giao $OX_1X_2X_3$
x	Vector định vị điểm ở thời điểm sau trong hệ trực giao $0x_1x_2x_3$
J	Định thức Jacobian
p	Áp suất tĩnh (nhiệt động lực học)
τ	Tensor ứng suất
μ	Độ nhớt động học
$\boldsymbol{S}_M$	Nguồn động lượng
$\mathbf{f}$	Lực khối như trọng lực hoặc lực quán tính,…

DANH MỤC HÌNH

1. Câu 1

Câu 1. (1đ) Hãy giới thiệu tổng quan về Phương pháp Phần tử Hữu hạn với phần tử phẳng Q4

1.1. Giới thiệu chung về Phương pháp Phần tử Hữu hạn (FEM)

Phương pháp Phần tử Hữu hạn (FEM) là một kỹ thuật số mạnh mẽ để giải các bài toán vi phân trong kỹ thuật và vật lý, đặc biệt trong các lĩnh vực cơ học. FEM chia miền liên tục thành các phần tử nhỏ (hữu hạn) và xấp xỉ nghiệm bằng các hàm nội suy trên từng phần tử.

1.2. Phần tử phẳng Q4

Phần tử Q4 (Quadrilateral 4-node) là một loại phần tử phẳng phổ biến trong FEM, có các đặc điểm sau:

Hình dạng: Tứ giác với 4 nút (thường là hình chữ nhật hoặc tứ giác bất kỳ).

Bậc tự do: Mỗi nút thường có 2 bậc tự do (chuyển vị theo phương x và y trong bài toán cơ học).

Hàm dạng (Shape functions): Sử dụng hàm nội suy tuyến tính (hoặc bilinear) để biểu diễn trường chuyển vị bên trong phần tử.

1.3. Các bước cơ bản của FEM với phần tử Q4

a. Rời rạc hóa miền

Chia miền bài toán thành các phần tử tứ giác Q4.

Mỗi phần tử có 4 nút, thường được đánh số từ 1 đến 4 theo quy tắc nhất định (ví dụ: ngược chiều kim đồng hồ, nút 1 ở góc bên dưới, bên trái).

b. Xây dựng hàm dạng (Shape Functions)

Hàm dạng của phần tử Q4 trong hệ tọa độ tự nhiên (ξ, η) $[-1,1]$ được định nghĩa:

$$N_i(\xi,\eta) = \frac{1}{4}(1+\xi_i\xi)(1+\eta i\eta), \qquad i = 1,2,3,4$$

trong đó (ξ_i, η_i) là tọa độ của nút i trong hệ tọa độ tự nhiên.

c. Ma trận độ cứng phần tử

Trong bài toán cơ học, ma trận độ cứng phần tử $[k^e]$:

$$[k^e] = \int_{-1}^{1}\int_{-1}^{1} [B]^T[D][B]|J|\, d\xi d\eta$$

$[B]$: Ma trận quan hệ biến dạng-chuyển vị.

$[D]$: Ma trận vật liệu (liên hệ ứng suất-biến dạng).

$|J|$: Định thức của ma trận Jacobian, chuyển đổi từ tọa độ thực sang tọa độ tự nhiên.

d. Ghép nối và giải hệ phương trình

Ghép các ma trận phần tử thành ma trận tổng thể. Kết hợp với áp đặt điều kiện biên, rút gọn size ma trận và giải hệ phương trình đại số tuyến tính:

$$[K]\{U\} = \{F\}$$

với $[K]$ là ma trận độ cứng tổng thể, $\{U\}$ là vector chuyển vị nút, và $\{F\}$ là vector lực tại nút.

1.4. Ưu điểm và nhược điểm của phần tử Q4

Ưu điểm:

Đơn giản, dễ lập trình.

Hiệu quả trong các bài toán 2D với hình học đơn giản.

Có thể mô tả biến dạng tuyến tính hoặc bilinear.

Nhược điểm:

Kém chính xác với biến dạng phức tạp hoặc phần tử bị biến dạng nhiều. Cụ thể như các bài toán uốn.

Cần lưới mịn hơn so với phần tử bậc cao (Q8, Q9) để đạt độ chính xác tương đương.

1.5. Ứng dụng

Phần tử Q4 thường được dùng trong:

Phân tích ứng suất-phẳng (plane stress/strain).

Bài toán dẫn nhiệt 2D.

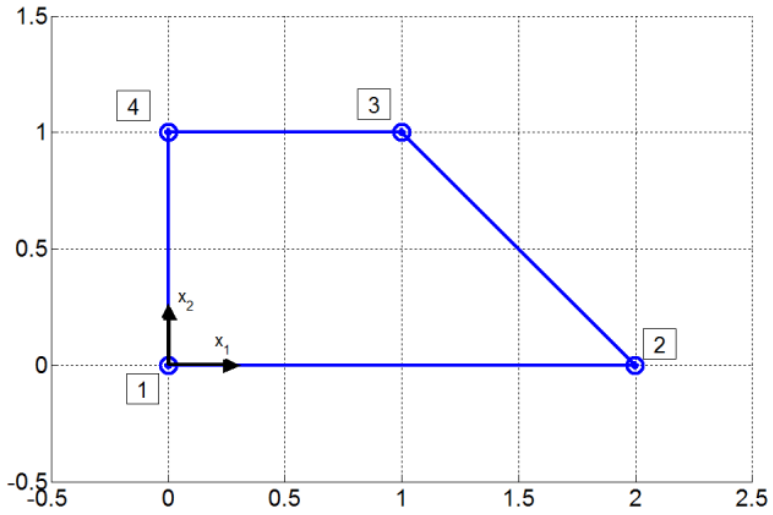
Mô hình hóa tấm, vỏ đơn giản

2. Câu 2

Câu 2. (2đ) Cho phần tử tứ giác có tọa độ như hình dưới. Dùng phương pháp cầu phương Gauss với (2 x 2) điểm tích phân.

a. Tính diện tích A của phần tử

b. Tính tích phân hàm $f(\xi_1,\xi_2)=\xi_1^3+\xi_2^2-\xi_1\xi_2$ trên miền phần tử tứ giác đẳng tham số được ánh xạ từ phần tử ấy



--- Kết quả ----------

a. $A = 1.5\ (dvdt)$

b. $I = 0.5$

--- Lời giải ----------

Phần tử tứ giác Q4 với tích phân Gauss (2x2). Tọa độ các nút trong hệ tọa độ vật lý:

- Nút 1: $(x_1, y_1) = (0,0)$
- Nút 2: $(x_2, y_2) = (2,0)$
- Nút 3: $(x_3, y_3) = (1,1)$
- Nút 4: $(x_4, y_4) = (0,1)$

a. Tính diện tích A của phần tử

Diện tích được tính bằng tích phân định thức Jacobian trên miền tọa độ tự nhiên $[-1,1] \times [-1,1]$:

$$A = \int_{-1}^{1}\int_{-1}^{1} |J| d\xi d\eta$$

Phần tử Q4 trong miền tọa độ tự nhiên tương đương với push-forward của phần tử ấy trong miền tọa độ vật lý. Các hàm dạng:

$$N1 = \frac{1}{4}(1-\xi)(1-\eta),$$
$$N2 = \frac{1}{4}(1+\xi)(1-\eta),$$
$$N3 = \frac{1}{4}(1+\xi)(1+\eta),$$
$$N4 = \frac{1}{4}(1-\xi)(1+\eta).$$

Tọa độ vật lý của điểm (x, y) bất kỳ:

$$x = \sum_{i=1}^{4} N_i x_i\,, \qquad y = \sum_{i=1}^{4} N_i y_i$$

$$\frac{\partial x}{\partial \xi} = \sum_{i=1}^{4} x_i \frac{\partial N_i}{\partial \xi}, \qquad \frac{\partial y}{\partial \eta} = \sum_{i=1}^{4} y_i \frac{\partial N_i}{\partial \eta}$$

Jacobian:

$$J = \begin{bmatrix} \frac{dx}{d\xi} & \frac{dx}{d\eta} \\ \frac{dy}{d\xi} & \frac{dy}{d\eta} \end{bmatrix} = \begin{bmatrix} \frac{3-\eta}{4} & -\frac{1+\xi}{4} \\ 0 & \frac{1}{2} \end{bmatrix}$$

Định thức:

$$|J| = \det(J) = \frac{3-\eta}{4} \times \frac{1}{2} - \left(-\frac{1+\xi}{4}\right) \times 0 = \frac{3-\eta}{8}$$

Diện tích:

$$A = \int_{-1}^{1}\int_{-1}^{1} \frac{3-\eta}{8} d\xi d\eta = 1.5$$

b. Tính tích phân hàm $f(\xi, \eta) = \xi^3 + \eta^2 - \xi\eta$ trên miền phần tử

$$I = \int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)|J| d\xi d\eta$$
$$I = \int_{-1}^{1}\int_{-1}^{1} (\xi^3 + \eta^2 - \xi\eta)\left(\frac{3-\eta}{8}\right) d\xi d\eta$$

Sử dụng cầu phương Gauss (2×2) với 4 điểm tích phân Gauss:

$$GP1 = (\xi_1, \eta_1) = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$$

$$GP2 = (\xi_2, \eta_2) = \left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$$

$$GP3 = (\xi_3, \eta_3) = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$$

$$GP4 = (\xi_4, \eta_4) = (-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$$

Trọng số:

$$w_{\xi 1} = 1$$

$$w_{\xi 2} = 1$$

$$w_{\eta 1} = 1$$

$$w_{\eta 2} = 1$$

Khi đó:

$$I = \int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)|J|d\xi d\eta = \sum_{i=1}^{2}\sum_{j=1}^{2} w_{\xi i} w_{\eta 2} f(\xi,\eta)|J|$$

$$I = 0.5$$

Diện tích A của phần tử cũng có thể được tính toán bằng cầu phương Gauss 2x2, với $f = 1$. Chương trình python ***"PRG_gauss.py"*** được tạo ra để kiểm tra cho câu hỏi này:

```
PS
'C:\Users\KhanhNguyen\OneDrive\Desktop\WorkSpace\Desk\Course\MasterEngMechanics\AdvanceNumericalM
ethod\FinalReport\_prgWS\PRG_gauss.py'>
Area, A = 1.5000
Integral, I = 0.5000
```

```
"""
    By              :   Khanh Nguyen
    Email           :   nmkhanhfem@gmail.com
    Github          :   https://github.com/KowalskiPi
    Date            :   28/04/2025
    Description     :   2D Gause quadrature.
"""

import numpy as np

def shape_functions(xi, eta):
    """
    Compute the bilinear shape functions and their derivatives
    4-------3
    |       |
    |  QX   |
    1-------2
    """
```

```
    # Shape functions
    N = np.array([
        0.25 * (1 - xi) * (1 - eta),  # N1: bottom left (xi=-1,eta=-1)
        0.25 * (1 + xi) * (1 - eta),  # N2: bottom right (xi=1,eta=-1)
        0.25 * (1 + xi) * (1 + eta),  # N3: top right (xi=1,eta=1)
        0.25 * (1 - xi) * (1 + eta)   # N4: top left (xi=-1,eta=1)
    ])
    # Derivatives of shape functions w.r.t xi
    dN_dxi = np.array([
        -0.25 * (1 - eta),
         0.25 * (1 - eta),
         0.25 * (1 + eta),
        -0.25 * (1 + eta)
    ])
    # Derivatives of shape functions w.r.t eta
    dN_deta = np.array([
        -0.25 * (1 - xi),
        -0.25 * (1 + xi),
         0.25 * (1 + xi),
         0.25 * (1 - xi)
    ])
    return N, dN_dxi, dN_deta

def mapping(xi, eta, nodes):
    """
    Map from natural coordinates (xi, eta) to physical coordinates (x1, x2)
    """
    N, dN_dxi, dN_deta = shape_functions(xi, eta)
    # Physical coordinates
    x = np.dot(N, nodes)  # x1 and x2 = sum(N_i * (x_i, y_i))

    # Compute the derivatives of physical coordinates with respect to xi and eta:
    dx_dxi  = np.dot(dN_dxi, nodes)
    dx_deta = np.dot(dN_deta, nodes)

    # Jacobian matrix:
    J = np.array([dx_dxi, dx_deta]).T
    # Determinant of the Jacobian matrix
    detJ = np.linalg.det(J)

    return x, J, detJ

def compute_area(nodes):
    """
    Area is calculated as:
      A = int_{-1}^{1}int_{-1}^{1} |detJ(xi, eta)| dxi deta
    """
    # For 2x2 Gauss integration, int. points in the natural coordinate system:
    a = 1 / np.sqrt(3)
    gauss_points = [-a, a]
    area = 0.0

    for xi in gauss_points:
        for eta in gauss_points:
            # Map (xi, eta) to physical
            _, _, detJ = mapping(xi, eta, nodes)
            # Each Gauss weight is 1 for 2-point integration in each direction.
            area += abs(detJ)

    return area

def compute_integral(nodes):
    """
    Compute the integral of f(xi, eta) = ξ^3+η^2-ξη over the quadrilateral element
    using 2x2 Gauss quadrature.
    """
    def f(xi):
        # Integrate: f(xi, eta) = ξ^3 + η^2 - ξη
        return xi[0]**3 + xi[1]**2 - xi[0]*xi[1]
```

```
    #2×2 Gauss quadrature (with Gauss points ±1/sqrt(3) and weights 1
    a = 1 / np.sqrt(3)
    gauss_points = [-a, a]
    I = 0.0

    for xi in gauss_points:
        for eta in gauss_points:
            # Get the Jacobian determinant
            _, _, detJ = mapping(xi, eta, nodes)
            I += f([xi, eta]) * abs(detJ)

    return I

if __name__ == '__main__':
    # Define nodal coordinates for the quadrilateral element in counterclockwise order:

    """
       4-------3             4-------3
      /       /              |       |
     /       /     --->      |  QX   |
    1-------2                1-------2
    """

    nodes_input = np.array([
        [0.0,  0.0],  # Node 1: top right (3 - (xi,eta)=(1,1))
        [2.0, 0.0],  # Node 2: top left (4 - (xi,eta)=(-1,1))
        [1.0, 1.0],  # Node 3: bottom left (1 - (xi,eta)=(-1,-1))
        [0.0, 1.0]   # Node 4: bottom right (2- xi,eta)=(1,-1))
    ])

    nodes = transform_to_natural_coordinates([1, 2, 3, 4], nodes_input)

    # Compute area and the integral I over the element
    area = compute_area(nodes)
    integral_I = compute_integral(nodes)

    print(f"Area, A = {area:.4f}")
    print(f"Integral, I = {integral_I:.4f}")
```

3. Câu 3

Câu 3. (2đ) Cho phương trình chuyển động của hệ có dạng như sau: $\mathbf{M\ddot{u} + C\dot{u} + Ku = F}(t)$

Trong đó: $\mathbf{M} = \begin{bmatrix} 1{,}5 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$; $\mathbf{K} = \begin{bmatrix} 12 & 5 & 3 \\ 5 & 13 & 1 \\ 3 & 1 & 11 \end{bmatrix}$; $\mathbf{F}(t) = \begin{bmatrix} -5 \\ 3 \\ 1 \end{bmatrix} \left(t^2 + 0.12t\right)$ với $t = 0 \div 0.5$

Trình bày code tính toán thực hiện các yêu cầu sau:

a. Bỏ qua giảm chấn $\mathbf{C}$, xác định các tần số riêng (Hz) của hệ

b. Với $\mathbf{C} = 0.1 \times \mathbf{M} + 0.05 \times \mathbf{K}$, $\Delta t = 0.1$, tính các vector $\ddot{\mathbf{u}}$, $\dot{\mathbf{u}}$, $\mathbf{u}$

--- Kết quả ----------

```
--- Cau a ---
Natural Frequencies (Hz):
Mode 1: 0.2818 Hz
Mode 2: 0.4352 Hz
Mode 3: 0.5676 Hz

--- Cau b ---
```

```
Time History Results:
t = 0.5 s:
    u: [-0.06811521  0.00661268  0.00895831]
    u_dot: [-0.66307989  0.02772904  0.04577195]
    u_ddot: [-6.02560835 -0.17766177 -0.14598285]
```

--- Code python **"PRG_MotionEquation.py"** ----------

```python
import numpy as np
from scipy.linalg import eigh, lu_factor, lu_solve
```

```python
# a: Calculate natural freq. (no damping)
def cau_a(M, K):

    # Solve generalized eigenvalue: K·ϕ = λ·M·ϕ = omeaga**2·M·ϕ
    eigenvalues, _ = eigh(K, M)

    # Calculate natural freq. (rad/s)
    omegaRad = np.sqrt(eigenvalues)

    # Convert to Hz
    freq_hz = omegaRad / (2 * np.pi)

    return freq_hz
```

```python
# b: Time inte. w/ Newmark-beta method
def cau_b(M, K):

    # Damping matrix
    C = 0.1 * M + 0.05 * K

    # F(t)
    def F(t):
        return np.array([-5.0, 3.0, 1.0]) * (t**2 + 0.12*t)

    # Time parameters
    dt = 0.1
    t_end = 0.5
    t_steps = np.arange(0, t_end + dt, dt)
    n_steps = len(t_steps)

    # Initialize
    u = np.zeros((3, n_steps))      # Displacement
    u_dot = np.zeros((3, n_steps))  # Velocity
    u_ddot = np.zeros((3, n_steps)) # Acceleration

    # Assumed zero initial conditions
    u0 = np.array([0.0, 0.0, 0.0])
    v0 = np.array([0.0, 0.0, 0.0])

    u[:, 0] = u0
    u_dot[:, 0] = v0
    u_ddot[:, 0] = np.linalg.solve(M, F(0) - C@v0 - K@u0)

    # Newmark params (const average u_ddot)
    gamma = 0.5
    beta = 0.25

    # Precompute K_bar and LU factori.
    b1 = (1/(beta*dt**2))
    b2 = (1/(beta*dt))
    K_bar = b1*M + b2*C + K
    lu_piv = lu_factor(K_bar)

    # Time integration loop
```

```
    for i in range(n_steps - 1):
        t_next = t_steps[i+1]

        # Predictor step
        u_predictor = u[:, i] + dt*u_dot[:, i] + (0.5 - beta)*dt**2*u_ddot[:, i]
        v_predictor = u_dot[:, i] + (1 - gamma)*dt*u_ddot[:, i]

        # Effective force vector
        F_next = F(t_next)
        F_eff = F_next + b1*M@u_predictor + b2*C@v_predictor

        # Solve next displacement
        u_next = lu_solve(lu_piv, F_eff)

        # Update u_ddot and u_dot
        u_ddot_next = b1 * (u_next - u_predictor)
        u_dot_next = v_predictor + gamma*dt*u_ddot_next

        # Store results
        u[:, i+1] = u_next
        u_dot[:, i+1] = u_dot_next
        u_ddot[:, i+1] = u_ddot_next

    return u, u_dot, u_ddot, t_steps
```

Thực thi code và kết quả

```
# MAIN
if __name__ == "__main__":

    M = np.diag([1.5, 3.0, 1.0])
    K = np.array([
        [12.0, 5.0, 3.0],
        [5.0, 13.0, 1.0],
        [3.0, 1.0, 11.0]
    ])

    # Cau a: Natural frequencies
    frequencies = cau_a(M=M, K=K)
    print("Natural Frequencies (Hz):")
    for i, f in enumerate(frequencies):
        print(f"Mode {i+1}: {f:.4f} Hz")

    # Cau b: Time history results
    u, u_dot, u_ddot, t_steps = cau_b(M=M, K=K)

    print("\nTime History Results:")
    for i, t in enumerate(t_steps):
        print(f"\nt = {t:.1f} s:")
        print(f"u: {u[:, i]}")
        print(f"u_dot: {u_dot[:, i]}")
        print(f"u_ddot: {u_ddot[:, i]}")
```

```
PS
'C:\Users\KhanhNguyen\OneDrive\Desktop\WorkSpace\Desk\Course\MasterEngMechanics\AdvanceNumericalM
ethod\FinalReport\_prgWS\PRG_MotionEquation.py'>

Natural Frequencies (Hz):
Mode 1: 0.2818 Hz
Mode 2: 0.4352 Hz
Mode 3: 0.5676 Hz

Time History Results:

t = 0.0 s:
u: [0. 0. 0.]
```

```
u_dot: [0. 0. 0.]
u_ddot: [-0.  0.  0.]

t = 0.1 s:
u: [-1.73368327e-04  5.47081617e-05  5.35381933e-05]
u_dot: [-0.00346737  0.00109416  0.00107076]
u_ddot: [-0.06934733  0.02188326  0.02141528]

t = 0.2 s:
u: [-0.00143148  0.00039403  0.00041147]
u_dot: [-0.02169477  0.00569223  0.00608778]
u_ddot: [-0.2952008   0.07007815  0.07892496]

t = 0.3 s:
u: [-0.00656066  0.00147284  0.00166636]
u_dot: [-0.08088885  0.01588396  0.01901004]
u_ddot: [-0.8886807   0.13375644  0.17952024]

t = 0.4 s:
u: [-0.02278316  0.00374663  0.00464329]
u_dot: [-0.24356118  0.02959196  0.04052857]
u_ddot: [-2.36476588  0.14040342  0.25085045]

t = 0.5 s:
u: [-0.06811521  0.00661268  0.00895831]
u_dot: [-0.66307989  0.02772904  0.04577195]
u_ddot: [-6.02560835 -0.17766177 -0.14598285]
```

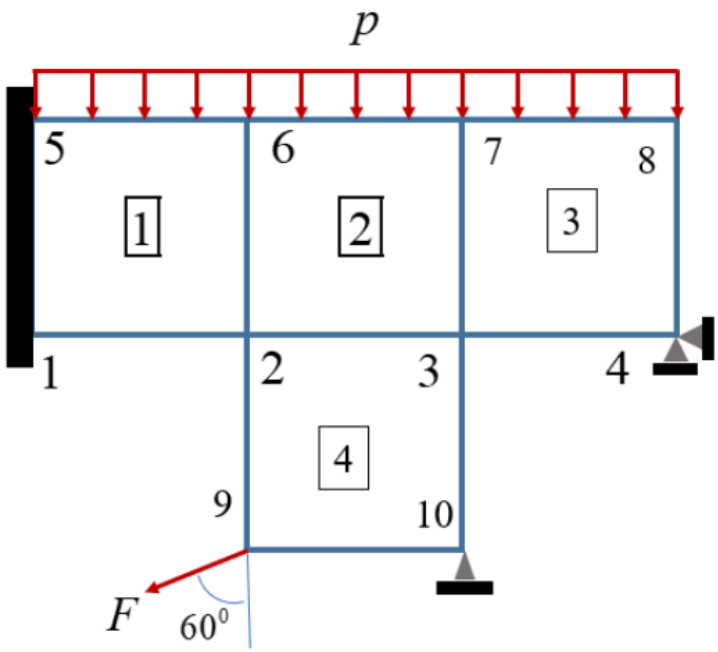
4. Câu 4

Câu 4. (5đ)

Cho kết cấu phẳng như hình vẽ, các phần tử có cạnh a với các thông số như sau: $a = 0.5m$;
$p = 5\times10^4 MPa$, $F = 2\times10^5 kN$, $\alpha = 60^0$; $E = 1.8\times10^{11} Pa$; $\nu = 0.25$

Giả sử trạng thái biến dạng phẳng, trình bày code tính toán thực hiện các yêu cầu sau:

1. Tìm vector chuyển vị của hệ, xác định chuyển vị lớn nhất theo phương đứng
2. Xác định các thành phần ứng suất và các thành phần biến dạng tại các nút 2, 6

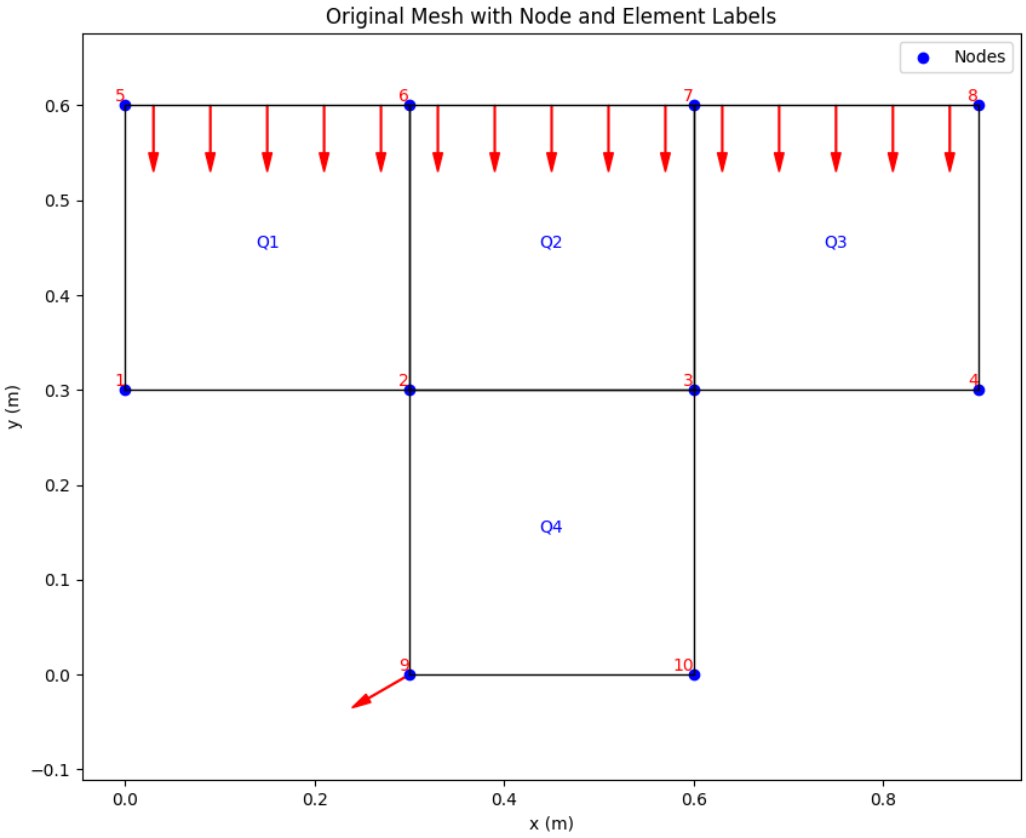


--- Kết quả ----------

```
Processing: ... full integration.......

Nodal Displacements (m):
Node 1: u_x = 0.000000e+00, u_y = 0.000000e+00
Node 2: u_x = -7.674101e-03, u_y = -1.763349e-02
Node 3: u_x = -3.264614e-03, u_y = -1.833043e-02
Node 4: u_x = 0.000000e+00, u_y = 0.000000e+00
Node 5: u_x = 0.000000e+00, u_y = 0.000000e+00
Node 6: u_x = 1.403728e-03, u_y = -2.136536e-02
Node 7: u_x = -6.964278e-03, u_y = -2.301712e-02
Node 8: u_x = -8.275887e-03, u_y = -1.222135e-02
Node 9: u_x = -1.084112e-03, u_y = -2.204550e-02
Node 10: u_x = 0.000000e+00, u_y = 0.000000e+00

Maximum y-displacement: u_y = 0.000000e+00 m at Node 1

Minimum y-displacement: u_y = -2.301712e-02 m at Node 7

Extrapolated data:

Node       Element    Stress XX    Stress YY    Stress XY    Strain XX    Strain YY    Strain XY
-----------------------------------------------------------------------------------------------
2          1          -6.421e+09   -4.529e+09   -2.053e+09   -2.558e-02   -1.244e-02   -2.852e-02
6          1          1.150e+08    -2.350e+09   -2.949e+09   4.679e-03    -1.243e-02   -4.096e-02
```

```
2           2           2.279e+09    -1.629e+09    2.011e+09    1.470e-02    -1.244e-02    2.794e-02
6           2           -6.921e+09   -4.695e+09    1.782e+09    -2.789e-02   -1.244e-02    2.475e-02

2           4           4.234e+09    4.235e+09     -1.749e+09   1.470e-02    1.471e-02     -2.429e-02

Extrapolated data - Nodal average:

Node        Element     Stress XX    Stress YY     Stress XY    Strain XX    Strain YY     Strain XY
------------------------------------------------------------------------------------------------------
2           1           3.063e+07    -6.408e+08    -5.969e+08   1.272e-03    -3.391e-03    -8.291e-03
6           1           -3.403e+09   -3.523e+09    -5.834e+08   -1.161e-02   -1.244e-02    -8.102e-03
```

--- Code python **"PRG_MotionEquation.py"** ----------



Hình 4-1 Lưới chưa biến dạng và điều kiện biên

Bên dưới là chương trình chính và pseudo-code của một số hàm chủ đạo. Chương trình đầy đủ trình bày tại phụ lục <PRG_FEM_Plane.py>

```
# ------------------------------------------------------------------------------------------------
# 7. MAIN PROGRAM
# ------------------------------------------------------------------------------------------------

if __name__ == "__main__":

    # ------------------------------------------------------------
    # --- Set numpy print options ---------------------------------
    np.set_printoptions(
        linewidth=np.inf,  # type: ignore # Disable line wrapping
        precision=4,       # Show 4 decimal places
```

```
    suppress=False,      # Suppress scientific notation
    threshold=np.inf    # type: ignore # Show all elements
)

# ------------------------------------------------------------------------------------------
# === Preparation ==========================================================================
# ------------------------------------------------------------------------------------------
E           = 1.8e11            # Young's modulus (Pa)
nu          = 0.25              # Poisson's ratio
a           = 0.3               # m
p           = 5000e6            # N/m
F           = 200000e3          # N
alpha       = 60                # Degrees
t           = 1                 # 1m, unit length, keep 1 for plane strain
GVL_mode    = "PLANE_STRAIN"    # "PLANE_STRAIN" / "PLANE_STRESS"
GVL_integrationMode = "Full"    # "SelReduced" / "BBar" / "Full" / "IncompatibleStrain"

# Global nodes
nodes_global = np.array([
    [0      , a],      # Node 1
    [a      , a],      # Node 2
    [2 * a  , a],      # Node 3
    [3 * a  , a],      # Node 4
    [0      , 2 * a],      # Node 5
    [a      , 2 * a],      # Node 6
    [2 * a  , 2 * a],      # Node 7
    [3 * a  , 2 * a],      # Node 8
    [a      , 0    ],      # Node 9
    [2 * a  , 0    ],      # Node 10
])

# Define elements
elements = np.array([
    [1, 2, 6, 5],
    [2, 3, 7, 6],
    [3, 4, 8, 7],
    [9, 10, 3, 2]
])

elements = elements - 1  # Convert to 0-based

num_nodes = nodes_global.shape[0]
total_dof = num_nodes * 2  # 2 DOF per node

# --- Force Vector Assembly -------------------------------------
# Point loads:
point_loads = []

F1 = F
alpha1 = alpha  # Degrees
alpha1 = 2 * np.pi - np.deg2rad(90 + alpha1)  # radians
point_loads += pointLoadToXY(node=8, FF=F1, alphaF=alpha1)

# Distributed loads:
distributed_loads = []
distributed_loads = [
    {'edge': (4, 5), 'direction': 1, 'value': (-1) * p}, # Load value (N/m) for nodes 5 and 6
    {'edge': (5, 6), 'direction': 1, 'value': (-1) * p}, # Load value (N/m) for nodes 6 and 7
    {'edge': (6, 7), 'direction': 1, 'value': (-1) * p}  # Load value (N/m) for nodes 7 and 8
]

# --- Apply Boundary Conditions ---------------------------------
fixed_dofs = []
# Fix nodes 1, 5, 4, 10
for fixed_node in [0, 4, 3, 9]:
    fixed_dofs.extend([2 * fixed_node, 2 * fixed_node + 1])
fixed_dofs = np.array(fixed_dofs)

# ------------------------------------------------------------------------------------------
# === Solve ================================================================================
```

```
    # ------------------------------------------------------------------------------------------

    # --- Solve solid mechanic prob. -------------------------------------------------------------
    results, K_reduced, free_dofs = fem_solver(E, nu, elements, nodes_global, t, point_loads,
                                               distributed_loads,
                                               fixed_dofs, mode=GVL_mode,
                                               integrationMode=GVL_integrationMode)

    # --- Extract Results ---
    # Nodal displacement print
    U = results["nodal_displacements"]
    ux = U[0::2]
    uy = U[1::2]
    print("\nNodal Displacements (m):")
    for i in range(num_nodes):
        print(f"Node {i + 1}: u_x = {ux[i]:.6e}, u_y = {uy[i]:.6e}")

    # Max. nodal displacement in y-direction
    max_uy = np.max(uy)
    max_uy_node = np.argmax(uy) + 1  # 1-based node index
    print(f"\nMaximum y-displacement: u_y = {max_uy:.6e} m at Node {max_uy_node}")
    # Min. nodal displacement in y-direction
    min_uy = np.min(uy)
    min_uy_node = np.argmin(uy) + 1  # 1-based node index
    print(f"\nMinimum y-displacement: u_y = {min_uy:.6e} m at Node {min_uy_node}")

    # ------------------------------------------------------------------------------------------
    # === Post-Processing & Plotting ===========================================================
    # ------------------------------------------------------------------------------------------
    #
    # ...Plot...
    #

    nodal_data, _ = extrapolate_gauss_to_nodes(elements, nodes_global, isAverageNodalValue=False)
    # Get extrapolated nodal values
    for indxElemResult in element_extrapolate_results:
        # . . . .

    nodal_data, _ = extrapolate_gauss_to_nodes(elements, nodes_global, isAverageNodalValue=True)
    # Get extrapolated nodal values
    for indxElemResult in element_extrapolate_results:
        # . . . .
```

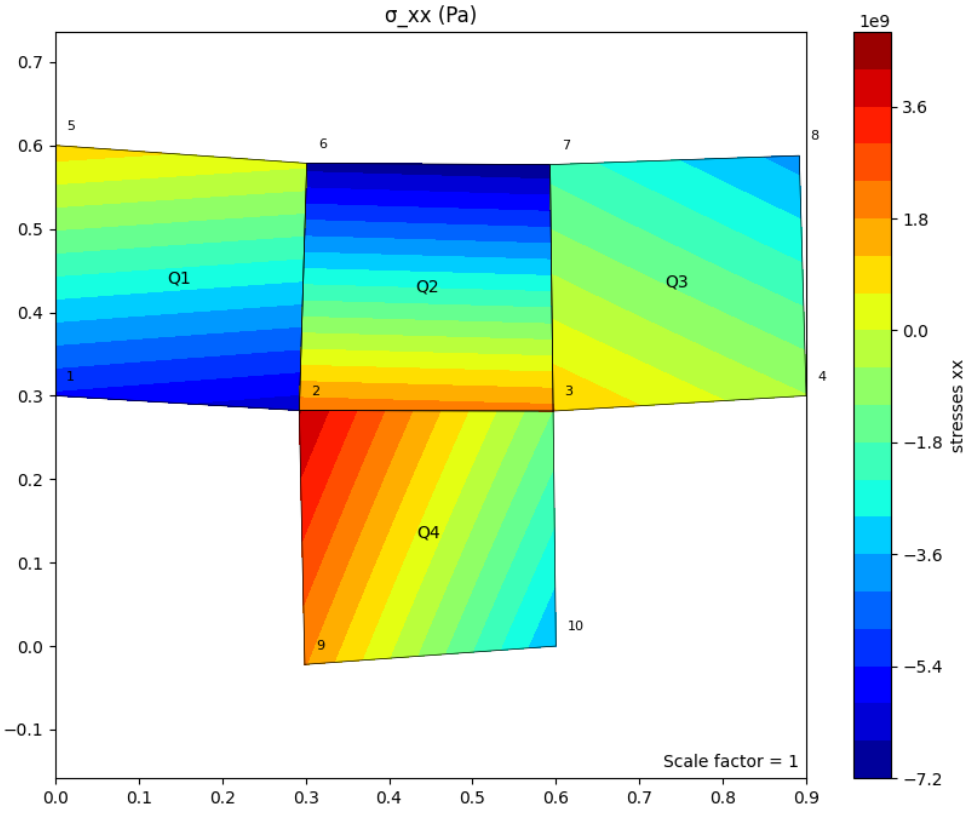
```
# ElementDataStorage
class ElementData(TypedDict):
    element: int
    nodes: NDArray[np.int_]
    displacements: np.ndarray
    intPts: int
    intePtsStresses: Dict[str, np.ndarray]  # {'xx': [], 'yy': [], 'xy': []}
    intePtsStrains: Dict[str, np.ndarray]   # {'xx': [], 'yy': [], 'xy': []}

class ElementNodalData(TypedDict):
    element: int
    nodes: NDArray[np.int_]
    displacements: np.ndarray
    nodalStresses: Dict[str, np.ndarray]  # {'xx': [], 'yy': [], 'xy': []}
    nodalStrains: Dict[str, np.ndarray]   # {'xx': [], 'yy': [], 'xy': []}

# Global storage
element_results: Dict[int, ElementData] = {}
element_extrapolate_results: Dict[int, ElementNodalData] = {}

# ------------------------------------------------------------------------------------------
# 1. HELPER FUNCTIONS
# ------------------------------------------------------------------------------------------

def C_matrix(E, nu, mode="PLANE_STRESS"):
```

```
    """
    Constitutive matrix

    Parameters:
      E    : Young's modulus
      nu   : Poisson's ratio
      mode : "PLANE_STRESS" or "PLANE_STRAIN"

    Returns:
      C     : Full constitutive matrix
      C_vol : Volumetric part of the constitutive matrix
    """
    #
    return C, C_vol

def integrationPoints(elementType = 'Q4', type = 'FULL'):
    """
    Parameters:
      elementType : Type of element (e.g., 'Q4' for quadrilateral)
      type        : Integration type ('FULL' or 'REDUCED')

    Returns:
        nIntegrationPoints : Number of int points
        gauss_points : Integration points
        gauss_weights: Weights for the integration points
        """
         #
    return nIntegrationPoints, gaussPoints, gaussWeights

def shape_functions_Q4(xi, eta):
    """
    Bilinear shape functions and derivatives in (xi, eta)
    @ 4-node quadrilateral element.

        4-------3
        |       |
        |  QX   |
        1-------2

    Parameters:
      xi, eta : natural coordinates (each in [-1, 1])

    Returns:
      N       : Array of shape functions [N1, N2, N3, N4]
      dN_dxi  : Array of derivatives with respect to xi
      dN_deta : Array of derivatives with respect to eta
    """
    N = np.array([
        0.25 * (1 - xi) * (1 - eta),  # N1 (bottom left)
        0.25 * (1 + xi) * (1 - eta),  # N2 (bottom right)
        0.25 * (1 + xi) * (1 + eta),  # N3 (top right)
        0.25 * (1 - xi) * (1 + eta)   # N4 (top left)
    ])
    dN_dxi = np.array([
        -0.25 * (1 - eta),
        0.25 * (1 - eta),
        0.25 * (1 + eta),
        -0.25 * (1 + eta)
    ])
    dN_deta = np.array([
        -0.25 * (1 - xi),
        -0.25 * (1 + xi),
        0.25 * (1 + xi),
        0.25 * (1 - xi)
    ])

    return N, dN_dxi, dN_deta

def mapping(xi, eta, nodes):
    """
```

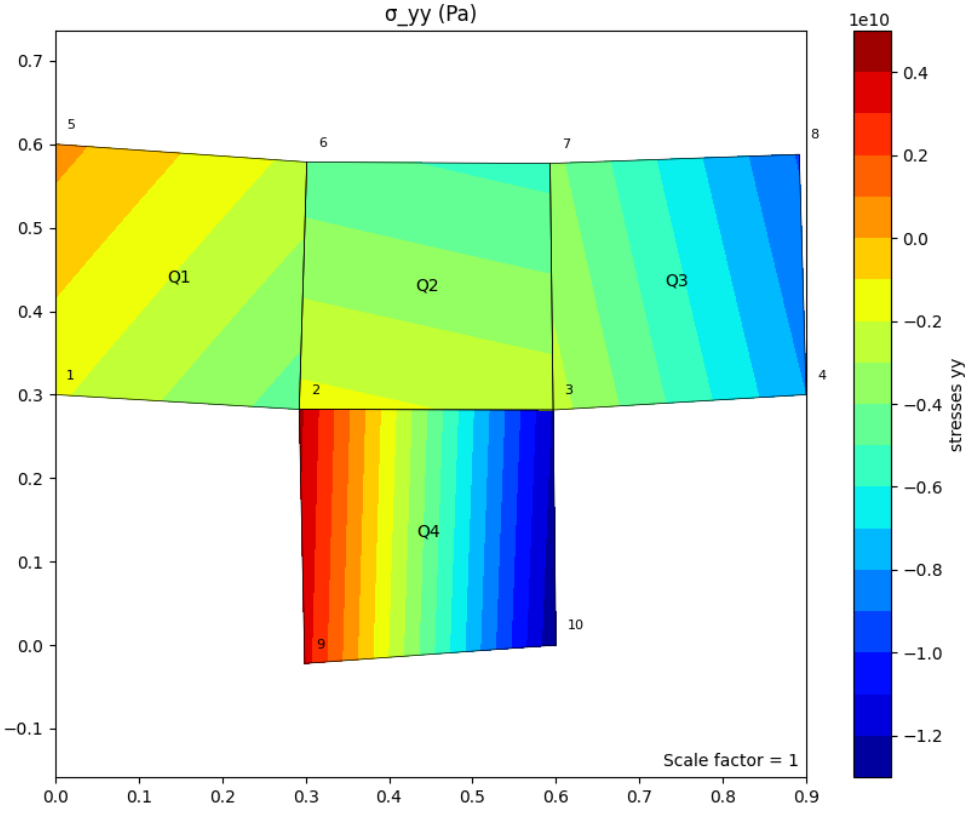
```
    Map from (xi, eta) to physical coord. (x, y).

    Parameters:
      xi, eta : Natural coordinates.
      nodes   : (4 x 2) array of nodal (x,y) coordinates for the element.

    Returns:
      x    : Physical coordinates (x, y) corresponding to (xi, eta)
      J    : 2x2 Jacobian matrix d(x,y)/d(xi,eta)
      detJ : Determinant of the Jacobian matrix.
    """
    #
    return x, J, detJ

# ------------------------------------------------------------------------------------------------
# 1.1 Element Stiffness Matrix Computation
# ------------------------------------------------------------------------------------------------

# Compute stiffness matrix using full integration ------------------------------------------------
def compute_quad_element_stiffness(E, nu, nodes, t=1, mode="PLANE_STRESS"):
    """
    Compute Q4 stiffness matrix (8 x 8) using 2x2 Gauss integration.

    Returns:
      Ke    : Element stiffness matrix (8 x 8)
      B_matrices : List of strain-displacement matrices (B) for each Gauss point
    """
    # Store B matrices for each Gauss point
    # --- Full Integration ---
    for i, gp in enumerate(gauss_points_full):
        # Compute global derivatives
        # Assemble strain-displacement matrix B (3 x 8)
        # Add contribution to the element stiffness matrix
        Ke += gp_weights_xi * gp_weights_eta * (B.T @ C @ B) * detJ * t
        # Store the B matrix for this Gauss point
    return Ke, B_matrices, C

# ------------------------------------------------------------------------------------------------
# 2. FORCE ASSEMBLY & B.C. APPLY
# ------------------------------------------------------------------------------------------------

def assemble_force_vector(total_dof, point_loads=None, distributed_loads=None,
nodes_global=None):
    """
    Assemble the global force vector.

    Parameters:
      total_dof         : Total degrees of freedom.
      point_loads       : List of tuples (node_index, dof, value)
                          dof = 0 (x-direction) or 1 (y-direction).
      distributed_loads : {'edge': (n1, n2), 'direction': 'y', 'value': load_value}
      nodes_global      : Global nodal coordinates.

    Returns:
      f : Global force vector.
    """
    # Apply point loads
    # Apply distributed loads (1D line loads on edges)
        # For each distributed load over an edge:
            # Compute length of edge
            # Use 2-point Gauss quadrature in 1D for line integration:
                # Linear shape functions along the edge:
                # Distribute load to the relevant DOFs (assume vertical load if direction=='y')
    return f

def apply_boundary_conditions(K, f, fixed_dofs):
    """
    Apply boundary conditions -> reduce global K and f.

    Parameters:
```

```
      K         : Global stiffness matrix.
      f         : Global force vector.
      fixed_dofs: Array or list of fixed degree-of-freedom.

    Returns:
      K_reduced, f_reduced, free_dofs : Reduced K, f, and free DOFs.
    """
    K_reduced = K[np.ix_(free_dofs, free_dofs)]
    f_reduced = f[free_dofs]
    return K_reduced, f_reduced, free_dofs

# ---------------------------------------------------------------------------------------------
# 3. ASSEMBLY
# ---------------------------------------------------------------------------------------------

def assemble_global_stiffness_matrix(E, nu, elements, nodes_global, t=1, mode="PLANE_STRESS",
integrationMode='Full'):
    """
    Assemble global stiffness matrix.

    Parameters:
      E           : Young's modulus (Pa)
      nu          : Poisson's ratio
      elements    : List of elements with node indices
      nodes_global: Global nodal coordinates
      t           : Thickness of the element (default=1 for unit thickness)
      mode        : default('PLANE_STRESS') / 'PLANE_STRAIN'
      integrationMode        : Mode of integration ('Full' / 'BBar' / 'SelReduced' /
'IncompatibleStrain')

    Returns:
      global_K    : Global stiffness matrix
    """
    # 2 DOF per node
        elif integrationMode == 'Full':
            Ke_local, _, _ = compute_quad_element_stiffness(E, nu, elem_nodes, t, mode)
        ###
        dof_indices = []
        for node in elem:
            dof_indices.extend([2 * node, 2 * node + 1])
        dof_indices = np.array(dof_indices)

        for i in range(len(dof_indices)):
            for j in range(len(dof_indices)):
                global_K[dof_indices[i], dof_indices[j]] += Ke_local[i, j]
    return global_K

# ---------------------------------------------------------------------------------------------
# 4. FEM SOLID MECHANIC SOLVER
# ---------------------------------------------------------------------------------------------

def fem_solver(E, nu, elements, nodes_global, t, point_loads, distributed_loads, fixed_dofs,
mode="PLANE_STRESS",
               integrationMode='Full'):
    """
    Finite Element Method Solver for 2D Plane Stress/Strain.

    Returns:
      results    : Dictionary of nodal displacements, element stresses, etc.
      K_reduced  : Reduced stiffness matrix (export for eigen.)
      free_dofs  : Free degrees of freedom (export for eigen.)
    """
    # --- Global Stiffness Matrix Assembly ---
    # --- Force Vector Assembly ---
    # --- Apply Boundary Conditions ---
    # --- Solve the System KU = f ---
    # --- Compute Stresses for Each Element ---
        # Elements displacements
        u_element = np.zeros(8)  # 4 nodes * 2 DOF per node
        for i, node in enumerate(element_nodes):
```

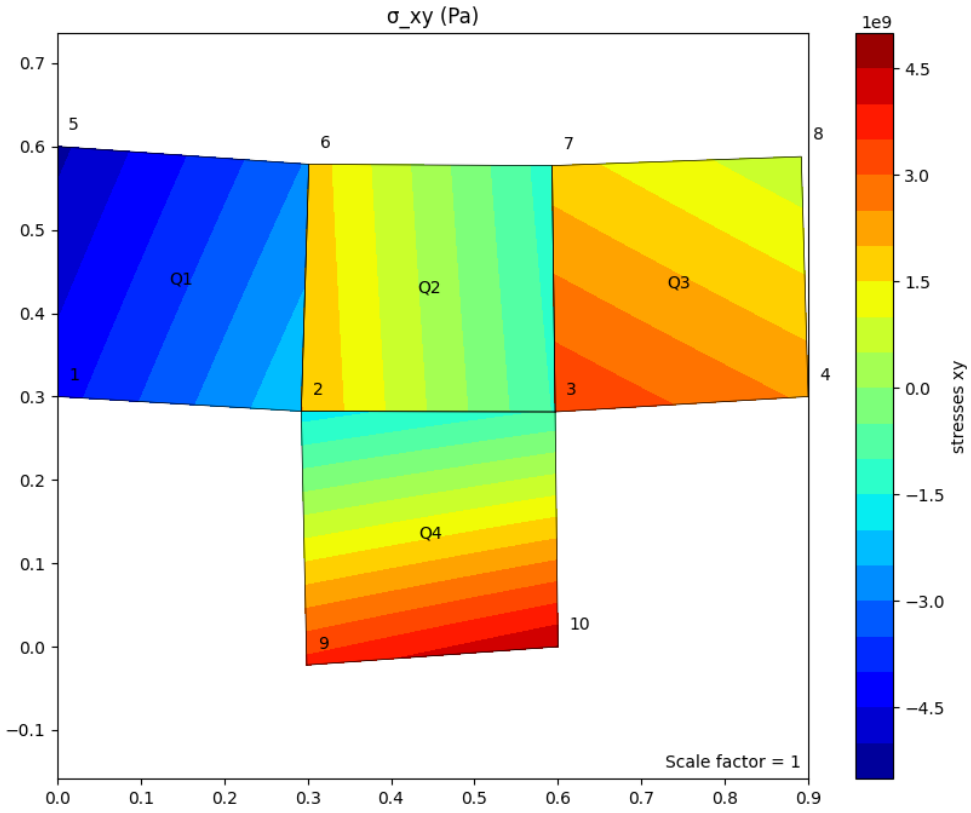
```
            # x-displacement
            # y-displacement
        # Get B and C
        # Stress at Gauss points
        # Store 1 element result
    # Store all elements results
    results = {
        "nodal_displacements": U,
        "element_results": element_results
    }
    return results, K_reduced, free_dofs

# ------------------------------------------------------------------------------------------------
# 6. POST-PROCESSING FUNCTIONS
# ------------------------------------------------------------------------------------------------

def extrapolate_gauss_to_nodes(elements: np.ndarray,
                               nodes_global: np.ndarray,
                               isAverageNodalValue: bool = True,
                               element_results: Dict = element_results) -> Tuple[Dict, Dict]:

    # Get shape functions and inverse matrix
    # Initialize data structures
    # Process each element
    for elem_idx, elem_data in element_results.items():
        # Extrapolate stresses and strains
        # Store extrapolated results
        element_extrapolate_results[elem_idx] = ElementNodalData(
                                                    element=elem_idx + 1,
                                                    nodes=elem_nodes,
                                                    displacements=elem_data['displacements'],
                                                    nodalStresses={
                                                        'xx': nodal_stresses[:, 0],
                                                        'yy': nodal_stresses[:, 1],
                                                        'xy': nodal_stresses[:, 2]
                                                    },
                                                    nodalStrains={
                                                        'xx': nodal_strains[:, 0],
                                                        'yy': nodal_strains[:, 1],
                                                        'xy': nodal_strains[:, 2]
                                                    }
                                                )
        # Accumulate for averaging
    # Average nodal
        for node in range(num_global_nodes):
            for comp in ['xx', 'yy', 'xy']:
                # Average stresses
                # Average strains
         for comp in ['xx', 'yy', 'xy']:
            nodal_average_data['stresses'][comp] = np.array(nodal_average_data['stresses'][comp])
            nodal_average_data['strains'][comp] = np.array(nodal_average_data['strains'][comp])
        # --- Assign averaged nodal values back to each element ---
        for elem_idx, elem_data in element_results.items():
            # For each node in the element, get the averaged value from nodal_data
            element_extrapolate_results[elem_idx] = ElementNodalData(
                element=elem_idx + 1,
                nodes=elem_nodes,
                displacements=elem_data['displacements'],
                nodalStresses=nodalStresses,
                nodalStrains=nodalStrains
            )
    return nodal_average_data, element_extrapolate
```



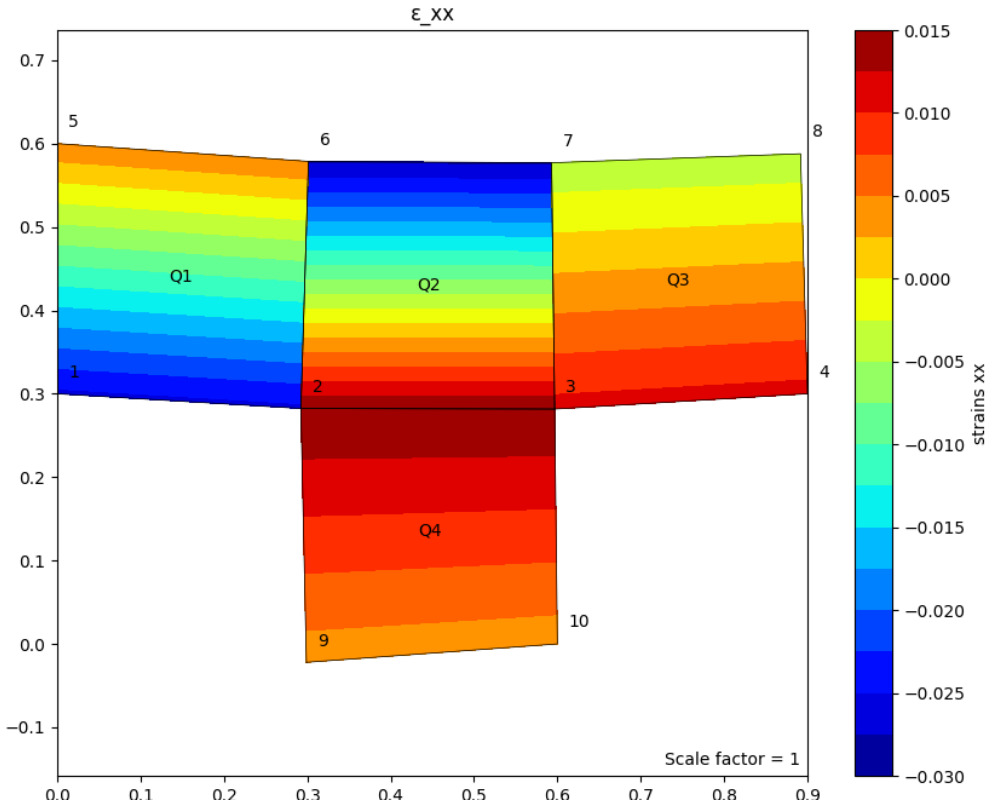
Hình 4-2 Ứng suất phương XX



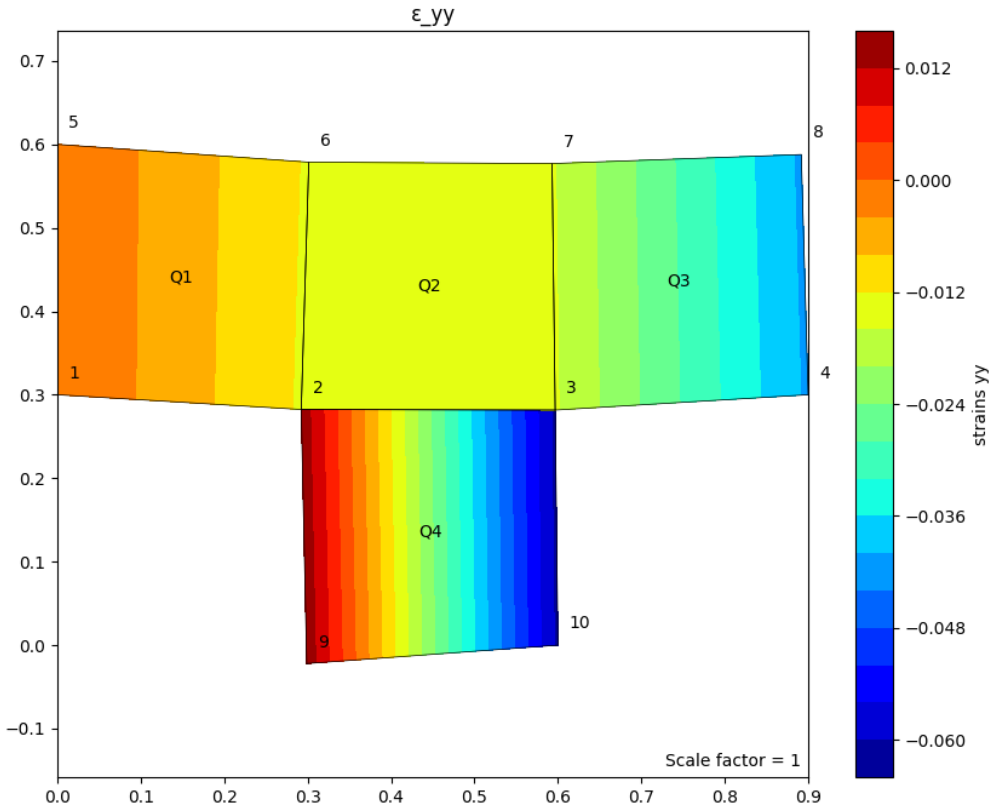
Hình 4-3 Ứng suất chương YY



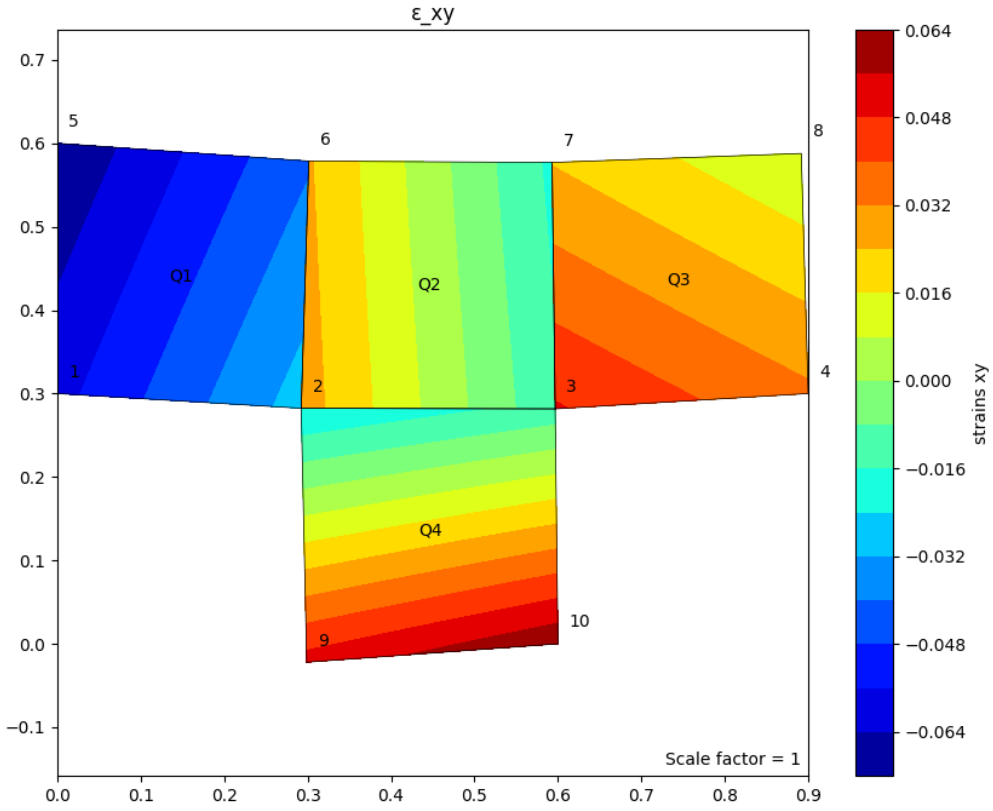
Hình 4-4 Ứng suất phương XY



Hình 4-5 Biến dạng phương XX



Hình 4-6 Biến dạng phương YY



Hình 4-7 Biến dạng phương XY

PHỤ LỤC A. PRG_FEM_Plane.py

```
"""
    By              :   Khanh Nguyen
    Email           :   nmkhanhfem@gmail.com
    Github          :   https://github.com/KowalskiPi
    Date            :   28/04/2025
    Description     :   2D Finite Element Method (FEM) Solver for Plane Stress/Strain Problems.
                        Modify for HCMUT Master's Course. + Eigenvalue Analysis.
"""

# ------------------------------------------------------------
# 0. LIBRARIES
# ------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.tri as tri
from matplotlib import patheffects as pe
from matplotlib.collections import PolyCollection
from scipy.linalg import eigh
from typing import Dict, List, Any
from typing import TypedDict, Tuple
from numpy.typing import NDArray


# ElementDataStorage
class ElementData(TypedDict):
    element: int
    nodes: NDArray[np.int_]
    displacements: np.ndarray
    intPts: int
    intePtsStresses: Dict[str, np.ndarray]  # {'xx': [], 'yy': [], 'xy': []}
    intePtsStrains: Dict[str, np.ndarray]   # {'xx': [], 'yy': [], 'xy': []}

class ElementNodalData(TypedDict):
    element: int
    nodes: NDArray[np.int_]
    displacements: np.ndarray
    nodalStresses: Dict[str, np.ndarray]  # {'xx': [], 'yy': [], 'xy': []}
    nodalStrains: Dict[str, np.ndarray]   # {'xx': [], 'yy': [], 'xy': []}

# Global storage
element_results: Dict[int, ElementData] = {}
element_extrapolate_results: Dict[int, ElementNodalData] = {}

# ------------------------------------------------------------------------------------------------------------
# 1. HELPER FUNCTIONS
# ------------------------------------------------------------------------------------------------------------

def C_matrix(E, nu, mode="PLANE_STRESS"):
    """
    Constitutive matrix

    Parameters:
      E    : Young's modulus
      nu   : Poisson's ratio
      mode : "PLANE_STRESS" or "PLANE_STRAIN"

    Returns:
      C     : Full constitutive matrix
      C_vol : Volumetric part of the constitutive matrix
    """
    # Bulk modulus
```

```
    # K = E / (3 * (1 - 2 * nu))
    # Shear modulus
    # G = E / (2 * (1 + nu))

    if mode == "PLANE_STRESS":
        # Full constitutive matrix
        C = (E / (1 - nu**2)) * np.array([
                                            [1, nu, 0],
                                            [nu, 1, 0],
                                            [0, 0, (1 - nu) / 2]
                                        ])

        # Volumetric part
        C_vol = (E / (1 - nu**2)) * np.array([
                                                [1 + nu, 0, 0],
                                                [0, 1 + nu, 0],
                                                [0, 0, 0]
                                            ])

    elif mode == "PLANE_STRAIN":
        # Full constitutive matrix
        C = E / ((1 + nu) * (1 - 2 * nu)) * np.array([
                                                        [1 - nu, nu, 0],
                                                        [nu, 1 - nu, 0],
                                                        [0, 0, (1 - 2 * nu) / 2]
                                                    ])

        # Volumetric part
        C_vol = E / ((1 + nu) * (1 - 2 * nu)) * np.array([
                                                            [1, 0, 0],
                                                            [0, 1, 0],
                                                            [0, 0, 0]
                                                        ])

    else:
        print("Unsupported mode. Use 'PLANE_STRESS' or 'PLANE_STRAIN'.")
        return None, None

    return C, C_vol

def integrationPoints(elementType = 'Q4', type = 'FULL'):
    """
    Parameters:
      elementType : Type of element (e.g., 'Q4' for quadrilateral)
      type        : Integration type ('FULL' or 'REDUCED')

    Returns:
        gauss_points : Integration points
        gauss_weights: Weights for the integration points
        """
    nIntegrationPoints, gaussPoints, gaussWeights = 0, None, None

    if type == 'FULL':
        if elementType == 'Q4':
            nIntegrationPoints = 4
            gaussPoints = np.array([
                                    [-1/np.sqrt(3), -1/np.sqrt(3)],
                                    [ 1/np.sqrt(3), -1/np.sqrt(3)],
                                    [ 1/np.sqrt(3),  1/np.sqrt(3)],
                                    [-1/np.sqrt(3),  1/np.sqrt(3)]
                                   ])
            gaussWeights = np.array([1.0, 1.0])
    elif type == 'REDUCED':
        if elementType == 'Q4':
            nIntegrationPoints = 1
            gaussPoints = np.array([[0, 0]])
            gaussWeights = np.array([4.0])

    return nIntegrationPoints, gaussPoints, gaussWeights
```

```
def shape_functions_Q4(xi, eta):
    """
    Bilinear shape functions and derivatives in (xi, eta)
    @ 4-node quadrilateral element.

        4-------3
        |       |
        |  QX   |
        1-------2

    Parameters:
      xi, eta : natural coordinates (each in [-1, 1])

    Returns:
      N       : Array of shape functions [N1, N2, N3, N4]
      dN_dxi  : Array of derivatives with respect to xi
      dN_deta : Array of derivatives with respect to eta
    """
    N = np.array([
        0.25 * (1 - xi) * (1 - eta),  # N1 (bottom left)
        0.25 * (1 + xi) * (1 - eta),  # N2 (bottom right)
        0.25 * (1 + xi) * (1 + eta),  # N3 (top right)
        0.25 * (1 - xi) * (1 + eta)   # N4 (top left)
    ])
    dN_dxi = np.array([
        -0.25 * (1 - eta),
        0.25 * (1 - eta),
        0.25 * (1 + eta),
        -0.25 * (1 + eta)
    ])
    dN_deta = np.array([
        -0.25 * (1 - xi),
        -0.25 * (1 + xi),
        0.25 * (1 + xi),
        0.25 * (1 - xi)
    ])

    return N, dN_dxi, dN_deta

def mapping(xi, eta, nodes):
    """
    Map from (xi, eta) to physical coord. (x, y).

    Parameters:
      xi, eta : Natural coordinates.
      nodes   : (4 x 2) array of nodal (x,y) coordinates for the element.

    Returns:
      x    : Physical coordinates (x, y) corresponding to (xi, eta)
      J    : 2x2 Jacobian matrix d(x,y)/d(xi,eta)
      detJ : Determinant of the Jacobian matrix.
    """
    N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
    x = np.dot(N, nodes)
    dx_dxi  = np.dot(dN_dxi, nodes)
    dx_deta = np.dot(dN_deta, nodes)
    J = np.array([dx_dxi, dx_deta]).T
    detJ = np.linalg.det(J)
    return x, J, detJ

def compute_area_Q4(nodes, gauss_points=[-1/np.sqrt(3), 1/np.sqrt(3)]):
    """
    Compute the area of Q4 using 2x2 Gauss integration.

    Parameters:
      nodes : (4 x 2) (physical space)
      gauss_points : Gauss points (default: [-1/sqrt(3), 1/sqrt(3)])

    Returns:
      area  : Element area.
```

```
    """
    area = 0.0
    for xi in gauss_points:
        for eta in gauss_points:
            _, _, detJ = mapping(xi, eta, nodes)
            area += abs(detJ)
    return area

# ------------------------------------------------------------------------------------------------
-------------------------
# 1.1 Element Stiffness Matrix Computation
# ------------------------------------------------------------------------------------------------
-------------------------

# Compute stiffness matrix using full integration ----------------------------------------------
--------------
def compute_quad_element_stiffness(E, nu, nodes, t=1, mode="PLANE_STRESS"):
    """
    Compute Q4 stiffness matrix (8 x 8) using 2x2 Gauss integration.

    Returns:
      Ke    : Element stiffness matrix (8 x 8)
      B_matrices : List of strain-displacement matrices (B) for each Gauss point
    """
    print("\nProcessing: Element stiffness matrix, full integration.......")

    nFullIntegrationPoints, gauss_points_full, gauss_weights_full = integrationPoints('Q4',
'FULL')
    if gauss_points_full is None or gauss_weights_full is None:
        print("Gauss points = None")
        return None, None, None
    if mode == "PLANE_STRAIN":
        t = 1  # Force thickness to 1 for plane strain

    C, _ = C_matrix(E, nu, mode)
    if C is None:
        print("C matrix = None")
        return None, None, None

    Ke = np.zeros((8, 8))
    B_matrices = []  # Store B matrices for each Gauss point

    # --- Full Integration ---
    print("\nFull Integration Points (2x2 Quadrature):")

    gp_weights_xi = gauss_weights_full[0]
    gp_weights_eta = gauss_weights_full[1]
    for i, gp in enumerate(gauss_points_full):
        xi, eta = gp
        print(f"GP{i}: (ξ={xi:.4f}, η={eta:.4f}), weight=({gp_weights_xi:.4f},
{gp_weights_eta:.4f})")

        N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
        _, J, detJ = mapping(xi, eta, nodes)
        if detJ <= 0:
            print("Jacobian determinant is non-positive. Check node ordering!")

        invJ = np.linalg.inv(J)

        # Compute global derivatives
        dN_dx = invJ[0, 0] * dN_dxi + invJ[0, 1] * dN_deta
        dN_dy = invJ[1, 0] * dN_dxi + invJ[1, 1] * dN_deta

        # Assemble strain-displacement matrix B (3 x 8)
        B = np.zeros((3, 8))
        for k in range(4):
            B[0, 2 * k] = dN_dx[k]
            B[1, 2 * k + 1] = dN_dy[k]
            B[2, 2 * k] = dN_dy[k]
            B[2, 2 * k + 1] = dN_dx[k]
```

```
        # Add contribution to the element stiffness matrix
        Ke += gp_weights_xi * gp_weights_eta * (B.T @ C @ B) * detJ * t

        # Store the B matrix for this Gauss point
        B_matrices.append(B)

    return Ke, B_matrices, C

# Compute stiffness matrix using B-Bar method ------------------------------------------------------
--------------
def compute_quad_element_stiffness_bbar(E, nu, nodes, t=1, mode="PLANE_STRESS"):
    """
    Compute Q4 stiffness matrix (8 x 8) using 2x2 Gauss integration with B-Bar method.

    Parameters:
      E     : Young's modulus
      nu    : Poisson's ratio
      nodes: (4 x 2) array of nodal coordinates
      t     : Thickness of the element
      mode : "PLANE_STRESS" or "PLANE_STRAIN"

    Returns:
      Ke     : Element stiffness matrix (8 x 8)
      B_matrices : List of strain-displacement matrices (B) for each Gauss point
    """
    print("\nProcessing: Element stiffness matrix, BBar.......")

    # Full integration points (2x2 Gauss quadrature)
    nFullIntegrationPoints, gauss_points_full, gauss_weights_full = integrationPoints('Q4',
'FULL')
    if gauss_points_full is None or gauss_weights_full is None:
        print("Gauss points = None")
        return None, None, None

    if mode == "PLANE_STRAIN":
        t = 1  # Force thickness to 1 for plane strain

    C, _ = C_matrix(E, nu, mode)
    if C is None:
        print("C matrix = None")
        return None, None, None

    Ke = np.zeros((8, 8))
    B_matrices = []  # Store B matrices for each Gauss point

    # Volumetric strain-displacement matrix normalization
    BvolNorm = np.zeros((4, 2))  # 4 nodes, 2 coord (x, y)
    element_volume = 0.0

    # --- First loop: Bvol and Ve ---
    print("\nNormalizing volumetric B_vol matrix......")
    gp_weights_xi = gauss_weights_full[0]
    gp_weights_eta = gauss_weights_full[1]
    for i, gp in enumerate(gauss_points_full):
        xi, eta = gp
        print(f"GP{i}: (ξ={xi:.4f}, η={eta:.4f}), weight=({gp_weights_xi:.4f},
{gp_weights_eta:.4f})")

        N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
        _, J, detJ = mapping(xi, eta, nodes)
        if detJ <= 0:
            print("Jacobian determinant is non-positive. Check node ordering!")

        invJ = np.linalg.inv(J)

        # Compute global derivatives
        dN_dx = invJ[0, 0] * dN_dxi + invJ[0, 1] * dN_deta
        dN_dy = invJ[1, 0] * dN_dxi + invJ[1, 1] * dN_deta
```

```
        # Accumulate volumetric strain-displacement matrix
        for k in range(4):  # Loop over nodes
            BvolNorm[k, 0] += dN_dx[k] * gp_weights_xi * gp_weights_eta * detJ
            BvolNorm[k, 1] += dN_dy[k] * gp_weights_xi * gp_weights_eta * detJ

        # Accumulate element volume
        element_volume += gp_weights_xi * gp_weights_eta * detJ

    # Normalize Bvol by Ve
    BvolNorm = (1 / 2) * BvolNorm / element_volume

    # Second loop: Compute stiffness matrix with B-Bar correction
    print("\nCorrection of B matrix......")
    gp_weights_xi = gauss_weights_full[0]
    gp_weights_eta = gauss_weights_full[1]
    for i, gp in enumerate(gauss_points_full):
        xi, eta = gp
        print(f"GP{i}: (ξ={xi:.4f}, η={eta:.4f}), weight=({gp_weights_xi:.4f},
{gp_weights_eta:.4f})")

        N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
        _, J, detJ = mapping(xi, eta, nodes)
        if detJ <= 0:
            print("Jacobian determinant is non-positive. Check node ordering!")

        invJ = np.linalg.inv(J)

        # Compute global derivatives
        dN_dx = invJ[0, 0] * dN_dxi + invJ[0, 1] * dN_deta
        dN_dy = invJ[1, 0] * dN_dxi + invJ[1, 1] * dN_deta

        # Assemble eps-displacement matrix B (3 x 8)
        B = np.zeros((3, 8))
        for k in range(4):
            B[0, 2 * k] = dN_dx[k]
            B[1, 2 * k + 1] = dN_dy[k]
            B[2, 2 * k] = dN_dy[k]
            B[2, 2 * k + 1] = dN_dx[k]

        # Correct B using B-Bar method
        for k in range(4):
            B[0, 2 * k] += (- dN_dx[k]/2 + BvolNorm[k, 0])
            B[1, 2 * k] += (- dN_dx[k]/2 + BvolNorm[k, 0])
            B[0, 2 * k + 1] += (- dN_dy[k]/2 + BvolNorm[k, 1])
            B[1, 2 * k + 1] += (- dN_dy[k]/2 + BvolNorm[k, 1])
        # Add contribution to the element stiffness matrix
        Ke += gp_weights_xi * gp_weights_eta * (B.T @ C @ B) * detJ * t

        # Store the B matrix for this Gauss point
        B_matrices.append(B)

    return Ke, B_matrices, C

# Compute stiffness matrix using selective reduced integration method ----------------------------
--------------
def compute_quad_element_stiffness_selective_reduced(E, nu, nodes, t=1, mode="PLANE_STRESS"):
    """
    Compute Q4 stiffness matrix (8 x 8) using selective reduced integration.

    Parameters:
      E    : Young's modulus
      nu   : Poisson's ratio
      nodes: (4 x 2) array of nodal coordinates
      t    : Thickness of the element
      mode : "PLANE_STRESS" or "PLANE_STRAIN"

    Returns:
      Ke    : Element stiffness matrix (8 x 8)
      B_matrices : List of strain-displacement matrices (B) for each Gauss point
    """
```

```
    print("\nProcessing: Element stiffness matrix, selective reduced integration......")

    # Full integration points (2x2 Gauss quadrature)
    nFullIntegrationPoints, gauss_points_full, gauss_weights_full = integrationPoints('Q4',
'FULL')
    # Reduced integration points (1x1 Gauss quadrature)
    nReducedIntegrationPoints, gauss_points_reduced, gauss_weights_reduced =
integrationPoints('Q4', 'REDUCED')
    if gauss_points_full is None or gauss_weights_full is None \
        or gauss_points_reduced is None or gauss_weights_reduced is None:
        print("Gauss points = None")
        return None, None, None

    if mode == "PLANE_STRAIN":
        t = 1  # Force thickness to 1 for plane strain

    C, C_vol = C_matrix(E, nu, mode)
    if C is None or C_vol is None:
        print("C matrices = None")
        return None, None, None

    Ke = np.zeros((8, 8))
    B_matrices = []

    # --- Full Integration: Deviatoric Part ---
    print("\nDeviatoric: Full Integration Points (2x2 Quadrature):")
    gp_weights_xi = gauss_weights_full[0]
    gp_weights_eta = gauss_weights_full[1]
    for i, gp in enumerate(gauss_points_full):
        xi, eta = gp
        print(f"GP{i}: (ξ={xi:.4f}, η={eta:.4f}), weight=({gp_weights_xi:.4f},
{gp_weights_eta:.4f})")

        N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
        _, J, detJ = mapping(xi, eta, nodes)
        if detJ <= 0:
            print("Jacobian determinant is non-positive. Check node ordering!")

        invJ = np.linalg.inv(J)

        # Compute global derivatives
        dN_dx = invJ[0, 0] * dN_dxi + invJ[0, 1] * dN_deta
        dN_dy = invJ[1, 0] * dN_dxi + invJ[1, 1] * dN_deta

        # Assemble strain-displacement matrix B (3 x 8)
        B = np.zeros((3, 8))
        for k in range(4):
            B[0, 2 * k] = dN_dx[k]
            B[1, 2 * k + 1] = dN_dy[k]
            B[2, 2 * k] = dN_dy[k]
            B[2, 2 * k + 1] = dN_dx[k]

        # Remove volumetric part from stiffness matrix
        Ke += gp_weights_xi * gp_weights_eta * (B.T @ C @ B) * detJ * t
        Ke -= (1/2) * gp_weights_xi * gp_weights_eta * (B.T @ C_vol @ B) * detJ * t

        # Store the B matrix for this Gauss point
        B_matrices.append(B)

    # --- Reduced Integration: Recover volumetric ---
    print("\nRecover volumetric: Reduced Integration Points (1x1 Quadrature):")
    for i, gp in enumerate(gauss_points_reduced):
        xi, eta = gp
        print(f"GP{i}: (ξ={xi:.4f}, η={eta:.4f}), weight={gauss_weights_reduced[0]:.4f}")

        N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
        _, J, detJ = mapping(xi, eta, nodes)
        if detJ <= 0:
            print("Jacobian determinant is non-positive. Check node ordering!")
```

```
        invJ = np.linalg.inv(J)

        # Global derivatives
        dN_dx = invJ[0, 0] * dN_dxi + invJ[0, 1] * dN_deta
        dN_dy = invJ[1, 0] * dN_dxi + invJ[1, 1] * dN_deta

        # Assemble strain-displacement matrix B (3 x 8)
        B = np.zeros((3, 8))
        for k in range(4):
            B[0, 2 * k] = dN_dx[k]
            B[1, 2 * k + 1] = dN_dy[k]
            B[2, 2 * k] = dN_dy[k]
            B[2, 2 * k + 1] = dN_dx[k]

        # Add back the volumetric part via reduced inter.
        Ke += (1/2) * gauss_weights_reduced[0] * (B.T @ C_vol @ B) * detJ * t
    # print("Type of Ke:", type(Ke))
    return Ke, B_matrices, C

# Compute stiffness matrix using incompatible simplified strain ---------------------------------
--------------
def compute_quad_element_stiffness_incompatible(E, nu, nodes, t=1, mode="PLANE_STRESS"):
    """
    Compute Q4 stiffness matrix (8 x 8) using incompatible modes and static condensation.

    Parameters:
      E     : Young's modulus
      nu    : Poisson's ratio
      nodes : (4 x 2) array of nodal coordinates
      t     : Thickness of the element
      mode  : "PLANE_STRESS" or "PLANE_STRAIN"

    Returns:
      Ke_condensed : Condensed element K (8 x 8)
      B_matrices   : List of standard B matrices
    """
    print("\nProcessing: Element stiffness matrix, incompatible mode.......")

    # Full integration points (2x2 Gauss quadrature)
    nFullIntegrationPoints, gauss_points_full, gauss_weights_full = integrationPoints('Q4',
'FULL')
    if gauss_points_full is None or gauss_weights_full is None:
            print("Gauss points = None")
            return None, None, None

    if mode == "PLANE_STRAIN":
        t = 1  # Force thickness to 1 for plane strain

    C, C_vol = C_matrix(E, nu, mode)
    if C is None or C_vol is None:
        print("C matrices = None")
        return None, None, None

    B_matrices = []
    B_std_list = []
    B_inc_list = []

    # Compute Jacobian at centroid (xi=0, eta=0)
    xi_center, eta_center = 0.0, 0.0
    _, J0, detJ0 = mapping(xi_center, eta_center, nodes)
    invJ0 = np.linalg.inv(J0)

    # Initialize
    Kuu = np.zeros((8, 8))
    Kua = np.zeros((8, 4))
    Kau = np.zeros((4, 8))
    Kaa = np.zeros((4, 4))

    # Integration loop
    gp_weights_xi = gauss_weights_full[0]
```

```
    gp_weights_eta = gauss_weights_full[1]
    for i, gp in enumerate(gauss_points_full):
        xi, eta = gp
        print(f"GP{i}: (ξ={xi:.4f}, η={eta:.4f}), weight=({gp_weights_xi:.4f},
{gp_weights_eta:.4f})")

        N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
        _, J, detJ = mapping(xi, eta, nodes)
        if detJ <= 0:
            print("Jacobian determinant is non-positive. Check node ordering!")

        invJ = np.linalg.inv(J)

        weight = gp_weights_xi * gp_weights_eta

        # Standard B matrix (Voigt)
        N, dN_dxi, dN_deta = shape_functions_Q4(xi, eta)
        dN_dx = invJ[0, 0] * dN_dxi + invJ[0, 1] * dN_deta
        dN_dy = invJ[1, 0] * dN_dxi + invJ[1, 1] * dN_deta

        B_std = np.zeros((3, 8))
        for a in range(4):
            B_std[0, 2*a] = dN_dx[a]     # ε_xx
            B_std[1, 2*a + 1] = dN_dy[a] # ε_yy
            B_std[2, 2*a] = dN_dy[a]     # ε_xy
            B_std[2, 2*a + 1] = dN_dx[a]

        # --- Incompatible Modes --------------
        scaling = (detJ0 / detJ)
        xi_scaled = xi * scaling
        eta_scaled = eta * scaling

        # Derivatives of incompatible modes (alpha1: xi-term, alpha2: eta-term)
        dalpha1_dx = invJ0[0, 0] * xi_scaled
        dalpha1_dy = invJ0[1, 0] * xi_scaled
        dalpha2_dx = invJ0[0, 1] * eta_scaled
        dalpha2_dy = invJ0[1, 1] * eta_scaled

        # B_alpha matrix (3x4)
        B_alpha = np.zeros((3, 4))
        # Mode 1 (xi-term)
        B_alpha[0, 0] = dalpha1_dx  # ε_xx
        B_alpha[1, 0] = 0.0
        B_alpha[2, 0] = dalpha1_dy  # ε_xy
        # Mode 2 (eta-term)
        B_alpha[0, 1] = dalpha2_dx
        B_alpha[1, 1] = 0.0
        B_alpha[2, 1] = dalpha2_dy
        # Modes 3 & 4
        B_alpha[0, 2] = 0.0
        B_alpha[1, 2] = dalpha1_dy  # ε_yy
        B_alpha[2, 2] = dalpha1_dx
        B_alpha[0, 3] = 0.0
        B_alpha[1, 3] = dalpha2_dy
        B_alpha[2, 3] = dalpha2_dx

        # --- Assemble Sub-Matrices ---
        Kuu += (B_std.T @ C @ B_std) * detJ * t * weight
        Kua += (B_std.T @ C @ B_alpha) * detJ * t * weight
        Kau += (B_alpha.T @ C @ B_std) * detJ * t * weight
        Kaa += (B_alpha.T @ C @ B_alpha) * detJ * t * weight

        # B_matrices.append(B_std)
        B_std_list.append(B_std)
        B_inc_list.append(B_alpha)

    # Static condensation
    Kaa += 1e-10 * np.eye(4)  # Stabilize
    Kaa_inv = np.linalg.inv(Kaa)
    Ke_condensed = Kuu - Kua @ Kaa_inv @ Kau
```

```
    B_eff_list = []
    for B_std, B_alpha in zip(B_std_list, B_inc_list):
        B_eff = B_std - B_alpha @ Kaa_inv @ Kau
        B_eff_list.append(B_eff)

    B_matrices = B_eff_list

    return Ke_condensed, B_matrices, C

# ------------------------------------------------------------------------------------------------
------------------------
# 2. FORCE ASSEMBLY & B.C. APPLY
# ------------------------------------------------------------------------------------------------
------------------------

def assemble_force_vector(total_dof, point_loads=None, distributed_loads=None,
nodes_global=None):
    """
    Assemble the global force vector.

    Parameters:
      total_dof         : Total degrees of freedom.
      point_loads       : List of tuples (node_index, dof, value)
                          dof = 0 (x-direction) or 1 (y-direction).
      distributed_loads : {'edge': (n1, n2), 'direction': 'y', 'value': load_value}
      nodes_global      : Global nodal coordinates.

    Returns:
      f : Global force vector.
    """
    f = np.zeros(total_dof)

    # Apply point loads
    if point_loads is not None:
        for load in point_loads:
            node, dof, value = load  # node index (0-indexed), dof: 0 => x, 1 => y.
            f[2*node + dof] += value

    # Apply distributed loads (1D line loads on edges)
    if distributed_loads is not None:
        if nodes_global is None:
            print("Global nodal coordinates must be provided for distributed loads.")
            return
        # For each distributed load over an edge:
        for dload in distributed_loads:
            n1, n2 = dload['edge']
            direction = dload['direction']
            load_value = dload['value']
            # Compute length of edge
            x1, y1 = nodes_global[n1]
            x2, y2 = nodes_global[n2]
            edge_length = np.sqrt((x2-x1)**2 + (y2-y1)**2)
            # Use 2-point Gauss quadrature in 1D for line integration:
            gp_1D = np.array([-1/np.sqrt(3), 1/np.sqrt(3)])
            gp_weights = np.array([1.0, 1.0])
            for gp, w in zip(gp_1D, gp_weights):
                # Linear shape functions along the edge:
                N_edge = np.array([0.5*(1-gp), 0.5*(1+gp)])
                f_edge = N_edge * load_value * (edge_length / 2) * w
                # Distribute load to the relevant DOFs (assume vertical load if direction=='y')
                if direction == 1:
                    f[2*n1+1] += f_edge[0]
                    f[2*n2+1] += f_edge[1]
                else:
                    f[2*n1] += f_edge[0]
                    f[2*n2] += f_edge[1]

    return f
```

```
def apply_boundary_conditions(K, f, fixed_dofs):
    """
    Apply boundary conditions -> reduce global K and f.

    Parameters:
      K         : Global stiffness matrix.
      f         : Global force vector.
      fixed_dofs: Array or list of fixed degree-of-freedom.

    Returns:
      K_reduced, f_reduced, free_dofs : Reduced K, f, and free DOFs.
    """
    total_dof = K.shape[0]
    all_dofs = np.arange(total_dof)
    free_dofs = np.setdiff1d(all_dofs, fixed_dofs)
    # print("Type of free_dofs:", type(free_dofs))
    # print("free_dofs dtype:", free_dofs.dtype)

    if f is None:
        f = np.zeros(total_dof)

    # Ensure no overlap
    assert len(np.intersect1d(fixed_dofs, free_dofs)) == 0, "Fixed DOFs overlap with Free DOFs!"
    assert np.max(free_dofs) < total_dof, "Free DOFs contain out-of-bounds indices!"

    # print("Type of K:", type(K))
    # print("Shape of K:", K.shape)
    K_reduced = K[np.ix_(free_dofs, free_dofs)]
    f_reduced = f[free_dofs]
    # print("Type of K_reduced:", type(K_reduced))
    # print("Shape of K_reduced:", K_reduced.shape)
    return K_reduced, f_reduced, free_dofs

# ------------------------------------------------------------------------------------------------
------------------------
# 3. ASSEMBLY
# ------------------------------------------------------------------------------------------------
------------------------

def assemble_global_stiffness_matrix(E, nu, elements, nodes_global, t=1, mode="PLANE_STRESS",
integrationMode='Full'):
    """
    Assemble global stiffness matrix.

    Parameters:
      E           : Young's modulus (Pa)
      nu          : Poisson's ratio
      elements    : List of elements with node indices
      nodes_global: Global nodal coordinates
      t           : Thickness of the element (default=1 for unit thickness)
      mode        : default('PLANE_STRESS') / 'PLANE_STRAIN'
      integrationMode        : Mode of integration ('Full' / 'BBar' / 'SelReduced' /
'IncompatibleStrain')

    Returns:
      global_K    : Global stiffness matrix
    """
    total_dof = nodes_global.shape[0] * 2  # 2 DOF per node
    global_K = np.zeros((total_dof, total_dof))
    Ke_local = None

    for e_idx, elem in enumerate(elements):
        elem_nodes = nodes_global[elem, :]

        # Use B-Bar
        if integrationMode == 'BBar':
            Ke_local, _, _ = compute_quad_element_stiffness_bbar(E, nu, elem_nodes, t, mode)
        elif integrationMode == 'Full':
            Ke_local, _, _ = compute_quad_element_stiffness(E, nu, elem_nodes, t, mode)
        elif integrationMode == 'SelReduced':
```

```
            Ke_local, _, _ = compute_quad_element_stiffness_selective_reduced(E, nu, elem_nodes,
t, mode)
        elif integrationMode == 'IncompatibleStrain':
            Ke_local, _, _ = compute_quad_element_stiffness_incompatible(E, nu, elem_nodes, t,
mode)
        else:
            print("Unsupported integration mode %s" % integrationMode)

        if Ke_local is None:
            print("Element stiffness matrix = None. Check element %d." % (e_idx + 1))
            return None
        print(f"\nElement {e_idx + 1}:\n", Ke_local)
        ###

        dof_indices = []
        for node in elem:
            dof_indices.extend([2 * node, 2 * node + 1])
        dof_indices = np.array(dof_indices)

        for i in range(len(dof_indices)):
            for j in range(len(dof_indices)):
                global_K[dof_indices[i], dof_indices[j]] += Ke_local[i, j]

    # print("Type of global_K:", type(global_K))
    # print("Shape of global_K:", global_K.shape)
    return global_K

# ------------------------------------------------------------------------------------------------
-------------------------
# 4. FEM SOLID MECHANIC SOLVER
# ------------------------------------------------------------------------------------------------
-------------------------

def fem_solver(E, nu, elements, nodes_global, t, point_loads, distributed_loads, fixed_dofs,
mode="PLANE_STRESS",
               integrationMode='Full'):
    """
    Finite Element Method Solver for 2D Plane Stress/Strain.

    Returns:
      results    : Dictionary of nodal displacements, element stresses, etc.
      K_reduced  : Reduced stiffness matrix (export for eigen.)
      free_dofs  : Free degrees of freedom (export for eigen.)
    """
    # --- Global Stiffness Matrix Assembly ---
    global_K = assemble_global_stiffness_matrix(E, nu, elements, nodes_global, t, mode,
integrationMode)
    print("\nGlobal Stiffness Matrix (K):\n", global_K)

    # --- Force Vector Assembly ---
    total_dof = nodes_global.shape[0] * 2
    f = assemble_force_vector(total_dof, point_loads=point_loads,
distributed_loads=distributed_loads,
                              nodes_global=nodes_global)

    # --- Apply Boundary Conditions ---
    K_reduced, f_reduced, free_dofs = apply_boundary_conditions(global_K, f, fixed_dofs)
    print("\nReduced Global Stiffness Matrix (K):\n", K_reduced)
    print("\nReduced Force Vector (f):\n", f_reduced)

    # --- Solve the System KU = f ---
    U_reduced = np.linalg.solve(K_reduced, f_reduced)
    U = np.zeros(total_dof)
    U[free_dofs] = U_reduced

    # --- Compute Stresses for Each Element ---
    for elem_idx, element_nodes in enumerate(elements):
        element_coords = nodes_global[element_nodes]  # Coordinates of element nodes

        # Elements displacements
```

```
        u_element = np.zeros(8)  # 4 nodes * 2 DOF per node
        for i, node in enumerate(element_nodes):
            u_element[2 * i] = U[2 * node]       # x-displacement
            u_element[2 * i + 1] = U[2 * node + 1]  # y-displacement

        # Get B and C
        B_matrices = None
        C = None
        if integrationMode == 'BBar':
            _, B_matrices, C = compute_quad_element_stiffness_bbar(E, nu, element_coords, t,
mode)
        elif integrationMode == 'Full':
            _, B_matrices, C = compute_quad_element_stiffness(E, nu, element_coords, t, mode)
        elif integrationMode == 'SelReduced':
            _, B_matrices, C = compute_quad_element_stiffness_selective_reduced(E, nu,
element_coords, t, mode)
        elif integrationMode == 'IncompatibleStrain':
            _, B_matrices, C = compute_quad_element_stiffness_incompatible(E, nu, element_coords,
t, mode)
        else:
            print("Unsupported integration mode %s" % integrationMode)

        # Stress at Gauss points
        stresses = []
        strains = []
        if B_matrices is not None and C is not None:
            for B in B_matrices:
                strain = B @ u_element
                stress = C @ strain
                stresses.append(stress)
                strains.append(strain)
        else:
            print(f"\nElement {elem_idx+1}: B_matrices or C = None, stress/strain computation
error.")
            stresses = np.zeros((4, 3))
            strains = np.zeros((4, 3))

        stresses = np.array(stresses)
        strains = np.array(strains)

        # Store 1 element result
        element_results[elem_idx] = {
                                        'element': elem_idx + 1,
                                        'intPts': 4,
                                        'nodes': element_nodes,
                                        'displacements': u_element,
                                        # 'intePtsStresses': stresses,
                                        # 'intePtsStrains': strains
                                        'intePtsStresses': {'xx':stresses[:, 0], 'yy':stresses[:,
1], 'xy':stresses[:, 2]},
                                        'intePtsStrains': {'xx':strains[:, 0], 'yy':strains[:,
1], 'xy':strains[:, 2]}
                                    }

    # Store all elements results
    results = {
        "nodal_displacements": U,
        "element_results": element_results
    }

    return results, K_reduced, free_dofs

# ------------------------------------------------------------------------------------------
------------------------
# 5. FEM EIGEN VALUE SOLVER
# ------------------------------------------------------------------------------------------
------------------------

def compute_Q4_element_mass(density, t, nodes):
    """
```

```
    Compute the consistent mass matrix Q4.

    Parameters:
      density : Material density (kg/m^3)
      t       : Thickness of the element (m)
      nodes   : (4 x 2) node coord.

    Returns:
      M_local : Element mass matrix (8 x 8)
    """
    # Compute the area of the element
    area = compute_area_Q4(nodes)

    # Total mass of the element
    mass = density * t * area

    # Consistent mass matrix for a 4-node quadrilateral element
    M_local = (mass / 36) * np.array([
        [4, 0, 2, 0, 1, 0, 2, 0],
        [0, 4, 0, 2, 0, 1, 0, 2],
        [2, 0, 4, 0, 2, 0, 1, 0],
        [0, 2, 0, 4, 0, 2, 0, 1],
        [1, 0, 2, 0, 4, 0, 2, 0],
        [0, 1, 0, 2, 0, 4, 0, 2],
        [2, 0, 1, 0, 2, 0, 4, 0],
        [0, 2, 0, 1, 0, 2, 0, 4]
    ])

    return M_local

def assemble_global_mass_matrix(density, t, elements, nodes_global):
    """
    Assemble the global mass matrix for the entire structure.

    Parameters:
      density      : Material density (kg/m^3)
      t            : Thickness of the element (m)
      elements     : Connectivity array (list of elements with node indices)
      nodes_global : Global nodal coordinates

    Returns:
      global_M : Global mass matrix
    """
    total_dof = nodes_global.shape[0] * 2  # 2 DOF per node
    global_M = np.zeros((total_dof, total_dof))

    for e_idx, elem in enumerate(elements):
        elem_nodes = nodes_global[elem, :]  # Get nodal coordinates for the element
        M_local = compute_Q4_element_mass(density, t, elem_nodes)  # Local mass matrix

        # Map local mass matrix to global mass matrix
        dof_indices = []
        for node in elem:
            dof_indices.extend([2 * node, 2 * node + 1])
        dof_indices = np.array(dof_indices)

        for i in range(len(dof_indices)):
            for j in range(len(dof_indices)):
                global_M[dof_indices[i], dof_indices[j]] += M_local[i, j]

    return global_M

def fem_eigen_solver(K_reduced, density, t, elements, nodes_global, free_dofs, num_modes=4,
modeShapeScale=[1]):
    """
    Compute the first few eigenfrequencies and mode shapes of the structure.

    Parameters:
      K_reduced   : Reduced stiffness matrix
      density     : Material density (kg/m^3)
```

```
      t           : Thickness of the element (m)
      elements    : Connectivity array (list of elements with node indices)
      nodes_global: Global nodal coordinates
      free_dofs   : List of free degrees of freedom
      num_modes   : Number of eigenfrequencies and mode shapes to compute

    Returns:
      eigenfrequencies : Array of eigenfrequencies (Hz)
      mode_shapes      : Array of mode shapes (eigenvectors)
    """
    # Assemble global mass matrix
    global_M = assemble_global_mass_matrix(density, t, elements, nodes_global)
    print("\nGlobal mass matrix:\n", global_M)

    # Apply boundary conditions
    M_reduced = global_M[np.ix_(free_dofs, free_dofs)]
    print("\nReduced mass matrix:\n", M_reduced)

    # Solve the generalized eigenvalue problem
    eigenvalues, eigenvectors = eigh(K_reduced, M_reduced, subset_by_index=[0, num_modes - 1])
    numModeScale = np.size(modeShapeScale)
    for i in range(numModeScale):
        eigenvectors[:, i] = modeShapeScale[i]*eigenvectors[:, i]

    # Compute eigenfrequencies (Hz)
    eigenfrequencies = np.sqrt(eigenvalues) / (2 * np.pi)

    # Normalize mode shapes
    mode_shapes = np.zeros((global_M.shape[0], num_modes))
    mode_shapes[free_dofs, :] = eigenvectors

    return eigenfrequencies, mode_shapes

# ------------------------------------------------------------------------------------------------
------------------------
# 6. POST-PROCESSING FUNCTIONS
# ------------------------------------------------------------------------------------------------
------------------------

def element_average(elemIdx: int) -> Dict[str, Dict[str, float]]:

    # Initialize averages
    averageValues = {
        'stresses': {'xx': 0.0, 'yy': 0.0, 'xy': 0.0},
        'strains': {'xx': 0.0, 'yy': 0.0, 'xy': 0.0}
    }

    # Get all (0-based)
    element_data = element_results.get(elemIdx)
    if not element_data:
        print(f"Element {elemIdx} not found in results")
        return averageValues

    # Calculate averages across all nodes
    num_nodes = len(element_data['nodes'])
    num_intPts = element_data['intPts']

    for i, comp in enumerate(['xx', 'yy', 'xy']):
        if len(element_data['intePtsStresses'][comp]) != num_intPts:
            print(f"σ_{comp} != no. of integration points")
            break
        averageValues['stresses'][comp] = float(np.mean(element_data['intePtsStresses'][comp]))

    for i, comp in enumerate(['xx', 'yy', 'xy']):
        if len(element_data['intePtsStrains'][comp]) != num_nodes:
            print(f"ε_{comp} != no. of integration points")
            break
        averageValues['strains'][comp] = float(np.mean(element_data['intePtsStrains'][comp]))

    return averageValues
```

```
def extrapolate_gauss_to_nodes(elements: np.ndarray,
                               nodes_global: np.ndarray,
                               isAverageNodalValue: bool = True,
                               element_results: Dict = element_results) -> Tuple[Dict, Dict]:

    # Get shape functions and inverse matrix
    n_gp, gp_points, gp_weights = integrationPoints('Q4', 'FULL')
    if gp_points is None or gp_weights is None:
        print("Gauss points = None")
        return {}, {}

    elemN = [shape_functions_Q4(xi, eta)[0] for xi, eta in gp_points]
    inv_N = np.linalg.inv(np.array(elemN))

    # Initialize data structures
    num_global_nodes = nodes_global.shape[0]
    element_extrapolate = {}
    nodal_accumulator = {
                            'stresses': {comp: np.zeros(num_global_nodes) for comp in ['xx',
'yy', 'xy']},
                            'strains': {comp: np.zeros(num_global_nodes) for comp in ['xx', 'yy',
'xy']},
                            'count': np.zeros(num_global_nodes, dtype=int)
                        } if isAverageNodalValue else None

    # Process each element
    for elem_idx, elem_data in element_results.items():
        elem_nodes = elements[elem_idx]
        num_elem_nodes = len(elem_nodes)

        # Extrapolate stresses and strains
        stresses_gp = np.column_stack([elem_data['intePtsStresses'][comp]
                                       for comp in ['xx', 'yy', 'xy']])
        strains_gp = np.column_stack([elem_data['intePtsStrains'][comp]
                                      for comp in ['xx', 'yy', 'xy']])

        nodal_stresses = inv_N @ stresses_gp  # Shape: (4 nodes, 3 components)
        nodal_strains = inv_N @ strains_gp

        # Store extrapolated results
        element_extrapolate_results[elem_idx] = ElementNodalData(
                                                    element=elem_idx + 1,
                                                    nodes=elem_nodes,
                                                    displacements=elem_data['displacements'],
                                                    nodalStresses={
                                                        'xx': nodal_stresses[:, 0],
                                                        'yy': nodal_stresses[:, 1],
                                                        'xy': nodal_stresses[:, 2]
                                                    },
                                                    nodalStrains={
                                                        'xx': nodal_strains[:, 0],
                                                        'yy': nodal_strains[:, 1],
                                                        'xy': nodal_strains[:, 2]
                                                    }
                                                )

        # Accumulate for averaging
        if isAverageNodalValue and nodal_accumulator is not None:
            for i, node in enumerate(elem_nodes):
                nodal_accumulator['count'][node] += 1
                for comp in ['xx', 'yy', 'xy']:
                    nodal_accumulator['stresses'][comp][node] +=
element_extrapolate_results[elem_idx]['nodalStresses'][comp][i]
                    nodal_accumulator['strains'][comp][node] +=
element_extrapolate_results[elem_idx]['nodalStrains'][comp][i]

    # Average nodal
    nodal_average_data = None
    if isAverageNodalValue and nodal_accumulator is not None:
```

```
        nodal_average_data = {
            'node': np.arange(num_global_nodes),
            'stresses': {comp: [] for comp in ['xx', 'yy', 'xy']},
            'strains': {comp: [] for comp in ['xx', 'yy', 'xy']}
        }

        for node in range(num_global_nodes):
            count = nodal_accumulator['count'][node]
            if count == 0:
                print(f"Node {node+1} not connected to any elements")
            for comp in ['xx', 'yy', 'xy']:
                # Average stresses
                val = nodal_accumulator['stresses'][comp][node] / count if count > 0 else 0.0
                nodal_average_data['stresses'][comp].append(val)
                # Average strains
                val = nodal_accumulator['strains'][comp][node] / count if count > 0 else 0.0
                nodal_average_data['strains'][comp].append(val)

        # To numpy arrays
        for comp in ['xx', 'yy', 'xy']:
            nodal_average_data['stresses'][comp] = np.array(nodal_average_data['stresses'][comp])
            nodal_average_data['strains'][comp] = np.array(nodal_average_data['strains'][comp])

        # --- Assign averaged nodal values back to each element ---
        for elem_idx, elem_data in element_results.items():
            elem_nodes = elements[elem_idx]
            # For each node in the element, get the averaged value from nodal_data
            nodalStresses = {comp: nodal_average_data['stresses'][comp][elem_nodes] for comp in
['xx', 'yy', 'xy']}
            nodalStrains = {comp: nodal_average_data['strains'][comp][elem_nodes] for comp in
['xx', 'yy', 'xy']}
            element_extrapolate_results[elem_idx] = ElementNodalData(
                element=elem_idx + 1,
                nodes=elem_nodes,
                displacements=elem_data['displacements'],
                nodalStresses=nodalStresses,
                nodalStrains=nodalStrains
            )
        element_extrapolate = element_extrapolate_results
    else:
        element_extrapolate = element_extrapolate_results

    if nodal_average_data is None:
        nodal_average_data = {}

    return nodal_average_data, element_extrapolate

def plotUndeformedMesh(nodes_global, elements, point_loads, distributed_loads, figsize = (10,
8)):

    """Plot original problem with undeformed mesh"""
    fig, ax = plt.subplots(figsize=figsize)

    for i, elem in enumerate(elements):
        x_orig = nodes_global[elem, 0]
        y_orig = nodes_global[elem, 1]
        x_orig = np.append(x_orig, x_orig[0])
        y_orig = np.append(y_orig, y_orig[0])
        ax.plot(x_orig, y_orig, 'k-', linewidth=1)

        # Annotate element numbers (center of the element)
        elem_center_x = np.mean(nodes_global[elem, 0])
        elem_center_y = np.mean(nodes_global[elem, 1])
        ax.text(elem_center_x, elem_center_y, f"Q{i + 1}", color="blue", fontsize=10,
ha="center")

    # Annotate node numbers
    for i, (x, y) in enumerate(nodes_global):
        ax.text(x, y, f"{i + 1}", color="red", fontsize=10, ha="right", va="bottom")
```

```
    # Plot point loads as arrows
    for i in range(0, len(point_loads), 2):  # Iterate over load pairs (fx, fy)
        node = point_loads[i][0]  # Node index is the same for both fx and fy
        x, y = nodes_global[node]
        Fx = point_loads[i][2]  # x-direction force
        Fy = point_loads[i + 1][2]  # y-direction force
        magnitude = np.sqrt(Fx**2 + Fy**2)
        if magnitude > 0:
            ax.arrow(x, y, Fx / magnitude * 0.05, Fy / magnitude * 0.05, head_width=0.01,
head_length=0.02,
                     fc='red', ec='red')

    if distributed_loads is not None:
        # Plot distributed loads as arrows
        for dload in distributed_loads:
            n1, n2 = dload['edge']
            direction = dload['direction']
            load_value = dload['value']
            x1, y1 = nodes_global[n1]
            x2, y2 = nodes_global[n2]
            edge_length = np.sqrt((x2 - x1)**2 + (y2 - y1)**2)
            num_arrows = 5
            for i in range(num_arrows):
                x_arrow = x1 + (x2 - x1) * (i + 0.5) / num_arrows
                y_arrow = y1 + (y2 - y1) * (i + 0.5) / num_arrows
                if direction == 0:  # x-direction
                    ax.arrow(x_arrow, y_arrow, load_value / abs(load_value) * 0.05, 0,
head_width=0.01, head_length=0.02,
                             fc='red', ec='red')
                elif direction == 1:  # y-direction
                    ax.arrow(x_arrow, y_arrow, 0, load_value / abs(load_value) * 0.05,
head_width=0.01, head_length=0.02,
                             fc='red', ec='red')

    # Set plot properties
    ax.scatter(nodes_global[:, 0], nodes_global[:, 1], c='blue', marker='o', label="Nodes")
    ax.set_title("Original Mesh with Node and Element Labels")
    ax.set_xlabel("x (m)")
    ax.set_ylabel("y (m)")
    ax.axis("equal")
    ax.legend()

    return fig

def plotDeformedMesh(nodes_global, elements, ux, uy, scaleFactor, figsize = (10, 8),
                     isShowUndeformedShap = True):

    # Deformed nodes
    original_nodes = nodes_global.copy()
    deformed_nodes = original_nodes.copy()
    if ux is not None and uy is not None:
        deformed_nodes += scaleFactor * np.column_stack((ux.ravel(), uy.ravel()))

    fig, ax = plt.subplots(figsize=figsize)

    if isShowUndeformedShap:
        for i, elem in enumerate(elements):
            x_orig = nodes_global[elem, 0]
            y_orig = nodes_global[elem, 1]
            x_orig = np.append(x_orig, x_orig[0])
            y_orig = np.append(y_orig, y_orig[0])
            ax.plot(x_orig, y_orig, 'k--', linewidth=1, label="Original Mesh" if i == 0 else "")

    for i, elem in enumerate(elements):
        x_def = deformed_nodes[elem, 0]
        y_def = deformed_nodes[elem, 1]
        x_def = np.append(x_def, x_def[0])
        y_def = np.append(y_def, y_def[0])
        ax.plot(x_def, y_def, 'r-', linewidth=2, label="Deformed Mesh" if i == 0 else "")
```

```
    ax.scatter(nodes_global[:, 0], nodes_global[:, 1], c='blue', marker='o', label="Original
Nodes")
    ax.scatter(deformed_nodes[:, 0], deformed_nodes[:, 1], c='red', marker='x', label="Deformed
Nodes")
    ax.set_title(f"Deformed Configuration (Scale factor = {scaleFactor})")
    ax.legend()

    return fig

def plotModeShape(nodes_global, elements, eigenfrequencies, mode_shapes, figsize=(10, 8),
isShowUndeformedShap=True):
    numMode = len(eigenfrequencies)
    fig, axes = plt.subplots(nrows=(numMode + 1) // 2, ncols=2, figsize=figsize)
    axes = axes.flatten()
    scale_mode = 1

    for mode in range(numMode):
        mode_shape = mode_shapes[:, mode]
        mode_displacements = np.column_stack((mode_shape[0::2], mode_shape[1::2]))
        deformed_mode_nodes = nodes_global + scale_mode * mode_displacements

        if isShowUndeformedShap:
            for i, elem in enumerate(elements):
                x_orig = nodes_global[elem, 0]
                y_orig = nodes_global[elem, 1]
                x_orig = np.append(x_orig, x_orig[0])
                y_orig = np.append(y_orig, y_orig[0])
                axes[mode].plot(x_orig, y_orig, 'k--', linewidth=1, label="Original Mesh" if i ==
0 else "")

        for i, elem in enumerate(elements):
            x_def = deformed_mode_nodes[elem, 0]
            y_def = deformed_mode_nodes[elem, 1]
            x_def = np.append(x_def, x_def[0])
            y_def = np.append(y_def, y_def[0])
            axes[mode].plot(x_def, y_def, 'b-', linewidth=2, label="Mode Shape" if i == 0 else
"")

        axes[mode].scatter(nodes_global[:, 0], nodes_global[:, 1], c='blue', marker='o',
label="Original Nodes")
        axes[mode].scatter(deformed_mode_nodes[:, 0], deformed_mode_nodes[:, 1], c='red',
marker='x', label="Deformed Nodes")
        axes[mode].set_title(f"Mode {mode + 1} (f = {eigenfrequencies[mode]:.2f} Hz)")
        axes[mode].set_xlabel("x (m)")
        axes[mode].set_ylabel("y (m)")
        axes[mode].axis("equal")
        axes[mode].legend()

    fig.tight_layout()
    return fig

def generate_triangulation(elements, nodes_global):
    """Convert quadrilateral mesh to triangular mesh for plotting"""
    triangles = []
    for elem in elements:
        n1, n2, n3, n4 = elem  # Q4
        triangles.append([n1, n2, n3])
        triangles.append([n1, n3, n4])

    return tri.Triangulation(nodes_global[:, 0], nodes_global[:, 1], triangles)

def plotSmoothQuadContours(element_extrapolate, nodes_global, elements,
                           component='stresses', field='xx',
                           ux=None, uy=None,
                           resolution=20, isShowNodes=True, isShowElements=True,
                           plotTitle="", scaleFactor=1,
                           isShowUndeformedShape=True,  # Controls undeformed mesh visibility
                           isShowScaleFactor=True,
                           node_fontsize=8, element_fontsize=10,
                           figsize=(10, 8)):
```

```
    fig, ax = plt.subplots(figsize=figsize)

    # Deformed nodes
    original_nodes = nodes_global.copy()
    deformed_nodes = original_nodes.copy()
    if ux is not None and uy is not None:
        deformed_nodes += scaleFactor * np.column_stack((ux.ravel(), uy.ravel()))

    # Generate Contour Data
    all_x = []
    all_y = []
    all_data = []
    all_triangles = []
    point_index = 0  # current index for triangle

    for elem_idx in element_extrapolate:
        elem_data = element_extrapolate[elem_idx]
        nodes = elem_data['nodes']
        elem_coords = deformed_nodes[nodes]  # Use deformed coordinates

        # Get extrapolated values for this component/field
        nodal_values = elem_data['nodalStresses' if component == 'stresses' else
'nodalStrains'][field]

        # Parametric grid
        xi = np.linspace(-1, 1, resolution)
        eta = np.linspace(-1, 1, resolution)
        xi_grid, eta_grid = np.meshgrid(xi, eta, indexing='ij')

        # Interpolate to grid
        x_phys = []
        y_phys = []
        data = []
        for xi_val, eta_val in zip(xi_grid.ravel(), eta_grid.ravel()):
            N, _, _ = shape_functions_Q4(xi_val, eta_val)
            x = N @ elem_coords[:, 0]
            y = N @ elem_coords[:, 1]
            val = N @ nodal_values

            x_phys.append(x)
            y_phys.append(y)
            data.append(val)

        # Store points
        current_indices = np.arange(point_index, point_index + len(x_phys))
        all_x.extend(x_phys)
        all_y.extend(y_phys)
        all_data.extend(data)

        # Create triangles for this element's grid
        indices = current_indices.reshape(resolution, resolution)
        for i in range(resolution-1):
            for j in range(resolution-1):
                all_triangles.append([indices[i, j], indices[i+1, j], indices[i, j+1]])
                all_triangles.append([indices[i+1, j], indices[i+1, j+1], indices[i, j+1]])

        point_index += len(x_phys)

    # --------------------------
    # Plotting
    # --------------------------
    triang = tri.Triangulation(all_x, all_y, triangles=all_triangles)
    tcf = ax.tricontourf(triang, all_data, levels=20, cmap='jet')
    fig.colorbar(tcf, ax=ax, label=f'{component} {field}')

    # Plot deformed mesh edges
    for elem in elements:
        elem_nodes = deformed_nodes[elem]
        elem_nodes_closed = np.vstack([elem_nodes, elem_nodes[0]])  # Close
```

```
        ax.plot(elem_nodes_closed[:, 0], elem_nodes_closed[:, 1], 'k-', linewidth=0.5)

    # Plot undeformed mesh
    if isShowUndeformedShape:
        for i, elem in enumerate(elements):
            x_orig = nodes_global[elem, 0]
            y_orig = nodes_global[elem, 1]
            x_orig = np.append(x_orig, x_orig[0])
            y_orig = np.append(y_orig, y_orig[0])
            ax.plot(x_orig, y_orig, 'k--', linewidth=1, label="Original Mesh" if i == 0 else "")

    # Annotations
    if isShowNodes:
        # Offset for node numbers (as a fraction of the mesh size)
        x_span = np.max(deformed_nodes[:, 0]) - np.min(deformed_nodes[:, 0])
        y_span = np.max(deformed_nodes[:, 1]) - np.min(deformed_nodes[:, 1])
        x_offset = 0.015 * x_span
        y_offset = 0.025 * y_span
        for node_idx, (x, y) in enumerate(deformed_nodes):
            ax.text(x + x_offset, y + y_offset, str(node_idx+1), color='black',
fontsize=node_fontsize,
                    ha='left', va='bottom')

    if isShowUndeformedShape and isShowNodes:
        ax.scatter(nodes_global[:, 0], nodes_global[:, 1], c='blue', marker='o', label="Original
Nodes")
    # ax.scatter(deformed_nodes[:, 0], deformed_nodes[:, 1], c='red', marker='x', label="Deformed
Nodes")

    if isShowElements:
        for elem_idx, elem in enumerate(elements):
            center = np.mean(deformed_nodes[elem], axis=0)
            ax.text(center[0], center[1], f"Q{elem_idx+1}", color='black',
                    fontsize=element_fontsize, ha='center', va='center')

    # Show scale factor on plot
    if isShowScaleFactor:
        ax.text(0.99, 0.01, f"Scale factor = {scaleFactor}", color='black',
                fontsize=10, ha='right', va='bottom', transform=ax.transAxes)

    ax.set_title(plotTitle or f"Deformed Mesh with {component} {field}")
    ax.axis('equal')
    return fig

def plotElementAverage(element_result, nodes_global,
                       elements,
                       component='stresses', field='xx',
                       ux=None, uy=None,
                       scaleFactor=1,
                       isShowNodes=True, isShowElements=True,
                       isShowUndeformedShape=True,
                       isShowScaleFactor=True,
                       node_fontsize=8, element_fontsize=10,
                       plotTitle="",
                       figsize=(10, 8)):
    """
    Plot element-based contours using average of all integration points in each element,
    and overlay both deformed and undeformed mesh.
    """
    fig, ax = plt.subplots(figsize=figsize)

    # Compute deformed node coordinates
    original_nodes = nodes_global.copy()
    deformed_nodes = original_nodes.copy()
    if ux is not None and uy is not None:
        deformed_nodes += scaleFactor * np.column_stack((ux.ravel(), uy.ravel()))

    quads = []
    values = []
```

```
    for elem_idx in element_result:
        elem_data = element_result[elem_idx]
        nodes = elem_data['nodes']
        coords = deformed_nodes[nodes]  # Use deformed coordinates for filled polygons

        # Compute average value for this element (over all integration points)
        if component == 'stresses':
            data_array = elem_data.get('intePtsStresses', {}).get(field, None)
        else:
            data_array = elem_data.get('intePtsStrains', {}).get(field, None)
        if data_array is not None:
            avg_val = np.mean(data_array)
        else:
            avg_val = 0.0

        quads.append(coords)
        values.append(avg_val)

    # Create PolyCollection for all elements (deformed mesh)
    pc = PolyCollection(quads, array=np.array(values), cmap='jet', edgecolors='k',
linewidths=0.5)
    ax.add_collection(pc)
    fig.colorbar(pc, ax=ax, label=f'{field} ({component})')

    # Overlay deformed mesh edges
    for elem in elements:
        elem_nodes = deformed_nodes[elem]
        elem_nodes_closed = np.vstack([elem_nodes, elem_nodes[0]])
        ax.plot(elem_nodes_closed[:, 0], elem_nodes_closed[:, 1], 'k-', linewidth=0.5)

    # Overlay undeformed mesh edges
    if isShowUndeformedShape:
        for i, elem in enumerate(elements):
            x_orig = nodes_global[elem, 0]
            y_orig = nodes_global[elem, 1]
            x_orig = np.append(x_orig, x_orig[0])
            y_orig = np.append(y_orig, y_orig[0])
            ax.plot(x_orig, y_orig, 'k--', linewidth=1, label="Original Mesh" if i == 0 else "")

    # Annotate node numbers (optional, offset from deformed node)
    if isShowNodes:
        x_span = np.max(deformed_nodes[:, 0]) - np.min(deformed_nodes[:, 0])
        y_span = np.max(deformed_nodes[:, 1]) - np.min(deformed_nodes[:, 1])
        x_offset = 0.015 * x_span
        y_offset = 0.025 * y_span
        for node_idx, (x, y) in enumerate(deformed_nodes):
            ax.text(x + x_offset, y + y_offset, str(node_idx+1), color='black',
fontsize=node_fontsize,
                    ha='left', va='bottom')

    if isShowUndeformedShape and isShowNodes:
        ax.scatter(nodes_global[:, 0], nodes_global[:, 1], c='blue', marker='o', label="Original
Nodes")
    # ax.scatter(deformed_nodes[:, 0], deformed_nodes[:, 1], c='red', marker='x', label="Deformed
Nodes")

    # Annotate element numbers (optional, at center of deformed element)
    if isShowElements:
        for elem_idx, elem in enumerate(elements):
            center = np.mean(deformed_nodes[elem], axis=0)
            ax.text(center[0], center[1], f"Q{elem_idx+1}", color='black',
                    fontsize=element_fontsize, ha='center', va='center')

    # Show scale factor on plot
    if isShowScaleFactor:
        ax.text(0.99, 0.01, f"Scale factor = {scaleFactor}", color='black',
                fontsize=10, ha='right', va='bottom', transform=ax.transAxes)

    ax.set_title(plotTitle or f"Element-Based {component} - {field}")
    ax.set_xlabel("X")
```

```
    ax.set_ylabel("Y")
    ax.axis('equal')

    return fig

# ------------------------------------------------------------------------------------------------
------------------------
# A. UTILITY FUNCTIONS
# ------------------------------------------------------------------------------------------------
------------------------

def pointLoadToXY(node, FF, alphaF):
    Fx = FF * np.cos(alphaF)
    Fy = FF * np.sin(alphaF)
    point_loads = [
        (node, 0, Fx),  # Node _, x-direction
        (node, 1, Fy)   # Node _, y-direction
    ]

    return point_loads

def transform_to_natural_coordinates(sequence, nodes_input):
    """
    Reorder nodal coordinates (by user) to Q4 natural ordering.
    Natural order: [bottom left, bottom right, top right, top left].

    Parameters:
      sequence    : List of order (e.g. [1,4,2,3]).
      nodes_input : (4 x 2) array of original nodal coordinates.

    Returns:
      nodes       : Reordered nodal coordinates in the natural order.
    """
    if sorted(sequence) != [1, 2, 3, 4]:
        print("Sequence must contain 1 & 2 & 3 & 4.")
    nodes = np.array([
        nodes_input[sequence[0]-1],  # Node 1: bottom left
        nodes_input[sequence[1]-1],  # Node 2: bottom right
        nodes_input[sequence[2]-1],  # Node 3: top right
        nodes_input[sequence[3]-1]   # Node 4: top left
    ])
    return nodes

# ------------------------------------------------------------------------------------------------
------------------------
# 7. MAIN PROGRAM
# ------------------------------------------------------------------------------------------------
------------------------

if __name__ == "__main__":

    # ------------------------------------------------------------
    # --- Set numpy print options ---------------------------------
    np.set_printoptions(
        linewidth=np.inf,  # type: ignore # Disable line wrapping
        precision=4,       # Show 4 decimal places
        suppress=False,      # Suppress scientific notation
        threshold=np.inf    # type: ignore # Show all elements
    )

    # ------------------------------------------------------------------------------------------------
--------------------
    # === Preparation
================================================================================================
    # ------------------------------------------------------------------------------------------------
--------------------
    E           = 1.8e11            # Young's modulus (Pa)
    nu          = 0.25              # Poisson's ratio
    # density     = 7830              # kg/m^3
    a           = 0.3               # m
```

```
    p           = 5000e6            # N/m
    F           = 200000e3          # N
    alpha       = 60                # Degrees
    t           = 1                 # 1m, unit length, keep 1 for plane strain
    GVL_mode    = "PLANE_STRAIN"    # "PLANE_STRAIN" / "PLANE_STRESS"
    GVL_integrationMode = "Full"  # "SelReduced" / "BBar" / "Full" / "IncompatibleStrain"

    # Global nodes
    nodes_global = np.array([
        [0      , a],      # Node 1
        [a      , a],      # Node 2
        [2 * a  , a],      # Node 3
        [3 * a  , a],      # Node 4
        [0      , 2 * a],      # Node 5
        [a      , 2 * a],      # Node 6
        [2 * a  , 2 * a],      # Node 7
        [3 * a  , 2 * a],      # Node 8
        [a      , 0    ],      # Node 9
        [2 * a  , 0    ],      # Node 10
    ])

    # Define elements
    elements = np.array([
        [1, 2, 6, 5],
        [2, 3, 7, 6],
        [3, 4, 8, 7],
        [9, 10, 3, 2]
    ])

    elements = elements - 1  # Convert to 0-based

    num_nodes = nodes_global.shape[0]
    total_dof = num_nodes * 2  # 2 DOF per node

    # --- Force Vector Assembly -------------------------------------
    # Point loads:
    point_loads = []

    F1 = F
    alpha1 = alpha  # Degrees
    alpha1 = 2 * np.pi - np.deg2rad(90 + alpha1)  # radians
    point_loads += pointLoadToXY(node=8, FF=F1, alphaF=alpha1)

    # Distributed loads:
    distributed_loads = []
    distributed_loads = [
        {'edge': (4, 5), 'direction': 1, 'value': (-1) * p},  # Load value (N/m) for nodes 5 and
6
        {'edge': (5, 6), 'direction': 1, 'value': (-1) * p},  # Load value (N/m) for nodes 6 and
7
        {'edge': (6, 7), 'direction': 1, 'value': (-1) * p}   # Load value (N/m) for nodes 7 and
8
    ]

    # --- Apply Boundary Conditions ---------------------------------
    fixed_dofs = []
    # Fix nodes 1, 5, 4, 10
    for fixed_node in [0, 4, 3, 9]:
        fixed_dofs.extend([2 * fixed_node, 2 * fixed_node + 1])
    fixed_dofs = np.array(fixed_dofs)

    # ------------------------------------------------------------------------------------------
-------------------
    # === Solve
==========================================================================================
====
    # ------------------------------------------------------------------------------------------
-------------------
```

```
    # --- Solve solid mechanic prob. ----------------------------------------------------------
--------------------
    results, K_reduced, free_dofs = fem_solver(E, nu, elements, nodes_global, t, point_loads,
distributed_loads,
                                               fixed_dofs, mode=GVL_mode,
integrationMode=GVL_integrationMode)

    # --- Extract Results ---
    # Nodal displacement print
    U = results["nodal_displacements"]
    ux = U[0::2]
    uy = U[1::2]
    print("\nNodal Displacements (m):")
    for i in range(num_nodes):
        print(f"Node {i + 1}: u_x = {ux[i]:.6e}, u_y = {uy[i]:.6e}")

    # Max. nodal displacement in y-direction
    max_uy = np.max(uy)
    max_uy_node = np.argmax(uy) + 1  # 1-based node index
    print(f"\nMaximum y-displacement: u_y = {max_uy:.6e} m at Node {max_uy_node}")
    # Min. nodal displacement in y-direction
    min_uy = np.min(uy)
    min_uy_node = np.argmin(uy) + 1  # 1-based node index
    print(f"\nMinimum y-displacement: u_y = {min_uy:.6e} m at Node {min_uy_node}")

    # Element stress prints
    print("\nElement Stresses (Pa):")
    for indxElemResult in element_results:
        element_result = element_results.get(indxElemResult)
        if element_result is None:
            print(f"Element {indxElemResult} not found in results")
            continue

        print(f"Element {element_result['element']}:")
        intPtsStress = element_result['intePtsStresses']
        for intePt in range(element_result['intPts']):
            print(f"  GP {intePt + 1}: σ_xx = {intPtsStress['xx'][intePt]:.6e}, \
                  σ_yy = {intPtsStress['yy'][intePt]:.6e}, σ_xy =
{intPtsStress['xy'][intePt]:.6e}")

    # Element strain prints
    print("\nElement Strain:")
    for indxElemResult in element_results:
        element_result = element_results.get(indxElemResult)
        if element_result is None:
            print(f"Element {indxElemResult} not found in results")
            continue

        print(f"Element {element_result['element']}:")
        intPtsStrain = element_result['intePtsStrains']
        for intePt in range(element_result['intPts']):
            print(f"  GP {intePt + 1}: ε_xx = {intPtsStrain['xx'][intePt]:.6e}, \
            ε_yy = {intPtsStrain['yy'][intePt]:.6e}, ε_xy = {intPtsStrain['xy'][intePt]:.6e}")

    # Element average
    for indxElemResult in element_results:
        element_result = element_results.get(indxElemResult)
        averageValues = element_average(indxElemResult)
        if element_result is None:
            print(f"Element {indxElemResult} not found in results")
            continue

        print(f"\nElement {element_result['element']} mean stress: σ_xx =
{averageValues['stresses']['xx']:.6e}, \
              σ_yy = {averageValues['stresses']['yy']:.6e}, σ_xy =
{averageValues['stresses']['xy']:.6e}")
    for indxElemResult in element_results:
        element_result = element_results.get(indxElemResult)
        averageValues = element_average(indxElemResult)
        if element_result is None:
```

```
            print(f"Element {indxElemResult} not found in results")
            continue

        print(f"\nElement {element_result['element']} mean strain: ε_xx =
{averageValues['strains']['xx']:.6e}, \
              ε_yy = {averageValues['strains']['yy']:.6e}, ε_xy =
{averageValues['strains']['xy']:.6e}")

    # ------------------------------------------------------------------------------------------
--------------------
    # === Post-Processing & Plotting
================================================================================
    # ------------------------------------------------------------------------------------------
--------------------

    # --- Plot Original Mesh with Node and Element Labels ---------------------------------
    fig1 = plotUndeformedMesh(nodes_global, elements, point_loads, distributed_loads, figsize =
(10,8))

    # --- Plot Deformed mesh --------------------------------------------------------------
    fig2 = plotDeformedMesh(nodes_global, elements, ux, uy, scaleFactor=1,
                            figsize = (10,8), isShowUndeformedShap=True)

    # --- Plot Stress and Strain----------------------------------------------------------

    nodal_data, _ = extrapolate_gauss_to_nodes(elements, nodes_global, isAverageNodalValue=False)
    # Get extrapolated nodal values
    print("\nExtrapolated data:")
    header = "{:<10} {:<10} {:<15} {:<15} {:<15} {:<15} {:<15} {:<15}".format("Node", "Element",
"Stress XX", "Stress YY", "Stress XY",
                                                                                "Strain XX",
"Strain YY", "Strain XY"
                                                                            )

    format_str = (
                    "{:<10} {:<10} "
                    "{:<15.6e} {:<15.6e} {:<15.6e} "
                    "{:<15.6e} {:<15.6e} {:<15.6e}"
                )

    print("\n" + header)
    print("-" * 116)

    for indxElemResult in element_extrapolate_results:
        element_ext_result = element_extrapolate_results[indxElemResult]
        for i, node in enumerate(element_ext_result['nodes']):
            print(format_str.format(
                int(node + 1),
                int(element_ext_result['element']),
                element_ext_result['nodalStresses']['xx'][i],
                element_ext_result['nodalStresses']['yy'][i],
                element_ext_result['nodalStresses']['xy'][i],
                element_ext_result['nodalStrains']['xx'][i],
                element_ext_result['nodalStrains']['yy'][i],
                element_ext_result['nodalStrains']['xy'][i]
            ))

    fig3a = plotSmoothQuadContours(
                                        element_extrapolate_results,
                                        nodes_global,
                                        elements,
                                        component='stresses',
                                        field='xx',
                                        ux=ux,
                                        uy=uy,
                                        resolution=20,
                                        isShowNodes=True,
                                        isShowElements=True,
                                        plotTitle= "σ_xx (Pa)",
                                        scaleFactor=1,
```

```
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=8,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

    fig3b = plotSmoothQuadContours(
                                    element_extrapolate_results,
                                    nodes_global,
                                    elements,
                                    component='stresses',
                                    field='yy',
                                    ux=ux,
                                    uy=uy,
                                    resolution=20,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "σ_yy (Pa)",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=8,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

    fig3c = plotSmoothQuadContours(
                                    element_extrapolate_results,
                                    nodes_global,
                                    elements,
                                    component='stresses',
                                    field='xy',
                                    ux=ux,
                                    uy=uy,
                                    resolution=20,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "σ_xy (Pa)",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=10,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

    fig4a = plotSmoothQuadContours(
                                    element_extrapolate_results,
                                    nodes_global,
                                    elements,
                                    component='strains',
                                    field='xx',
                                    ux=ux,
                                    uy=uy,
                                    resolution=20,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "ε_xx",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=10,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

    fig4b = plotSmoothQuadContours(
                                    element_extrapolate_results,
                                    nodes_global,
```

```
                                    elements,
                                    component='strains',
                                    field='yy',
                                    ux=ux,
                                    uy=uy,
                                    resolution=20,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "ε_yy",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=10,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig4c = plotSmoothQuadContours(
                                    element_extrapolate_results,
                                    nodes_global,
                                    elements,
                                    component='strains',
                                    field='xy',
                                    ux=ux,
                                    uy=uy,
                                    resolution=20,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "ε_xy",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=10,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig5a = plotElementAverage(
                                    element_results,
                                    nodes_global,
                                    elements,
                                    component='stresses',
                                    field='xx',
                                    ux=ux,
                                    uy=uy,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "σ_xx (Pa)",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=8,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig5b = plotElementAverage(
                                    element_results,
                                    nodes_global,
                                    elements,
                                    component='stresses',
                                    field='yy',
                                    ux=ux,
                                    uy=uy,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "σ_yy (Pa)",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
```

```
                                    node_fontsize=8,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig5c = plotElementAverage(
                                    element_results,
                                    nodes_global,
                                    elements,
                                    component='stresses',
                                    field='xy',
                                    ux=ux,
                                    uy=uy,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "σ_xy (Pa)",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=8,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig6a = plotElementAverage(
                                    element_results,
                                    nodes_global,
                                    elements,
                                    component='strains',
                                    field='xx',
                                    ux=ux,
                                    uy=uy,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "ε_xx",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=8,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig6b = plotElementAverage(
                                    element_results,
                                    nodes_global,
                                    elements,
                                    component='stresses',
                                    field='yy',
                                    ux=ux,
                                    uy=uy,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "ε_yy",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=8,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig6c = plotElementAverage(
                                    element_results,
                                    nodes_global,
                                    elements,
                                    component='stresses',
                                    field='xy',
                                    ux=ux,
                                    uy=uy,
```

```
                                        isShowNodes=True,
                                        isShowElements=True,
                                        plotTitle= "ε_xy",
                                        scaleFactor=1,
                                        isShowUndeformedShape=False,
                                        isShowScaleFactor=True,
                                        node_fontsize=8,
                                        element_fontsize=10,
                                        figsize=(10, 8)
                                    )

    # --- Plot Stress and strains with nodal average values-----------------------------------

    nodal_data, _ = extrapolate_gauss_to_nodes(elements, nodes_global, isAverageNodalValue=True)
    # Get extrapolated nodal values
    print("\nExtrapolated data - Nodal average:")
    header = "{:<10} {:<10} {:<15} {:<15} {:<15} {:<15} {:<15} {:<15}".format("Node", "Element",
"Stress XX", "Stress YY", "Stress XY",
                                                                                "Strain XX",
"Strain YY", "Strain XY"
                                                                            )

    format_str = (
                    "{:<10} {:<10} "
                    "{:<15.6e} {:<15.6e} {:<15.6e} "
                    "{:<15.6e} {:<15.6e} {:<15.6e}"
                )

    print("\n" + header)
    print("-" * 116)

    for indxElemResult in element_extrapolate_results:
        element_ext_result = element_extrapolate_results[indxElemResult]
        for i, node in enumerate(element_ext_result['nodes']):
            print(format_str.format(
                int(node + 1),
                int(element_ext_result['element']),
                element_ext_result['nodalStresses']['xx'][i],
                element_ext_result['nodalStresses']['yy'][i],
                element_ext_result['nodalStresses']['xy'][i],
                element_ext_result['nodalStrains']['xx'][i],
                element_ext_result['nodalStrains']['yy'][i],
                element_ext_result['nodalStrains']['xy'][i]
            ))

    fig8a = plotSmoothQuadContours(
                                        element_extrapolate_results,
                                        nodes_global,
                                        elements,
                                        component='stresses',
                                        field='xx',
                                        ux=ux,
                                        uy=uy,
                                        resolution=20,
                                        isShowNodes=True,
                                        isShowElements=True,
                                        plotTitle= "σ_xx (Pa) - Nodal average",
                                        scaleFactor=1,
                                        isShowUndeformedShape=False,
                                        isShowScaleFactor=True,
                                        node_fontsize=8,
                                        element_fontsize=10,
                                        figsize=(10, 8)
                                    )

    fig8b = plotSmoothQuadContours(
                                        element_extrapolate_results,
                                        nodes_global,
                                        elements,
                                        component='stresses',
```

```
                                        field='yy',
                                        ux=ux,
                                        uy=uy,
                                        resolution=20,
                                        isShowNodes=True,
                                        isShowElements=True,
                                        plotTitle= "σ_yy (Pa) - Nodal average",
                                        scaleFactor=1,
                                        isShowUndeformedShape=False,
                                        isShowScaleFactor=True,
                                        node_fontsize=8,
                                        element_fontsize=10,
                                        figsize=(10, 8)
                                    )

    fig8c = plotSmoothQuadContours(
                                        element_extrapolate_results,
                                        nodes_global,
                                        elements,
                                        component='stresses',
                                        field='xy',
                                        ux=ux,
                                        uy=uy,
                                        resolution=20,
                                        isShowNodes=True,
                                        isShowElements=True,
                                        plotTitle= "σ_xy (Pa) - Nodal average",
                                        scaleFactor=1,
                                        isShowUndeformedShape=False,
                                        isShowScaleFactor=True,
                                        node_fontsize=10,
                                        element_fontsize=10,
                                        figsize=(10, 8)
                                    )

    fig9a = plotSmoothQuadContours(
                                        element_extrapolate_results,
                                        nodes_global,
                                        elements,
                                        component='strains',
                                        field='xx',
                                        ux=ux,
                                        uy=uy,
                                        resolution=20,
                                        isShowNodes=True,
                                        isShowElements=True,
                                        plotTitle= "ε_xx - Nodal average",
                                        scaleFactor=1,
                                        isShowUndeformedShape=False,
                                        isShowScaleFactor=True,
                                        node_fontsize=10,
                                        element_fontsize=10,
                                        figsize=(10, 8)
                                    )

    fig9b = plotSmoothQuadContours(
                                        element_extrapolate_results,
                                        nodes_global,
                                        elements,
                                        component='strains',
                                        field='yy',
                                        ux=ux,
                                        uy=uy,
                                        resolution=20,
                                        isShowNodes=True,
                                        isShowElements=True,
                                        plotTitle= "ε_yy - Nodal average",
                                        scaleFactor=1,
                                        isShowUndeformedShape=False,
                                        isShowScaleFactor=True,
```

```
                                    node_fontsize=10,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

fig9c = plotSmoothQuadContours(
                                    element_extrapolate_results,
                                    nodes_global,
                                    elements,
                                    component='strains',
                                    field='xy',
                                    ux=ux,
                                    uy=uy,
                                    resolution=20,
                                    isShowNodes=True,
                                    isShowElements=True,
                                    plotTitle= "ε_xy - Nodal average",
                                    scaleFactor=1,
                                    isShowUndeformedShape=False,
                                    isShowScaleFactor=True,
                                    node_fontsize=10,
                                    element_fontsize=10,
                                    figsize=(10, 8)
                                )

plt.show()
```

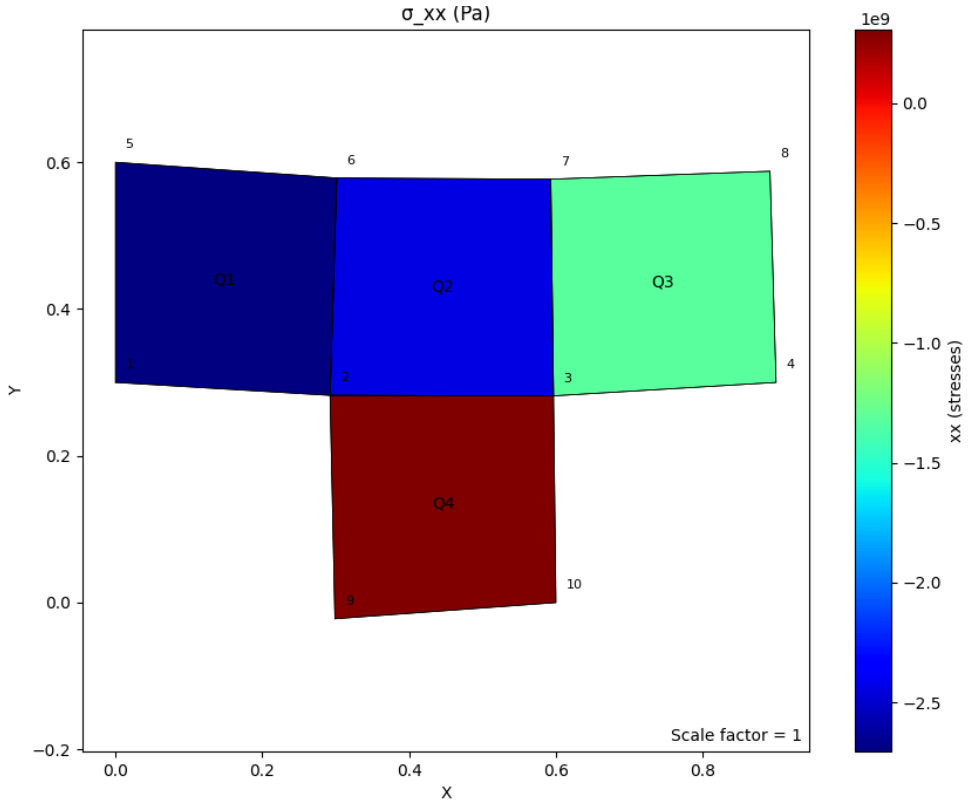
PHỤ LỤC B. Output – Câu 4 Full integration

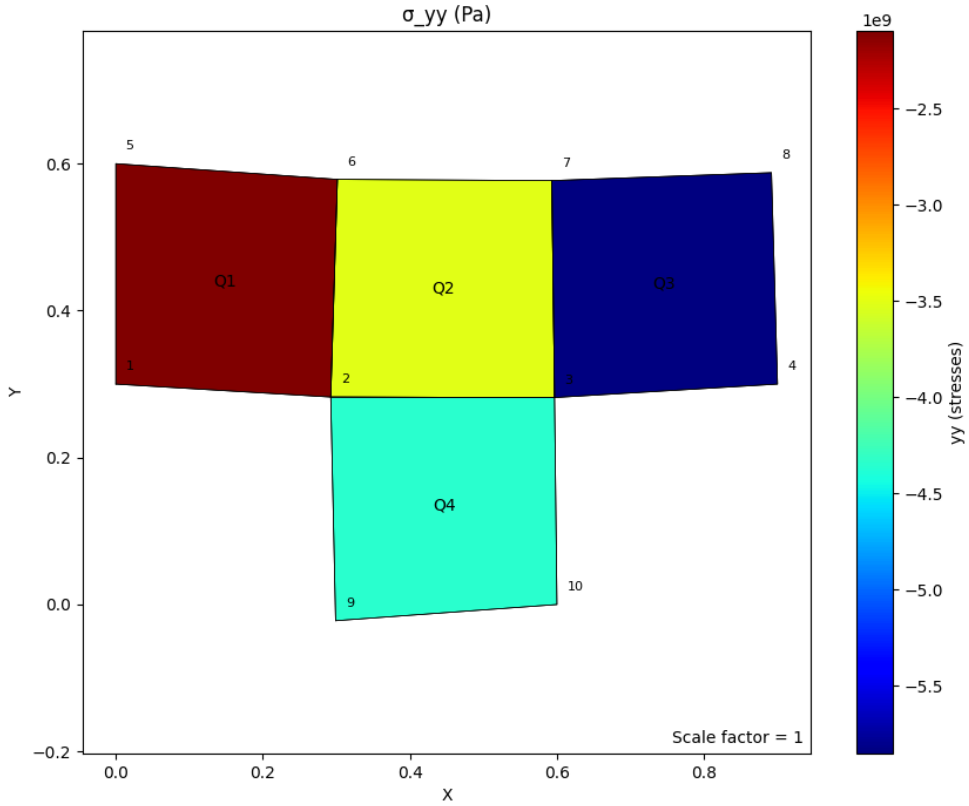
```
Processing: Element stiffness matrix, full integration.......

Full Integration Points (2x2 Quadrature):
GP0: (ξ=-0.5774, η=-0.5774), weight=(1.0000, 1.0000)
GP1: (ξ=0.5774, η=-0.5774), weight=(1.0000, 1.0000)
GP2: (ξ=0.5774, η=0.5774), weight=(1.0000, 1.0000)
GP3: (ξ=-0.5774, η=0.5774), weight=(1.0000, 1.0000)

Element 1:
 [[ 9.6000e+10  3.6000e+10 -6.0000e+10 -4.7684e-07 -4.8000e+10 -3.6000e+10  1.2000e+10 -9.5367e-
07]
 [ 3.6000e+10  9.6000e+10 -4.7684e-07  1.2000e+10 -3.6000e+10 -4.8000e+10 -9.5367e-07 -
6.0000e+10]
 [-6.0000e+10  4.7684e-07  9.6000e+10 -3.6000e+10  1.2000e+10  4.7684e-07 -4.8000e+10
3.6000e+10]
 [ 4.7684e-07  1.2000e+10 -3.6000e+10  9.6000e+10  4.7684e-07 -6.0000e+10  3.6000e+10 -
4.8000e+10]
 [-4.8000e+10 -3.6000e+10  1.2000e+10  0.0000e+00  9.6000e+10  3.6000e+10 -6.0000e+10 -9.5367e-
07]
 [-3.6000e+10 -4.8000e+10  0.0000e+00 -6.0000e+10  3.6000e+10  9.6000e+10 -9.5367e-07
1.2000e+10]
 [ 1.2000e+10  0.0000e+00 -4.8000e+10  3.6000e+10 -6.0000e+10  9.5367e-07  9.6000e+10 -
3.6000e+10]
 [ 0.0000e+00 -6.0000e+10  3.6000e+10 -4.8000e+10  9.5367e-07  1.2000e+10 -3.6000e+10
9.6000e+10]]

.
.
.

Reduced Global Stiffness Matrix (K):
 [[ 2.8800e+11 -3.6000e+10 -1.2000e+11  4.7684e-07  2.4000e+10 -4.7684e-07 -4.8000e+10 -
3.6000e+10  0.0000e+00  0.0000e+00  1.2000e+10  0.0000e+00]
 [-3.6000e+10  2.8800e+11  4.7684e-07  2.4000e+10 -4.7684e-07 -1.2000e+11 -3.6000e+10 -4.8000e+10
0.0000e+00  0.0000e+00  0.0000e+00 -6.0000e+10]
 [-1.2000e+11 -4.7684e-07  2.8800e+11  3.6000e+10 -4.8000e+10  3.6000e+10  2.4000e+10 -1.4305e-06
-4.8000e+10 -3.6000e+10 -4.8000e+10 -3.6000e+10]
 [-4.7684e-07  2.4000e+10  3.6000e+10  2.8800e+11  3.6000e+10 -4.8000e+10 -1.4305e-06 -1.2000e+11
-3.6000e+10 -4.8000e+10 -3.6000e+10 -4.8000e+10]
 [ 2.4000e+10  0.0000e+00 -4.8000e+10  3.6000e+10  1.9200e+11  0.0000e+00 -6.0000e+10  9.5367e-07
0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
 [ 0.0000e+00 -1.2000e+11  3.6000e+10 -4.8000e+10  0.0000e+00  1.9200e+11  9.5367e-07  1.2000e+10
0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
 [-4.8000e+10 -3.6000e+10  2.4000e+10  9.5367e-07 -6.0000e+10 -9.5367e-07  1.9200e+11 -7.6294e-06
-6.0000e+10  0.0000e+00  0.0000e+00  0.0000e+00]
 [-3.6000e+10 -4.8000e+10  9.5367e-07 -1.2000e+11 -9.5367e-07  1.2000e+10 -7.6294e-06  1.9200e+11
0.0000e+00  1.2000e+10  0.0000e+00  0.0000e+00]
 [ 0.0000e+00  0.0000e+00 -4.8000e+10 -3.6000e+10  0.0000e+00  0.0000e+00 -6.0000e+10  0.0000e+00
9.6000e+10  3.6000e+10  0.0000e+00  0.0000e+00]
 [ 0.0000e+00  0.0000e+00 -3.6000e+10 -4.8000e+10  0.0000e+00  0.0000e+00  0.0000e+00  1.2000e+10
3.6000e+10  9.6000e+10  0.0000e+00  0.0000e+00]
 [ 1.2000e+10 -9.5367e-07 -4.8000e+10 -3.6000e+10  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00
0.0000e+00  0.0000e+00  9.6000e+10  3.6000e+10]
 [-9.5367e-07 -6.0000e+10 -3.6000e+10 -4.8000e+10  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00
0.0000e+00  0.0000e+00  3.6000e+10  9.6000e+10]]

Reduced Force Vector (f):
 [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00 -1.5000e+09  0.0000e+00 -1.5000e+09
0.0000e+00 -7.5000e+08 -1.7321e+08 -1.0000e+08]

Nodal Displacements (m):
Node 1: u_x = 0.000000e+00, u_y = 0.000000e+00
Node 2: u_x = -7.674101e-03, u_y = -1.763349e-02
Node 3: u_x = -3.264614e-03, u_y = -1.833043e-02
Node 4: u_x = 0.000000e+00, u_y = 0.000000e+00
Node 5: u_x = 0.000000e+00, u_y = 0.000000e+00
Node 6: u_x = 1.403728e-03, u_y = -2.136536e-02
```

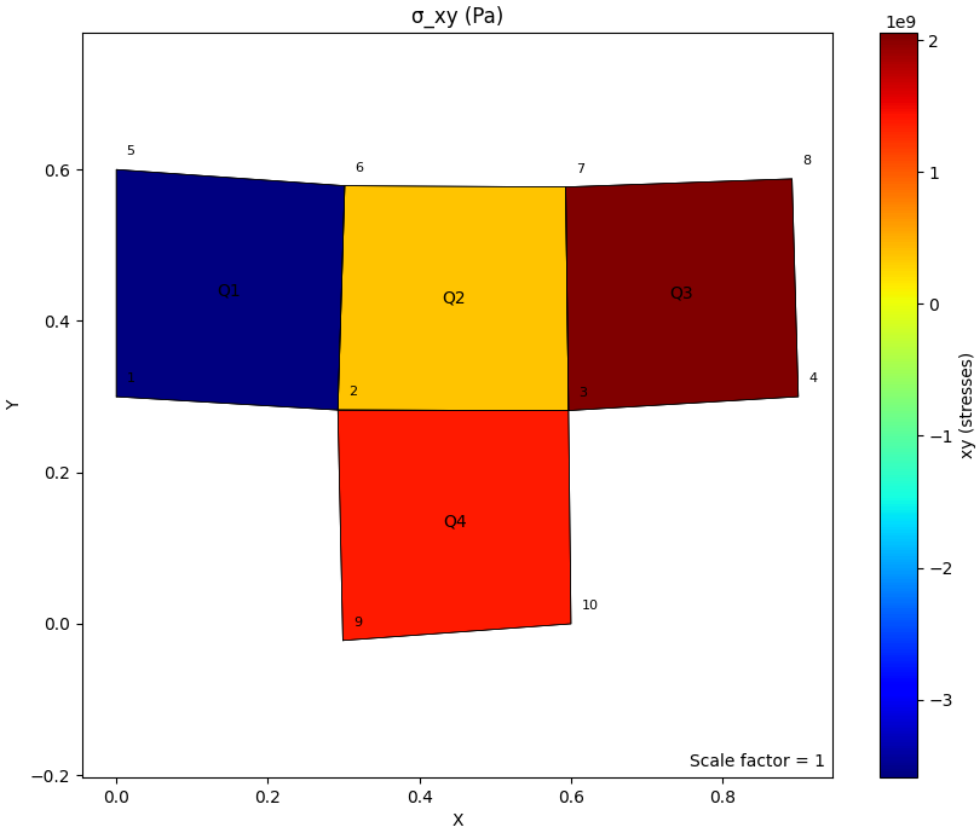
```
Node 7: u_x = -6.964278e-03, u_y = -2.301712e-02
Node 8: u_x = -8.275887e-03, u_y = -1.222135e-02
Node 9: u_x = -1.084112e-03, u_y = -2.204550e-02
Node 10: u_x = 0.000000e+00, u_y = 0.000000e+00

Maximum y-displacement: u_y = 0.000000e+00 m at Node 1

Minimum y-displacement: u_y = -2.301712e-02 m at Node 7

Element Stresses (Pa):
Element 1:
  GP 1: σ_xx = -4.333398e+09,                  σ_yy = -1.949193e+09, σ_xy = -3.960902e+09
  GP 2: σ_xx = -4.850500e+09,                  σ_yy = -3.500499e+09, σ_xy = -2.703041e+09
  GP 3: σ_xx = -1.076917e+09,                  σ_yy = -2.242638e+09, σ_xy = -3.220143e+09
  GP 4: σ_xx = -5.598153e+08,                  σ_yy = -6.913318e+08, σ_xy = -4.478004e+09
Element 2:
  GP 1: σ_xx = 2.866115e+08,                  σ_yy = -2.421993e+09, σ_xy = 1.314938e+09
  GP 2: σ_xx = 1.543074e+08,                  σ_yy = -2.818905e+09, σ_xy = -4.555630e+08
  GP 3: σ_xx = -5.157197e+09,                  σ_yy = -4.589407e+09, σ_xy = -5.878671e+08
  GP 4: σ_xx = -5.024893e+09,                  σ_yy = -4.192495e+09, σ_xy = 1.182634e+09
Element 3:
  GP 1: σ_xx = 1.472844e+08,                  σ_yy = -3.969432e+09, σ_xy = 2.897145e+09
  GP 2: σ_xx = -8.967492e+08,                  σ_yy = -7.101532e+09, σ_xy = 2.263045e+09
  GP 3: σ_xx = -2.799049e+09,                  σ_yy = -7.735632e+09, σ_xy = 1.219011e+09
  GP 4: σ_xx = -1.755016e+09,                  σ_yy = -4.603532e+09, σ_xy = 1.853111e+09
Element 4:
  GP 1: σ_xx = 1.191962e+09,                  σ_yy = 1.451397e+08, σ_xy = 2.724529e+09
  GP 2: σ_xx = -1.959324e+09,                  σ_yy = -9.308716e+09, σ_xy = 3.185307e+09
  GP 3: σ_xx = -5.769912e+08,                  σ_yy = -8.847939e+09, σ_xy = 3.402145e+07
  GP 4: σ_xx = 2.574294e+09,                  σ_yy = 6.059172e+08, σ_xy = -4.267561e+08

Element Strain:
Element 1:
  GP 1: ε_xx = -1.918577e-02,             ε_yy = -2.628785e-03, ε_xy = -5.501252e-02
  GP 2: ε_xx = -1.918577e-02,             ε_yy = -9.810759e-03, ε_xy = -3.754223e-02
  GP 3: ε_xx = -1.715476e-03,             ε_yy = -9.810759e-03, ε_xy = -4.472421e-02
  GP 4: ε_xx = -1.715476e-03,             ε_yy = -2.628785e-03, ε_xy = -6.219450e-02
Element 2:
  GP 1: ε_xx = 5.697617e-03,             ε_yy = -1.311214e-02, ε_xy = 1.826303e-02
  GP 2: ε_xx = 5.697617e-03,             ε_yy = -1.494969e-02, ε_xy = -6.327264e-03
  GP 3: ε_xx = -1.889268e-02,             ε_yy = -1.494969e-02, ε_xy = -8.164821e-03
  GP 4: ε_xx = -1.889268e-02,             ε_yy = -1.311214e-02, ε_xy = 1.642548e-02
Element 3:
  GP 1: ε_xx = 7.658481e-03,             ε_yy = -2.092983e-02, ε_xy = 4.023812e-02
  GP 2: ε_xx = 7.658481e-03,             ε_yy = -3.543029e-02, ε_xy = 3.143118e-02
  GP 3: ε_xx = -1.148464e-03,             ε_yy = -3.543029e-02, ε_xy = 1.693071e-02
  GP 4: ε_xx = -1.148464e-03,             ε_yy = -2.092983e-02, ε_xy = 2.573766e-02
Element 4:
  GP 1: ε_xx = 5.956155e-03,             ε_yy = -1.313442e-03, ε_xy = 3.784068e-02
  GP 2: ε_xx = 5.956155e-03,             ε_yy = -4.508129e-02, ε_xy = 4.424037e-02
  GP 3: ε_xx = 1.235584e-02,             ε_yy = -4.508129e-02, ε_xy = 4.725201e-04
  GP 4: ε_xx = 1.235584e-02,             ε_yy = -1.313442e-03, ε_xy = -5.927168e-03

Element 1 mean stress: σ_xx = -2.705158e+09, σ_yy = -2.095916e+09, σ_xy = -3.590522e+09

Element 2 mean stress: σ_xx = -2.435293e+09, σ_yy = -3.505700e+09, σ_xy = 3.635356e+08

Element 3 mean stress: σ_xx = -1.325882e+09, σ_yy = -5.852532e+09, σ_xy = 2.058078e+09

Element 4 mean stress: σ_xx = 3.074852e+08, σ_yy = -4.351400e+09, σ_xy = 1.379275e+09

Element 1 mean strain: ε_xx = -1.045062e-02, ε_yy = -6.219772e-03, ε_xy = -4.986837e-02

Element 2 mean strain: ε_xx = -6.597532e-03, ε_yy = -1.403092e-02, ε_xy = 5.049106e-03

Element 3 mean strain: ε_xx = 3.255008e-03, ε_yy = -2.818006e-02, ε_xy = 2.858442e-02

Element 4 mean strain: ε_xx = 9.155999e-03, ε_yy = -2.319737e-02, ε_xy = 1.915660e-02

Extrapolated data:
```

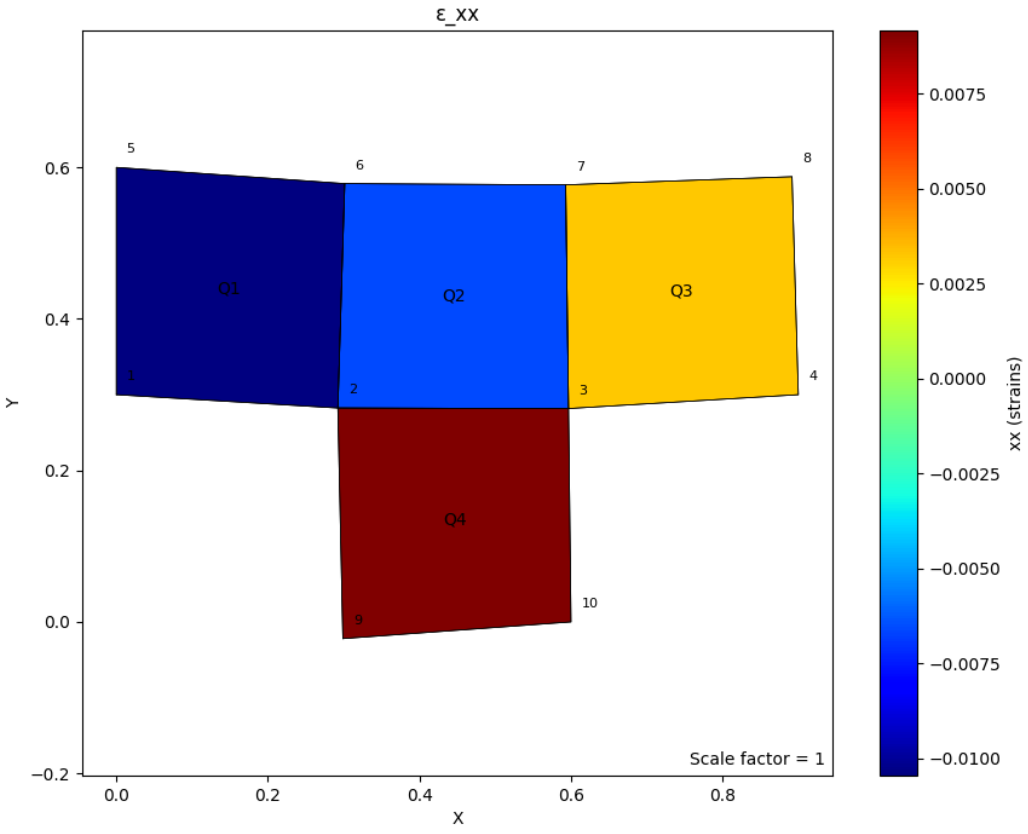
```
Node       Element    Stress XX       Stress YY       Stress XY       Strain XX       Strain YY
Strain XY
-------------------------------------------------------------------------------------------------
------------------
1          1          -5.525353e+09   -1.841784e+09   -4.232038e+09   -2.558034e-02   -8.714615e-
19   -5.877831e-02
2          1          -6.421000e+09   -4.528726e+09   -2.053359e+09   -2.558034e-02   -1.243954e-
02   -2.851888e-02
6          1          1.150372e+08    -2.350047e+09   -2.949006e+09   4.679095e-03    -1.243954e-
02   -4.095842e-02
5          1          1.010684e+09    3.368948e+08    -5.127685e+09   4.679095e-03    -1.182414e-
18   -7.121785e-02
2          2          2.279184e+09    -1.628665e+09   2.011414e+09    1.469829e-02    -1.243954e-
02   2.793630e-02
3          2          2.050026e+09    -2.316137e+09   -1.055185e+09   1.469829e-02    -1.562229e-
02   -1.465535e-02
7          2          -7.149769e+09   -5.382735e+09   -1.284342e+09   -2.789335e-02   -1.562229e-
02   -1.783809e-02
6          2          -6.920612e+09   -4.695263e+09   1.782256e+09    -2.789335e-02   -1.243954e-
02   2.475356e-02
3          3          1.225717e+09    -2.590907e+09   3.511384e+09    1.088205e-02    -1.562229e-
02   4.876923e-02
4          3          -5.826018e+08   -8.015864e+09   2.413091e+09    1.088205e-02    -4.073783e-
02   3.351515e-02
8          3          -3.877482e+09   -9.114157e+09   6.047718e+08    -4.372030e-03   -4.073783e-
02   8.399608e-03
7          3          -2.069163e+09   -3.689200e+09   1.703065e+09    -4.372030e-03   -1.562229e-
02   2.365368e-02
9          4          1.839443e+09    3.436835e+09    3.709324e+09    3.613706e-03    1.470670e-
02    5.151838e-02
10         4          -3.618743e+09   -1.293772e+10   4.507414e+09    3.613706e-03    -6.110144e-
02   6.260297e-02
3          4          -1.224473e+09   -1.213963e+10   -9.507728e+08   1.469829e-02    -6.110144e-
02   -1.320518e-02
2          4          4.233714e+09    4.234925e+09    -1.748863e+09   1.469829e-02    1.470670e-
02    -2.428976e-02

Extrapolated data - Nodal average:

Node       Element    Stress XX       Stress YY       Stress XY       Strain XX       Strain YY
Strain XY
-------------------------------------------------------------------------------------------------
------------------
1          1          -5.525353e+09   -1.841784e+09   -4.232038e+09   -2.558034e-02   -8.714615e-
19   -5.877831e-02
2          1          3.063242e+07    -6.408218e+08   -5.969361e+08   1.272082e-03    -3.390795e-
03   -8.290780e-03
6          1          -3.402787e+09   -3.522655e+09   -5.833751e+08   -1.160713e-02   -1.243954e-
02   -8.102432e-03
5          1          1.010684e+09    3.368948e+08    -5.127685e+09   4.679095e-03    -1.182414e-
18   -7.121785e-02
2          2          3.063242e+07    -6.408218e+08   -5.969361e+08   1.272082e-03    -3.390795e-
03   -8.290780e-03
3          2          6.837569e+08    -5.682226e+09   5.018089e+08    1.342621e-02    -3.078200e-
02   6.969568e-03
7          2          -4.609466e+09   -4.535968e+09   2.093614e+08    -1.613269e-02   -1.562229e-
02   2.907798e-03
6          2          -3.402787e+09   -3.522655e+09   -5.833751e+08   -1.160713e-02   -1.243954e-
02   -8.102432e-03
3          3          6.837569e+08    -5.682226e+09   5.018089e+08    1.342621e-02    -3.078200e-
02   6.969568e-03
4          3          -5.826018e+08   -8.015864e+09   2.413091e+09    1.088205e-02    -4.073783e-
02   3.351515e-02
8          3          -3.877482e+09   -9.114157e+09   6.047718e+08    -4.372030e-03   -4.073783e-
02   8.399608e-03
7          3          -4.609466e+09   -4.535968e+09   2.093614e+08    -1.613269e-02   -1.562229e-
02   2.907798e-03
9          4          1.839443e+09    3.436835e+09    3.709324e+09    3.613706e-03    1.470670e-
02    5.151838e-02
```

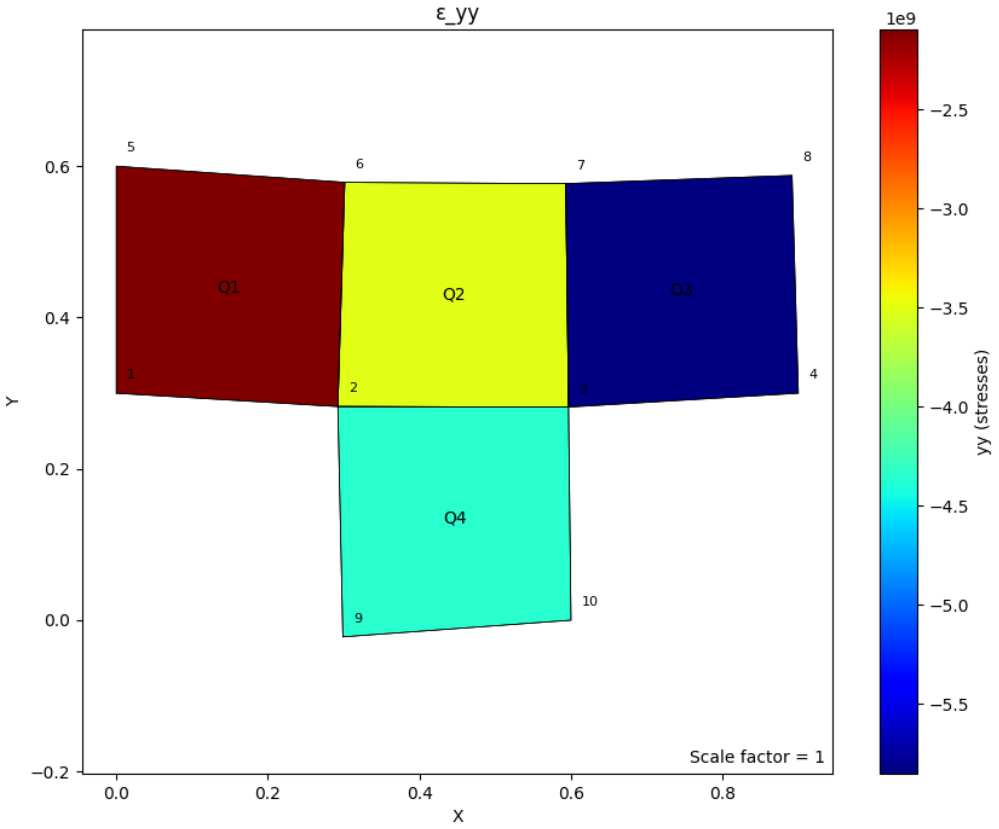
```
10          4           -3.618743e+09   -1.293772e+10   4.507414e+09    3.613706e-03    -6.110144e-
02    6.260297e-02
3           4           6.837569e+08    -5.682226e+09   5.018089e+08    1.342621e-02    -3.078200e-
02    6.969568e-03
2           4           3.063242e+07    -6.408218e+08   -5.969361e+08   1.272082e-03    -3.390795e-
03    -8.290780e-03
```

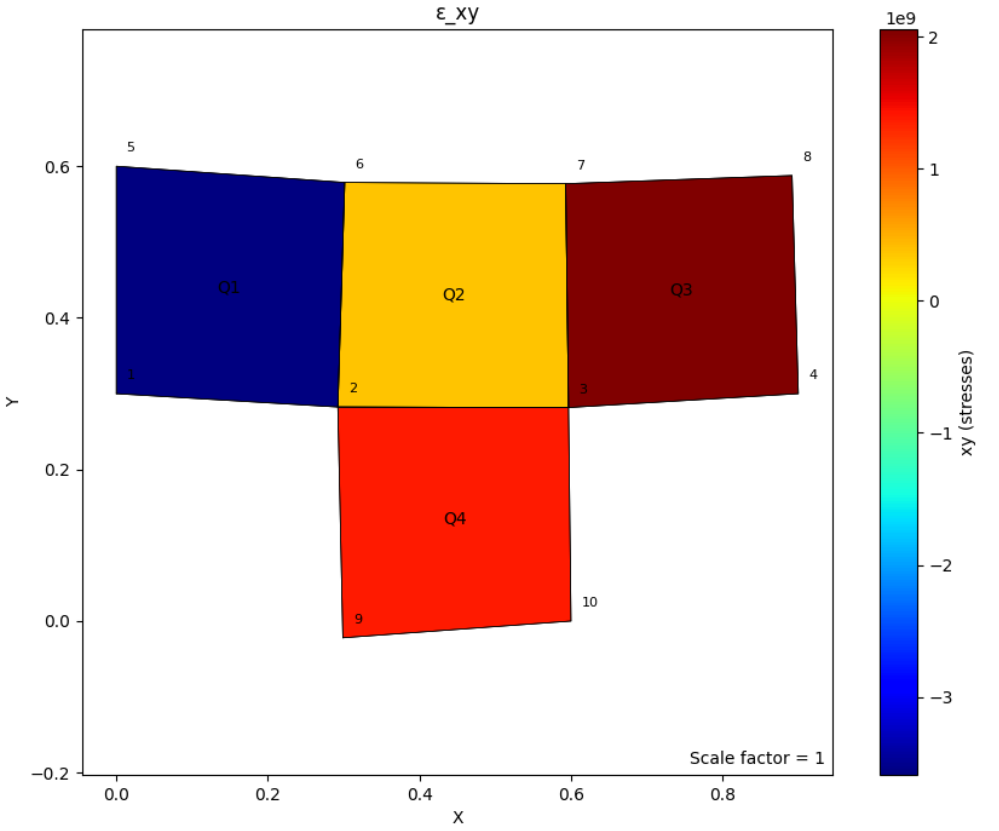


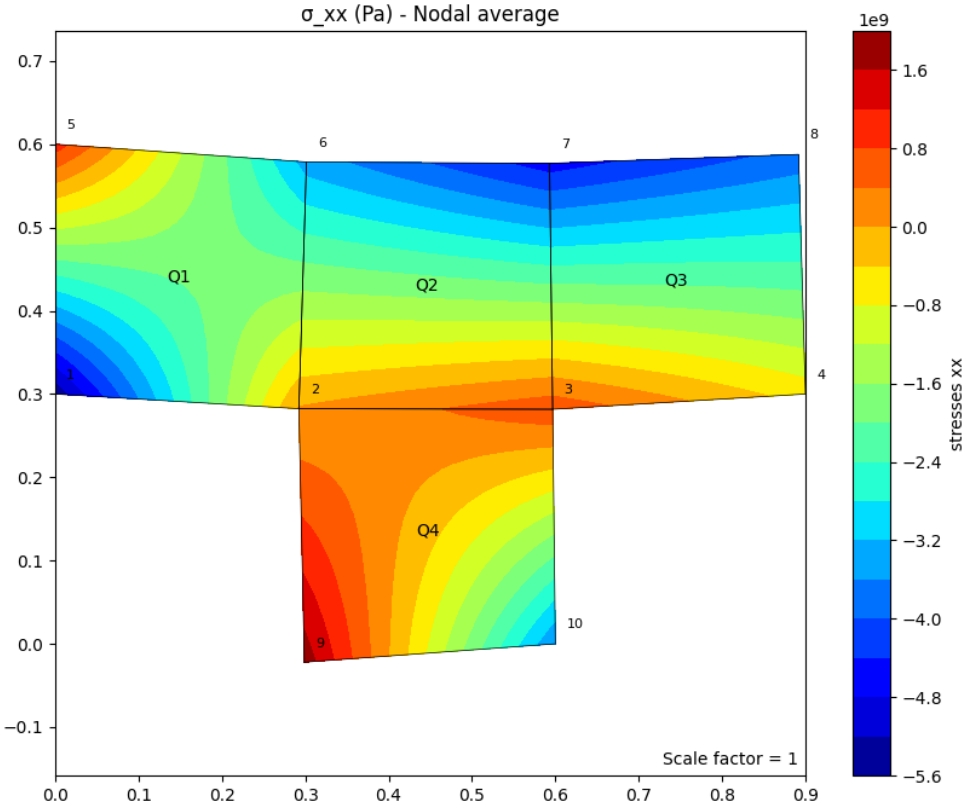


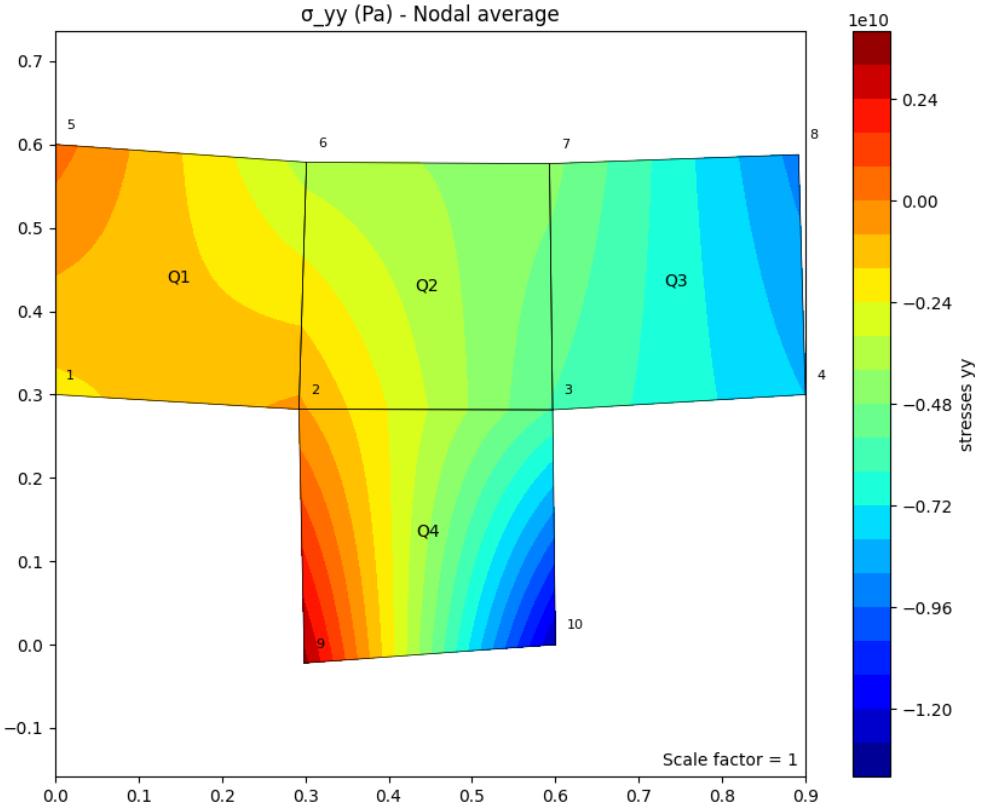

σ_xy (Pa)
1e9
2
1
0
−1
−2
−3
xy (stresses)
5
6
7
8
Q1
Q2
Q3
1
2
3
4
Q4
9
10
Scale factor = 1
0.6
0.4
0.2
0.0
−0.2
Y
0.0
0.2
0.4
0.6
0.8
X

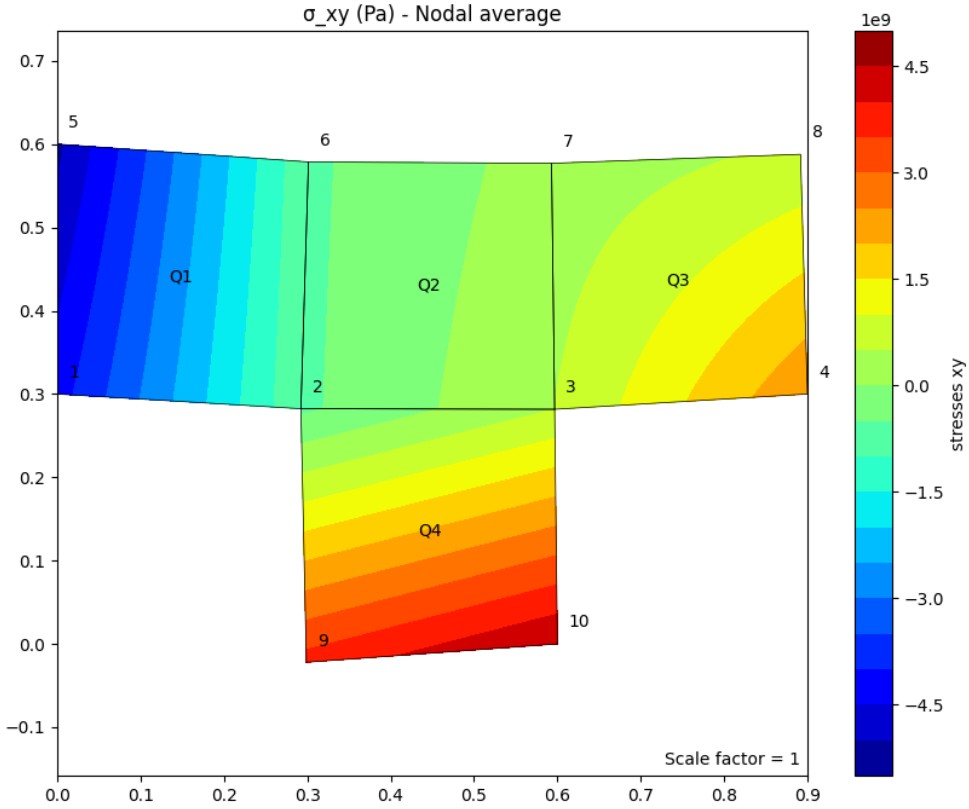

ε_xx
0.0075
0.0050
0.0025
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
xx (strains)
5
6
7
8
Q1
Q2
Q3
1
2
3
4
Q4
9
10
Scale factor = 1
0.6
0.4
0.2
0.0
−0.2
Y
0.0
0.2
0.4
0.6
0.8
X

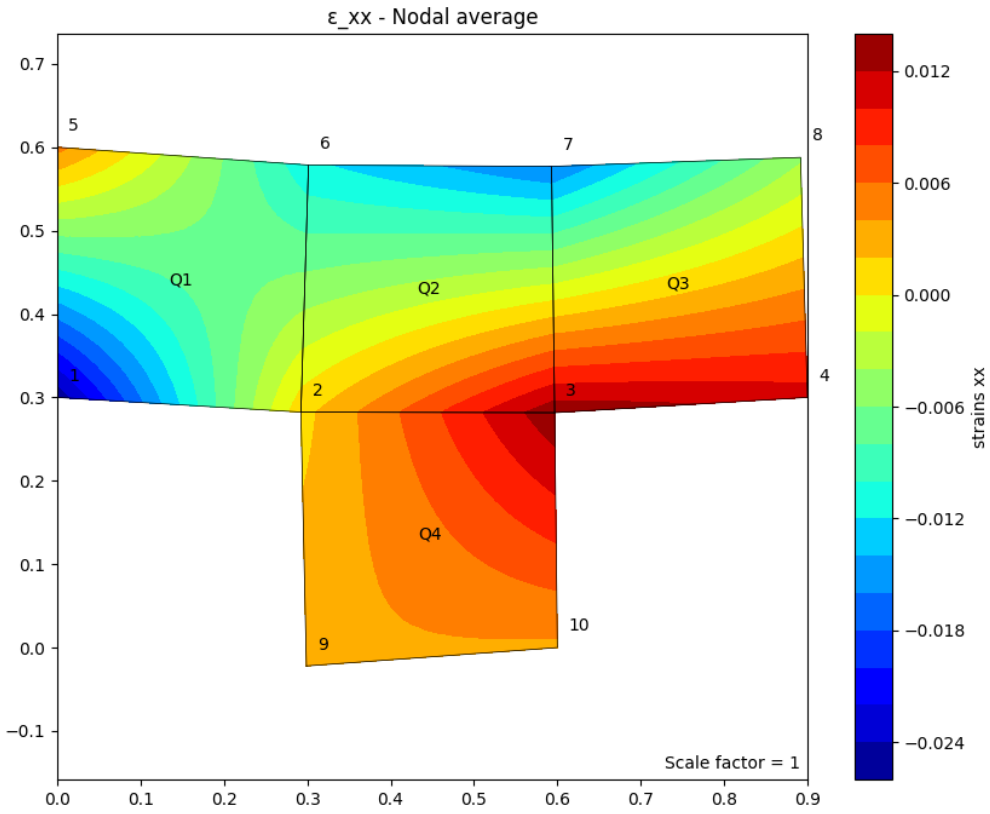

ε_yy
Q1
Q2
Q3
Q4
1
2
3
4
5
6
7
8
9
10
Scale factor = 1
X
Y
1e9
yy (stresses)

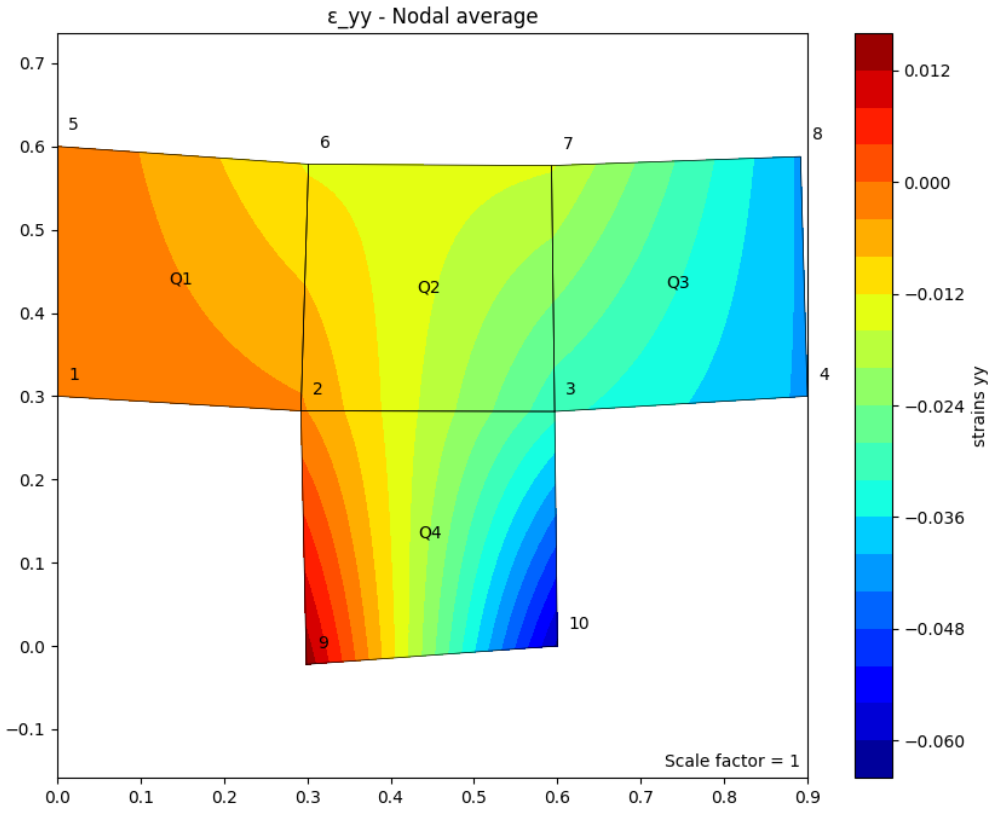

ε_xy
Q1
Q2
Q3
Q4
1
2
3
4
5
6
7
8
9
10
Scale factor = 1
X
Y
1e9
xy (stresses)


σ_xx (Pa) - Nodal average
1e9
1.6
0.8
0.0
−0.8
−1.6
−2.4
−3.2
−4.0
−4.8
−5.6
stresses xx
Q1
Q2
Q3
Q4
Scale factor = 1


σ_yy (Pa) - Nodal average
1e10
0.24
0.00
−0.24
−0.48
−0.72
−0.96
−1.20
stresses yy
Q1
Q2
Q3
Q4
Scale factor = 1


σ_xy (Pa) - Nodal average
1e9
4.5
3.0
1.5
0.0
−1.5
−3.0
−4.5
stresses xy
Q1
Q2
Q3
Q4
1
2
3
4
5
6
7
8
9
10
Scale factor = 1


ε_xx - Nodal average
0.012
0.006
0.000
−0.006
−0.012
−0.018
−0.024
strains xx
Q1
Q2
Q3
Q4
1
2
3
4
5
6
7
8
9
10
Scale factor = 1



Hình B-1 (a) - (l). Biểu diễn các kết quả khác của câu 4

TÀI LIỆU THAM KHẢO

[1] ANSYS, Inc., ANSYS CFX-Solver Theory Guide, ANSYS, Inc., 2017.

[2] ANSYS, Inc., ANSYS CFX Introduction, ANSYS, Inc., 2017.

[3] ANSYS, Inc., "ANSYS Mechanical APDL Element Reference," ANSYS, Inc., 2011.

[4] ANSYS, Inc., "ANSYS Mechanical APDL Command Reference," ANSYS, Inc., 2010.

[5] A. F. Bower, Applied Mechanics of Solids, ISBN 978-1439802472: CRC Press, 2009.

[6] K.-J. Bathe, Finite Element Procedures, ISBN 0-13-301458-4: Prentice Hall, 1996.