**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

[MỤC LỤC i](#_Toc199428540)

[ĐỊNH NGHĨA TỪ VIẾT TẮT ii](#_Toc199428541)

[KÝ HIỆU iii](#_Toc199428542)

[DANH MỤC HÌNH iv](#_Toc199428543)

[1. Câu 1 5](#_Toc199428544)

[1.1. Giới thiệu chung về Phương pháp Phần tử Hữu hạn (FEM) 5](#_Toc199428545)

[1.2. Phần tử phẳng Q4 5](#_Toc199428546)

[1.3. Các bước cơ bản của FEM với phần tử Q4 5](#_Toc199428547)

[1.4. Ưu điểm và nhược điểm của phần tử Q4 6](#_Toc199428548)

[1.5. Ứng dụng 6](#_Toc199428549)

[2. Câu 2 7](#_Toc199428550)

[3. Câu 3 11](#_Toc199428551)

[4. Câu 4 15](#_Toc199428552)

[PHỤ LỤC A. PRG\_FEM\_Plane.py 26](#_Toc199428553)

[PHỤ LỤC B. Output – Câu 4 Full integration 58](#_Toc199428554)

# 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 |
|  | Khối lượng riêng |
|  | Vector vận tốc của |
|  | Thể tích mà môi trường chiếm chỗ ở thời điểm |
|  | Thể tích mà môi trường chiếm chỗ ở thời điểm |
|  | Vector định vị điểm ở thời điểm ban đầu trong hệ trực giao |
|  | Vector định vị điểm ở thời điểm sau trong hệ trực giao |
|  | Định thức Jacobian |
|  | Áp suất tĩnh (nhiệt động lực học) |
|  | Tensor ứng suất |
|  | Độ nhớt động học |
|  | Nguồn động lượng |
| **f** | Lực khối như trọng lực hoặc lực quán tính,… |

# DANH MỤC HÌNH

Hình Trang

[Hình 4‑1 Lưới chưa biến dạng và điều kiện biên 16](#_Toc199428454)

[Hình 4‑2 Ứng suất phương XX 23](#_Toc199428455)

[Hình 4‑3 Ứng suất chương YY 23](#_Toc199428456)

[Hình 4‑4 Ứng suất phương XY 24](#_Toc199428457)

[Hình 4‑5 Biến dạng phương XX 24](#_Toc199428458)

[Hình 4‑6 Biến dạng phương YY 25](#_Toc199428459)

[Hình 4‑7 Biến dạng phương XY 25](#_Toc199428460)

[Hình B‑1 (a) - (l). Biểu diễn các kết quả khác của câu 4 66](#_Toc199428462)

# Câu 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.

## 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.

## 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 được định nghĩa:

trong đó là tọa độ của nút trong hệ tọa độ tự nhiên.

c. Ma trận độ cứng phần tử

Trong bài toán cơ học, ma trận độ cứng phần tử :

: Ma trận quan hệ biến dạng-chuyển vị.

: Ma trận vật liệu (liên hệ ứng suất-biến dạng).

: 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:

với là ma trận độ cứng tổng thể, là vector chuyển vị nút, và là vector lực tại nút.

## 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.

## Ứng dụng

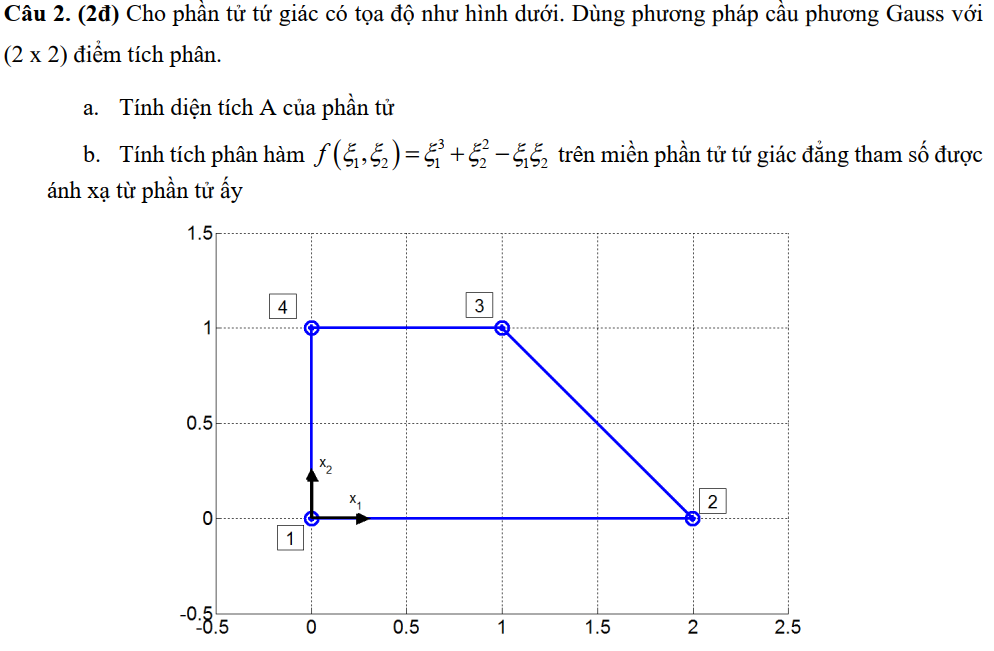
Phần tử Q4 thường được dùng trong:

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

Bài toán dẫn nhiệt 2D.

Mô hình hóa tấm, vỏ đơn giản

# Câu 2



--- Kết quả ---------------------------------------------------------------------------------------------------

--- Lời giải ---------------------------------------------------------------------------------------------------

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

* Nút 1:
* Nút 2:
* Nút 3:
* Nút 4:

a. Tính diện tích A của phần tử

Diện tích được tính bằng tích phân định thức Jacobian trên miền tọa độ tự nhiên :

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:

Tọa độ vật lý của điểm bất kỳ:

Jacobian:

Định thức:

Diện tích:

b. Tính tích phân hàm trên miền phần tử

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

Trọng số:

Khi đó:

Diện tích của phần tử cũng có thể được tính toán bằng cầu phương Gauss 2x2, với . 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\AdvanceNumericalMethod\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}")

# Câu 3

A math problem with numbers and lines

AI-generated content may be incorrect.

--- 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”** ---------------------------------------------------------

import numpy as np

from scipy.linalg import eigh, lu\_factor, lu\_solve

# 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

# 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\AdvanceNumericalMethod\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]

# Câu 4

A diagram of a line with numbers and lines

AI-generated content may be incorrect.

--- 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”** ---------------------------------------------------------

A screen shot of a graph

AI-generated content may be incorrect.

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,

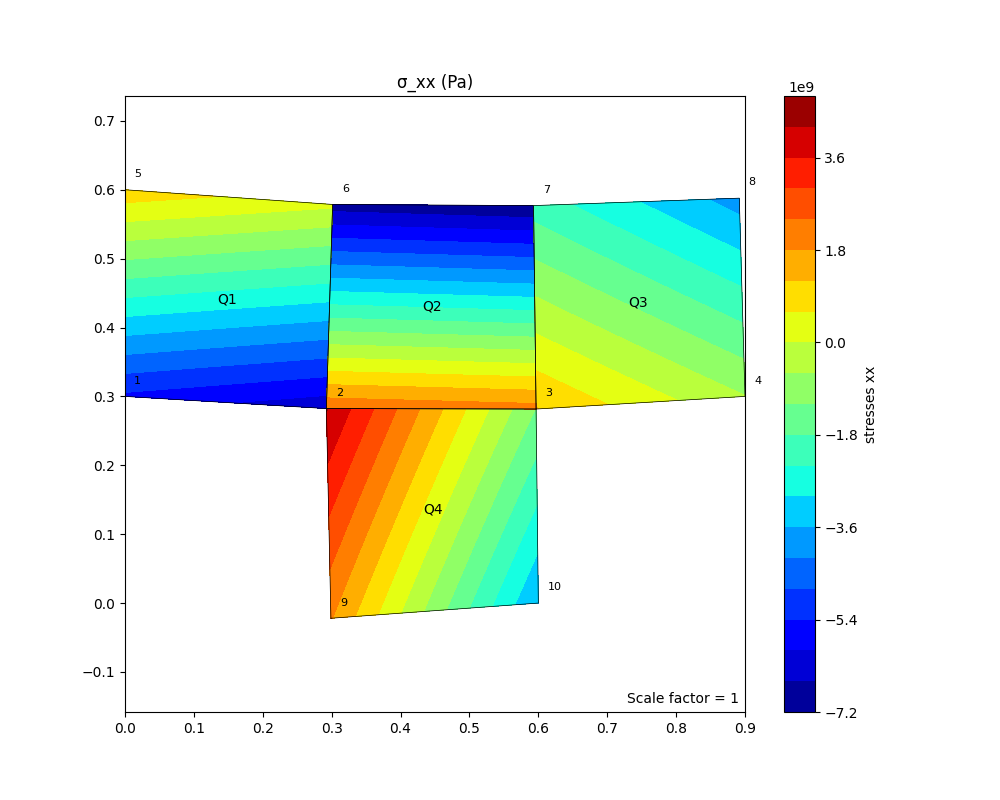
displacements=elem\_data['displacements'],

nodalStresses=nodalStresses,

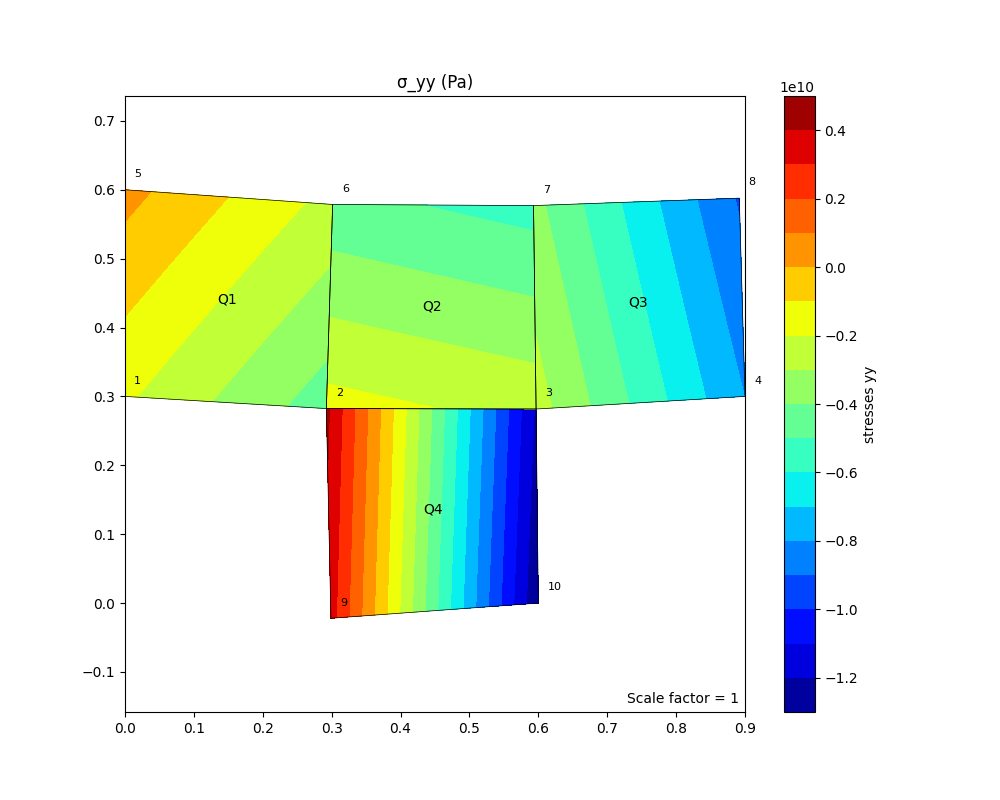
nodalStrains=nodalStrains

)

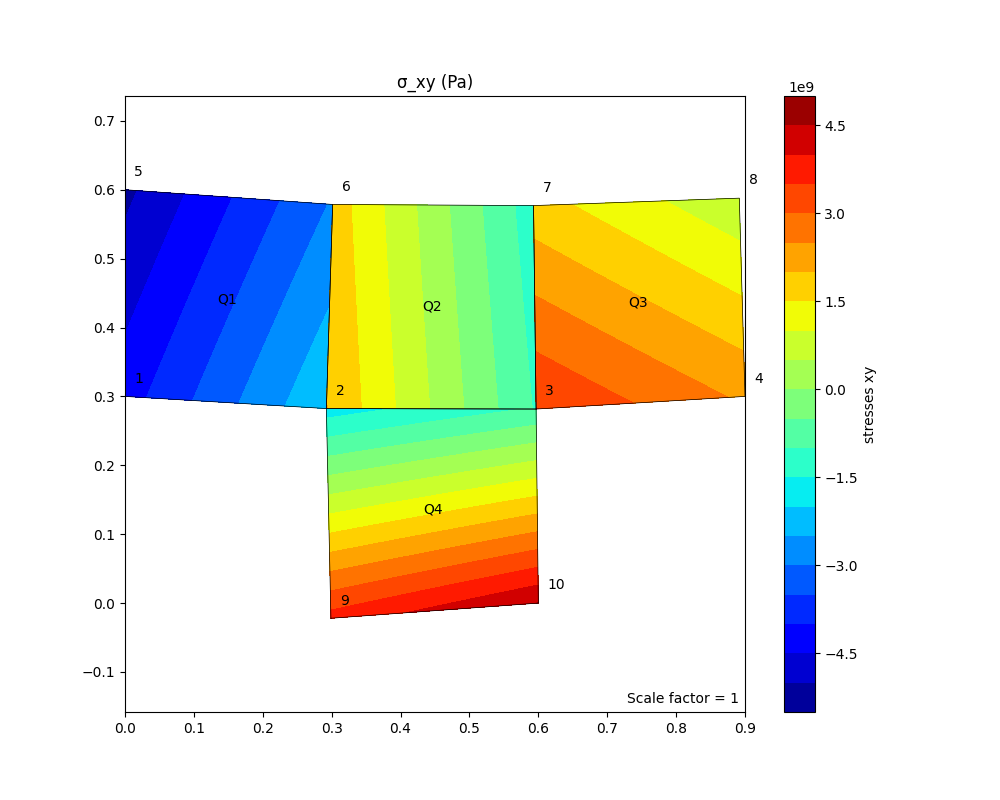
return nodal\_average\_data, element\_extrapolate



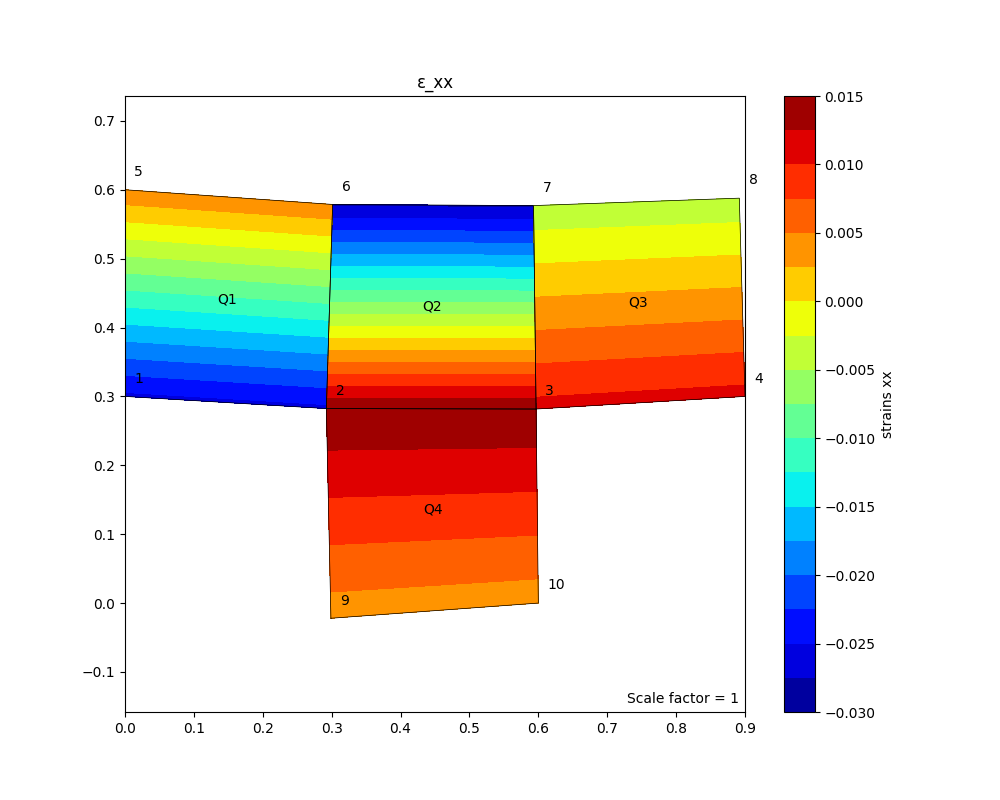
Hình 4‑2 Ứng suất phương XX



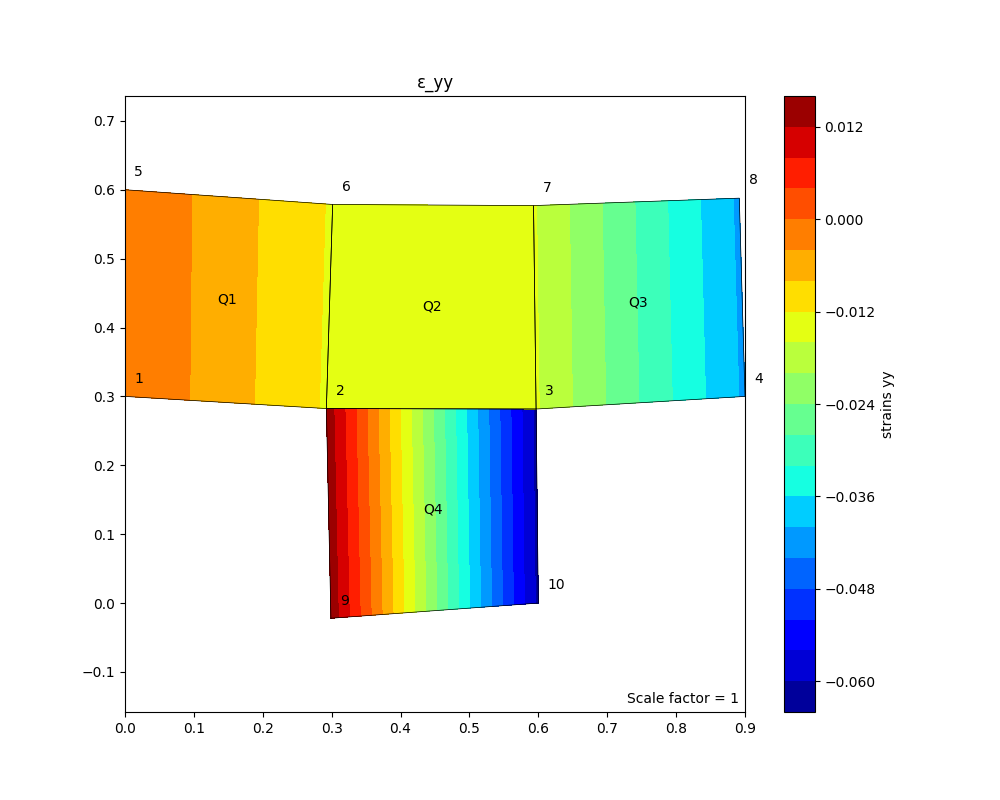
Hình 4‑3 Ứng suất chương YY



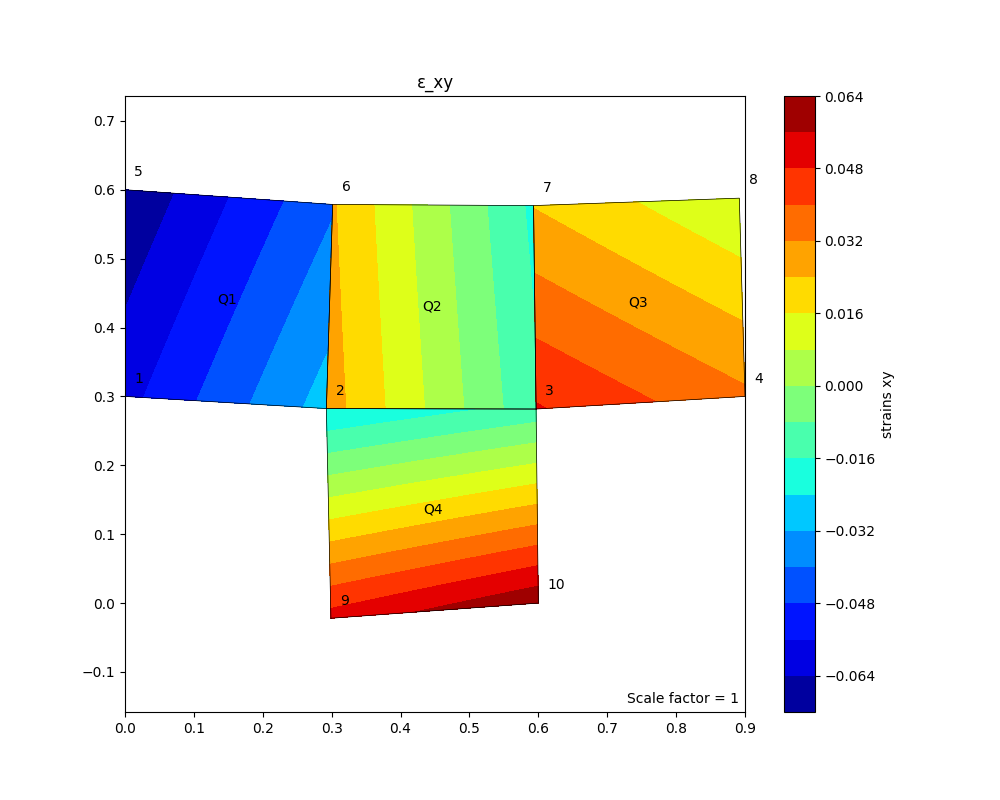
Hình 4‑4 Ứng suất phương XY



Hình 4‑5 Biến dạng phương XX



Hình 4‑6 Biến dạng phương YY



Hình 4‑7 Biến dạng phương XY

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()

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

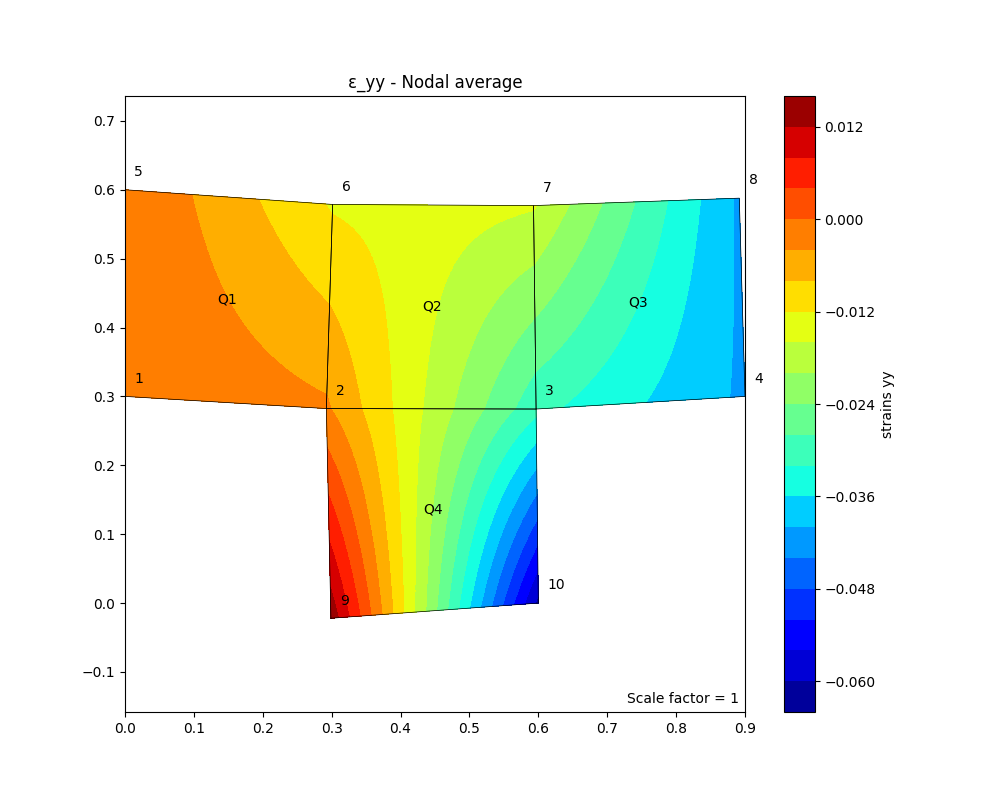
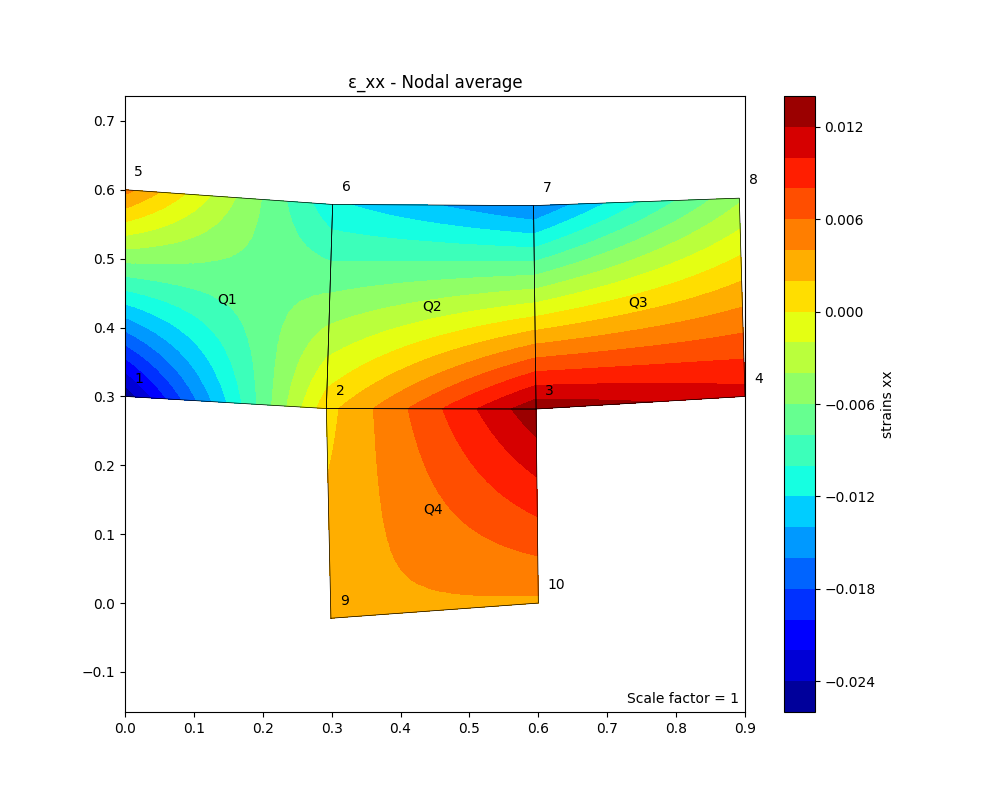
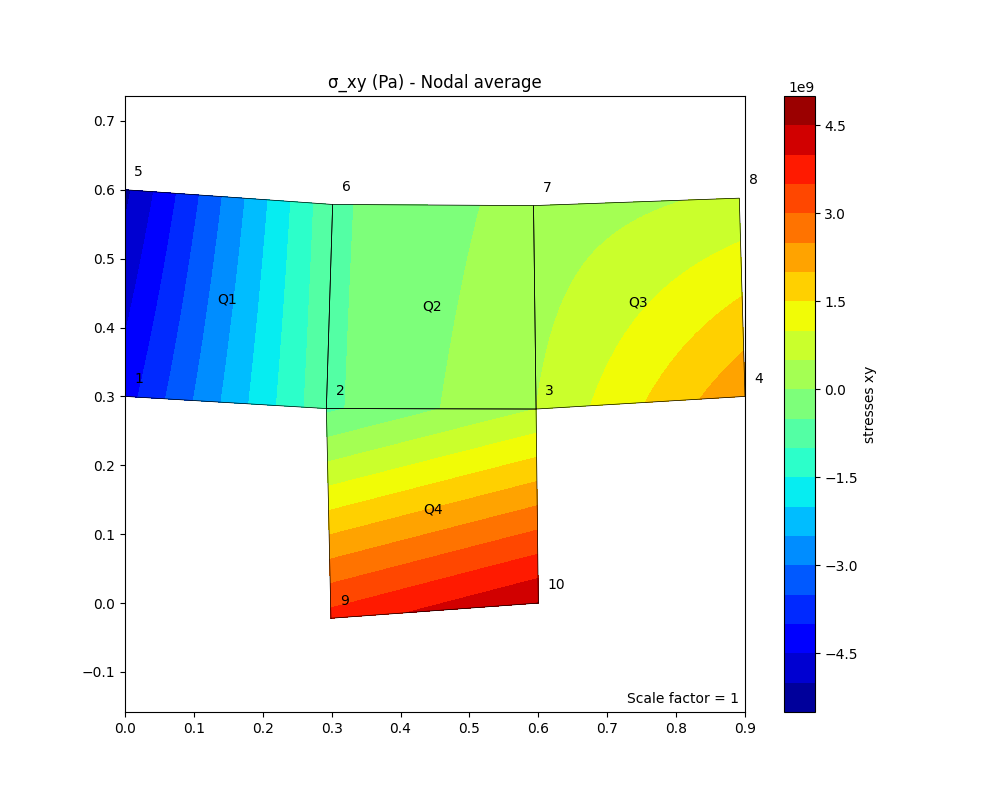
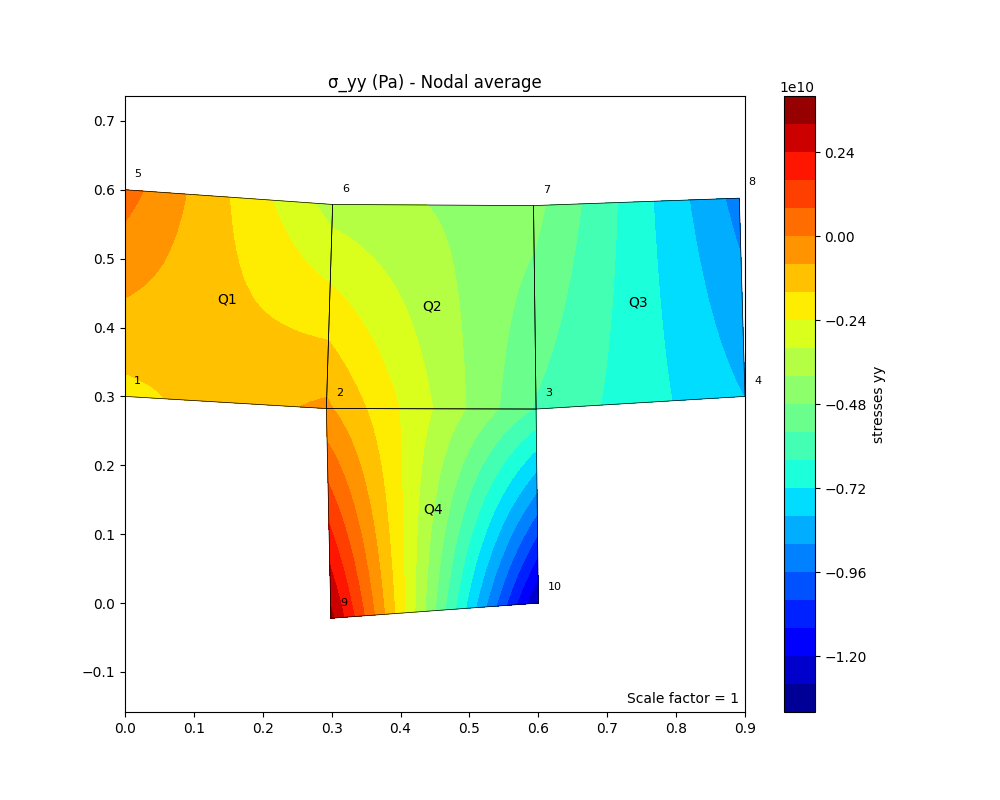
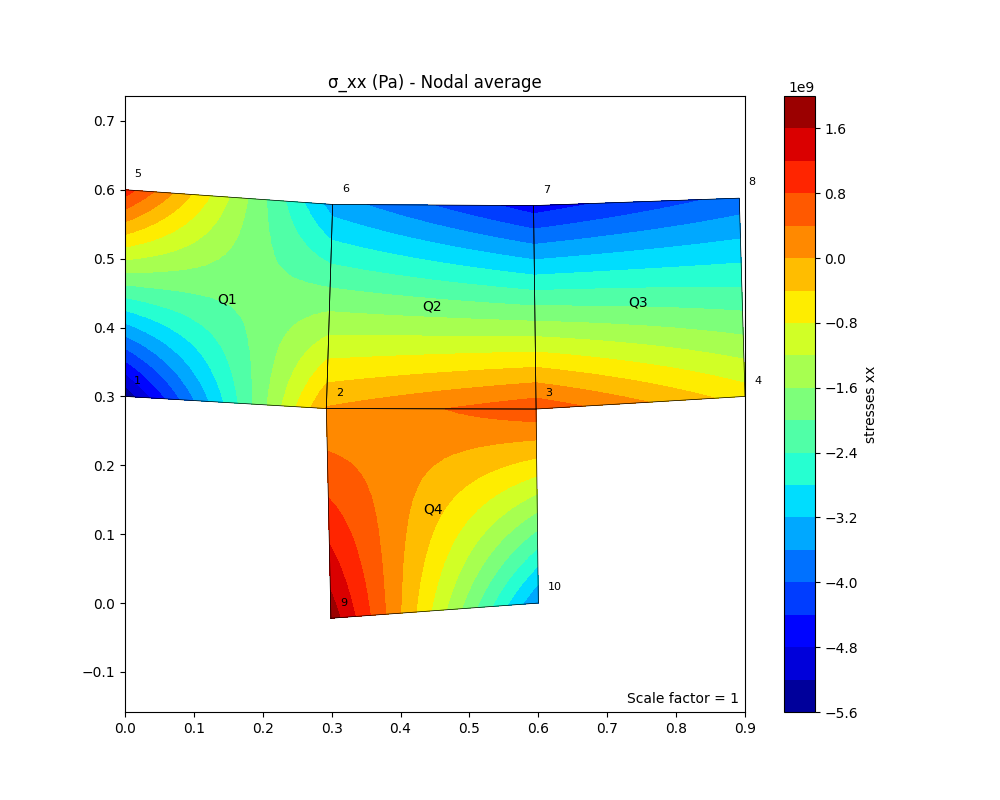
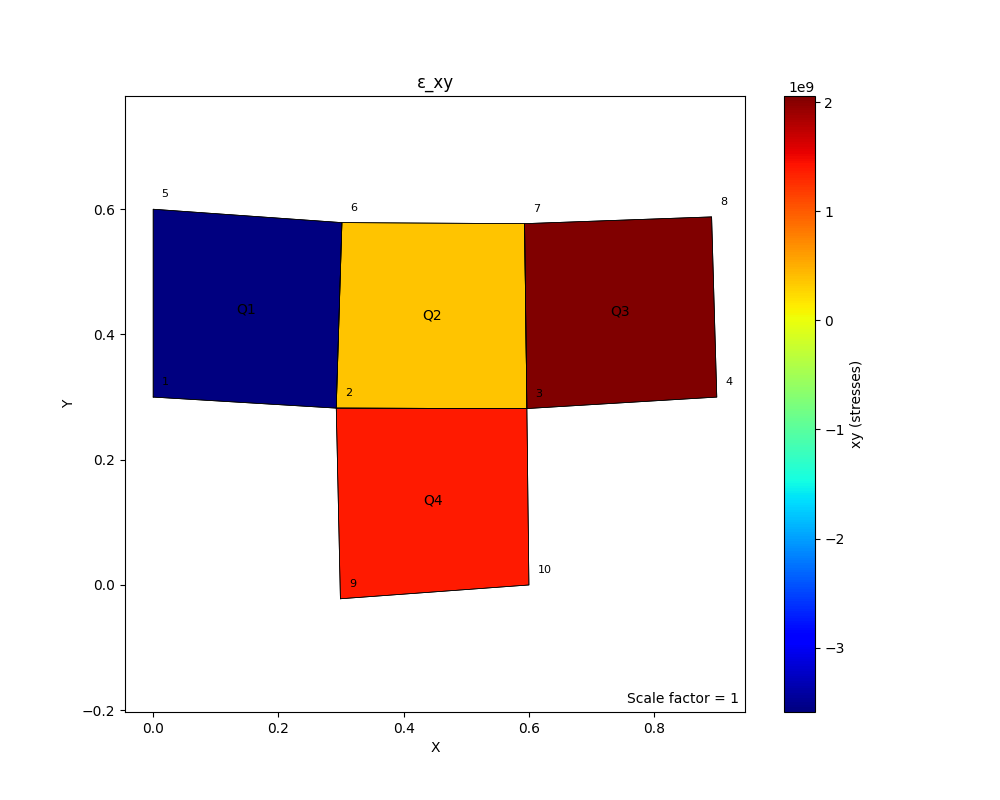
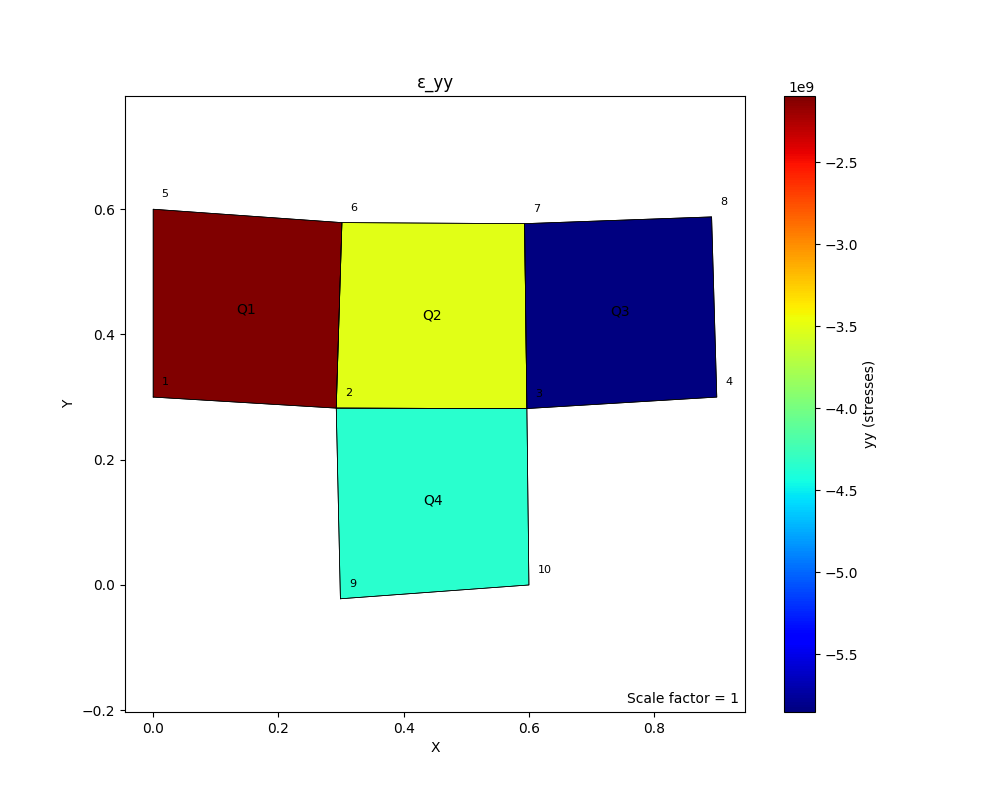
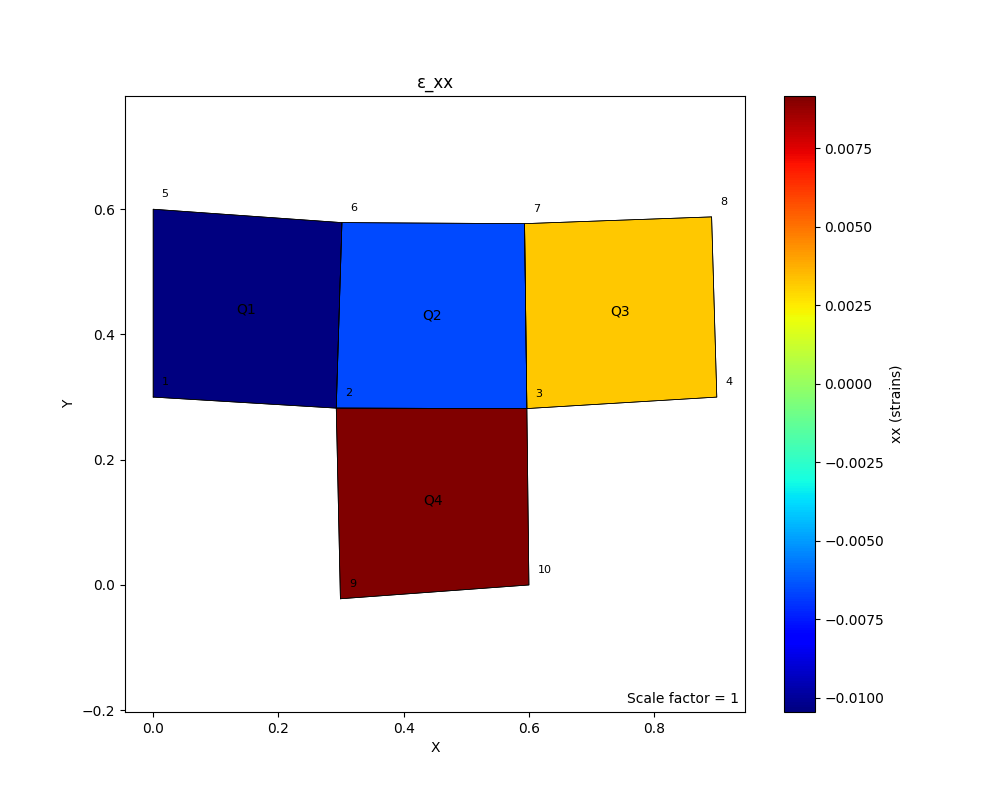
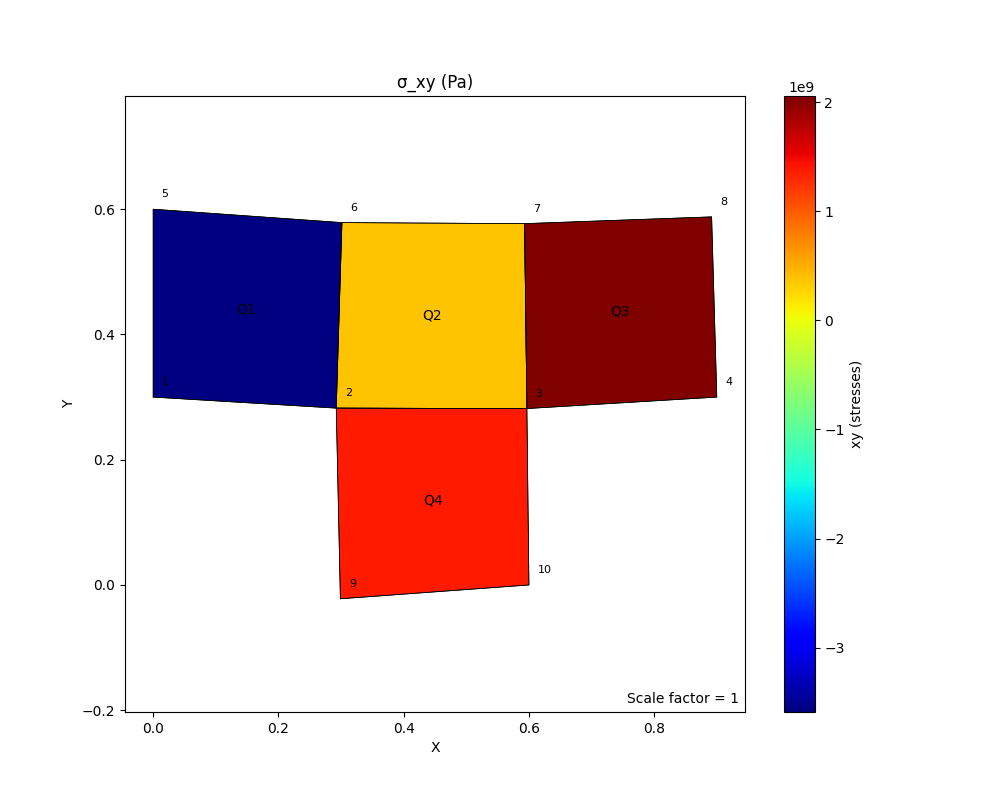
2 4 3.063242e+07 -6.408218e+08 -5.969361e+08 1.272082e-03 -3.390795e-03 -8.290780e-03

A diagram of a color scale

AI-generated content may be incorrect.

A diagram of a multicolored square

AI-generated content may be incorrect.



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. |