Solution to Exam (2025-08-25)

MHA021 Finite Element Method

Problem 1

1.1 Task 1a

Multiply the differential equation with an (arbitrary) test function $v(x)$ and integrate over the interval

$$-\int_0^L v\frac{d}{dx}\left[EA\frac{du}{dx}\right]\,\mathrm{d}x = \int_0^L v\,b\,\mathrm{d}x$$

IBP. of the left-hand side gives (after moving the boundary terms to the right-hand side)

$$\int_0^L EA\,\frac{dv}{dx}\,\frac{du}{dx}\,\mathrm{d}x = \int_0^L v\,b\,\mathrm{d}x + v(L)\left[EA\,\frac{du}{dx}\right]_{x=L} - v(0)\,\left[EA\,\frac{du}{dx}\right]_{x=0}$$

At $x = L$, we can substitute the the boundary term given in the strong form $(EA\,\frac{\mathrm{d}u}{\mathrm{d}x}(0) = 0) \Rightarrow$ the weak form:

Find u such that

$$\int_0^L EA\,\frac{dv}{dx}\,\frac{du}{dx}\,\mathrm{d}x = \int_0^L v\,b\,\mathrm{d}x - v(0)\,\underbrace{\left[EA\,\frac{du}{dx}\right]_{x=0}}_{\text{reaction force}=R} = \int_0^L v\,b\,\mathrm{d}x - v(0)\,R$$

where $u(0) = 0$.

1.2 Task 1b

Approximate u using a linear combination of shape functions: $u \approx u_h = \sum_i N_i(x)a_i = \mathbf{N}\,\mathbf{a}$, where $N = [N_1(x)\ N_2(x)\dots N_n(x)]$ is a row vector with the shape functions and $a = \left[a_1\ a_2\ \dots\ a_n\right]^{\mathrm{T}}$ are the degrees of freedom. Use the same shape functions for the test function $v(x) = \sum_i N_i(x)c_i = \mathbf{N}\,\mathbf{c}$. Inserting into the weak form while using the notation $\frac{\mathrm{d}}{\mathrm{d}x}\mathbf{N}(x) = \mathbf{B}$ gives

$$\mathbf{c}^{\mathrm{T}}\int_0^L EA\,\mathbf{B}^{\mathrm{T}}\,\mathbf{B}\,\mathrm{d}x\,\mathbf{a} = \mathbf{c}^{\mathrm{T}}\int_0^L \mathbf{N}^{\mathrm{T}}\,b\,\mathrm{d}x - \mathbf{c}^{\mathrm{T}}\,\mathbf{N}^{\mathrm{T}}(L)R$$

which should hold for arbitrary $\mathbf{c} \Rightarrow$...

$$\int_0^L EA\,\mathbf{B}^{\mathrm{T}}\,\mathbf{B}\,\mathrm{d}x\,\mathbf{a} = \int_0^L \mathbf{N}^{\mathrm{T}}\,b\,\mathrm{d}x - \mathbf{N}^{\mathrm{T}}(0)R$$

or simply $\mathbf{K}\,\mathbf{a} = \mathbf{f}_l + \mathbf{f}_{\mathrm{b}}$ with

$$\mathbf{K} = \int_0^L EA\,\mathbf{B}^{\mathrm{T}}\,\mathbf{B}\,\mathrm{d}x, \quad \mathbf{f}_{\mathrm{l}} = \int_0^L \mathbf{N}^{\mathrm{T}}\,b\,\mathrm{d}x, \quad \mathbf{f}_{\mathrm{b}} = -\mathbf{N}^{\mathrm{T}}(0)R$$

1.3 Task 1c

Assembly of $\mathbf{K}$ and $\mathbf{f}_1$ (using the mid-point of each element to evaluate b) gives the system of equations

$$\begin{bmatrix} 12000000. & -12000000. & 0. & 0. \\ -12000000. & 24000000. & -12000000. & 0. \\ 0. & -12000000. & 24000000. & -12000000. \\ 0. & 0. & -12000000. & 12000000. \end{bmatrix} \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 277.8 \\ 1111.1 \\ 2222.2 \\ 1388.9 \end{bmatrix} + \begin{bmatrix} -R \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

1.4 Task 1d

Displacement at the free end corresponds to the last element in $\mathbf{a}$ which gives (after solving the system of equations above) $u_h \approx 0.00081$ m which gives an error $e_u = \frac{u(L)-u_h}{u} \approx -2.8\%$.

Using linear shape functions: $N_1^e = -\frac{1}{L}(x - x_{i+1})$ and $N_2^e = \frac{1}{L}(x - x_i)$ gives

$$\mathbf{N}^e = [N_1^e, N_2^e] \quad \Rightarrow \mathbf{B}^e = \frac{\mathrm{d}}{\mathrm{d}x}\mathbf{N}^e = \frac{1}{L^e}[-1, 1]$$

The normal force is computed as $N_h^e = EA\,\mathbf{B}^e\,\mathbf{a}^e$ which gives for element 1:

$$N_h^1 = EA\,\mathbf{B}^1\,\mathbf{a}^1 = EA\frac{1}{L^e}[-1, 1]\begin{bmatrix} 0 \\ a_2 \end{bmatrix} \approx 4722\,\mathrm{N}$$

which gives an error $e_N = \frac{N(0)-N_h^1}{N(0)} \approx -5.6\,\%$

1.5 Task 1e

The consistent element load vector becomes (see code for integration)

$$\mathbf{f}_1^e = \int_{x_1}^{x_2} (\mathbf{N}^e)^{\mathrm{T}} b(x)\,\mathrm{d}x = \ldots = \begin{bmatrix} -3333x_1^2 + 1667x_1x_2 + 1667x_2^2 \\ -1667x_1^2 - 1667x_1x_2 + 3333x_2^2 \end{bmatrix}$$

Assembling using this load vector gives

$$\begin{bmatrix} 12000000. & -12000000. & 0. & 0. \\ -12000000. & 24000000. & -12000000. & 0. \\ 0. & -12000000. & 24000000. & -12000000. \\ 0. & 0. & -12000000. & 12000000. \end{bmatrix} \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 185.2 \\ 1111.1 \\ 2222.2 \\ 1481.5 \end{bmatrix} + \begin{bmatrix} -R \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

which gives the following errors instead: $e_u = 0\,\%$, $e_N \approx -3.7\,\%$

```python
import numpy as np
import calfem.core as cfc

# c)
EA = 200e9*2e-5
P = 10e3
L = 1
b0 = P/L
K = np.zeros((4, 4))
f = np.zeros((4, 1))

def b(x):
    return b0*x/L

Le = L/3
Ke = EA/Le * np.array([[1, -1],[-1, 1]])
fl1 = b(L/6)*Le/2 * np.array([[1],[1]])
fl2 = b(3*L/6)*Le/2 * np.array([[1],[1]])
fl3 = b(5*L/6)*Le/2 * np.array([[1],[1]])
cfc.assem(np.array([1, 2]), K, Ke, f, fl1)
cfc.assem(np.array([2, 3]), K, Ke, f, fl2)
cfc.assem(np.array([3, 4]), K, Ke, f, fl3)

# Boundary conditions and solve the system of equations

bc_dofs = np.array([1]) # first dof prescribed
bc_vals = np.array([0]) # displacement is zero
a, r = cfc.solveq(K, f, bc_dofs, bc_vals)

# d)
# Postprocessing
u_FEM = a[-1, 0]
N_FEM = EA*(a[1,0] - a[0,0]) / Le


# e)
import sympy as sp
x, x1, x2 = sp.symbols("x x1 x2")

N = sp.Matrix([(x2-x)/(x2-x1), (x-x1)/(x2-x1)]) # shape functions
fle = sp.simplify(sp.integrate(N*b(x), (x, x1, x2)))
fle_ = sp.lambdify((x1, x2), fle, "numpy")

# c) Solve the problem again but with new load vector
```

```
K = np.zeros((4, 4))
f_new = np.zeros((4, 1))
fl1 = fle_(0, Le)
fl2 = fle_(Le, 2*Le)
fl3 = fle_(2*Le, 3*Le)
cfc.assem(np.array([1, 2]), K, Ke, f_new, fl1)
cfc.assem(np.array([2, 3]), K, Ke, f_new, fl2)
cfc.assem(np.array([3, 4]), K, Ke, f_new, fl3)

a_new, r_new = cfc.solveq(K, f_new, bc_dofs, bc_vals)
u_FEM_new = a_new[-1, 0]
N_FEM_new = EA*(a_new[1,0] - a_new[0,0]) / Le
```

Problem 2: Heat equation

2.1 Task 2a

Since nothing is given on the top and bottom boundaries, we assume that these are insulated, i.e. $q_n = 0$. We then set up the boundary conditions formally as

$$T(\mathbf{x}) = 0\,^\circ\mathrm{C}, \quad \mathbf{x} \text{ on } \Gamma_{\text{left}} \tag{2.1}$$

$$T(\mathbf{x}) = 0\,^\circ\mathrm{C}, \quad \mathbf{x} \text{ on } \Gamma_{\text{right}} \tag{2.2}$$

$$\mathbf{q}^{\mathrm{T}}(\mathbf{x})\mathbf{n} = 0\,\mathrm{W/m^2}, \quad \mathbf{x} \text{ on } \Gamma_{\text{bottom}} \tag{2.3}$$

$$\mathbf{q}^{\mathrm{T}}(\mathbf{x})\mathbf{n} = 0\,\mathrm{W/m^2}, \quad \mathbf{x} \text{ on } \Gamma_{\text{top}} \tag{2.4}$$

$$\mathbf{q}^{\mathrm{T}}(\mathbf{x})\mathbf{n} = 2\,\mathrm{W/m^2}, \quad \mathbf{x} \text{ on } \Gamma_{\text{inner}} \tag{2.5}$$

2.2 Task 2b

We multiply the Partial Differential Equation (PDE) by an arbitrary scalar test function, $v(\mathbf{x})$, and integrate over the domain, Ω, resulting in

$$\int_{\Omega} v \ \nabla^{\mathrm{T}}\mathbf{q} \ \mathrm{d}\Omega = 0 \tag{2.6}$$

Applying Green-Gauss theorem to the left hand side, results in

$$\int_{\Gamma} v \ \mathbf{n}^{\mathrm{T}}\mathbf{q} \ \mathrm{d}\Gamma - \int_{\Omega} [\nabla v]^{\mathrm{T}} \ \mathbf{q} \ \mathrm{d}\Omega = 0 \tag{2.7}$$

Finally, we split the boundary terms into the Dirichlet ($\Gamma_g = \Gamma_{\text{left}} \cup \Gamma_{\text{right}}$) and Neumann ($\Gamma_h = \Gamma_{\text{top}} \cup \Gamma_{\text{bottom}}$ and Γ_{inner}) parts, insert the constitutive law, $\mathbf{q} = -k\nabla T$, and re-arrange to get

$$\int_{\Omega} [\nabla v]^{\mathrm{T}} \, [k \ \nabla T] \ \mathrm{d}\Omega = -\int_{\Gamma_h} v \ q_{\mathrm{n}} \ \mathrm{d}\Gamma - \int_{\Gamma_{\text{inner}}} v \ q_{\mathrm{n}} \ \mathrm{d}\Gamma - \int_{\Gamma_g} v \ \mathbf{n}^{\mathrm{T}}\mathbf{q} \ \mathrm{d}\Gamma \tag{2.8}$$

Still subject to the constraint, $T(\mathbf{x}) = 0\,^\circ\mathrm{C}$ for x on Γ_{g}. Inserting that $q_{\mathrm{n}} = 0$ on Γ_h, we get

$$\int_{\Omega} [\nabla v]^{\mathrm{T}} \, [k \ \nabla T] \ \mathrm{d}\Omega = -\int_{\Gamma_{\text{inner}}} v \ q_{\mathrm{n}} \ \mathrm{d}\Gamma - \int_{\Gamma_g} v \ \mathbf{n}^{\mathrm{T}}\mathbf{q} \ \mathrm{d}\Gamma \tag{2.9}$$

2.3 Task 2c

To get the FE form, we introduce the approximation of the test function, $v(\mathbf{x}) \approx v_h(\mathbf{x}) = \sum_{i=1}^{N_d} N_i(\mathbf{x})c_i = \mathbf{N} \ \mathbf{c}$, where c_i are arbitrary weights, that are coordinate independent. This results in

$$\sum_{i=1}^{N_d} c_i \underbrace{\left[\int\limits_\Omega [\nabla N_i(\mathbf{x})]^{\mathrm{T}} [k\ \nabla T]\ \mathrm{d}\Omega - \left[-\int\limits_{\Gamma_{\text{inner}}} N_i(\mathbf{x})\ q_{\mathrm{n}}\ \mathrm{d}\Gamma - \int\limits_{\Gamma_g} N_i(\mathbf{x})\ \mathbf{n}^{\mathrm{T}}\mathbf{q}\ \mathrm{d}\Gamma\right]\right]}_{r_i} = 0 \tag{2.10}$$

Since c_i are arbitrary coefficients, and this has to hold for any c_i, this implies that $r_i = 0$ has to hold. We can see this by considering the case that $c_\alpha = 1$, and $c_i = 0$ if $i \neq 0$. Then $r_\alpha = 0$ follows, and this has to hold for any $1 \leq \alpha \leq N_d$. We thus obtain the FE form,

$$\int\limits_\Omega [\nabla N_i(\mathbf{x})]^{\mathrm{T}} [k\ \nabla T]\ \mathrm{d}\Omega = -\int\limits_{\Gamma_{\text{inner}}} N_i(\mathbf{x})\ q_{\mathrm{n}}\ \mathrm{d}\Gamma - \int\limits_{\Gamma_g} N_i(\mathbf{x})\ \mathbf{n}^{\mathrm{T}}\mathbf{q}\ \mathrm{d}\Gamma \tag{2.11}$$

and can finally insert the FE-approximation for the temperature, $T(\mathbf{x}) \approx T_h(\mathbf{x}) = \sum_{i=1}^{N_d} N_i(\mathbf{x}) a_i = \mathbf{N}\mathbf{a}$, to obtain,

$$\underbrace{\int\limits_\Omega [\nabla N_i(\mathbf{x})]^{\mathrm{T}} [k\ \nabla N_j]\ \mathrm{d}\Omega}_{K_{ij}}\ a_j = \underbrace{-\int\limits_{\Gamma_{\text{inner}}} N_i(\mathbf{x})\ q_{\mathrm{n}}\ \mathrm{d}\Gamma - \int\limits_{\Gamma_g} N_i(\mathbf{x})\ \mathbf{n}^{\mathrm{T}}\mathbf{q}\ \mathrm{d}\Gamma}_{f_i} \tag{2.12}$$

$$T_h(\mathbf{x}) = 0\,^\circ\mathrm{C} \text{ on } \Gamma_{\text{left}} \tag{2.13}$$

$$T_h(\mathbf{x}) = 0\,^\circ\mathrm{C} \text{ on } \Gamma_{\text{right}} \tag{2.14}$$

where the reaction flux, $q_n(\mathbf{x})$ is unknown on Γ_g and $q_n(\mathbf{x}) = 2\,\mathrm{W/m^2}$ on Γ_{inner}.

2.4 Task 2d & 2e

```python
# Problem 2

# General packages
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import matplotlib.tri as tri
# CALFEM packages
import calfem.core as cfc
import calfem.vis_mpl as cfv

# Load mesh data
mesh = scipy.io.loadmat('mesh_task2.mat')

Coord = mesh['Coord']                          # [x, y] coords for each node
Dofs = mesh['Dofs']
```

```python
Edof = mesh['Edof']                             # [element number, dof1, dof2, dof3]
Ex = mesh['Ex']
Ey = mesh['Ey']
right_dofs = mesh['right_dofs']
left_dofs = mesh['left_dofs']
inner_edgedofs = mesh['inner_edgedofs']
inner_ex = mesh['inner_ex']
inner_ey = mesh['inner_ey']

# Plot the mesh
plotpar = np.array([1, 1, 2]) # parameters for line style, color, marker
# cfv.eldraw2(Ex, Ey)
# plt.xlabel("x [m]")
# plt.ylabel("y [m]")
# plt.show()

nel=np.shape(Edof)[0]
ndofs=np.max(Edof[:,1:])
dimension = 2

T1 = 0 # Prescribed temperature on right boundary [C]
T2 = 0 # Prescribed temperature on left boundary [C]
t = 0.01 # thickness [m]
qn = -2.0 # [W/m^2]
k = 1.0 # Heat conductivity [W/mC]

D = np.eye(dimension) * k
K = np.zeros([ndofs,ndofs])
f = np.zeros([ndofs, 1])

for el in range(nel):
    Ke = cfc.flw2te(Ex[el,:], Ey[el,:], [t], D)
    K = cfc.assem(Edof[el,1:], K, Ke)

# Assemble Neumann boundary conditions
num_edges = inner_edgedofs.shape[0]
for edge in range(num_edges):
    ex = inner_ex[edge, :]
    ey = inner_ey[edge, :]
    edof = inner_edgedofs[edge, :]-1
    Le = np.sqrt( (ex[1]-ex[0])**2 + (ey[1] - ey[0])**2 )
    fe = -(t * Le * qn / 2) * np.array([[1], [1]])
    f[edof] += fe
bcpresc=np.vstack([right_dofs,left_dofs]).flatten()
```

```
bcVal1=np.ones_like(right_dofs)*T2
bcVal2=np.ones_like(left_dofs)*T1
bcVal=np.vstack([bcVal1,bcVal2]).flatten()
a, q = cfc.solveq(K,f,bcpresc,bcVal)

print(a)
np.linalg.norm(a)

# Heat flux vector in element 10, 20, 30

ed = cfc.extract_ed(Edof[:, 1:], a)
el = 10
es10, et10 = cfc.flw2ts(Ex[el-1, :], Ey[el-1, :], D, ed[el-1, :])
el = 20
es20, et20 = cfc.flw2ts(Ex[el-1, :], Ey[el-1, :], D, ed[el-1, :])
el = 30
es30, et30 = cfc.flw2ts(Ex[el-1, :], Ey[el-1, :], D, ed[el-1, :])
display(es10, es20, es30) # heat flux vectors
```

Problem 3

See matlab file at the bottom for full calculations.

3.1 Task 3a

The local coordinate of point A is, $\xi_A = [0, 1]^T$, which gives with the shape functions in the formula sheet for a 4-noded quadrilateral,

$$\mathbf{x}_A = \sum_{\alpha=1}^{4} \mathbf{x}_\alpha \hat{N}_\alpha(\xi_A) = \begin{bmatrix} 0.0150 \\ 0.0170 \end{bmatrix} \text{m} \tag{3.1}$$

3.2 Task 3b

Given the displacement vector, we transform this into the dof vector as $\mathbf{a}^e = [u^e_{x,1}, u^e_{y,1}, u^e_{x,2}, \cdots]^T$, and then we have the vectorized shape functions,

$$\mathbf{N}^e = \begin{bmatrix} M_1 & 0 & M_2 & \cdots \\ 0 & M_1 & 0 & \cdots \end{bmatrix} \tag{3.2}$$

and the displacement at the point $\mathbf{x}_A$ is then given as

$$\mathbf{u}_A = \mathbf{N}^e \mathbf{a}^e = \begin{bmatrix} 0.0375 \\ 0.0160 \end{bmatrix} \text{mm} \tag{3.3}$$

To calculate the strains, we need the spatial gradients of the shape functions, i.e. the B-matrix. So we need the isoparametric mapping, and from the formula sheet we use

$$\mathbf{J} = \frac{\partial \mathbf{x}}{\partial \xi} = \sum_{\alpha=1}^{N_{\text{nodes}}} \mathbf{x}^e_\alpha \left[\frac{\partial \hat{N}_\alpha}{\partial \xi} \right]^{\mathrm{T}}, \quad \frac{\partial N^e_i}{\partial \mathbf{x}} = \mathbf{J}^{-T} \frac{\partial \hat{N}^e_i}{\partial \xi} \tag{3.4}$$

$$\mathbf{B} = \begin{bmatrix} \frac{\partial N^e_1}{\partial x_1} & 0 & \frac{\partial N^e_2}{\partial x_1} & 0 & \cdots \\ 0 & \frac{\partial N^e_1}{\partial x_2} & 0 & \frac{\partial N^e_2}{\partial x_2} & \cdots \\ \frac{\partial N^e_1}{\partial x_2} & \frac{\partial N^e_1}{\partial x_1} & \frac{\partial N^e_2}{\partial x_2} & \frac{\partial N^e_2}{\partial x_1} & \cdots \end{bmatrix} \tag{3.5}$$

The strain is then given as

$$\epsilon = \mathbf{B}^e \mathbf{a}^e = \begin{bmatrix} 0.017 \\ 0.046 \\ 0.017 \end{bmatrix} \% \tag{3.6}$$

and the stress as

$$\sigma = \mathbf{D}\,\epsilon = \begin{bmatrix} 71.1 \\ 117.9 \\ 13.7 \end{bmatrix}\ \text{MPa} \tag{3.7}$$

3.3 Task 3c

Calculating the integral implies that we have

$$\int_{\Omega^e} \mathrm{d}\Omega \approx \sum_{\alpha=1}^{4} w_\alpha \det(\mathbf{J}(\xi_\alpha)) = 50\,\text{mm}^2 \tag{3.8}$$

3.4 Python code

```python
# Solution Problem 3, Re-Exam MHA021, 2025-08-25
# This code is a AI-translation from the Matlab solution written by Knut Andreas Meyer
import numpy as np

def shape_values(xi):
    """
    xi: array-like of length 2 (xi1, xi2)
    returns N as a 1D array of length 4
    """
    xi = np.asarray(xi).reshape(2,)
    N = np.array([
        (1 - xi[0]) * (1 - xi[1]),
        (1 + xi[0]) * (1 - xi[1]),
        (1 + xi[0]) * (1 + xi[1]),
        (1 - xi[0]) * (1 + xi[1]),
    ]) / 4.0
    return N


def shape_gradients(xi):
    """
    xi: array-like of length 2
    returns dN/dxi as a (2 x 4) array
    Row 0 = dN/dxi1, Row 1 = dN/dxi2
    """
    xi = np.asarray(xi).reshape(2,)
    dNdxi = np.array([
        [-(1 - xi[1]), -(1 - xi[0])],   # node 1
        [+(1 - xi[1]), -(1 + xi[0])],   # node 2
        [+(1 + xi[1]), +(1 + xi[0])],   # node 3
        [-(1 + xi[1]), +(1 - xi[0])],   # node 4
    ]).T / 4.0  # transpose to get (2 x 4)
    return dNdxi


def element_jacobian(xi, x):
    """
    xi: array-like of length 2
    x: (2 x 4) nodal coordinates array (each column is a node: [x; y])
    returns J as a (2 x 2) array
    """
    dNdxi = shape_gradients(xi)      # (2 x 4)
    J = x @ dNdxi.T                  # (2 x 4) @ (4 x 2) = (2 x 2)
    return J


E = 210e9
nu = 0.3
t = 0.025  # thickness

# Plane-stress constitutive matrix D
D = (E / (1 - nu**2)) * np.array([
    [1.0,  nu,   0.0],
    [nu,   1.0,  0.0],
    [0.0,  0.0, (1 - nu) / 2.0]
], dtype=float)

# Nodal coordinates (2 x 4): each column is a node [x; y]
x = np.array([
    [0.014, 0.021, 0.018, 0.012],
    [0.010, 0.009, 0.018, 0.016],
], dtype=float)

# Element dofs (8 x 1): [u1 v1 u2 v2 u3 v3 u4 v4]^T
ae = (1e-5) * np.array([4.1, 1.0, 3.5, 1.6, 3.8, 1.7, 3.7, 1.5], dtype=float)

xi_A = np.array([0.0, 1.0], dtype=float)

# Shape values and point in physical coordinates
Ns = shape_values(xi_A)      # (4,)
xA = x @ Ns                  # (2,) equivalent to (Ns * x')' in MATLAB
```

```
# Interpolation matrix Nv and displacement at A
Nv = np.array([
    [Ns[0], 0.0,   Ns[1], 0.0,   Ns[2], 0.0,   Ns[3], 0.0],
    [0.0,   Ns[0], 0.0,   Ns[1], 0.0,   Ns[2], 0.0,   Ns[3]],
], dtype=float)
uA = Nv @ ae                 # (2,)

# Gradients in parent coordinates
dNdxi = shape_gradients(xi_A)     # (2 x 4)

# Jacobian (direct)
J = x @ dNdxi.T                   # (2 x 2)

# Alternative Jacobian (constructed in a loop)
J1 = np.zeros((2, 2))
for i in range(x.shape[1]):       # i = 0..3
    J1 += np.outer(x[:, i], dNdxi[:, i])

# Map gradients to physical coordinates: dN/dx = inv(J') * dN/dxi
dNdx = np.linalg.inv(J.T) @ dNdxi  # (2 x 4)

# Strain-displacement matrix B (3 x 8)
B = np.zeros((3, 8))
B[0, 0::2] = dNdx[0, :]  # epsilon_xx terms
B[1, 1::2] = dNdx[1, :]  # epsilon_yy terms
B[2, 0::2] = dNdx[1, :]  # gamma_xy terms (shear)
B[2, 1::2] = dNdx[0, :]

# Strain and stress at xi_A
strain = B @ ae                 # (3,)
stress = D @ strain             # (3,)

# 2x2 Gauss quadrature for area integration
qr_points_1d = np.array([-1.0 / np.sqrt(3.0), 1.0 / np.sqrt(3.0)])
qr_weights_1d = np.array([1.0, 1.0])

# Build 2D rule
qr_points_2d = np.zeros((2, qr_points_1d.size ** 2))  # (2 x 4)
qr_weights_2d = np.zeros(qr_points_1d.size ** 2)      # (4,)
k = 0
for i in range(qr_points_1d.size):
    for j in range(qr_points_1d.size):
        qr_points_2d[:, k] = [qr_points_1d[i], qr_points_1d[j]]
        qr_weights_2d[k] = qr_weights_1d[i] * qr_weights_1d[j]
        k += 1

# Integrate area A = \int det(J) dxi over the parent domain
A = 0.0
for i in range(qr_weights_2d.size):
    xi = qr_points_2d[:, i]
    Ji = element_jacobian(xi, x)
    A += qr_weights_2d[i] * np.linalg.det(Ji)

# --- Output ---
np.set_printoptions(precision=6, suppress=True)
print("Ns =", Ns)
print("xA =", xA)
print("uA =", uA)
print("J =\n", J)
print("J1 (loop) =\n", J1)
print("dNdx =\n", dNdx)
print("B =\n", B)
print("strain =", strain)
print("stress =", stress)
print("A =", A)
```