Project Report

AE675

INTRODUCTION TO FINITE ELEMENT METHODS

Submitted by:

Atharv Soni

210229

Mayankit

210599

Department of Aerospace Engineering

Indian Institute of Technology, Kanpur

Beam Bending Problem

PROBLEM STATEMENT

Write a one dimensional finite element code using Hermite cubic shape functions with the following details for the beam bending problem.

1) Uniform cross section: 1 cm X 1 cm
2) Length of the beam: 10 cm
3) $E = 200$ GPa
4) The code should be capable of handling the transverse loads of the type
   a. Concentrated/point load
   b. Uniformly distributed load
   c. Point moments ***at the centre of the beam length only***
5) Further, it should be capable of applying the appropriate combination of boundary conditions at either of the ends as:
   a. Specified transverse displacement
   b. Specified slope of the transverse displacement
   c. Shear force
   d. Bending moment

Now, take appropriate values of loads as mentioned in Point # 4 above and perform the following finite element analysis using your code for 1, 4, 10, 50 and 100 elements.

1) Give continuous variation of transverse displacement and its slope
2) Give continuous variation of shear force and bending moment
3) Bending stress on the top most line of beam along its entire length.

Discuss your results and verify those using Euler Bernoulli beam theory closed form solutions.

SOLUTION

PYTHON CODE

```python
import numpy as np
import matplotlib.pyplot as plt

def shape_functions(h):
    """Compute coefficients of shape functions."""
    co=[[0.5, -0.75, 0, 0.25],[-0.125*h, 0.125*h, 0.125*h, -0.125*h],[0.5,
   0.75, 0, -0.25],[0.125*h, 0.125*h, -0.125*h, -0.125*h]]
    cod=[[-0.75, 0, 0.75], [0.125*h, 0.25*h, -0.375*h],[0.75, 0, -0.75],[0.125*
   h, -0.25*h, -0.375*h]]
    codd=[[0, 1.5], [0.25*h, -0.75*h], [0, -1.5], [-0.25*h, -0.75*h]]
```

```
    coddd=[[1.5],[-0.75*h],[-1.5],[-0.75*h]]
    return co, cod, codd, coddd

def integration_approximation(re):
    """Approximate integration coefficients."""
    coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
    coi=np.append(coi,
    [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
    s=0
    for i in range(0, 6):
        su = 0
        for j in range(0, int(re.shape[0])):
            su += re[j] * coi[i][0] ** j
        s += su * coi[i][1]
    return s

def elemental_matrix(f, co, h):
    """Compute elemental matrix."""
    fe = np.zeros((4, 1))
    for i in range(0, 4):
        fe[i][0] = (h / 2) * integration_approximation(np.polynomial.polynomial
    .polymul(co[i], f[0]))
    return fe

def get_value(co, x, p, d):
    """Map element position to global matrix position."""
    s = 0
    for i in range(0, p - d +1):
        s += co[i] * x ** i
    return s

def main():
    n = int(input("Enter number of elements: "))
    h = 1 / n
    co, cod, codd, coddd = shape_functions(h)
    bc1 = int(input("Type of BC at x=0 (1 or 2): "))
    bc2 = int(input("Type of BC at x=1 (1 or 2): "))
    if bc1 == 2:
```

```
        force1 = float(input("Enter force: "))
        moment1 = float(input("Enter moment: "))
    else:
        dis1 = float(input("Enter displacement: "))
        slope1 = float(input("Enter slope: "))
    if bc2 == 2:
        force2 = float(input("Enter force: "))
        moment2 = float(input("Enter moment: "))
    else:
        dis2 = float(input("Enter displacement: "))
        slope2 = float(input("Enter slope: "))
    fp = int(input("Enter order of t: "))
    fco = np.zeros((1, fp+1))
    for i in range(0, fp + 1):
        fco[0][i] = int(input("Enter coeff: "))
    if n % 2 == 0:
        forcemid = int(input("Enter force at midpoint: "))
        momentmid = int(input("Enter moment at midpoint: "))
    nodloc = np.zeros((n, 2))
    for i in range(0, n):
        for j in range(0, 2):
            if i == j == 0:
                continue
            elif j % 2 == 0:
                nodloc[i][j] = nodloc[i - 1][j + 1]
            else:
                nodloc[i][j] = nodloc[i][j - 1] + h
    K = np.zeros((2 * n + 2, 2 * n + 2))
    F = np.zeros((2 * n + 2, 1))
    Q = np.zeros((2 * n + 2, 1))
    ke = (1000 / (3 * h ** 3)) * np.array([[6, -3 * h, -6, -3 * h],
                                           [-3 * h, 2 * h ** 2, 3 * h, h ** 2],
                                           [-6, 3 * h, 6, 3 * h],
                                           [-3 * h, h ** 2, 3 * h, 2 * h ** 2]])
    re = np.zeros((1, 5))
    for i in range(0, n):
        fe = np.zeros((4, 1))
```

```
        sunod = (nodloc[i][0] + nodloc[i][1]) / 2
        fcof = np.array([[fco[0][0] + fco[0][1] * sunod + fco[0][2] * sunod * 2, fco[0][1] * (h / 2) + fco[0][2] * h * sunod,
                          fco[0][2] * (h / 4)]])
        fe = elemental_matrix(fcof, co, h)
        if i == 0:
            K[:4, :4] += ke
            F[:4, 0:1] += fe
        else:
            K[2 * i:2 * (i + 2), 2 * i:2 * (i + 2)] += ke
            F[2 * i:2 * (i + 2), 0:1] += fe
    KF = K
    if bc1 == 1:
        for i in range(1, 2 * n + 2):
            F[i] = F[i] - dis1 * KF[i][0]
        for i in range(0, 2 * n + 2):
            for j in range(0, n * 2 + 2):
                if i == 0 or j == 0:
                    KF[i][j] = 0
        KF[0][0] = 1
        F[0][0] = dis1
        Q[0][0] = 0
        for i in range(2, 2 * n + 2):
            F[i] = F[i] - slope1 * KF[i][1]
        for i in range(0, 2 * n + 2):
            for j in range(0, 2 * n + 2):
                if i == 1 or j == 1:
                    KF[i][j] = 0
        KF[1][1] = 1
        F[1][0] = slope1
        Q[1][0] = 0
    if bc2 == 1:
        for i in range(0, 2 * n + 1):
            F[i] = F[i] - slope2 * KF[i][2 * n + 1]
        for i in range(0, n * 2 + 2):
            for j in range(0, 2 * n + 2):
                if i == 2 * n + 1 or j == n * 2 + 1:
                    KF[i][j] = 0
```

```
        KF[n * 2 + 1][n * 2 + 1] = 1
        F[n * 2 + 1][0] = slope2
        Q[n * 2 + 1][0] = 0
        for i in range(0, 2 * n):
            F[i] = F[i] - dis2 * KF[i][2 * n]
        for i in range(0, n * 2 + 2):
            for j in range(0, 2 * n + 2):
                if i == 2 * n or j == n * 2:
                    KF[i][j] = 0
        KF[n * 2][n * 2] = 1
        F[n * 2][0] = dis2
        Q[n * 2][0] = 0
    if bc1 == 2:
        Q[0][0] = force1
        Q[1][0] = -moment1
    if bc2 == 2:
        Q[2 * n][0] = force2
        Q[2 * n + 1][0] = -moment2
    if n % 2 == 0:
        Q[n][0] = forcemid
        Q[n + 1][0] = -momentmid
    U = np.linalg.inv(KF) @ (F + Q)

    # Plotting
    x = np.linspace(0, 1, 1000)
    yh, yhslope, moment, shear, stress = [], [], [], [], []

    for i in range(n):
        for j in range(1000):
            if nodloc[i][0] <= x[j] <= nodloc[i][1]:  # Check if x[j] is within the element range
                aux = (2 * x[j] - (nodloc[i][0] + nodloc[i][1])) / h
                w, wslope, mom, sh, st, b = 0, 0, 0, 0, 0, 0
                for m in range(i * 2, i * 2 + 4):
                    w=w+(U[m][0]*get_value(co[b],aux,3,0))
                    wslope=wslope+(U[m][0]*get_value(cod[b],aux,3,1))*(2/h)
                    mom=mom+(500/3)*(U[m][0]*get_value(codd[b],aux,3,2))*(2/h)**2
```

```
                    sh=sh+(-500/3)*(U[m][0]*get_value(coddd[b],aux,3,3))*(2/h)**3
                    st=st+(-1000)*(U[m][0]*get_value(codd[b],aux,3,2))*(2/h)**2
                    b=b+1
                yh.append(w)
                yhslope.append(wslope)
                moment.append(mom)
                shear.append(sh)
                stress.append(st)

    # Ensure yh and other lists have the same length as x
    if len(yh) < 1000:
        yh.extend([yh[-1]] * (1000 - len(yh)))
        yhslope.extend([yhslope[-1]] * (1000 - len(yhslope)))
        moment.extend([moment[-1]] * (1000 - len(moment)))
        shear.extend([shear[-1]] * (1000 - len(shear)))
        stress.extend([stress[-1]] * (1000 - len(stress)))
    elif len(yh) > 1000:
        yh = yh[:1000]
        yhslope = yhslope[:1000]
        moment = moment[:1000]
        shear = shear[:1000]
        stress = stress[:1000]

    ye, yed = [], []
    i = 0
    while i <= 1:
        ye.append((3 / 500) * ((i ** 4) / 24 - (i ** 3) / 3 + 5 * i * i / 4))
        yed.append(((i ** 3) / 6 - i * i + 5 * i / 2) * (3 / 500))
        i += 0.001
    er = 0
    for i in range(0, 1000):
        er += ((ye[i] - yh[i]) ** 2)
    e = np.sqrt(er / 1000) * 100
    print("RMSE error%:", e, "%")

    # Plotting results
    line1, = plt.plot(x, yh, label="FEM")
```

```
    line2, = plt.plot(x, ye, label="Exact")
    plt.legend(handles=[line1, line2])
    plt.title("Plot of deflection with x")
    plt.ylabel("Deflection")
    plt.xlabel("x")
    plt.show()

    line1, = plt.plot(x, yhslope, label="FEM")
    line2, = plt.plot(x, yed, label="Exact")
    plt.legend(handles=[line1, line2])
    plt.title("Plot of slope with x")
    plt.ylabel("Slope")
    plt.xlabel("x")
    plt.show()

    plt.plot(x, moment)
    plt.title("Plot of moment with x")
    plt.ylabel("Moment")
    plt.xlabel("x")
    plt.show()

    plt.plot(x, shear)
    plt.title("Plot of shear force with x")
    plt.ylabel("Shear Force")
    plt.xlabel("x")
    plt.show()

    plt.plot(x, stress)
    plt.title("Plot of stress with x")
    plt.ylabel("Stress")
    plt.xlabel("x")
    plt.show()

if __name__ == "__main__":
    main()
```

PYTHON CODE WITH SAMPLE INPUT

At the start of the beam:

- Displacement: 0
- Slope: 0

At the end of the beam:

- Force: 5
- Moment: 5

```
import numpy as np
import matplotlib.pyplot as plt

def solve_beam_finite_element(n_elements, bc_start, bc_end, force_coefficients)
    :
    def shape_functions(co,cod,codd,coddd,h):
        # Coefficients of shape functions
        co = [[0.5, -0.75, 0, 0.25],[-0.125*h, 0.125*h, 0.125*h, -0.125*h],
              [0.5, 0.75, 0, -0.25],[0.125*h, 0.125*h, -0.125*h, -0.125*h]]
        cod = [[-0.75, 0, 0.75], [0.125*h, 0.25*h, -0.375*h],
               [0.75, 0, -0.75],[0.125*h, -0.25*h, -0.375*h]]
        codd = [[0, 1.5], [0.25*h, -0.75*h], [0, -1.5], [-0.25*h, -0.75*h]]
        coddd = [[1.5], [-0.75*h], [-1.5], [-0.75*h]]
        return co, cod, codd, coddd

    def integrate(re):
        # Integration approximating coefficients
        coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
        coi=np.append(coi,
    [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
        s = 0
        for i in range(0, 6):
            su = 0
            for j in range(0, int(re.shape[0])):
                su = su + (re[j] * coi[i][0]**j)
            s = s + su * coi[i][1]
```

```
        return s

    def element_matrix(f):
      # Polynomial multiplication of coefficients in an elemental matrix
      for i in range(0, 4):
          element_forces[i][0] = (h / 2) * integrate(np.polynomial.polynomial.
polymul(co[i], f[0]))
      return element_forces

    def get_value(co, x, p, d):
        # Mapping of element position to global matrix position
        if d == 0:
            s = 0
            for i in range(0, p+1):
                s = s + co[i] * x**i
        elif d == 1:
            s = 0
            for i in range(0, p):
                s = s + co[i] * x**i
        elif d == 2:
            s = 0
            for i in range(0, p-1):
                s = s + co[i] * x**i
        else:
            s = 0
            for i in range(0, p-2):
                s = s + co[i] * x**i
        return s

    h = 1 / n_elements
    co = np.zeros((5, 5))
    cod = np.zeros((5, 5))
    codd = np.zeros((5, 5))
    coddd = np.zeros((5, 5))
    co, cod, codd, coddd = shape_functions(co, cod, codd, coddd, h)

    if bc_start == 2:
        force1 = float(input("Enter force= "))
```

```
        moment1 = float(input("Enter moment= "))
    else:
        displacement1 = 0
        slope1 = 0
    if bc_end == 2:
        force2 = 5
        moment2 = 5
    else:
        displacement2 = float(input("Enter displacement= "))
        slope2 = float(input("Enter slope= "))

    node_locations = np.zeros((n_elements, 2))
    for i in range(0, n_elements):
        for j in range(0, 2):
            if (i == j == 0):
                continue
            elif (j % 2 == 0):
                node_locations[i][j] = node_locations[i - 1][j + 1]
            else:
                node_locations[i][j] = node_locations[i][j - 1] + h

    stiffness_matrix = np.zeros((2 * n_elements + 2, 2 * n_elements + 2))
    forces = np.zeros((2 * n_elements + 2, 1))
    loads = np.zeros((2 * n_elements + 2, 1))
    element_stiffness = np.zeros((4, 4))
    element_stiffness += np.array([[6, -3*h, -6, -3*h], [-3*h, 2*h*h, 3*h, h*h],
                    [-6, 3*h, 6, 3*h], [-3*h, h*h, 3*h, 2*h*h]])
    element_stiffness = (1000 / (3 * h ** 3)) * element_stiffness

    for i in range(0, n_elements):
        element_forces = np.zeros((4, 1))
        sunod = (node_locations[i][0] + node_locations[i][1]) / 2
        forces_of_element = np.array([[force_coefficients[0][0] + force_coefficients[0][1] * sunod +
                                       force_coefficients[0][2] * sunod * 2,
                                       force_coefficients[0][1] * (h / 2) + force_coefficients[0][2] * h * sunod,
```

```
                                       force_coefficients[0][2] * (h / 4)]])
        element_forces = element_matrix(forces_of_element)
        if (i == 0):
            stiffness_matrix[:4, :4] += element_stiffness
            forces[:4, 0:1] += element_forces
        else:
            stiffness_matrix[2 * i:2 * (i + 2), 2 * i:2 * (i + 2)] += element_stiffness
            forces[2 * i:2 * (i + 2), 0:1] += element_forces

    stiffness_matrix_final = stiffness_matrix

    if (bc_start == 1):
        for i in range(1, 2 * n_elements + 2):
            forces[i] = forces[i] - displacement1 * stiffness_matrix_final[i][0]
        for i in range(0, 2 * n_elements + 2):
            for j in range(0, n_elements * 2 + 2):
                if (i == 0 or j == 0):
                    stiffness_matrix_final[i][j] = 0
        stiffness_matrix_final[0][0] = 1
        forces[0][0] = displacement1
        loads[0][0] = 0
        for i in range(2, 2 * n_elements + 2):
            forces[i] = forces[i] - slope1 * stiffness_matrix_final[i][1]
        for i in range(0, 2 * n_elements + 2):
            for j in range(0, 2 * n_elements + 2):
                if (i == 1 or j == 1):
                    stiffness_matrix_final[i][j] = 0
        stiffness_matrix_final[1][1] = 1
        forces[1][0] = slope1
        loads[1][0] = 0

    if (bc_end == 1):
        for i in range(0, 2 * n_elements + 1):
            forces[i] = forces[i] - slope2 * stiffness_matrix_final[i][2 * n_elements + 1]
        for i in range(0, n_elements * 2 + 2):
```

```
            for j in range(0, 2 * n_elements + 2):
                if (i == 2 * n_elements + 1 or j == n_elements * 2 + 1):
                    stiffness_matrix_final[i][j] = 0
        stiffness_matrix_final[n_elements * 2 + 1][n_elements * 2 + 1] = 1
        forces[n_elements * 2 + 1][0] = slope2
        loads[n_elements * 2 + 1][0] = 0
        for i in range(0, 2 * n_elements):
            forces[i] = forces[i] - displacement2 * stiffness_matrix_final[i][2 * n_elements]
        for i in range(0, n_elements * 2 + 2):
            for j in range(0, 2 * n_elements + 2):
                if (i == 2 * n_elements or j == n_elements * 2):
                    stiffness_matrix_final[i][j] = 0
        stiffness_matrix_final[n_elements * 2][n_elements * 2] = 1
        forces[n_elements * 2][0] = displacement2
        loads[n_elements * 2][0] = 0

    if (bc_start == 2):
        loads[0][0] = force1
        loads[1][0] = -moment1
    if (bc_end == 2):
        loads[2 * n_elements][0] = force2
        loads[2 * n_elements + 1][0] = -moment2

    U = np.linalg.inv(stiffness_matrix_final) @ (forces + loads)

    x = np.linspace(0, 1, 1000)
    y_deflection = []
    y_slope = []
    moment = []
    shear_force = []
    stress = []

    for i in range(0, n_elements):
        for j in range(0, 1000):
            if x[j] >= node_locations[i][0] and x[j] <= node_locations[i][1]:
                aux = (2 * x[j] - (node_locations[i][0] + node_locations[i][1])) / h
```

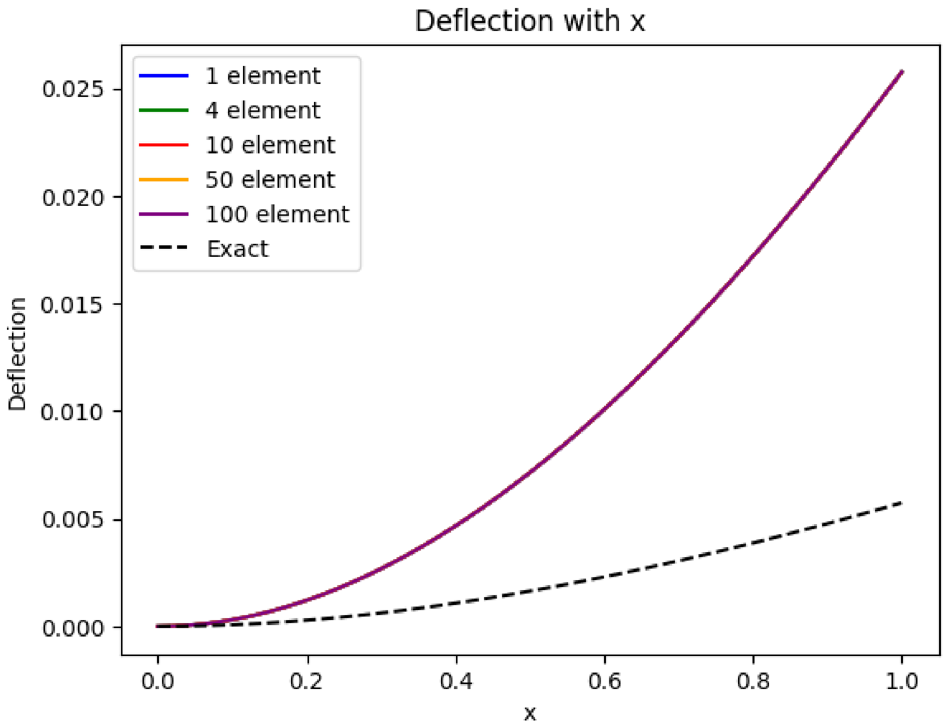
```
            w = 0
            wslope = 0
            mom = 0
            sh = 0
            st = 0
            b = 0
            for m in range(i * 2, i * 2 + 4):
                w = w + (U[m][0] * get_value(co[b], aux, 3, 0))
                wslope = wslope + (U[m][0] * get_value(cod[b], aux, 3, 1)) * (2 / h)
                mom = mom + (500 / 3) * (U[m][0] * get_value(codd[b], aux, 3, 2)) * (2 / h) ** 2
                sh = sh + (-500 / 3) * (U[m][0] * get_value(coddd[b], aux, 3, 3)) * (2 / h) ** 3
                st = st + (-1000) * (U[m][0] * get_value(codd[b], aux, 3, 2)) * (2 / h) ** 2
                b = b + 1
            y_deflection.append(w)
            y_slope.append(wslope)
            moment.append(mom)
            shear_force.append(sh)
            stress.append(st)

y_exact_deflection = []
y_exact_slope = []
i = 0
while i <= 1:
    y_exact_deflection.append((3 / 500) * ((i ** 4) / 24 - (i ** 3) / 3 + (5 * i * i)/ 4))
    y_exact_slope.append(((i ** 3) / 6 - i * i + 5 * i / 2) * (3 / 500))
    i = i + 0.001

error = 0
for i in range(0, 999):
    error = error + ((y_exact_deflection[i] - y_deflection[i]) ** 2)

rmse_error = np.sqrt(error / 1000) * 100
print("RMSE error% = ", rmse_error, "%")
```

```

    if (np.size(x) < np.size(y_deflection)):
        for i in range(0, (np.size(y_deflection) - np.size(x))):
            y_deflection.pop()
            y_slope.pop()
    elif (np.size(y_deflection) < np.size(x)):
        for i in range(0, (np.size(x) - np.size(y_deflection))):
            x = x[:-1]
            y_exact_deflection = y_exact_deflection[:-1]
            y_exact_slope = y_exact_slope[:-1]

    return y_deflection, y_exact_deflection, y_slope, y_exact_slope, moment,
    shear_force, stress, x

n_elements_list = [1, 4, 10, 50, 100]
force_coefficients = np.zeros((1, 3))
y_deflection_final = []
y_exact_deflection_final = []
y_slope_final = []
y_exact_slope_final = []
moment_final = []
shear_force_final = []
stress_final = []
x_final = []
force_coefficients[0][0] = 1

for i in range(0, 5):
    y_deflection, y_exact_deflection, y_slope, y_exact_slope, moment,
    shear_force, stress, x = \
        solve_beam_finite_element(n_elements_list[i], 1, 2, force_coefficients)
    y_deflection_final.append(y_deflection)
    y_exact_deflection_final.append(y_exact_deflection)
    y_slope_final.append(y_slope)
    y_exact_slope_final.append(y_exact_slope)
    moment_final.append(moment)
    shear_force_final.append(shear_force)
    stress_final.append(stress)
    x_final.append(x)
```

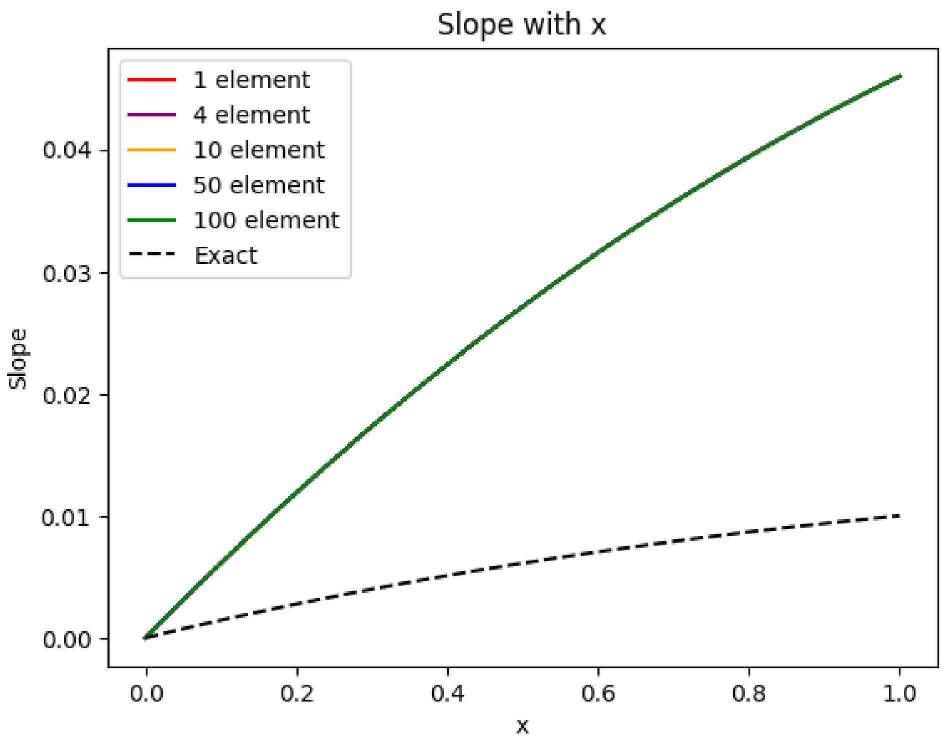
```
plt.figure()
plt.plot(x_final[0], y_deflection_final[0], label="1 element", color='blue')
plt.plot(x_final[1], y_deflection_final[1], label="4 element", color='green')
plt.plot(x_final[2], y_deflection_final[2], label="10 element", color='red')
plt.plot(x_final[3], y_deflection_final[3], label="50 element", color='orange')
plt.plot(x_final[4], y_deflection_final[4], label="100 element", color='purple'
    )
plt.plot(x_final[0], y_exact_deflection_final[0], label="Exact", color='black',
     linestyle='dashed')
plt.legend()
plt.title("Deflection with x")
plt.ylabel("Deflection")
plt.xlabel("x")
plt.show()

plt.figure()
plt.plot(x_final[0], y_slope_final[0], label="1 element", color='red')
plt.plot(x_final[1], y_slope_final[1], label="4 element", color='purple')
plt.plot(x_final[2], y_slope_final[2], label="10 element", color='orange')
plt.plot(x_final[3], y_slope_final[3], label="50 element", color='blue')
plt.plot(x_final[4], y_slope_final[4], label="100 element", color='green')
plt.plot(x_final[0], y_exact_slope_final[0], label="Exact", color='black',
    linestyle='dashed')
plt.legend()
plt.title("Slope with x")
plt.ylabel("Slope")
plt.xlabel("x")
plt.show()

plt.figure()
plt.plot(x_final[0], moment_final[0], label="1 element", color='red')
plt.plot(x_final[1], moment_final[1], label="4 element", color='purple')
plt.plot(x_final[2], moment_final[2], label="10 element", color='orange')
plt.plot(x_final[3], moment_final[3], label="50 element", color='blue')
plt.plot(x_final[4], moment_final[4], label="100 element", color='green')
plt.plot(x_final[4], moment_final[4], label="Exact", color='black', linestyle='
    dashed')
```

```
plt.legend()
plt.title("Moment with x")
plt.ylabel("Moment")
plt.xlabel("x")
plt.show()

plt.figure()
plt.plot(x_final[0], shear_force_final[0], label="1 element", color='red')
plt.plot(x_final[1], shear_force_final[1], label="4 element", color='purple')
plt.plot(x_final[2], shear_force_final[2], label="10 element", color='orange')
plt.plot(x_final[3], shear_force_final[3], label="50 element", color='blue')
plt.plot(x_final[4], shear_force_final[4], label="100 element", color='green')
plt.plot(x_final[4], shear_force_final[4], label="Exact", color='black',
    linestyle='dashed')
plt.legend()
plt.title("Shear force with x")
plt.ylabel("Shear force")
plt.xlabel("x")
plt.show()

plt.figure()
plt.plot(x_final[0], stress_final[0], label="1 element", color='red')
plt.plot(x_final[1], stress_final[1], label="4 element", color='purple')
plt.plot(x_final[2], stress_final[2], label="10 element", color='orange')
plt.plot(x_final[3], stress_final[3], label="50 element", color='blue')
plt.plot(x_final[4], stress_final[4], label="100 element", color='green')
plt.plot(x_final[4], stress_final[4], label="Exact", color='black', linestyle='
    dashed')
plt.legend()
plt.title("Stress with x")
plt.ylabel("Stress (MPa)")
plt.xlabel("x")
plt.show()
```

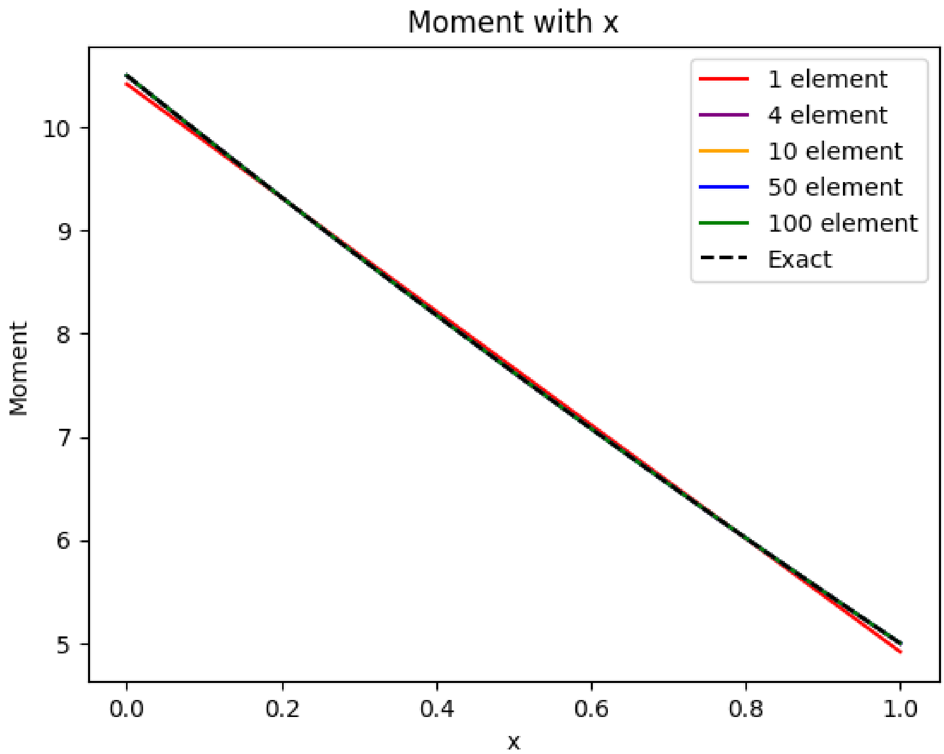
OUTPUTS PLOTS



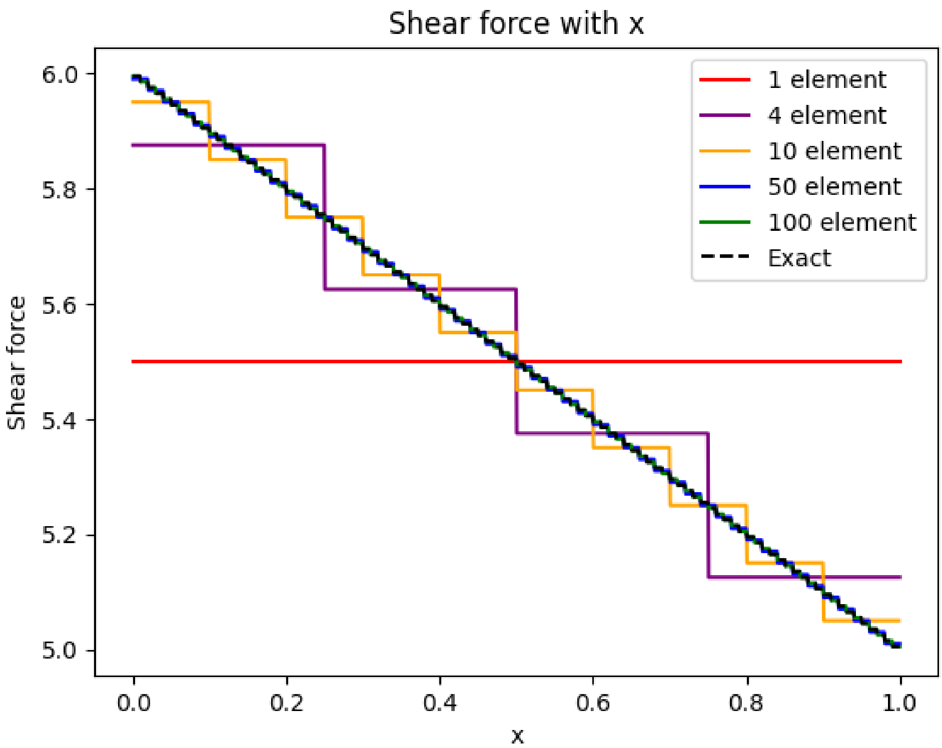
RMSE % = 0.9229651204807908 %



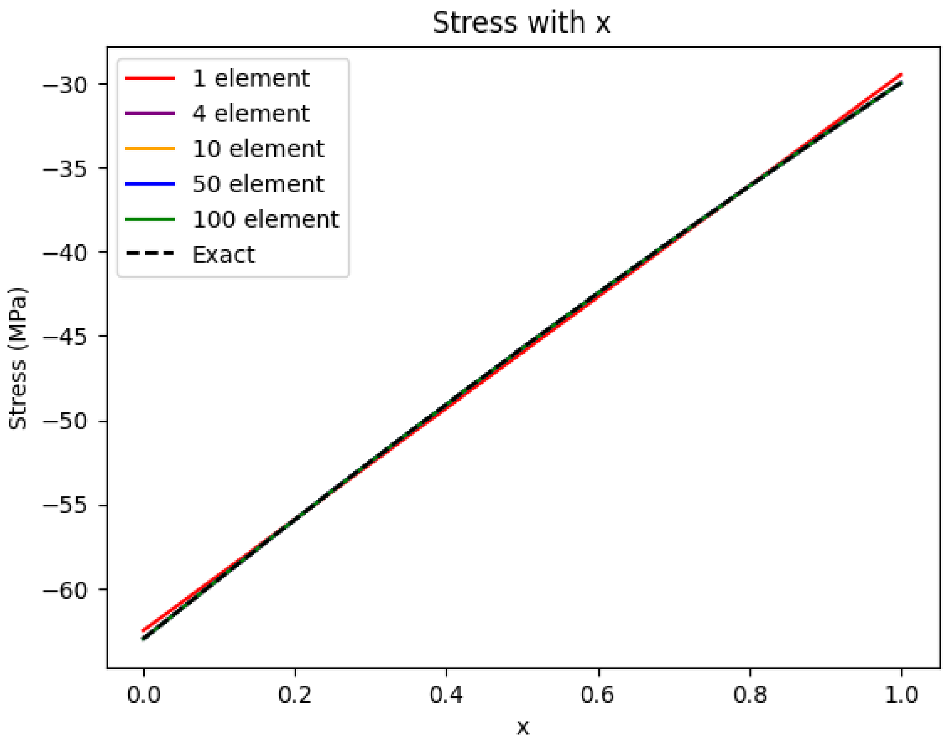
RMSE % = 0.9235163544395251 %



RMSE % = 0.9235187394198392 %



RMSE % = 0.9235188021954196 %



RMSE % = 0.9235188025278858 %

DISCUSSION

We observe that when a constant forcing is applied to the beam and the exact solution is plotted, it yields a quartic solution following the Euler-Bernoulli beam bending theory. Consequently, when approximating it using cubic Hermite polynomials, some approximation error is expected. However, as the number of elements increases, the approximation converges towards the exact solution.

Similarly, as we analyze the slope, cubic polynomials are approximated by quadratic polynomials. Consequently, similar convergence towards the exact solution is observed with increasing elements.

The most noticeable difference in approximation occurs in the moment curves, where quadratic curves are approximated by linear curves, resulting in minimal RMS error. However, beyond 10 elements, the difference becomes less apparent.

In the case of shear force, which is linear, approximation is done using constant functions, where the average value at the element boundaries represents the constant value. On increasing the elements, the approximation tends towards a sloped line.

One-dimensional hp Code

PROBLEM STATEMENT

Figure 1 shows an elastic bar under traction load and constrained at the ends A and B. Develop a generic finite element code to get the approximate solution to the resulting governing differential equation for the bar shown in Figure 1.

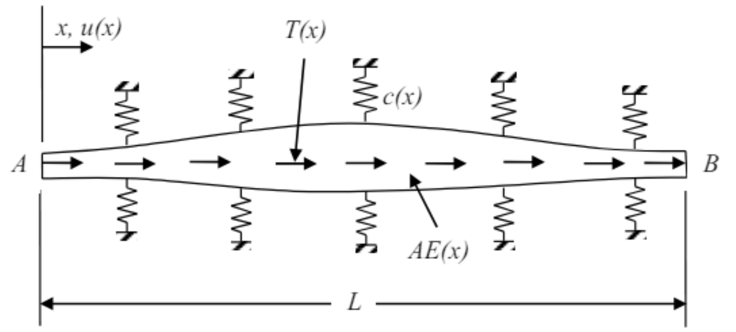


Figure 1: Elastic bar loaded in traction and constrained at ends A and B.

The code should have the following capabilities:

1. Boundary conditions/End Constraints: Both ends can be constrained by specifying (a) primary variable (Dirichlet/Displacement/Essential), (b) secondary variable or force (Force/Neumann/Natural) and (c) springs (Mixed/Robin)
2. The variables $T(x), c(x)$ and $AE(x)$ can vary from a constant to a quadratic function.
3. The length L and the number of elements will be input values. Discretize the domain into given number of elements with equal lengths.
4. There should be a provision to put at least one concentrated load at any given location (excluding the ends).
5. Use of either Lagrange interpolation or hierarchic shape functions upto quartic order should be possible.
6. Postprocessing must be able to represent the primary, secondary and other variables over the domain either continuously or discretely as required.

SOLUTION

1. Do the patch test for the following cases:

   $AE(x) = 1$ and $c(x) = 0$ with $u(x)|_{x=0} = 0$ and $\left.\frac{du}{dx}\right|_{x=1} = 0$. When $T(x) = 1$, use 1, 2, 5, 10 and 100 number of linear and quadratic elements and when $T(x) = x$ use 1, 2, 5, 10 and 100 number of linear, quadratic and cubic elements and superimpose your solutions with the respective exact solutions. Plot the error in the solution. Also plot the derivative of the exact solution and finite element solutions. Discuss the results.

2. For the problem in Point 1, plot the strain energy of the exact and finite element solution against the number of elements in the mesh for all the cases. Also plot the strain energy of the solution. Discuss the results.

PYTHON CODE WITH GIVEN INPUTS

```
import numpy as np
import matplotlib.pyplot as plt

def finite_element_solver(n_elements, degree, selection, bc_left, bc_right,
    a_coeff, c_coeff, f_coeff):
    def shape_function(degree, coordinates, coordinates_derivative, selection):
        if selection == 1:
            if degree == 1:
                coordinates = [[0.5, -0.5],
                               [0.5, 0.5]]
                coordinates_derivative = [[-0.5],
                                          [0.5]]
            elif degree == 2:
                coordinates = [[0, -0.5, 0.5],
                               [1, 0, -1],
                               [0, 0.5, 0.5]]
                coordinates_derivative = [[-0.5, 1],
                                          [0, -2],
                                          [0.5, 1]]
            elif degree == 3:
                coordinates = [[-0.0625, 0.0625, 0.5625, -0.5625],
                               [0.5625, -1.6875, -0.5625, 1.6875],
                               [0.5625, 1.6875, -0.5625, -1.6875],
                               [-0.0625, -0.0625, 0.5625, 0.5625]]
                coordinates_derivative = [[0.0625, 1.125, -1.6875],
                                          [-1.6875, -1.125, 5.0625],
                                          [1.6875, -1.125, -5.0625],
                                          [-0.0625, 1.125, 1.6875]]
            elif degree == 4:
                coordinates = [[0, 0.1667, -0.1667, -0.6667, 0.6667],
                               [0, -1.3333, 2.6667, 1.3333, -2.6667],
                               [1, 0, -5, 0, 4],
                               [0, 1.3333, 2.6667, -1.3333, -2.6667],
```

```
                              [0, -0.1667, -0.1667, 0.6667, 0.6667]]
                coordinates_derivative = [[0.1667, -0.3333, -2, 2.6667],
                                          [-1.3333, 5.3333, 4, -10.6667],
                                          [0, -10, 0, 16],
                                          [1.3333, 5.3333, -4, -10.6667],
                                          [-0.1667, -0.3333, 2, 2.6667]]
        else:
            if degree == 1:
                coordinates = [[0.5, -0.5],
                               [0.5, 0.5]]
                coordinates_derivative = [[-0.5],
                                          [0.5]]
            elif degree == 2:
                coordinates = [[0.5, -0.5, 0, 0, 0],
                               [-0.6124, 0, 0.6124, 0, 0],
                               [0.5, 0.5, 0, 0, 0]]
                coordinates_derivative = [[-0.5, 0],
                                          [0, 1.2247],
                                          [0.5, 0]]
            elif degree == 3:
                coordinates = [[0.5, -0.5, 0, 0, 0],
                               [0, -0.7906, 0, 0.7906, 0],
                               [-0.6124, 0, 0.6124, 0, 0],
                               [0.5, 0.5, 0, 0, 0]]
                coordinates_derivative = [[-0.5, 0, 0, 0],
                                          [-0.7906, 0, 2.3717, 0],
                                          [0, 1.2247, 0, 0],
                                          [0.5, 0, 0, 0]]
            elif degree == 4:
                coordinates = [[0.5, -0.5, 0, 0, 0],
                               [0.2338, 0, -1.4031, 0, 1.1693],
                               [0, -0.7906, 0, 0.7906, 0],
                               [-0.6124, 0, 0.6124, 0, 0],
                               [0.5, 0.5, 0, 0, 0]]
                coordinates_derivative = [[-0.5, 0, 0, 0],
                                          [0, -2.8062, 0, 4.6771],
                                          [-0.7906, 0, 2.3717, 0],
                                          [0, 1.2247, 0, 0],
```

```
                                          [0.5, 0, 0, 0]]
    return coordinates, coordinates_derivative

def integration_rule(re):
    coordinates_integration = np.array([[0.23862, 0.46791], [-0.23862, 0.46791], [0.66121, 0.36076],
                                        [-0.66121, 0.36076], [0.93247, 0.17132], [-0.93247, 0.17132]])
    s = 0
    for i in range(0, 6):
        su = 0
        for j in range(0, int(re.shape[0])):
            su = su + (re[j] * coordinates_integration[i][0] ** j)
        s = s + su * coordinates_integration[i][1]
    return s

def element_matrix(ae, c, f, p):
    for i in range(0, p + 1):
        for j in range(0, p + 1):
            ke[i][j] = (2 / element_length) * integration_rule(
                np.polynomial.polynomial.polymul(np.polynomial.polynomial.polymul(cod[i], cod[j]), ae[0]))
            ge[i][j] = (element_length / 2) * integration_rule(
                np.polynomial.polynomial.polymul(np.polynomial.polynomial.polymul(co[i], co[j]), c[0]))
        fe[i][0] = (element_length / 2) * integration_rule(
            np.polynomial.polynomial.polymul(co[i], f[0]))
    return ke, ge, fe

def get_value(co, x, p, d):
    if d == 0:
        s = 0
        for i in range(0, p + 1):
            s = s + co[i] * x ** i
    else:
        s = 0
        for i in range(0, p):
            s = s + co[i] * x ** i
```

```
    return s

co = np.zeros((5, 5))
cod = np.zeros((5, 5))
co, cod = shape_function(degree, co, cod, selection)

if bc_left == 2:
    force_left = float(input("Enter force= "))
elif bc_left == 1:
    displacement_left = 0
else:
    spring_constant_left = float(input("Enter spring constant= "))
    deviation_left = float(input("Enter spring deviation= "))

if bc_right == 2:
    force_right = 0
elif bc_right == 1:
    displacement_right = float(input("Enter displacement= "))
else:
    spring_constant_right = float(input("Enter spring constant= "))
    deviation_right = float(input("Enter spring deviation= "))

element_length = 1 / n_elements
node_location = np.zeros((n_elements, 2))

for i in range(0, n_elements):
    for j in range(0, 2):
        if i == j == 0:
            continue
        elif j % 2 == 0:
            node_location[i][j] = node_location[i - 1][j + 1]
        else:
            node_location[i][j] = node_location[i][j - 1] + element_length

stiffness_matrix = np.zeros((n_elements * degree + 1, n_elements * degree + 1))
gradient_matrix = np.zeros((n_elements * degree + 1, n_elements * degree + 1))
```

```python
    force_matrix = np.zeros((n_elements * degree + 1, 1))
    load_matrix = np.zeros((n_elements * degree + 1, 1))

    for i in range(0, n_elements):
        ke = np.zeros((degree + 1, degree + 1))
        ge = np.zeros((degree + 1, degree + 1))
        fe = np.zeros((degree + 1, 1))

        midpoint = (node_location[i][0] + node_location[i][1]) / 2

        ae_coefficient = np.array([[a_coeff[0][0] + a_coeff[0][1] * midpoint + a_coeff[0][2] * midpoint ** 2,
                                    a_coeff[0][1] * (element_length / 2) + a_coeff[0][2] * element_length * midpoint,
                                    a_coeff[0][2] * (element_length / 4)]])
        c_coefficient = np.array([[c_coeff[0][0] + c_coeff[0][1] * midpoint + c_coeff[0][2] * midpoint ** 2,
                                    c_coeff[0][1] * (element_length / 2) + c_coeff[0][2] * element_length * midpoint,
                                    c_coeff[0][2] * (element_length / 4)]])
        f_coefficient = np.array([[f_coeff[0][0] + f_coeff[0][1] * midpoint + f_coeff[0][2] * midpoint ** 2,
                                    f_coeff[0][1] * (element_length / 2) + f_coeff[0][2] * element_length * midpoint,
                                    f_coeff[0][2] * (element_length / 4)]])

        ke, ge, fe = element_matrix(ae_coefficient, c_coefficient, f_coefficient, degree)

        if i == 0:
            stiffness_matrix[:degree + 1, :degree + 1] += ke
            gradient_matrix[:degree + 1, :degree + 1] += ge
            force_matrix[:degree + 1, 0:1] += fe
        else:
            stiffness_matrix[i * degree:i * degree + degree + 1, i * degree:i * degree + degree + 1] += ke
            gradient_matrix[i * degree:i * degree + degree + 1, i * degree:i * degree + degree + 1] += ge
```

```
            force_matrix[i * degree:i * degree + degree + 1, 0:1] += fe

    stiffness_plus_gradient = stiffness_matrix + gradient_matrix

    if bc_left == 1:
        for i in range(1, n_elements * degree + 1):
            force_matrix[i] = force_matrix[i] - displacement_left * stiffness_plus_gradient[i][0]
        for i in range(0, n_elements * degree + 1):
            for j in range(0, n_elements * degree + 1):
                if i == 0 or j == 0:
                    stiffness_plus_gradient[i][j] = 0
        stiffness_plus_gradient[0][0] = 1
        force_matrix[0][0] = displacement_left
        load_matrix[0][0] = 0

    if bc_right == 1:
        for i in range(0, n_elements * degree):
            force_matrix[i] = force_matrix[i] - displacement_right * stiffness_plus_gradient[i][n_elements * degree]
        for i in range(0, n_elements * degree + 1):
            for j in range(0, n_elements * degree + 1):
                if i == n_elements * degree or j == n_elements * degree:
                    stiffness_plus_gradient[i][j] = 0
        stiffness_plus_gradient[n_elements * degree][n_elements * degree] = 1
        force_matrix[n_elements * degree][0] = displacement_right
        load_matrix[n_elements * degree][0] = 0

    if bc_left == 2:
        load_matrix[0][0] = force_left

    if bc_right == 2:
        load_matrix[n_elements * degree][0] = force_right

    if bc_left == 3:
        stiffness_plus_gradient[0][0] += spring_constant_left
        load_matrix[0][0] = spring_constant_left * deviation_left

```

```
    if bc_right == 3:
        stiffness_plus_gradient[n_elements * degree][n_elements * degree] += spring_constant_right
        load_matrix[n_elements * degree][0] = spring_constant_right * deviation_right

    displacement_solution = np.linalg.inv(stiffness_plus_gradient) @ (force_matrix + load_matrix)

    x_values = np.linspace(0, 1, 1000)
    y_values = []
    y_derivative_values = []

    for i in range(0, n_elements):
        for j in range(0, 1000):
            if x_values[j] >= node_location[i][0] and x_values[j] <= node_location[i][1]:
                auxiliary = (2 * x_values[j] - (node_location[i][0] + node_location[i][1])) / element_length
                fr = 0
                frd = 0
                b = 0
                for m in range(i * degree, i * degree + degree + 1):
                    fr = fr + (displacement_solution[m][0] * get_value(co[b], auxiliary, degree, 0))
                    frd = frd + (displacement_solution[m][0] * get_value(cod[b], auxiliary, degree, 1)) * (2 / element_length)
                    b = b + 1
                y_values.append(fr)
                y_derivative_values.append(frd)

    exact_values = []
    exact_derivative_values = []
    i = 0

    while i < 1:
        exact_values.append(-i ** 2 / 2 + (force_right + 1) * i + displacement_left)
```

```
        exact_derivative_values.append(-i + (force_right + 1))
        i = i + 0.001

    if np.size(y_values) < np.size(exact_values):
        for i in range(0, (np.size(exact_values) - np.size(y_values))):
            exact_values.pop()
            x_values = x_values[:-1]
    elif np.size(exact_values) < np.size(y_values):
        for i in range(0, (np.size(y_values) - np.size(exact_values))):
            y_values.pop()

    return y_values, y_derivative_values, exact_values, exact_derivative_values
    , x_values


number_of_elements = [1, 2, 5, 10, 100]
orders = [1, 2]
selection = 1
ae_coefficient = np.zeros((1, 3))
c_coefficient = np.zeros((1, 3))
f_coefficient = np.zeros((1, 3))
ae_order = 0

for i in range(0, ae_order + 1):
    ae_coefficient[0][i] = 1

c_order = 0

for i in range(0, c_order + 1):
    c_coefficient[0][i] = 0

f_order = 0

for i in range(0, f_order + 1):
    f_coefficient[0][i] = 1

bc_left = 1
bc_right = 2
```

```
for order in orders:
    y_values, y_derivative_values, exact_values, exact_derivative_values,
    x_values = finite_element_solver(1, order, selection, bc_left, bc_right,
    ae_coefficient, c_coefficient, f_coefficient)
    err=[exact_values[i]-exact_values[i] for i in range(len(exact_values))]
    line1, = plt.plot(x_values, err, label="Exact solution")
    err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
    line2, = plt.plot(x_values, err, label="1 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values,
    x_values = finite_element_solver(2, order, selection, bc_left, bc_right,
    ae_coefficient, c_coefficient, f_coefficient)
    err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
    line3, = plt.plot(x_values, err, label="2 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values,
    x_values = finite_element_solver(5, order, selection, bc_left, bc_right,
    ae_coefficient, c_coefficient, f_coefficient)
    err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
    line4, = plt.plot(x_values, err, label="5 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values,
    x_values = finite_element_solver(10, order, selection, bc_left, bc_right,
    ae_coefficient, c_coefficient, f_coefficient)
    err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
    line5, = plt.plot(x_values, err, label="10 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values,
    x_values = finite_element_solver(100, order, selection, bc_left, bc_right,
    ae_coefficient, c_coefficient, f_coefficient)
    err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
    line6, = plt.plot(x_values, err, label="100 element")
    plt.legend(handles=[line1, line2, line3, line4, line5, line6])
    plt.title("Plot of error between exact solution and FEM solution")
    plt.xlabel("x")
    plt.ylabel("Difference in values(Error)")
    plt.show()

for order in orders:
    line1, = plt.plot(x_values, exact_derivative_values, label="Exact solution"
    )
```

```
    y_values, y_derivative_values, exact_values, exact_derivative_values, x_values = finite_element_solver(1, order, selection, bc_left, bc_right, ae_coefficient, c_coefficient, f_coefficient)
    line2, = plt.plot(x_values, y_derivative_values, label="1 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values, x_values = finite_element_solver(2, order, selection, bc_left, bc_right, ae_coefficient, c_coefficient, f_coefficient)
    line3, = plt.plot(x_values, y_derivative_values, label="2 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values, x_values = finite_element_solver(5, order, selection, bc_left, bc_right, ae_coefficient, c_coefficient, f_coefficient)
    line4, = plt.plot(x_values, y_derivative_values, label="5 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values, x_values = finite_element_solver(10, order, selection, bc_left, bc_right, ae_coefficient, c_coefficient, f_coefficient)
    line5, = plt.plot(x_values, y_derivative_values, label="10 element")
    y_values, y_derivative_values, exact_values, exact_derivative_values, x_values = finite_element_solver(100, order, selection, bc_left, bc_right, ae_coefficient, c_coefficient, f_coefficient)
    line6, = plt.plot(x_values, y_derivative_values, label="100 element")
    plt.legend(handles=[line1, line2, line3, line4, line5, line6])
    plt.title("Plot of derivatives")
    plt.xlabel("x")
    plt.ylabel("Force")
    plt.show()

errors = []
log_n = []
energies = []

for n in number_of_elements:
    y_values, y_derivative_values, exact_values, exact_derivative_values, x_values = finite_element_solver(n, 1, selection, bc_left, bc_right, ae_coefficient, c_coefficient, f_coefficient)
    error = np.log(np.linalg.norm(np.array(exact_values) - np.array(y_values)))
    errors.append(error)
    log_n.append(np.log(n))
    total_energy = 0
```

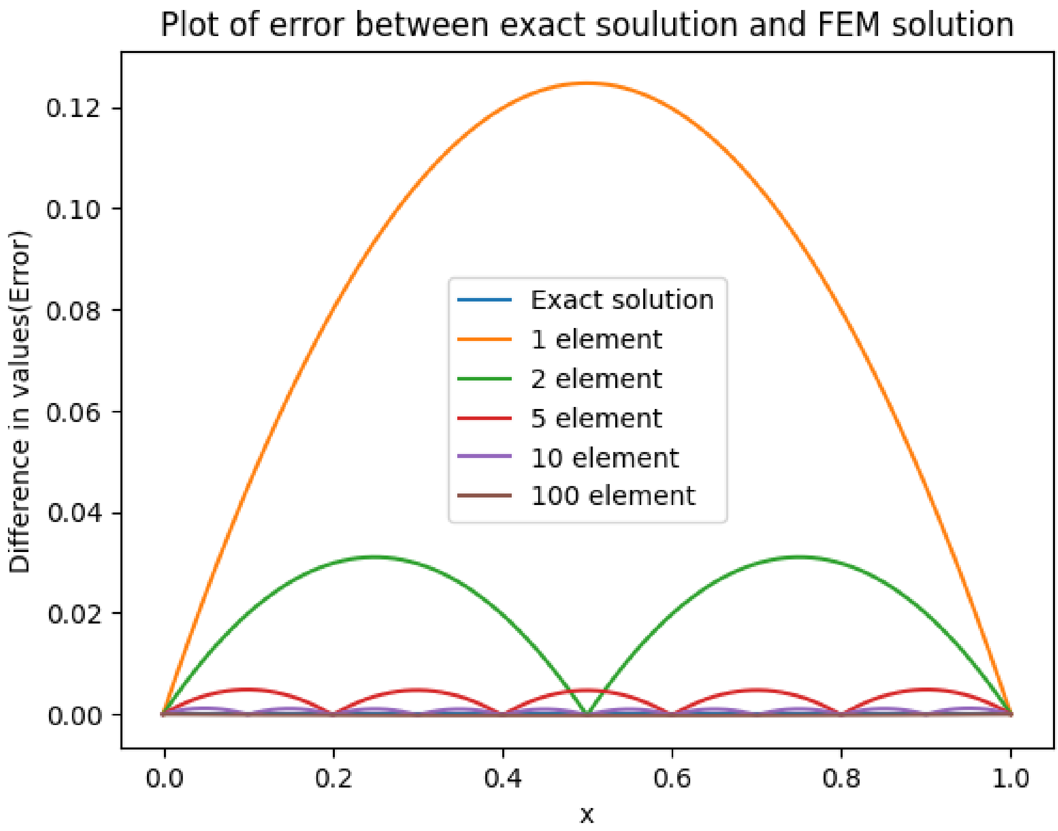
```python

    for i in range(0, 998):
        fx1 = y_derivative_values[i] ** 2
        fx2 = y_derivative_values[i + 1] ** 2
        total_energy = total_energy + 0.5 * (fx1 + fx2) * (x_values[i + 1] -
    x_values[i])

    energies.append(total_energy)

print(log_n, errors)

line1, = plt.plot(number_of_elements, energies, label="linear", marker=".")
errors = []
log_n = []
energies = []
exact_energies = []

for n in number_of_elements:
    y_values, y_derivative_values, exact_values, exact_derivative_values,
    x_values = finite_element_solver(n, 2, selection, bc_left, bc_right,
    ae_coefficient, c_coefficient, f_coefficient)
    error = np.log(np.linalg.norm(np.array(exact_values) - np.array(y_values)))
    errors.append(error)
    log_n.append(np.log(n))
    total_energy = 0

    for i in range(0, 998):
        fx1 = y_derivative_values[i] ** 2
        fx2 = y_derivative_values[i + 1] ** 2
        total_energy = total_energy + 0.5 * (fx1 + fx2) * (x_values[i + 1] -
    x_values[i])

    energies.append(total_energy)

    total_exact_energy = 0

    for i in range(0, 998):
        fx1 = exact_derivative_values[i] ** 2
```

```
            fx2 = exact_derivative_values[i + 1] ** 2
            total_exact_energy = total_exact_energy + 0.5 * (fx1 + fx2) * (x_values
    [i + 1] - x_values[i])

        exact_energies.append(total_exact_energy)

line2, = plt.plot(number_of_elements, energies, label="quadratic", marker=".")
plt.legend(handles=[line1, line2])
plt.xlabel("Number of elements")
plt.ylabel("Energy")
plt.title("Strain energy by approximating degree")
plt.show()

line1, = plt.plot(number_of_elements, energies, label="FEM solution", marker=".
    ")
line2, = plt.plot(number_of_elements, exact_energies, label="Exact solution",
    marker=".")
plt.legend(handles=[line1, line2])
plt.xlabel("Number of elements")
plt.ylabel("Energy")
plt.title("Strain Energy of FEM solution and Exact solution")
plt.show()
```

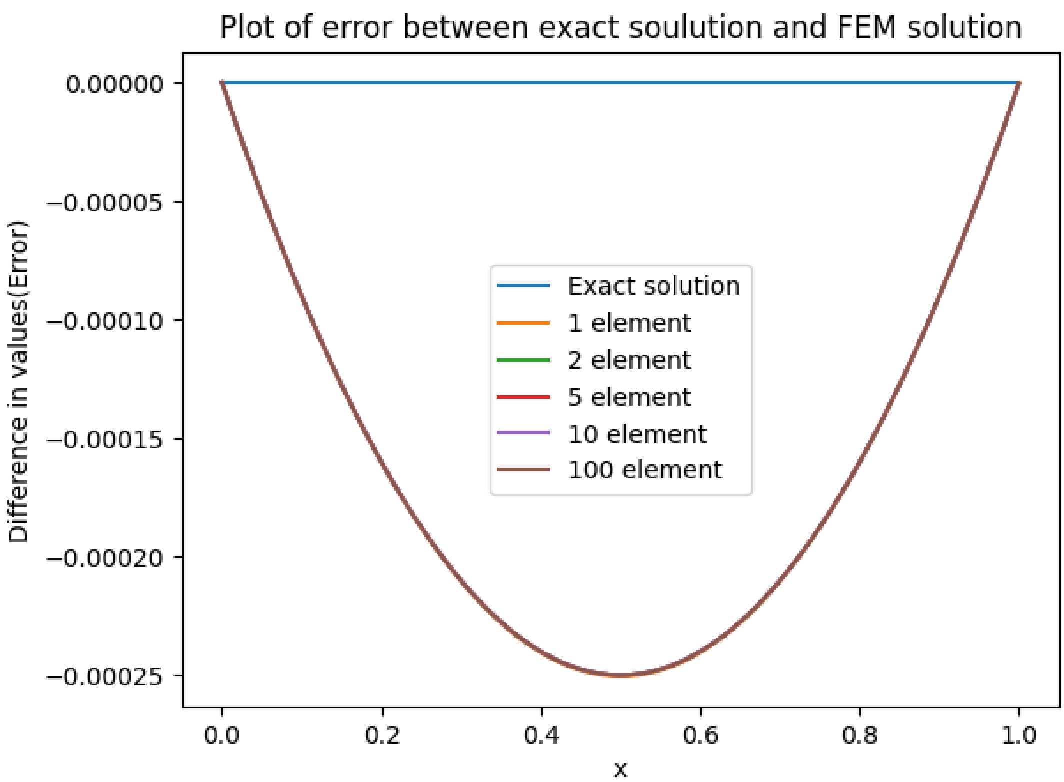
This code solves the weak form equation using finite element analysis and gives us primary and secondary variables, which we can plot to analyze the accuracy of our model with varying numbers of elements starting from 1 and going all the way to 100. We can observe that the more the number of elements and the higher the order of approximation of function, the more accurate the result will be, as shown in the plots. Also, here we can edit boundary conditions and values of traction to achieve all the plots required for part 2 of the question.

OUTPUT PLOTS

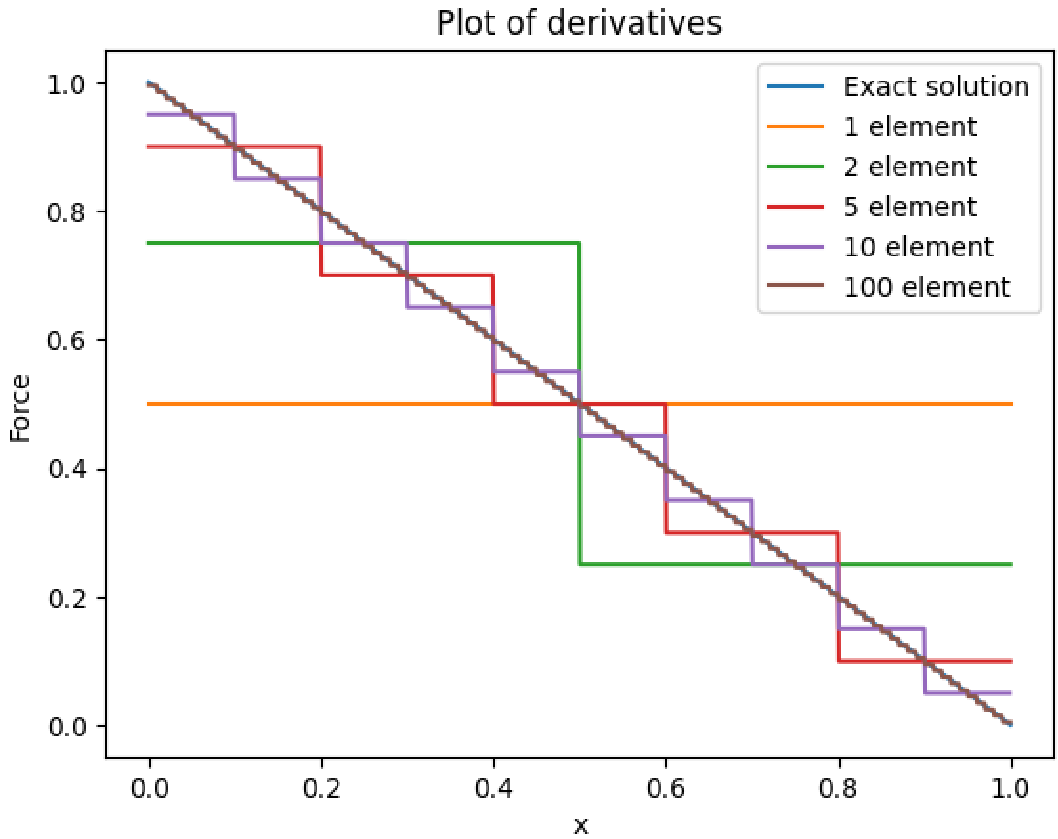
For AE(x) = 1, C(x) = 0, T(x)=1



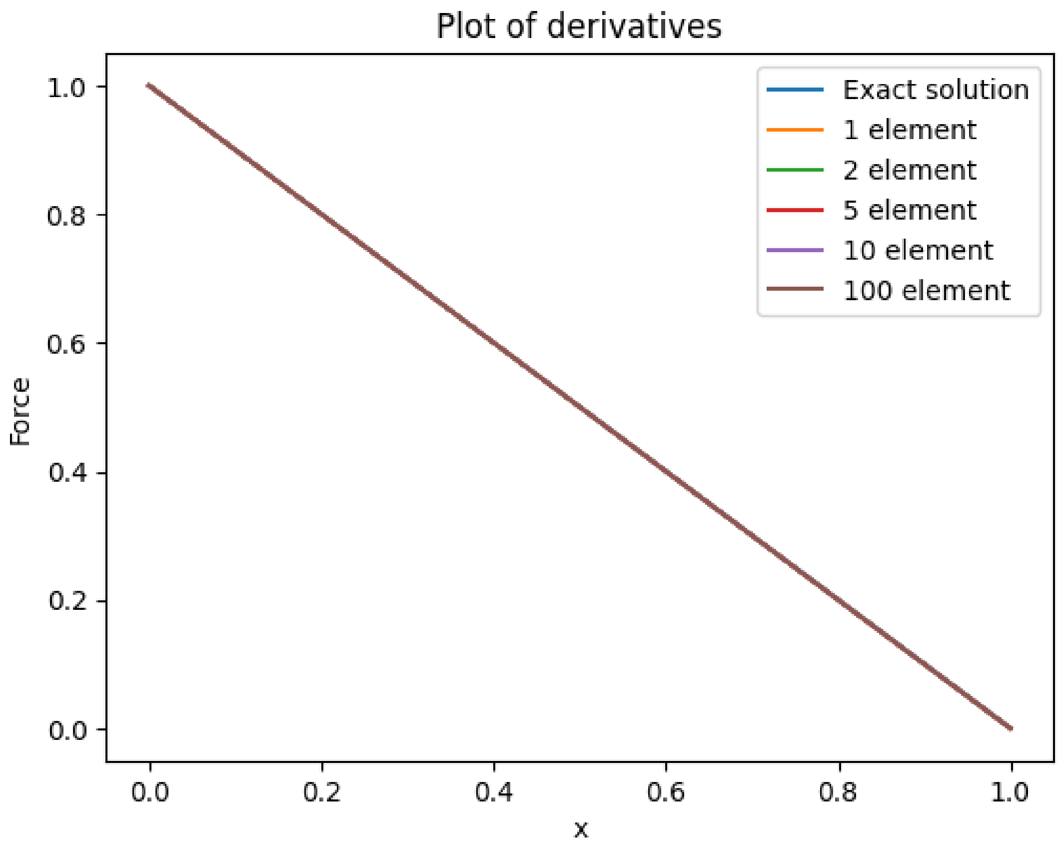
Linear Approximation



Quadratic Approximation

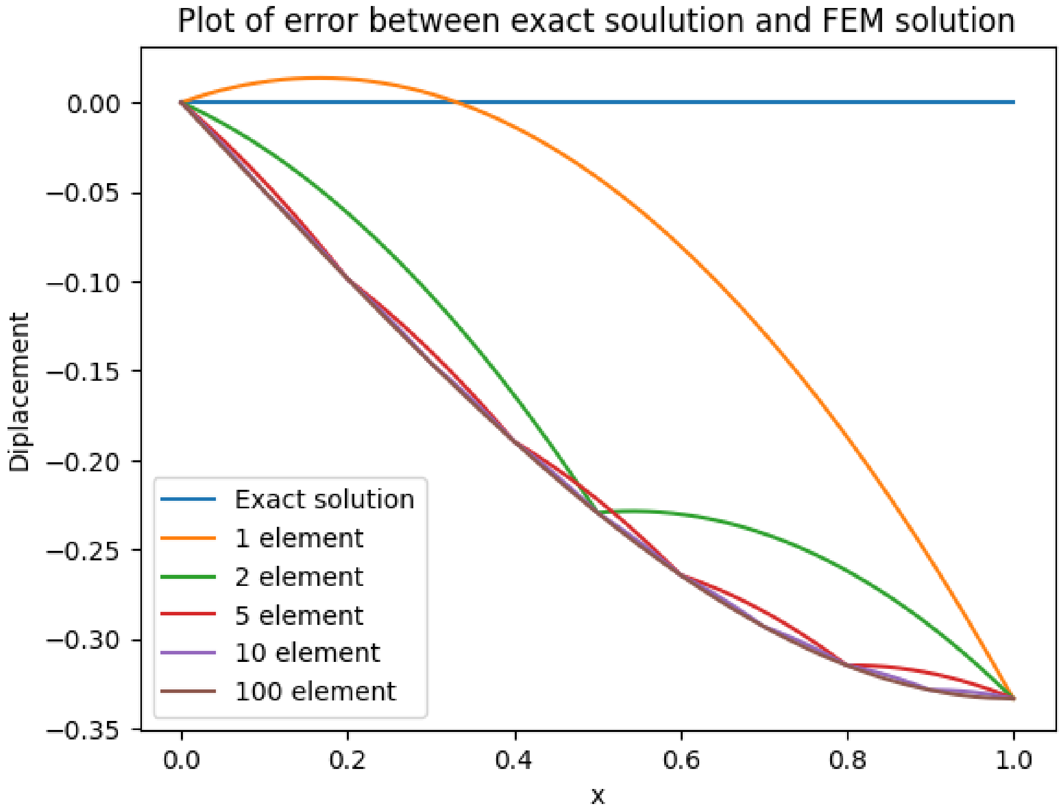


Linear Approximation

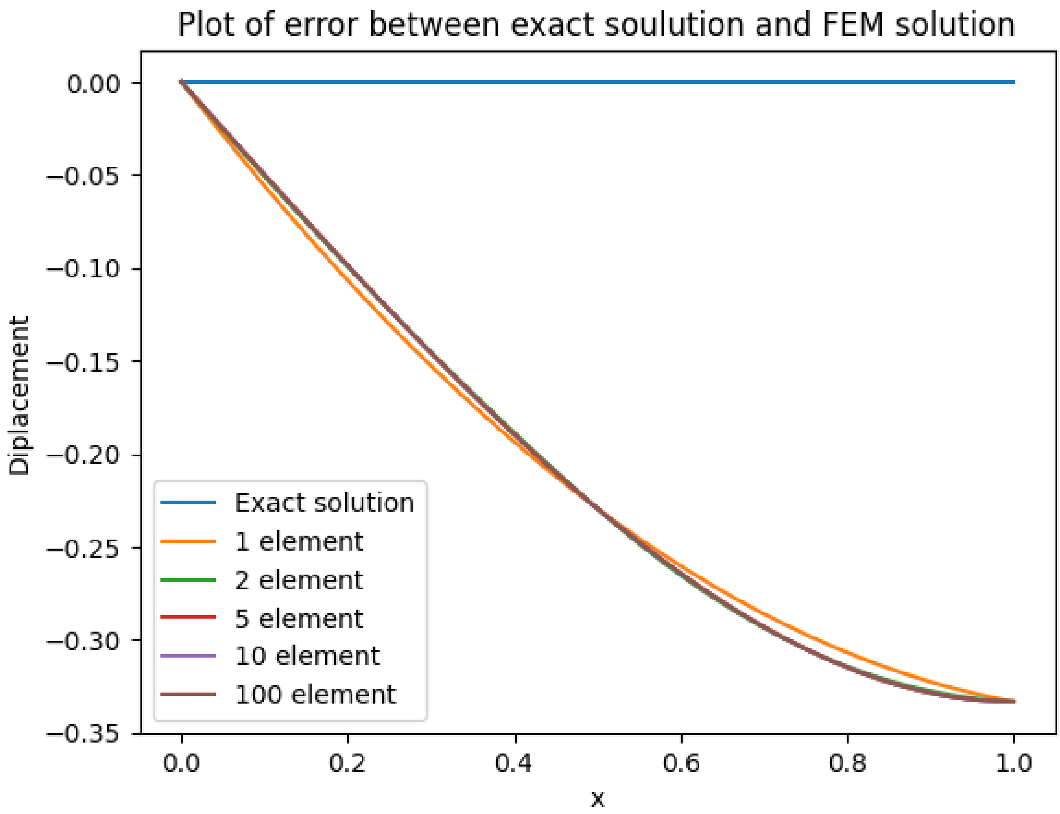


Quadratic Approximation

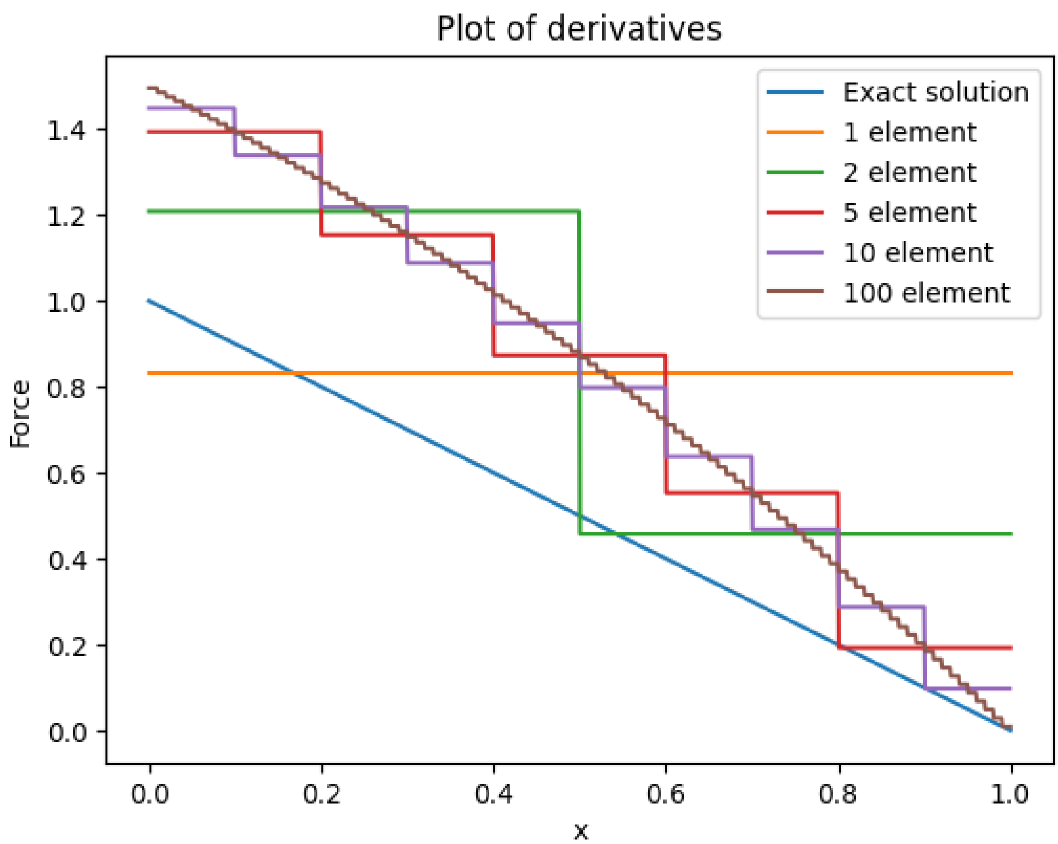
For AE(x) = 1, C(x) = 0, T(x)=x



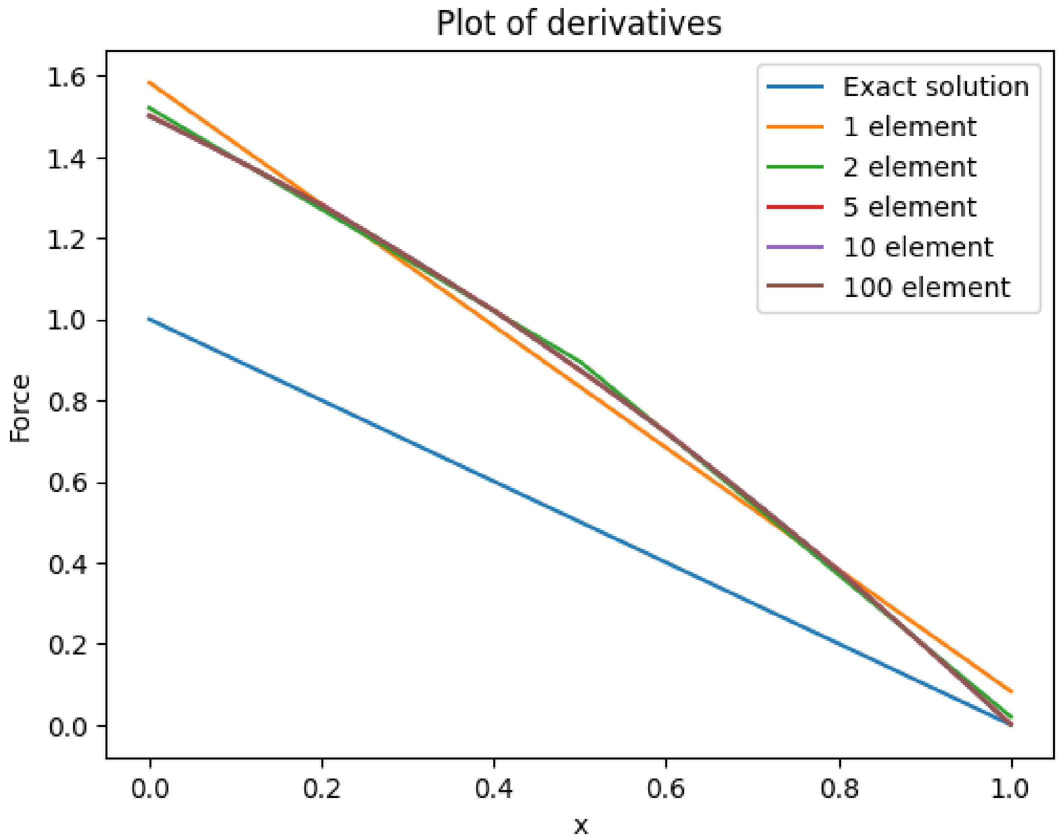
Linear Approximation



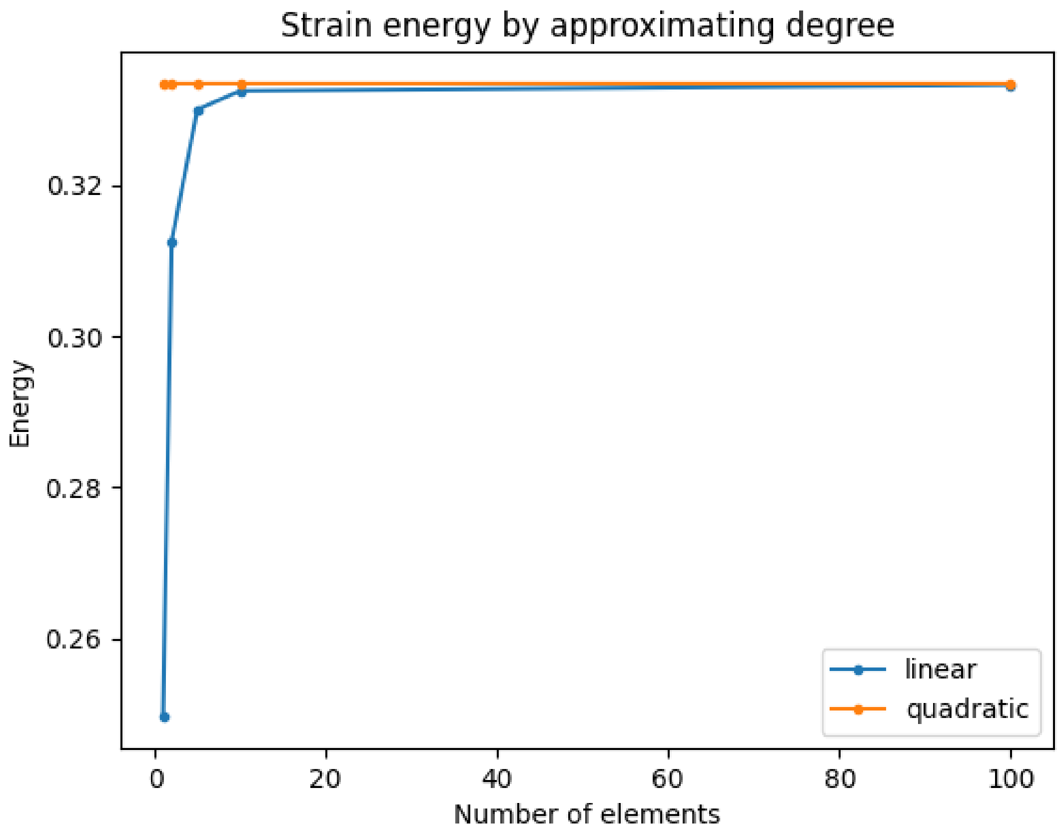
Quadratic Approximation

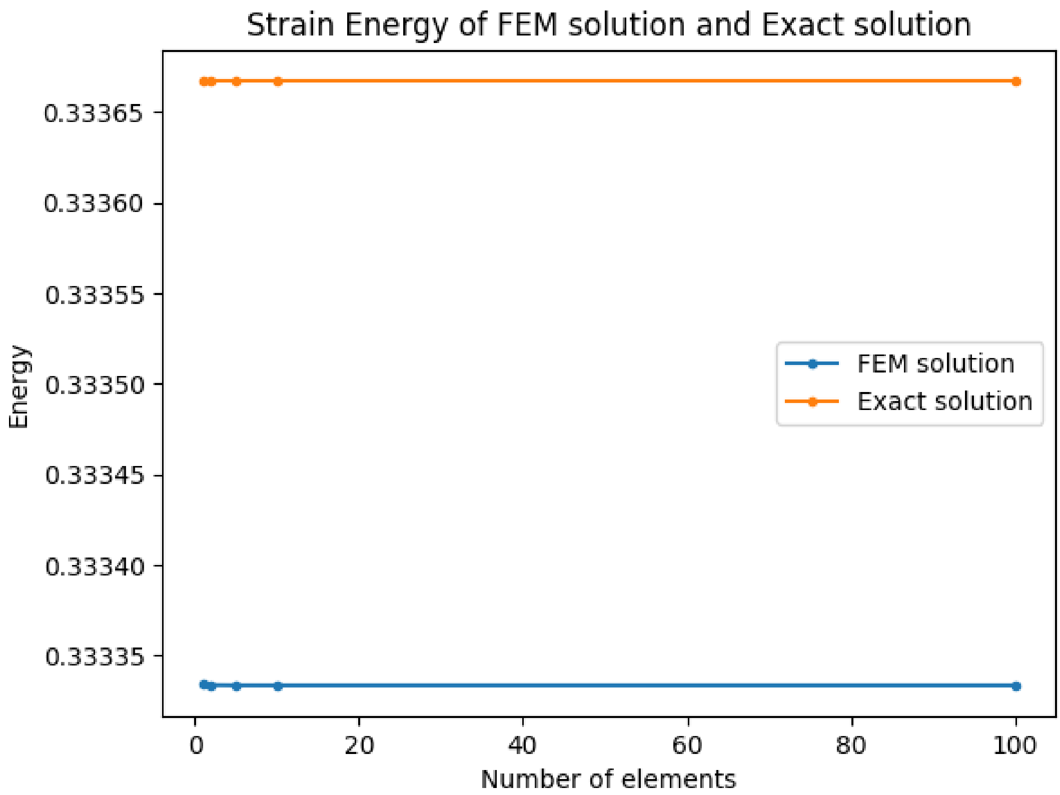


Linear Approximation



Quadratic Approximation





3. Take $AE(x) = 1$, $c(x) = 1$ and $T(x) = 1$ with $u(x)|_{x=0} = 0$ and $\left.\frac{du}{dx}\right|_{x=1} = 0$. Obtain the finite element solution with linear, quadratic, cubic and quartic elements respectively for 1, 10, 20, 40, 80 and 100 number of elements. For these cases:
   a) Plot the exact and finite element solutions together for these cases.
   b) Plot the error in the solution for these cases.
   c) Plot the strain energy of the finite element and exact solution as a function of number of elements.
   d) Plot the strain energy of the error as a function of number of elements.
   e) Plot the log of the relative error in the energy norm versus the log of number of elements.
   f) Try to estimate the convergence rate.

   Discuss the results.

PYTHON CODE WITH GIVEN INPUTS

```
import numpy as np
import matplotlib.pyplot as plt
def FEM_hp(n,p,choice,bc1,bc2,aeco,cco,fco):
    def shapeFunctions(p,co,cod,choice):
        if(choice==1):
            if (p==1):
                co=[[0.5,-0.5],
                    [0.5,0.5]]
```

```
            cod=[[-0.5],
                [0.5]]
        elif (p==2):
            co=[[0,-0.5,0.5],
                [1,0,-1],
                [0,0.5,0.5]]
            cod=[[-0.5,1],
                [0,-2],
                [0.5,1]]
        elif (p==3):
            co=[[-0.0625,0.0625,0.5625,-0.5625],
                [0.5625,-1.6875,-0.5625,1.6875],
                [0.5625,1.6875,-0.5625,-1.6875],
                [-0.0625,-0.0625,0.5625,0.5625]]
            cod=[[0.0625,1.125,-1.6875],
                [-1.6875,-1.125,5.0625],
                [1.6875,-1.125,-5.0625],
                [-0.0625,1.125,1.6875]]
        elif (p==4):
            co=[[0,0.1667,-0.1667,-0.6667,0.6667],
                [0,-1.3333,2.6667,1.3333,-2.6667],
                [1,0,-5,0,4],
                [0,1.3333,2.6667,-1.3333,-2.6667],
                [0,-0.1667,-0.1667,0.6667,0.6667]]
            cod=[[0.1667,-0.3333,-2,2.6667],
                [-1.3333,5.3333,4,-10.6667],
                [0,-10,0,16],
                [1.3333,5.3333,-4,-10.6667],
                [-0.1667,-0.3333,2,2.6667]]
    else:
        if (p==1):
            co=[[0.5,-0.5],
                [0.5,0.5]]
            cod=[[-0.5],
                [0.5]]
        elif (p==2):
            co=[[0.5,-0.5,0,0,0],
                [-0.6124,0,0.6124,0,0],
```

```
                    [0.5,0.5,0,0,0]]
                cod=[[-0.5,0],
                    [0,1.2247],
                    [0.5,0]]
            elif (p==3):
                co=[[0.5,-0.5,0,0,0],
                    [0,-0.7906,0,0.7906,0],
                    [-0.6124,0,0.6124,0,0],
                    [0.5,0.5,0,0,0]]
                cod=[[-0.5,0,0,0],
                    [-0.7906,0,2.3717,0],
                    [0,1.2247,0,0],
                    [0.5,0,0,0]]
            elif (p==4):
                co=[[0.5,-0.5,0,0,0],
                    [0.2338,0,-1.4031,0,1.1693],
                    [0,-0.7906,0,0.7906,0],
                    [-0.6124,0,0.6124,0,0],
                    [0.5,0.5,0,0,0]]
                cod=[[-0.5,0,0,0],
                    [0,-2.8062,0,4.6771],
                    [-0.7906,0,2.3717,0],
                    [0,1.2247,0,0],
                    [0.5,0,0,0]]
        return co,cod
    def integrate(re):
        coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
        coi=np.append(coi,
    [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
        s=0
        for i in range(0,6):
            su=0
            for j in range(0,int(re.shape[0])):
                su=su+(re[j]*coi[i][0]**j)
            s=s+su*coi[i][1]
        return s
    def elementMatrix(ae,c,f,p):
        for i in range(0, p+1):
```

```
                for j in range(0, p+1):
                    ke[i][j]=(2/h)*integrate(np.polynomial.polynomial.polymul(np.polynomial.polynomial.polymul(cod[i],cod[j]),ae[0]))
                    ge[i][j]=(h/2)*integrate(np.polynomial.polynomial.polymul(np.polynomial.polynomial.polymul(co[i],co[j]),c[0]))
                fe[i][0]=(h/2)*integrate(np.polynomial.polynomial.polymul(co[i],f[0]))
        return ke,ge,fe
    def getValue(co,x,p,d):
        if(d==0):
            s=0
            for i in range(0,p+1):
                s=s+co[i]*x**i
        else:
            s=0
            for i in range(0,p):
                s=s+co[i]*x**i
        return s
    co=np.zeros((5,5))
    cod=np.zeros((5,5))
    co,cod=shapeFunctions(p,co,cod,choice)
    if(bc1==2):
        force1=float(input("Enter force = "))
    elif (bc1==1):
        dis1=0
    else:
        spr1=float(input("Enter spring constant = "))
        dev1=float(input("Enter spring deviation = "))

    if(bc2==2):
        force2=0
    elif(bc2==1):
        dis2=float(input("Enter displacement = "))
    else:
        spr2=float(input("Enter spring constant = "))
        dev2=float(input("Enter spring deviation = "))

    h=1/n
```

```
    nodeLocations=np.zeros((n,2))
    for i in range(0,n):
        for j in range(0,2):
            if (i==j==0):
                continue
            elif (j%2==0):
                nodeLocations[i][j]=nodeLocations[i-1][j+1]
            else:
                nodeLocations[i][j]=nodeLocations[i][j-1]+h

    K=np.zeros((n*p+1,n*p+1))
    G=np.zeros((n*p+1,n*p+1))
    F=np.zeros((n*p+1,1))
    Q=np.zeros((n*p+1,1))
    for i in range(0,n):
        ke=np.zeros((p+1,p+1))
        ge=np.zeros((p+1,p+1))
        fe=np.zeros((p+1,1))
        sunod=(nodeLocations[i][0]+nodeLocations[i][1])/2
        aecof=np.array([[aeco[0][0]+aeco[0][1]*sunod+aeco[0][2]*sunod**2,aeco[0][1]*(h/2)+aeco[0][2]*h*sunod,aeco[0][2]*(h/4)]])
        ccof=np.array([[cco[0][0]+cco[0][1]*sunod+cco[0][2]*sunod**2,cco[0][1]*(h/2)+cco[0][2]*h*sunod,cco[0][2]*(h/4)]])
        fcof=np.array([[fco[0][0]+fco[0][1]*sunod+fco[0][2]*sunod**2,fco[0][1]*(h/2)+fco[0][2]*h*sunod,fco[0][2]*(h/4)]])
        ke,ge,fe=elementMatrix(aecof,ccof,fcof,p)

        if (i==0):
            K[:p+1,:p+1]+=ke
            G[:p+1,:p+1]+=ge
            F[:p+1,0:1]+=fe
        else:
            K[i*p:i*p+p+1,i*p:i*p+p+1]+=ke
            G[i*p:i*p+p+1,i*p:i*p+p+1]+=ge
            F[i*p:i*p+p+1,0:1]+=fe
    KF=K+G
    if (bc1==1):
        for i in range (1,n*p+1):
```

```
            F[i]=F[i]-dis1*KF[i][0]
        for i in range(0,n*p+1):
            for j in range (0,n*p+1):
                if(i==0 or j==0):
                    KF[i][j]=0
        KF[0][0]=1
        F[0][0]=dis1
        Q[0][0]=0
    if (bc2==1):
        for i in range (0,n*p):
            F[i]=F[i]-dis2*KF[i][n*p]
        for i in range(0,n*p+1):
            for j in range (0,n*p+1):
                if(i==n*p or j==n*p):
                    KF[i][j]=0
        KF[n*p][n*p]=1
        F[n*p][0]=dis2
        Q[n*p][0]=0
    if (bc1==2):
        Q[0][0]=force1
    if (bc2==2):
        Q[n*p][0]=force2
    if (bc1==3):
        KF[0][0]+=spr1
        Q[0][0]=spr1*dev1
    if (bc2==3):
        KF[n*p][n*p]+=spr2
        Q[n*p][0]=spr2*dev2

    U=np.linalg.inv(KF)@(F+Q)

    x=np.linspace(0,1,1000)
    yh=[]
    yhd=[]
    for i in range(0,n):
        for j in range(0,1000):
            if(x[j]>=nodeLocations[i][0] and x[j]<=nodeLocations[i][1]):
                aux=(2*x[j]-(nodeLocations[i][0]+nodeLocations[i][1]))/h
```

```
                fr=0
                frd=0
                b=0
                for m in range(i*p,i*p+p+1):
                    fr=fr+(U[m][0]*getValue(co[b],aux,p,0))
                    frd=frd+(U[m][0]*getValue(cod[b],aux,p,1))*(2/h)
                    b=b+1
                yh.append(fr)
                yhd.append(frd)
    ye=[]
    yed=[]
    i=0
    while (i<1):
        ye.append(((np.e**(-i))*(np.e**(i) - 1)*(-np.e**(i) + np.e**(2)))/(np.e
    **(2) + 1))
        yed.append((np.e**(2-i)-np.e**(i))/(np.e**(2)+1))
        i=i+0.001

    if (np.size(yh)<np.size(ye)):
        for i in range(0,(np.size(ye)-np.size(yh))):
            ye.pop()
            yed.pop()
            x=x[:-1]
    elif (np.size(ye)<np.size(yh)):
        for i in range(0,(np.size(yh)-np.size(ye))):
            yh.pop()
            yhd.pop()
    return yh,yhd,ye,yed,x

def strainEnergy(ae,fx,fxd,c):
    s=0
    for i in range(0,998):
        fx1=ae*fxd[i]**2+c*fx[i]**2
        fx2=ae*fxd[i+1]**2+c*fx[i]**2
        s=s+0.5*(fx1+fx2)*(x[i+1]-x[i])
    return s

numElements = [1,10,20,40,80,100]
```

```
order = [1,2,3,4]
choice = 1
aeco = np.zeros((1,3))
cco = np.zeros((1,3))
fco = np.zeros((1,3))
aep = 0

for i in range (0,aep+1):
    aeco[0][i]=1
cp=0
for i in range (0,cp+1):
    cco[0][0]=1
fp=0
for i in range (0,fp+1):
    fco[0][0]=1
bc1=1
bc2=2
for p in order:
    yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)
    line1, = plt.plot(x,yh, label="1 element", color='purple')
    yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
    line2, = plt.plot(x,yh, label="10 element", color='brown')
    yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
    line3, = plt.plot(x,yh, label="20 element", color='green')
    yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
    line4, = plt.plot(x,yh, label="40 element", color='red')
    yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
    line7, = plt.plot(x,yh, label="80 element", color='green')
    yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
    line5, = plt.plot(x,yh, label="100 element", color='blue')
    line6, = plt.plot(x,ye, label="exact", color='black', linestyle='dashed')
    plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])
    plt.title("FEM with exact solution")
    plt.xlabel("x")
    plt.ylabel("u")
    plt.show()
for p in order:
    yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)
```

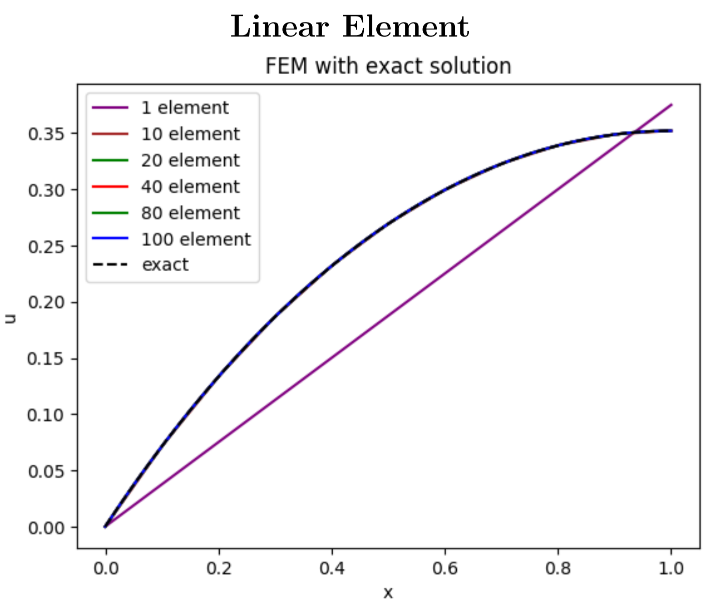
```
  #  line1, =plt.plot(x,yhd, label="1 element")
    yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
  #  line2, =plt.plot(x,yhd, label="10 element")
    yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
  #  line3, =plt.plot(x,yhd, label="20 element")
    yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
  #  line4, =plt.plot(x,yhd, label="40 element")
    yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
  #  line7, =plt.plot(x,yhd, label="80 element")
    yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
  #  line5, =plt.plot(x,yhd, label="100 element")
  #  line6, =plt.plot(x,yed, label="exact")
  #  plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])
  #  plt.title("Plot of derivatives")
  #  plt.xlabel("x")
  #  plt.ylabel("F")
  #  plt.show()
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
for n in numElements    :
    yh,yhd,ye,yed,x=FEM_hp(n,1,choice,bc1,bc2,aeco,cco,fco)
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
    energy.append(eg)
print(logElements,error)
line1,=plt.plot(logElements,error, label="linear",marker=".", color='purple')
```

```
# line1,=plt.plot(logElements,energy, label="linear",marker=".", color='purple
    ')
# line1,=plt.plot(logElements,logEnergyError, label="linear",marker=".", color
    ='purple')
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
for n in numElements    :
    yh,yhd,ye,yed,x=FEM_hp(n,2,choice,bc1,bc2,aeco,cco,fco)
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
    energy.append(eg)
print(logElements,error)
line2,=plt.plot(logElements,error, label="quadratic",marker=".", color='green')
# line2,=plt.plot(logElements,energy, label="quadratic",marker=".", color='
    green')
# line2,=plt.plot(logElements,logEnergyError, label="quadratic",marker=".",
    color='green')
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
for n in numElements    :
    yh,yhd,ye,yed,x=FEM_hp(n,3,choice,bc1,bc2,aeco,cco,fco)
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
```

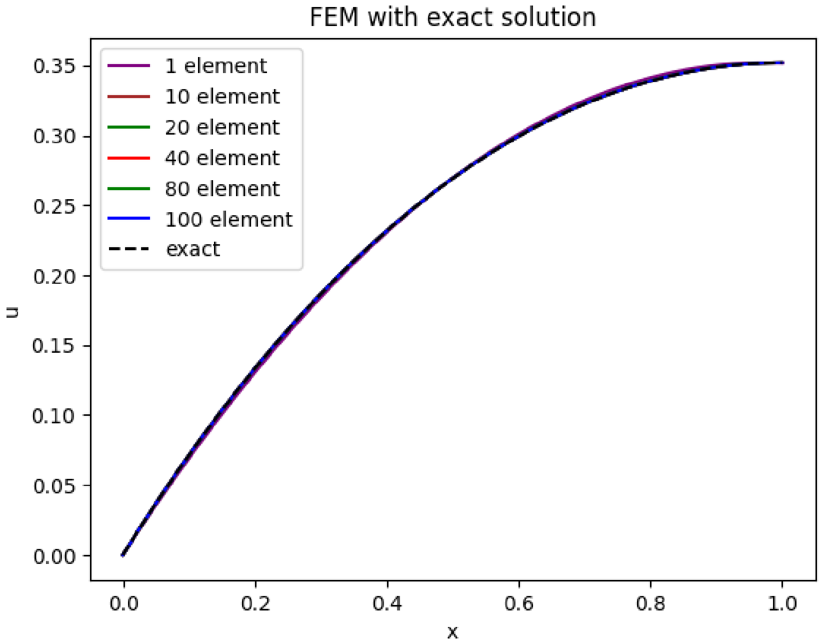
```
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
    energy.append(eg)
print(logElements,error)
line3,=plt.plot(logElements,error, label="cubic",marker=".", color='blue')
# line3,=plt.plot(logElements,energy, label="cubic",marker=".", color='blue')
# line3,=plt.plot(logElements,logEnergyError, label="cubic",marker=".", color='
    blue')
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
energyexact=[]
for n in numElements    :
    yh,yhd,ye,yed,x=FEM_hp(n,4,choice,bc1,bc2,aeco,cco,fco)
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    energyexact.append(strainEnergy(1,yed,ye,0))
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
    energy.append(eg)
print(logElements,error)
line4,=plt.plot(logElements,error, label="quartic",marker=".", color='orange')
```

```
# line4,=plt.plot(logElements,energy, label="quartic",marker=".", color='orange
    ')
# line4,=plt.plot(logElements,logEnergyError, label="quartic",marker=".", color
    ='orange')

plt.legend(handles=[line1, line2, line3, line4])
plt.title("Error with increasing number of elements")
plt.ylabel("ln(error)")
plt.xlabel("ln(No. of elements)")
plt.show()
# plt.title("Strain energy with increasing number of elements")
# plt.ylabel("ln(Strain energy)")
# plt.xlabel("ln(No. of elements)")
# plt.show()
# plt.title("Strain energy of the error with increasing number of elements")
# plt.ylabel("ln(Strain energy of the error)")
# plt.xlabel("ln(No. of elements)")
# plt.show()
# plt.title("Relative error with increasing number of elements")
# plt.ylabel("ln(Relative error)")
# plt.xlabel("ln(No. of elements)")
# plt.show()
```

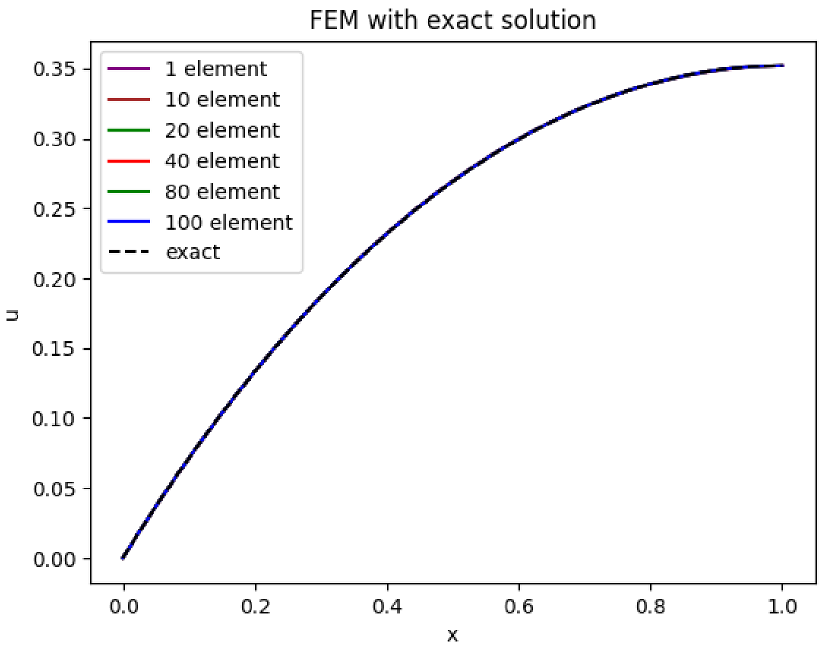
OUTPUT PLOTS



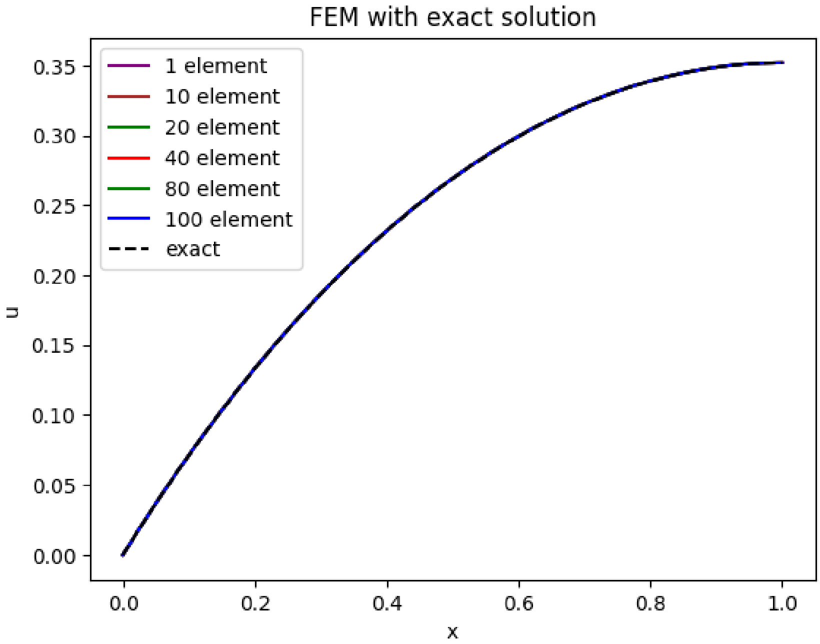
Quadratic Element



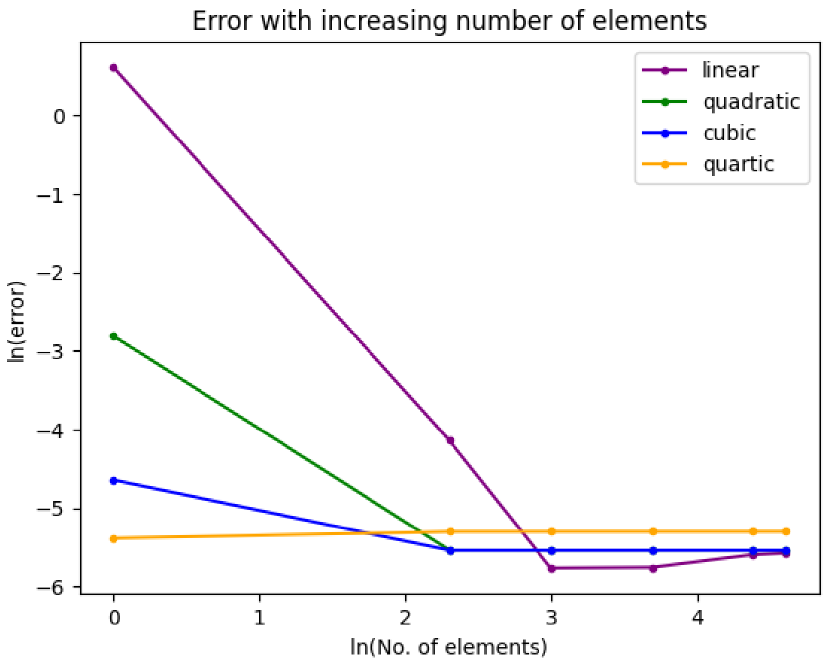
Cubic Element



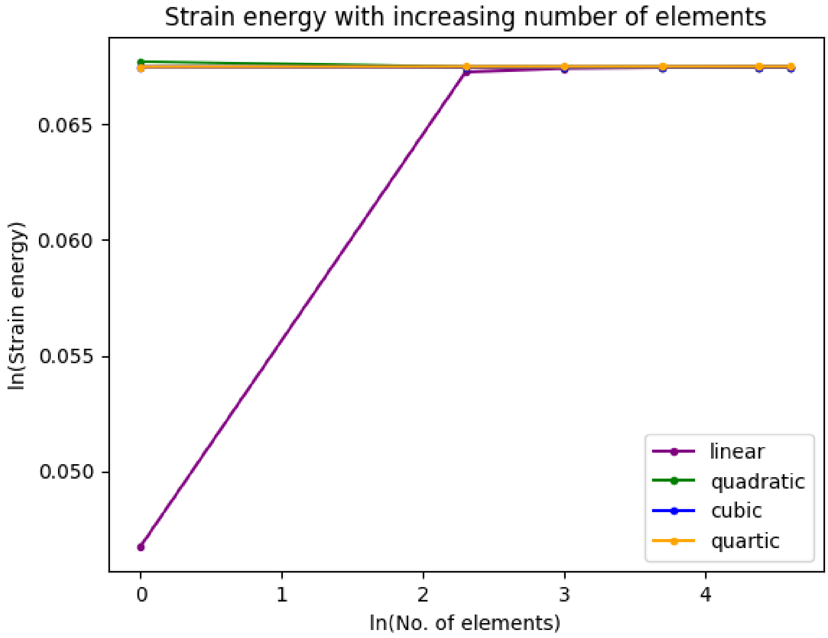
Quartic Element



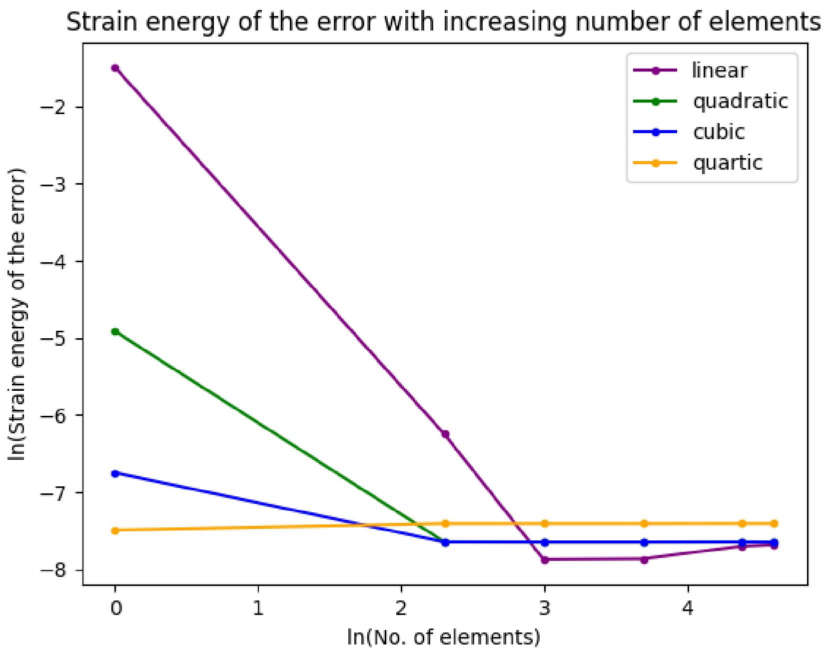
Error



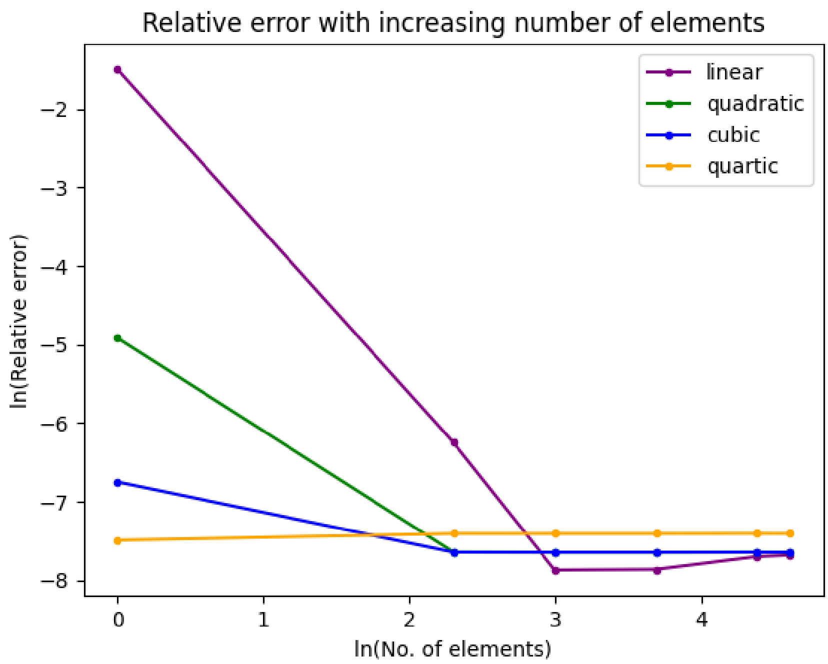
Strain Energy



Strain Energy of the error



Relative Error



INFERENCE FROM THE ABOVE PLOTS

As the discretization of the domain increases, the Finite Element Method (FEM) solution gradually converges towards the exact solution. This convergence is attributed to the finer mesh allowing for a more accurate representation of the exact solution. With a finer mesh, each element captures local variations more effectively, and as the mesh refinement progresses, these variations combine to yield a more faithful representation of the overall solution.

Furthermore, the choice of the order of approximation functions within each element significantly influences the accuracy of the FEM solution. Higher-order approximation functions are capable of better capturing complex variations in the solution. They excel in representing phenomena such as steep gradients or sharp changes in behavior. By increasing the order of approximation functions, the FEM can provide a closer approximation to the exact solution.

When employing quartic functions for approximation, the impact of the 6-point integration errors becomes more pronounced in the approximated solution. This is evident as we increase the number of elements in the discretization. The dominance of these errors underscores the importance of carefully managing integration schemes and considering higher-order approximations to improve the accuracy of the FEM solution.

4. Take $AE(x) = 1$, $c(x) = 0$ and $T(x) = \sin\frac{\pi}{L}x$ with $AE\frac{du}{dx}\Big|_{x=0} = \frac{1}{\pi}$ and $AE\frac{du}{dx}\Big|_{x=1} = k_L(\delta_L - u(L))$ with $k_L = 10$ and $\delta_L = 0$. Then repeat the exercise given a) through f) in Point 3.

PYTHON CODE WITH GIVEN INPUTS

```
import numpy as np
import matplotlib.pyplot as plt
def FEM_hp(n,p,choice,bc1,bc2,aeco,cco,fco):
    def shapeFunctions(p,co,cod,choice):
        if(choice==1):
            if (p==1):
                co=[[0.5,-0.5],
                    [0.5,0.5]]
                cod=[[-0.5],
                    [0.5]]
            elif (p==2):
                co=[[0,-0.5,0.5],
                    [1,0,-1],
                    [0,0.5,0.5]]
                cod=[[-0.5,1],
                    [0,-2],
                    [0.5,1]]
            elif (p==3):
                co=[[-0.0625,0.0625,0.5625,-0.5625],
                    [0.5625,-1.6875,-0.5625,1.6875],
                    [0.5625,1.6875,-0.5625,-1.6875],
                    [-0.0625,-0.0625,0.5625,0.5625]]
                cod=[[0.0625,1.125,-1.6875],
                    [-1.6875,-1.125,5.0625],
                    [1.6875,-1.125,-5.0625],
                    [-0.0625,1.125,1.6875]]
            elif (p==4):
                co=[[0,0.1667,-0.1667,-0.6667,0.6667],
                    [0,-1.3333,2.6667,1.3333,-2.6667],
```

```
                    [1,0,-5,0,4],
                    [0,1.3333,2.6667,-1.3333,-2.6667],
                    [0,-0.1667,-0.1667,0.6667,0.6667]]
                cod=[[0.1667,-0.3333,-2,2.6667],
                    [-1.3333,5.3333,4,-10.6667],
                    [0,-10,0,16],
                    [1.3333,5.3333,-4,-10.6667],
                    [-0.1667,-0.3333,2,2.6667]]
        else:
            if (p==1):
                co=[[0.5,-0.5],
                    [0.5,0.5]]
                cod=[[-0.5],
                    [0.5]]
            elif (p==2):
                co=[[0.5,-0.5,0,0,0],
                    [-0.6124,0,0.6124,0,0],
                    [0.5,0.5,0,0,0]]
                cod=[[-0.5,0],
                    [0,1.2247],
                    [0.5,0]]
            elif (p==3):
                co=[[0.5,-0.5,0,0,0],
                    [0,-0.7906,0,0.7906,0],
                    [-0.6124,0,0.6124,0,0],
                    [0.5,0.5,0,0,0]]
                cod=[[-0.5,0,0,0],
                    [-0.7906,0,2.3717,0],
                    [0,1.2247,0,0],
                    [0.5,0,0,0]]
            elif (p==4):
                co=[[0.5,-0.5,0,0,0],
                    [0.2338,0,-1.4031,0,1.1693],
                    [0,-0.7906,0,0.7906,0],
                    [-0.6124,0,0.6124,0,0],
                    [0.5,0.5,0,0,0]]
                cod=[[-0.5,0,0,0],
                    [0,-2.8062,0,4.6771],
```

```
                    [-0.7906,0,2.3717,0],
                    [0,1.2247,0,0],
                    [0.5,0,0,0]]
    return co,cod
def integrate(re):
    coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
    coi=np.append(coi, [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
    s=0
    for i in range(0,6):
        su=0
        for j in range(0,int(re.shape[0])):
            su=su+(re[j]*coi[i][0]**j)
        s=s+su*coi[i][1]
    return s
def elementMatrix(ae,c,f,p):
    for i in range(0, p+1):
        for j in range(0, p+1):
            ke[i][j]=(2/h)*integrate(np.polynomial.polynomial.polymul(np.polynomial.polynomial.polymul(cod[i],cod[j]),ae[0]))
            ge[i][j]=(h/2)*integrate(np.polynomial.polynomial.polymul(np.polynomial.polynomial.polymul(co[i],co[j]),c[0]))
        fe[i][0]=(h/2)*integrate(np.polynomial.polynomial.polymul(co[i],f[0]))
    return ke,ge,fe
def getValue(co,x,p,d):
    if(d==0):
        s=0
        for i in range(0,p+1):
            s=s+co[i]*x**i
    else:
        s=0
        for i in range(0,p):
            s=s+co[i]*x**i
    return s
co=np.zeros((5,5))
cod=np.zeros((5,5))
co,cod=shapeFunctions(p,co,cod,choice)
```

```
    if(bc1==2):
        force1=0.3183
    elif (bc1==1):
        dis1=0
    else:
        spr1=float(input("Enter spring constant= "))
        dev1=float(input("Enter spring deviation= "))

    if(bc2==2):
        force2=0
    elif(bc2==1):
        dis2=float(input("Enter displacement= "))
    else:
        spr2=10
        dev2=0
    h=1/n
    nodeLocations=np.zeros((n,2))
    for i in range(0,n):
        for j in range(0,2):
            if (i==j==0):
                continue
            elif (j%2==0):
                nodeLocations[i][j]=nodeLocations[i-1][j+1]
            else:
                nodeLocations[i][j]=nodeLocations[i][j-1]+h

    K=np.zeros((n*p+1,n*p+1))
    G=np.zeros((n*p+1,n*p+1))
    F=np.zeros((n*p+1,1))
    Q=np.zeros((n*p+1,1))
    for i in range(0,n):
        ke=np.zeros((p+1,p+1))
        ge=np.zeros((p+1,p+1))
        fe=np.zeros((p+1,1))
        sunod=(nodeLocations[i][0]+nodeLocations[i][1])/2
        aecof=np.array([[aeco[0][0]+aeco[0][1]*sunod+aeco[0][2]*sunod**2,aeco[0][1]*(h/2)+aeco[0][2]*h*sunod,aeco[0][2]*(h/4)]])
        ccof=np.array([[cco[0][0]+cco[0][1]*sunod+cco[0][2]*sunod**2,cco
```

```
[0][1]*(h/2)+cco[0][2]*h*sunod,cco[0][2]*(h/4)]])
    fcof=np.array([[fco[0][0]+fco[0][1]*sunod+fco[0][2]*sunod**2+fco[0][3]*sunod**3+fco[0][5]*sunod**5+fco[0][7]*sunod**7+fco[0][9]*sunod**9, fco[0][1]*(h/2)+fco[0][2]*h*sunod+1.5*fco[0][3]*h*sunod**2+2.5*h*fco[0][5]*sunod**4+3.5*h*sunod**6*fco[0][7]+4.5*h*(sunod**8)*fco[0][9], fco[0][2]*(h**2/4)+0.75*fco[0][3]*(h**2)*sunod+2.5*h**2*sunod**3*fco[0][5]+(21/4)*h**2*sunod**5*fco[0][7]+9*h**2*sunod**7*fco[0][9], 0.375*(h**3)*fco[0][3]+1.25*h**3*sunod**2*fco[0][5]+(35/8)*h**3*sunod**4*fco[0][7]+10.5**h**3*sunod**6*fco[0][9], (5/8)*(h**4)*sunod*fco[0][5]+(35/16)*(h**4)*sunod**3*fco[0][7]+(63/8)*(h**4)*sunod**5*fco[0][5], (1/32)*(h**5)*fco[0][5]+(21/32)*(h**5)*sunod**2*fco[0][7]+(63/16)*(h**5)*sunod**4*fco[0][9], (7/64)*h**6*sunod*fco[0][7]+(21/16)*(h**6)*sunod**3*fco[0][9], (1/128)*(h**7)*fco[0][7]+(9/32)*(h**7)*sunod**2*fco[0][9], (9/256)*(h**8)*sunod*fco[0][9],(1/512)*(h**9)*fco[0][9]]])
    ke,ge,fe=elementMatrix(aecof,ccof,fcof,p)

    if (i==0):
        K[:p+1,:p+1]+=ke
        G[:p+1,:p+1]+=ge
        F[:p+1,0:1]+=fe
    else:
        K[i*p:i*p+p+1,i*p:i*p+p+1]+=ke
        G[i*p:i*p+p+1,i*p:i*p+p+1]+=ge
        F[i*p:i*p+p+1,0:1]+=fe
KF=K+G
if (bc1==1):
    for i in range (1,n*p+1):
        F[i]=F[i]-dis1*KF[i][0]
    for i in range(0,n*p+1):
        for j in range (0,n*p+1):
            if(i==0 or j==0):
                KF[i][j]=0
    KF[0][0]=1
    F[0][0]=dis1
    Q[0][0]=0
if (bc2==1):
    for i in range (0,n*p):
        F[i]=F[i]-dis2*KF[i][n*p]
```

```
        for i in range(0,n*p+1):
            for j in range (0,n*p+1):
                if(i==n*p or j==n*p):
                    KF[i][j]=0
        KF[n*p][n*p]=1
        F[n*p][0]=dis2
        Q[n*p][0]=0
    if (bc1==2):
        Q[0][0]=force1
    if (bc2==2):
        Q[n*p][0]=force2
    if (bc1==3):
        KF[0][0]+=spr1
        Q[0][0]=spr1*dev1
    if (bc2==3):
        KF[n*p][n*p]+=spr2
        Q[n*p][0]=spr2*dev2

    U=np.linalg.inv(KF)@(F+Q)

    x=np.linspace(0,1,1000)
    yh=[]
    yhd=[]
    for i in range(0,n):
        for j in range(0,1000):
            if(x[j]>=nodeLocations[i][0] and x[j]<=nodeLocations[i][1]):
                aux=(2*x[j]-(nodeLocations[i][0]+nodeLocations[i][1]))/h
                fr=0
                frd=0
                b=0
                for m in range(i*p,i*p+p+1):
                    fr=fr+(U[m][0]*getValue(co[b],aux,p,0))
                    frd=frd+(U[m][0]*getValue(cod[b],aux,p,1))*(2/h)
                    b=b+1
                yh.append(fr)
                yhd.append(frd)
    ye=[]
    yed=[]
```

```
    i=0
    while (i<1):
        ye.append(((np.sin(np.pi*i))/(np.pi**2))+(0.1/np.pi))
        yed.append((np.cos(np.pi*i))/np.pi)
        i=i+0.001

    if (np.size(yh)<np.size(ye)):
        for i in range(0,(np.size(ye)-np.size(yh))):
            ye.pop()
            yed.pop()
            x=x[:-1]
    elif (np.size(ye)<np.size(yh)):
        for i in range(0,(np.size(yh)-np.size(ye))):
            yh.pop()
            yhd.pop()
    return yh,yhd,ye,yed,x

def strainEnergy(ae,fx,fxd,c):
    s=0
    for i in range(0,998):
        fx1=ae*fxd[i]**2+c*fx[i]**2
        fx2=ae*fxd[i+1]**2+c*fx[i]**2
        s=s+0.5*(fx1+fx2)*(x[i+1]-x[i])
    return s

numElements=[1,10,20,40,80,100]
order=[1,2,3,4]
choice=1
aeco=np.zeros((1,3))
cco=np.zeros((1,3))
fco=np.zeros((1,10))
aeco[0][1]=1
cco[0][0]=0
fco[0][0]=0
fco[0][1]=3.1416
fco[0][2]=0
fco[0][3]=-5.1677
fco[0][5]=2.5501
```

```
fco[0][7]=-0.599264
fco[0][9]=0.08214588
bc1=2
bc2=3
for p in order:
    yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)
    line1, =plt.plot(x,yh, label="1 element", color='purple')
    yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
    line2, =plt.plot(x,yh, label="10 element", color='brown')
    yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
    line3, =plt.plot(x,yh, label="20 element", color='green')
    yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
    line4, =plt.plot(x,yh, label="40 element", color='red')
    yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
    line7, =plt.plot(x,yh, label="80 element", color='orange')
    yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
    line5, =plt.plot(x,yh, label="100 element", color='blue')
    line6, =plt.plot(x,ye, label="exact", color='black', linestyle='dashed')
    plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])
    plt.title("Plot of FEM with exact solution")
    plt.xlabel("x")
    plt.ylabel("u")
    plt.show()
for p in order:
    yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)
  #  line1, =plt.plot(x,yhd, label="1 element")
    yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
 #   line2, =plt.plot(x,yhd, label="10 element")
    yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
#    line3, =plt.plot(x,yhd, label="20 element")
    yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
  #  line4, =plt.plot(x,yhd, label="40 element")
    yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
 #   line7, =plt.plot(x,yhd, label="80 element")
    yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
 #   line5, =plt.plot(x,yhd, label="100 element")
#    line6, =plt.plot(x,yed, label="exact")
#    plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])
```

```
 #   plt.title("Plot of derivatives")
  #  plt.xlabel("x")
    # plt.ylabel("F")
#     plt.show()
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
for n in numElements:
    yh,yhd,ye,yed,x=FEM_hp(n,1,choice,bc1,bc2,aeco,cco,fco)
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
    energy.append(eg)
print(logElements,error)
line1,=plt.plot(logElements,error, label="linear",marker=".", color='purple')
# line1,=plt.plot(logElements,energy, label="linear",marker=".", color='purple
    ')
# line1,=plt.plot(logElements,energyerr, label="linear",marker=".", color='
    purple')
# line1,=plt.plot(logElements,logEnergyError, label="linear",marker=".", color
    ='purple')
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
for n in numElements:
    yh,yhd,ye,yed,x=FEM_hp(n,2,choice,bc1,bc2,aeco,cco,fco)
```

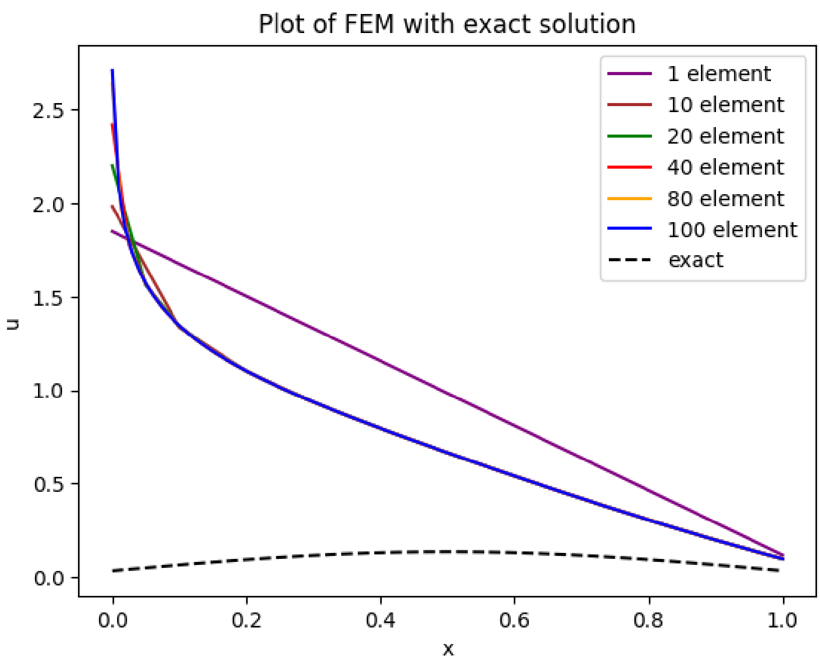
```
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
    energy.append(eg)
print(logElements,error)
line2,=plt.plot(logElements,error, label="quadratic",marker=".", color='green')
# line2,=plt.plot(logElements,energy, label="quadratic",marker=".", color='
    green')
# line2,=plt.plot(logElements,energyerr, label="quadratic",marker=".", color='
    green')
# line2,=plt.plot(logElements,logEnergyError, label="quadratic",marker=".",
    color='green')
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
for n in numElements:
    yh,yhd,ye,yed,x=FEM_hp(n,3,choice,bc1,bc2,aeco,cco,fco)
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
```

```
    energy.append(eg)
print(logElements,error)
line3,=plt.plot(logElements,error, label="cubic",marker=".", color='blue')
# line3,=plt.plot(logElements,energy, label="cubic",marker=".", color='blue')
# line3,=plt.plot(logElements,energyerr, label="cubic",marker=".", color='blue
    ')
# line3,=plt.plot(logElements,logEnergyError, label="cubic",marker=".", color='
    blue')
error=[]
logElements=[]
energy=[]
energyerr=[]
logEnergyError=[]
energyexact=[]
for n in numElements:
    yh,yhd,ye,yed,x=FEM_hp(n,4,choice,bc1,bc2,aeco,cco,fco)
    er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
    uex=np.linalg.norm(np.array(ye))
    energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
    error.append(er)
    logEnergyError.append(np.log(energynorm/uex))
    logElements.append(np.log(n))
    strar=np.array(ye)-np.array(yh)
    strard=np.array(yed)-np.array(yhd)
    eg=strainEnergy(1,yhd,yh,0)
    energyexact.append(strainEnergy(1,yed,ye,0))
    erreg=strainEnergy(1,strard,strar,0)
    energyerr.append(erreg)
    energy.append(eg)
print(logElements,error)
line4,=plt.plot(logElements,error, label="quartic",marker=".", color='orange')
# line4,=plt.plot(logElements,energy, label="quartic",marker=".", color='orange
    ')
# line4,=plt.plot(logElements,energyerr, label="quartic",marker=".", color='
    orange')
# line4,=plt.plot(logElements,logEnergyError, label="quartic",marker=".", color
    ='orange')

```

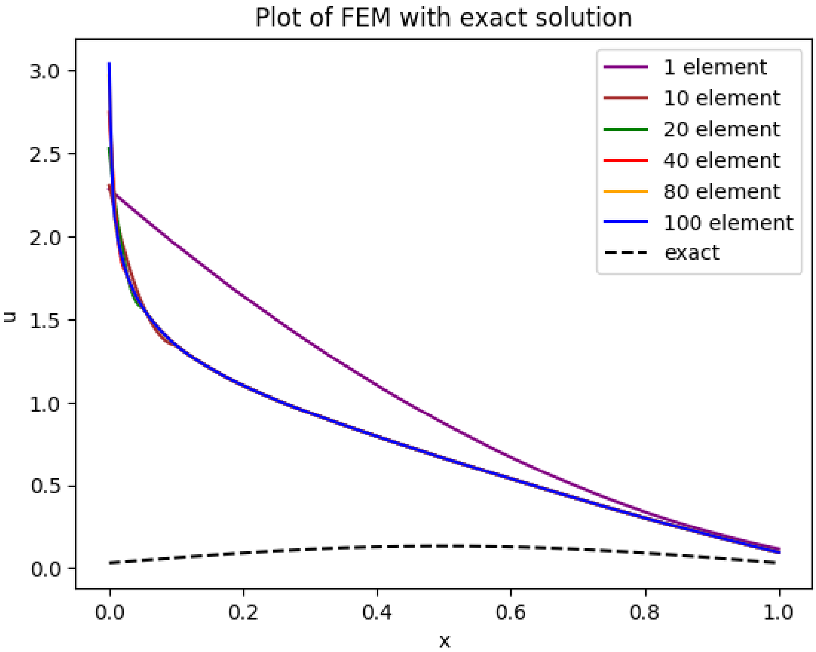
```
plt.legend(handles=[line1, line2, line3, line4])
plt.title("Error with increasing number of elements")
plt.ylabel("ln(error)")
plt.xlabel("ln(No. of elements)")
plt.show()
# plt.title("Strain energy with increasing number of elements")
# plt.ylabel("ln(Strain energy)")
# plt.xlabel("ln(No. of elements)")
# plt.show()
# plt.title("Strain energy of the error with increasing number of elements")
# plt.ylabel("ln(Strain energy of the error)")
# plt.xlabel("ln(No. of elements)")
# plt.show()
# plt.title("Relative error with increasing number of elements")
# plt.ylabel("ln(Relative error)")
# plt.xlabel("ln(No. of elements)")
# plt.show()
```

OUTPUT PLOTS

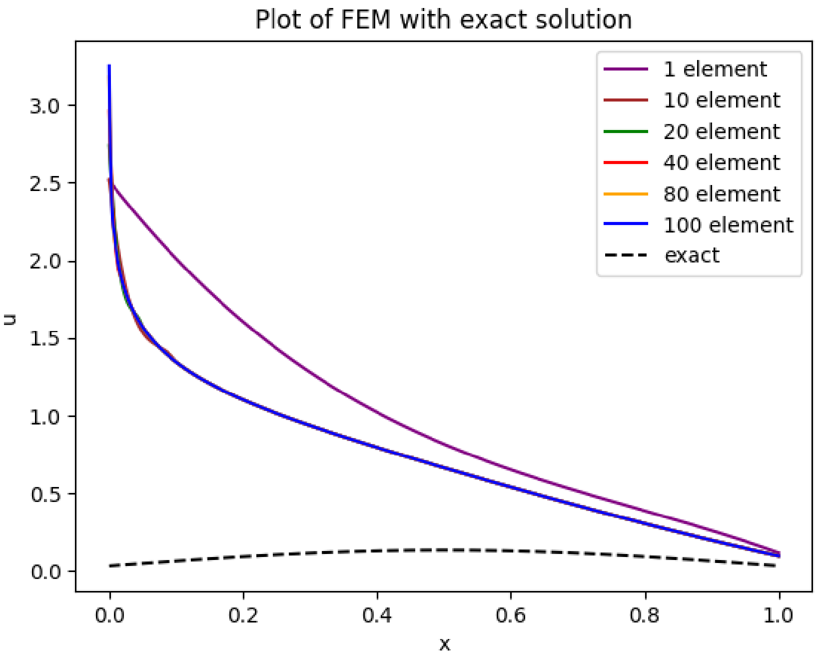
Linear Element



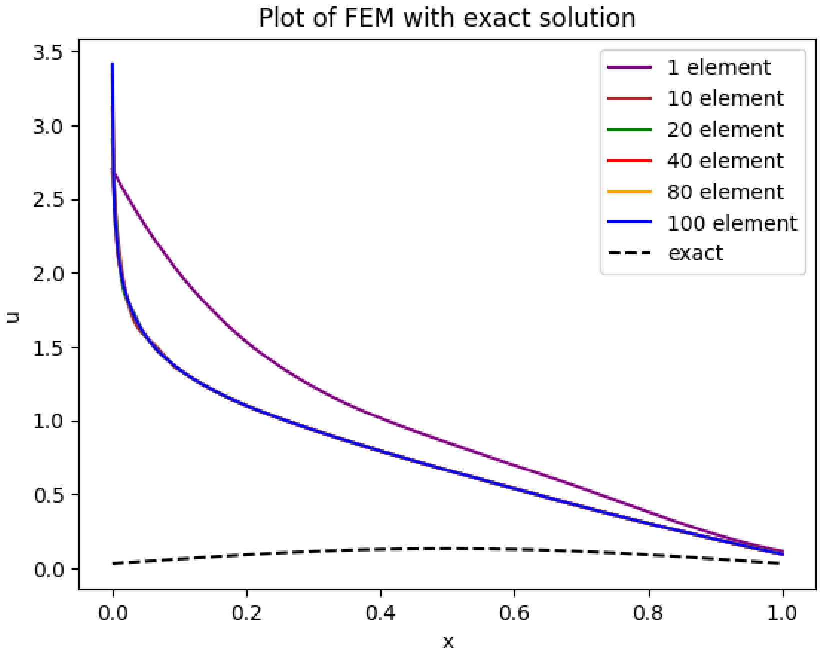
Quadratic Element



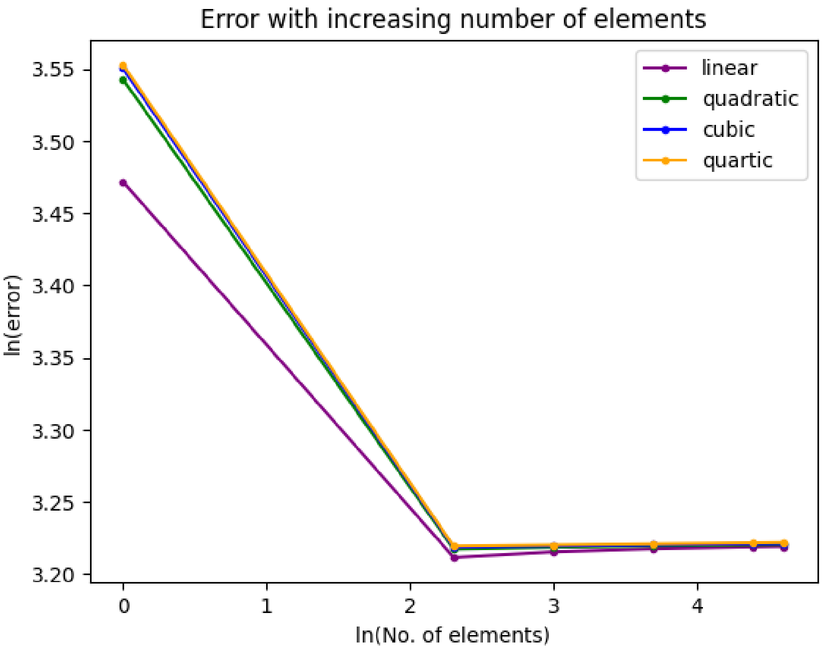
Cubic Element



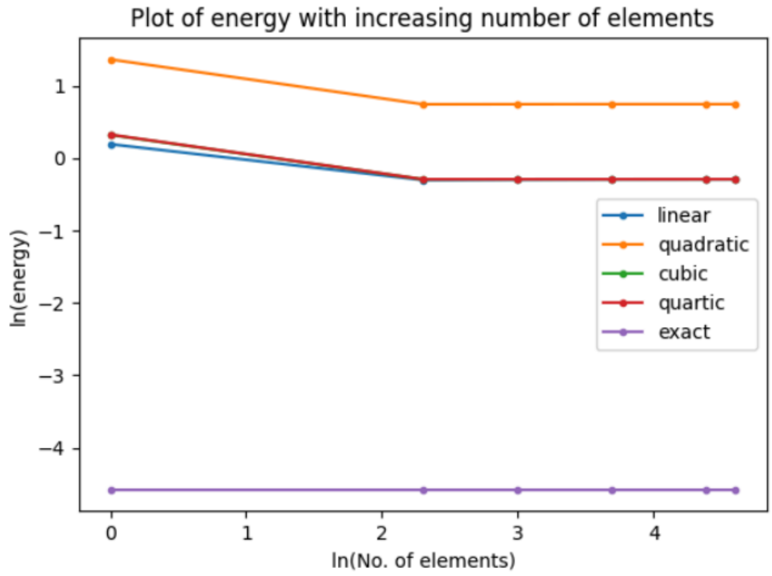
Quartic Element



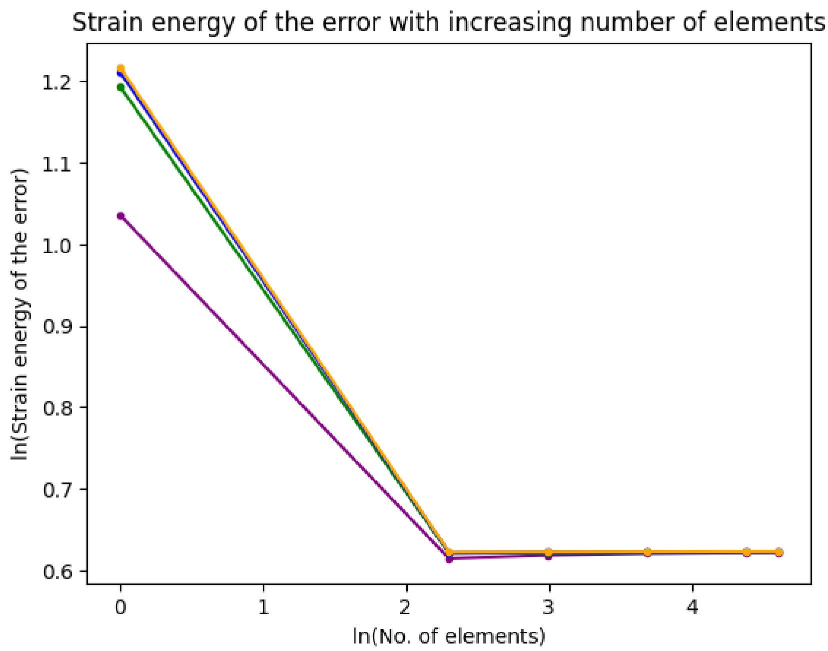
Error



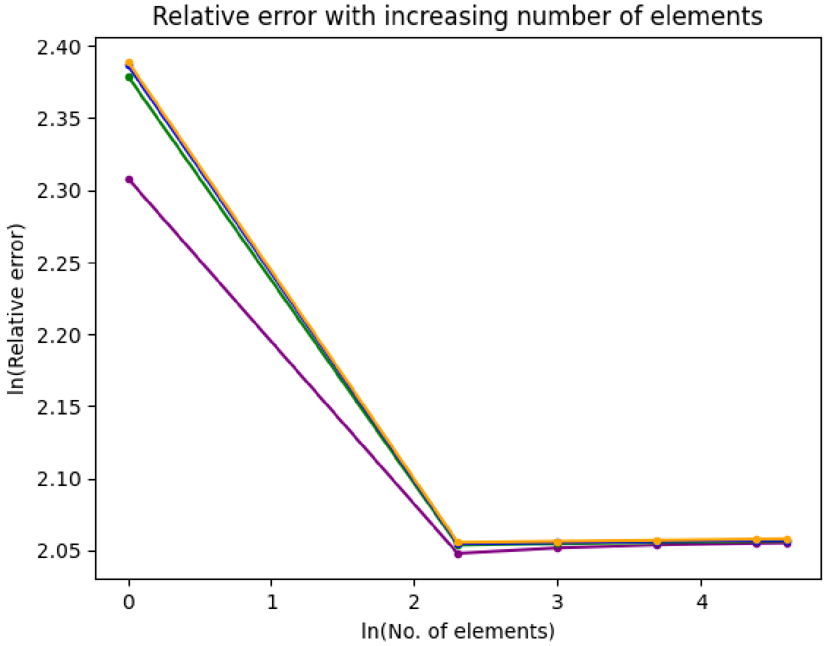
Strain Energy



Strain Energy of the Error



Relative Error



INFERENCE FROM THE ABOVE PLOTS

Incorporating a forcing term as $\sin(\pi x)$ necessitates a Taylor series approximation extended up to at least the 9th power. Failing to extend the series adequately, such as truncating it at the 3rd power, results in significant discrepancies in matching the boundary conditions. This discrepancy stems from the inherent limitations of our code, which is not explicitly designed to handle sinusoidal forces. It's worth noting that employing sinusoidal approximating functions could potentially yield solutions closer to the exact solution.

By utilizing 9th-order polynomials for approximating the forcing term, the error in boundary conditions is notably diminished. However, due to our reliance on a 6-point integration scheme over the master element, errors escalate as we progress from linear to quartic approximations. Unfortunately, this error is inherent and cannot be entirely mitigated, thus leading to solutions that appear increasingly divergent from the exact solution.