

# 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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def shape_functions(h):
5     """Compute coefficients of shape functions."""
6     co=[[0.5, -0.75, 0, 0.25],[-0.125*h, 0.125*h, 0.125*h, -0.125*h],[0.5,
7         0.75, 0, -0.25],[0.125*h, 0.125*h, -0.125*h, -0.125*h]]
8     cod=[[-0.75, 0, 0.75], [0.125*h, 0.25*h, -0.375*h],[0.75, 0, -0.75],[0.125*
9         h, -0.25*h, -0.375*h]]
10    codd=[[0, 1.5], [0.25*h, -0.75*h], [0, -1.5], [-0.25*h, -0.75*h]]
```

```

9     codddd=[[1.5],[-0.75*h],[-1.5],[-0.75*h]]
10     return co, cod, codd, codddd
11
12 def integration_approximation(re):
13     """Approximate integration coefficients."""
14     coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
15     coi=np.append(coi,
16     [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
17     s=0
18     for i in range(0, 6):
19         su = 0
20         for j in range(0, int(re.shape[0])):
21             su += re[j] * coi[i][0] ** j
22             s += su * coi[i][1]
23     return s
24
25 def elemental_matrix(f, co, h):
26     """Compute elemental matrix."""
27     fe = np.zeros((4, 1))
28     for i in range(0, 4):
29         fe[i][0] = (h / 2) * integration_approximation(np.polynomial.polynomial
30         .polymul(co[i], f[0]))
31     return fe
32
33 def get_value(co, x, p, d):
34     """Map element position to global matrix position."""
35     s = 0
36     for i in range(0, p - d +1):
37         s += co[i] * x ** i
38     return s
39
40 def main():
41     n = int(input("Enter number of elements: "))
42     h = 1 / n
43     co, cod, codd, codddd = shape_functions(h)
44     bc1 = int(input("Type of BC at x=0 (1 or 2): "))
45     bc2 = int(input("Type of BC at x=1 (1 or 2): "))
46     if bc1 == 2:

```

```

45     force1 = float(input("Enter force: "))
46     moment1 = float(input("Enter moment: "))
47 else:
48     dis1 = float(input("Enter displacement: "))
49     slope1 = float(input("Enter slope: "))
50 if bc2 == 2:
51     force2 = float(input("Enter force: "))
52     moment2 = float(input("Enter moment: "))
53 else:
54     dis2 = float(input("Enter displacement: "))
55     slope2 = float(input("Enter slope: "))
56 fp = int(input("Enter order of t: "))
57 fco = np.zeros((1, fp+1))
58 for i in range(0, fp + 1):
59     fco[0][i] = int(input("Enter coeff: "))
60 if n % 2 == 0:
61     forcemid = int(input("Enter force at midpoint: "))
62     momentmid = int(input("Enter moment at midpoint: "))
63 nodloc = np.zeros((n, 2))
64 for i in range(0, n):
65     for j in range(0, 2):
66         if i == j == 0:
67             continue
68         elif j % 2 == 0:
69             nodloc[i][j] = nodloc[i - 1][j + 1]
70         else:
71             nodloc[i][j] = nodloc[i][j - 1] + h
72 K = np.zeros((2 * n + 2, 2 * n + 2))
73 F = np.zeros((2 * n + 2, 1))
74 Q = np.zeros((2 * n + 2, 1))
75 ke = (1000 / (3 * h ** 3)) * np.array([[6, -3 * h, -6, -3 * h],
76                                         [-3 * h, 2 * h ** 2, 3 * h, h ** 2],
77                                         [-6, 3 * h, 6, 3 * h],
78                                         [-3 * h, h ** 2, 3 * h, 2 * h **
2]])
79 re = np.zeros((1, 5))
80 for i in range(0, n):
81     fe = np.zeros((4, 1))

```

```

82     sunod = (nodloc[i][0] + nodloc[i][1]) / 2
83     fcof = np.array([[fco[0][0] + fco[0][1] * sunod + fco[0][2] * sunod *
152     2, fco[0][1] * (h / 2) + fco[0][2] * h * sunod,
84                     fco[0][2] * (h / 4)]])
85     fe = elemental_matrix(fcof, co, h)
86     if i == 0:
87         K[:4, :4] += ke
88         F[:4, 0:1] += fe
89     else:
90         K[2 * i:2 * (i + 2), 2 * i:2 * (i + 2)] += ke
91         F[2 * i:2 * (i + 2), 0:1] += fe
92     KF = K
93     if bc1 == 1:
94         for i in range(1, 2 * n + 2):
95             F[i] = F[i] - dis1 * KF[i][0]
96         for i in range(0, 2 * n + 2):
97             for j in range(0, n * 2 + 2):
98                 if i == 0 or j == 0:
99                     KF[i][j] = 0
100     KF[0][0] = 1
101     F[0][0] = dis1
102     Q[0][0] = 0
103     for i in range(2, 2 * n + 2):
104         F[i] = F[i] - slope1 * KF[i][1]
105     for i in range(0, 2 * n + 2):
106         for j in range(0, 2 * n + 2):
107             if i == 1 or j == 1:
108                 KF[i][j] = 0
109     KF[1][1] = 1
110     F[1][0] = slope1
111     Q[1][0] = 0
112     if bc2 == 1:
113         for i in range(0, 2 * n + 1):
114             F[i] = F[i] - slope2 * KF[i][2 * n + 1]
115         for i in range(0, n * 2 + 2):
116             for j in range(0, 2 * n + 2):
117                 if i == 2 * n + 1 or j == n * 2 + 1:
118                     KF[i][j] = 0

```

```

119     KF[n * 2 + 1][n * 2 + 1] = 1
120     F[n * 2 + 1][0] = slope2
121     Q[n * 2 + 1][0] = 0
122     for i in range(0, 2 * n):
123         F[i] = F[i] - dis2 * KF[i][2 * n]
124     for i in range(0, n * 2 + 2):
125         for j in range(0, 2 * n + 2):
126             if i == 2 * n or j == n * 2:
127                 KF[i][j] = 0
128     KF[n * 2][n * 2] = 1
129     F[n * 2][0] = dis2
130     Q[n * 2][0] = 0
131     if bc1 == 2:
132         Q[0][0] = force1
133         Q[1][0] = -moment1
134     if bc2 == 2:
135         Q[2 * n][0] = force2
136         Q[2 * n + 1][0] = -moment2
137     if n % 2 == 0:
138         Q[n][0] = forcemid
139         Q[n + 1][0] = -momentmid
140     U = np.linalg.inv(KF) @ (F + Q)
141
142     # Plotting
143     x = np.linspace(0, 1, 1000)
144     yh, yhslope, moment, shear, stress = [], [], [], [], []
145
146     for i in range(n):
147         for j in range(1000):
148             if nodloc[i][0] <= x[j] <= nodloc[i][1]: # Check if x[j] is within
149                 the element range
150                 aux = (2 * x[j] - (nodloc[i][0] + nodloc[i][1])) / h
151                 w, wslope, mom, sh, st, b = 0, 0, 0, 0, 0, 0
152                 for m in range(i * 2, i * 2 + 4):
153                     w=w+(U[m][0]*get_value(co[b],aux,3,0))
154                     wslope=wslope+(U[m][0]*get_value(cod[b],aux,3,1))*(2/h)
155                     mom=mom+(500/3)*(U[m][0]*get_value(codd[b],aux,3,2))*(2/h)

```

\*\*2

```

155         sh=sh+(-500/3)*(U[m][0]*get_value(coddd[b],aux,3,3))*(2/h)
156
157     **3
158
159         st=st+(-1000)*(U[m][0]*get_value(codd[b],aux,3,2))*(2/h)**2
160
161         b=b+1
162
163         yh.append(w)
164         yhslope.append(wslope)
165         moment.append(mom)
166         shear.append(sh)
167         stress.append(st)
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
# 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")

```

```

192     line2, = plt.plot(x, ye, label="Exact")
193     plt.legend(handles=[line1, line2])
194     plt.title("Plot of deflection with x")
195     plt.ylabel("Deflection")
196     plt.xlabel("x")
197     plt.show()
198
199     line1, = plt.plot(x, yhslope, label="FEM")
200     line2, = plt.plot(x, yed, label="Exact")
201     plt.legend(handles=[line1, line2])
202     plt.title("Plot of slope with x")
203     plt.ylabel("Slope")
204     plt.xlabel("x")
205     plt.show()
206
207     plt.plot(x, moment)
208     plt.title("Plot of moment with x")
209     plt.ylabel("Moment")
210     plt.xlabel("x")
211     plt.show()
212
213     plt.plot(x, shear)
214     plt.title("Plot of shear force with x")
215     plt.ylabel("Shear Force")
216     plt.xlabel("x")
217     plt.show()
218
219     plt.plot(x, stress)
220     plt.title("Plot of stress with x")
221     plt.ylabel("Stress")
222     plt.xlabel("x")
223     plt.show()
224
225 if __name__ == "__main__":
226     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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def solve_beam_finite_element(n_elements, bc_start, bc_end, force_coefficients)
   :
5     def shape_functions(co,cod,codd,coddd,h):
6         # Coefficients of shape functions
7         co = [[0.5, -0.75, 0, 0.25],[-0.125*h, 0.125*h, 0.125*h, -0.125*h],
8               [0.5, 0.75, 0, -0.25],[0.125*h, 0.125*h, -0.125*h, -0.125*h]]
9         cod = [[-0.75, 0, 0.75], [0.125*h, 0.25*h, -0.375*h],
10               [0.75, 0, -0.75],[0.125*h, -0.25*h, -0.375*h]]
11         codd = [[0, 1.5], [0.25*h, -0.75*h], [0, -1.5], [-0.25*h, -0.75*h]]
12         coddd = [[1.5], [-0.75*h], [-1.5], [-0.75*h]]
13         return co, cod, codd, coddd
14
15     def integrate(re):
16         # Integration approximating coefficients
17         coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
18         coi=np.append(coi,
19                       [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
20         s = 0
21         for i in range(0, 6):
22             su = 0
23             for j in range(0, int(re.shape[0])):
24                 su = su + (re[j] * coi[i][0]**j)
25             s = s + su * coi[i][1]
```

```

25     return s
26
27 def element_matrix(f):
28     # Polynomial multiplication of coefficients in an elemental matrix
29     for i in range(0, 4):
30         element_forces[i][0] = (h / 2) * integrate(np.polynomial.polynomial.
31             polymul(co[i], f[0]))
32     return element_forces
33
34 def get_value(co, x, p, d):
35     # Mapping of element position to global matrix position
36     if d == 0:
37         s = 0
38         for i in range(0, p+1):
39             s = s + co[i] * x**i
40     elif d == 1:
41         s = 0
42         for i in range(0, p):
43             s = s + co[i] * x**i
44     elif d == 2:
45         s = 0
46         for i in range(0, p-1):
47             s = s + co[i] * x**i
48     else:
49         s = 0
50         for i in range(0, p-2):
51             s = s + co[i] * x**i
52     return s
53
54 h = 1 / n_elements
55 co = np.zeros((5, 5))
56 cod = np.zeros((5, 5))
57 codd = np.zeros((5, 5))
58 coddd = np.zeros((5, 5))
59 co, cod, codd, coddd = shape_functions(co, cod, codd, coddd, h)
60
61 if bc_start == 2:
62     force1 = float(input("Enter force= "))

```

```

62     moment1 = float(input("Enter moment= "))
63 else:
64     displacement1 = 0
65     slope1 = 0
66 if bc_end == 2:
67     force2 = 5
68     moment2 = 5
69 else:
70     displacement2 = float(input("Enter displacement= "))
71     slope2 = float(input("Enter slope= "))
72
73 node_locations = np.zeros((n_elements, 2))
74 for i in range(0, n_elements):
75     for j in range(0, 2):
76         if (i == j == 0):
77             continue
78         elif (j % 2 == 0):
79             node_locations[i][j] = node_locations[i - 1][j + 1]
80         else:
81             node_locations[i][j] = node_locations[i][j - 1] + h
82
83 stiffness_matrix = np.zeros((2 * n_elements + 2, 2 * n_elements + 2))
84 forces = np.zeros((2 * n_elements + 2, 1))
85 loads = np.zeros((2 * n_elements + 2, 1))
86 element_stiffness = np.zeros((4, 4))
87 element_stiffness += np.array([[6, -3*h, -6, -3*h], [-3*h, 2*h*h, 3*h, h*h],
88                                [-6, 3*h, 6, 3*h], [-3*h, h*h, 3*h, 2*h*h]])
89 element_stiffness = (1000 / (3 * h ** 3)) * element_stiffness
90
91 for i in range(0, n_elements):
92     element_forces = np.zeros((4, 1))
93     sunod = (node_locations[i][0] + node_locations[i][1]) / 2
94     forces_of_element = np.array([[force_coefficients[0][0] +
95                                     force_coefficients[0][2] * sunod * 2,
96                                     force_coefficients[0][1] * (h / 2) +
97                                     force_coefficients[0][2] * h * sunod,

```

```

97         force_coefficients[0][2] * (h / 4)]]
98     element_forces = element_matrix(forces_of_element)
99     if (i == 0):
100         stiffness_matrix[:4, :4] += element_stiffness
101         forces[:4, 0:1] += element_forces
102     else:
103         stiffness_matrix[2 * i:2 * (i + 2), 2 * i:2 * (i + 2)] +=
104         element_stiffness
105         forces[2 * i:2 * (i + 2), 0:1] += element_forces
106
107     stiffness_matrix_final = stiffness_matrix
108
109     if (bc_start == 1):
110         for i in range(1, 2 * n_elements + 2):
111             forces[i] = forces[i] - displacement1 * stiffness_matrix_final[i
112 ] [0]
113
114         for i in range(0, 2 * n_elements + 2):
115             for j in range(0, n_elements * 2 + 2):
116                 if (i == 0 or j == 0):
117                     stiffness_matrix_final[i][j] = 0
118             stiffness_matrix_final[0][0] = 1
119             forces[0][0] = displacement1
120             loads[0][0] = 0
121
122         for i in range(2, 2 * n_elements + 2):
123             forces[i] = forces[i] - slope1 * stiffness_matrix_final[i][1]
124
125         for i in range(0, 2 * n_elements + 2):
126             for j in range(0, 2 * n_elements + 2):
127                 if (i == 1 or j == 1):
128                     stiffness_matrix_final[i][j] = 0
129             stiffness_matrix_final[1][1] = 1
130             forces[1][0] = slope1
131             loads[1][0] = 0
132
133     if (bc_end == 1):
134         for i in range(0, 2 * n_elements + 1):
135             forces[i] = forces[i] - slope2 * stiffness_matrix_final[i][2 *
136 n_elements + 1]
137
138         for i in range(0, n_elements * 2 + 2):

```

```

132         for j in range(0, 2 * n_elements + 2):
133             if (i == 2 * n_elements + 1 or j == n_elements * 2 + 1):
134                 stiffness_matrix_final[i][j] = 0
135             stiffness_matrix_final[n_elements * 2 + 1][n_elements * 2 + 1] = 1
136             forces[n_elements * 2 + 1][0] = slope2
137             loads[n_elements * 2 + 1][0] = 0
138         for i in range(0, 2 * n_elements):
139             forces[i] = forces[i] - displacement2 * stiffness_matrix_final[i][2
* n_elements]
140         for i in range(0, n_elements * 2 + 2):
141             for j in range(0, 2 * n_elements + 2):
142                 if (i == 2 * n_elements or j == n_elements * 2):
143                     stiffness_matrix_final[i][j] = 0
144                 stiffness_matrix_final[n_elements * 2][n_elements * 2] = 1
145                 forces[n_elements * 2][0] = displacement2
146                 loads[n_elements * 2][0] = 0
147
148         if (bc_start == 2):
149             loads[0][0] = force1
150             loads[1][0] = -moment1
151         if (bc_end == 2):
152             loads[2 * n_elements][0] = force2
153             loads[2 * n_elements + 1][0] = -moment2
154
155         U = np.linalg.inv(stiffness_matrix_final) @ (forces + loads)
156
157         x = np.linspace(0, 1, 1000)
158         y_deflection = []
159         y_slope = []
160         moment = []
161         shear_force = []
162         stress = []
163
164         for i in range(0, n_elements):
165             for j in range(0, 1000):
166                 if x[j] >= node_locations[i][0] and x[j] <= node_locations[i][1]:
167                     aux = (2 * x[j] - (node_locations[i][0] + node_locations[i][1])
) / h

```

```

168         w = 0
169         wslope = 0
170         mom = 0
171         sh = 0
172         st = 0
173         b = 0
174         for m in range(i * 2, i * 2 + 4):
175             w = w + (U[m][0] * get_value(co[b], aux, 3, 0))
176             wslope = wslope + (U[m][0] * get_value(cod[b], aux, 3, 1))
177             * (2 / h)
178             mom = mom + (500 / 3) * (U[m][0] * get_value(codd[b], aux,
179             3, 2)) * (2 / h) ** 2
180             sh = sh + (-500 / 3) * (U[m][0] * get_value(coddd[b], aux,
181             3, 3)) * (2 / h) ** 3
182             st = st + (-1000) * (U[m][0] * get_value(codd[b], aux, 3,
183             2)) * (2 / h) ** 2
184             b = b + 1
185             y_deflection.append(w)
186             y_slope.append(wslope)
187             moment.append(mom)
188             shear_force.append(sh)
189             stress.append(st)
190
191     y_exact_deflection = []
192     y_exact_slope = []
193     i = 0
194     while i <= 1:
195         y_exact_deflection.append((3 / 500) * ((i ** 4) / 24 - (i ** 3) / 3 +
196         (5 * i * i) / 4))
197         y_exact_slope.append(((i ** 3) / 6 - i * i + 5 * i / 2) * (3 / 500))
198         i = i + 0.001
199
200     error = 0
201     for i in range(0, 999):
202         error = error + ((y_exact_deflection[i] - y_deflection[i]) ** 2)
203
204     rmse_error = np.sqrt(error / 1000) * 100
205     print("RMSE error% = ", rmse_error, "%")

```

```

201
202     if (np.size(x) < np.size(y_deflection)):
203         for i in range(0, (np.size(y_deflection) - np.size(x))):
204             y_deflection.pop()
205             y_slope.pop()
206     elif (np.size(y_deflection) < np.size(x)):
207         for i in range(0, (np.size(x) - np.size(y_deflection))):
208             x = x[:-1]
209             y_exact_deflection = y_exact_deflection[:-1]
210             y_exact_slope = y_exact_slope[:-1]
211
212     return y_deflection, y_exact_deflection, y_slope, y_exact_slope, moment,
        shear_force, stress, x
213
214 n_elements_list = [1, 4, 10, 50, 100]
215 force_coefficients = np.zeros((1, 3))
216 y_deflection_final = []
217 y_exact_deflection_final = []
218 y_slope_final = []
219 y_exact_slope_final = []
220 moment_final = []
221 shear_force_final = []
222 stress_final = []
223 x_final = []
224 force_coefficients[0][0] = 1
225
226 for i in range(0, 5):
227     y_deflection, y_exact_deflection, y_slope, y_exact_slope, moment,
        shear_force, stress, x = \
228         solve_beam_finite_element(n_elements_list[i], 1, 2, force_coefficients)
229     y_deflection_final.append(y_deflection)
230     y_exact_deflection_final.append(y_exact_deflection)
231     y_slope_final.append(y_slope)
232     y_exact_slope_final.append(y_exact_slope)
233     moment_final.append(moment)
234     shear_force_final.append(shear_force)
235     stress_final.append(stress)
236     x_final.append(x)

```

```

237
238 plt.figure()
239 plt.plot(x_final[0], y_deflection_final[0], label="1 element", color='blue')
240 plt.plot(x_final[1], y_deflection_final[1], label="4 element", color='green')
241 plt.plot(x_final[2], y_deflection_final[2], label="10 element", color='red')
242 plt.plot(x_final[3], y_deflection_final[3], label="50 element", color='orange')
243 plt.plot(x_final[4], y_deflection_final[4], label="100 element", color='purple'
    )
244 plt.plot(x_final[0], y_exact_deflection_final[0], label="Exact", color='black',
    linestyle='dashed')
245 plt.legend()
246 plt.title("Deflection with x")
247 plt.ylabel("Deflection")
248 plt.xlabel("x")
249 plt.show()
250
251 plt.figure()
252 plt.plot(x_final[0], y_slope_final[0], label="1 element", color='red')
253 plt.plot(x_final[1], y_slope_final[1], label="4 element", color='purple')
254 plt.plot(x_final[2], y_slope_final[2], label="10 element", color='orange')
255 plt.plot(x_final[3], y_slope_final[3], label="50 element", color='blue')
256 plt.plot(x_final[4], y_slope_final[4], label="100 element", color='green')
257 plt.plot(x_final[0], y_exact_slope_final[0], label="Exact", color='black',
    linestyle='dashed')
258 plt.legend()
259 plt.title("Slope with x")
260 plt.ylabel("Slope")
261 plt.xlabel("x")
262 plt.show()
263
264 plt.figure()
265 plt.plot(x_final[0], moment_final[0], label="1 element", color='red')
266 plt.plot(x_final[1], moment_final[1], label="4 element", color='purple')
267 plt.plot(x_final[2], moment_final[2], label="10 element", color='orange')
268 plt.plot(x_final[3], moment_final[3], label="50 element", color='blue')
269 plt.plot(x_final[4], moment_final[4], label="100 element", color='green')
270 plt.plot(x_final[4], moment_final[4], label="Exact", color='black', linestyle='
    dashed')

```

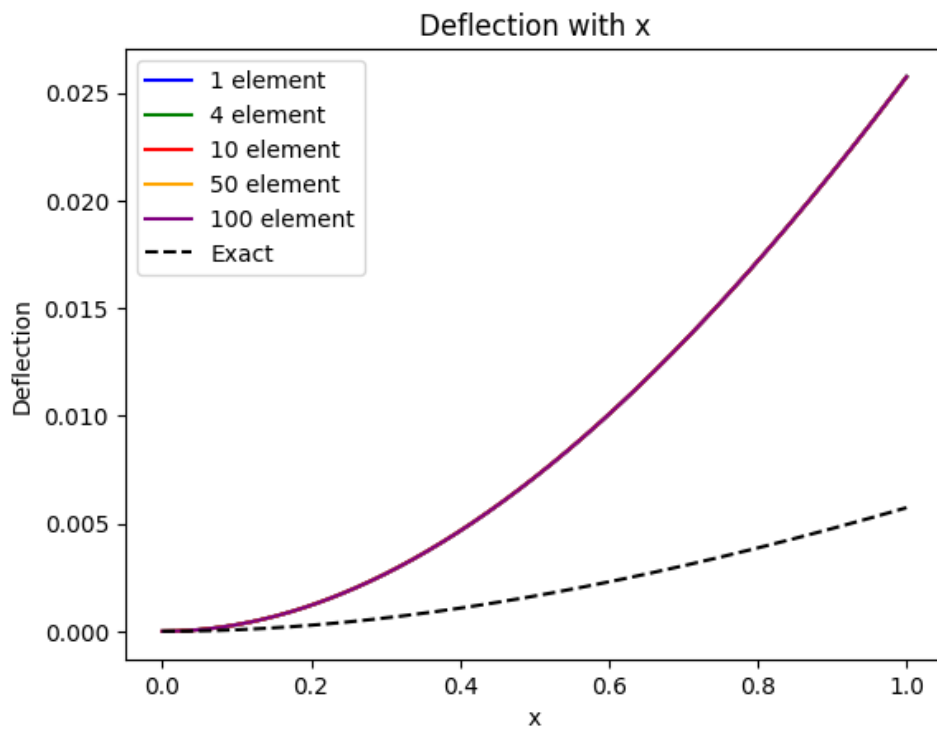


```

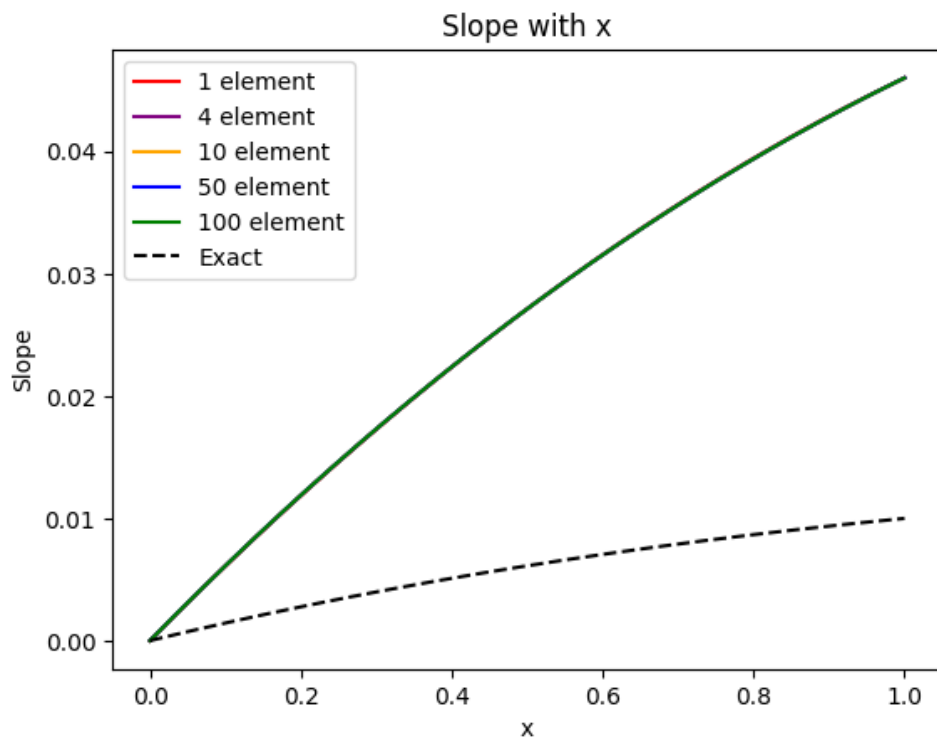
271 plt.legend()
272 plt.title("Moment with x")
273 plt.ylabel("Moment")
274 plt.xlabel("x")
275 plt.show()
276
277 plt.figure()
278 plt.plot(x_final[0], shear_force_final[0], label="1 element", color='red')
279 plt.plot(x_final[1], shear_force_final[1], label="4 element", color='purple')
280 plt.plot(x_final[2], shear_force_final[2], label="10 element", color='orange')
281 plt.plot(x_final[3], shear_force_final[3], label="50 element", color='blue')
282 plt.plot(x_final[4], shear_force_final[4], label="100 element", color='green')
283 plt.plot(x_final[4], shear_force_final[4], label="Exact", color='black',
          linestyle='dashed')
284 plt.legend()
285 plt.title("Shear force with x")
286 plt.ylabel("Shear force")
287 plt.xlabel("x")
288 plt.show()
289
290 plt.figure()
291 plt.plot(x_final[0], stress_final[0], label="1 element", color='red')
292 plt.plot(x_final[1], stress_final[1], label="4 element", color='purple')
293 plt.plot(x_final[2], stress_final[2], label="10 element", color='orange')
294 plt.plot(x_final[3], stress_final[3], label="50 element", color='blue')
295 plt.plot(x_final[4], stress_final[4], label="100 element", color='green')
296 plt.plot(x_final[4], stress_final[4], label="Exact", color='black', linestyle='
          dashed')
297 plt.legend()
298 plt.title("Stress with x")
299 plt.ylabel("Stress (MPa)")
300 plt.xlabel("x")
301 plt.show()

```

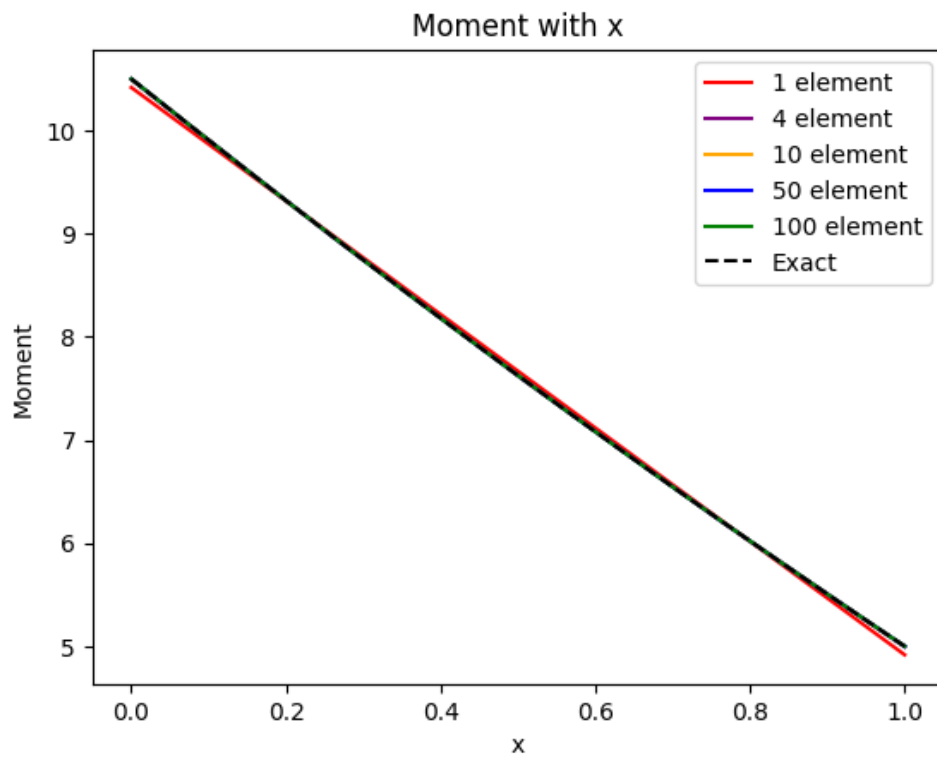
## OUTPUTS PLOTS



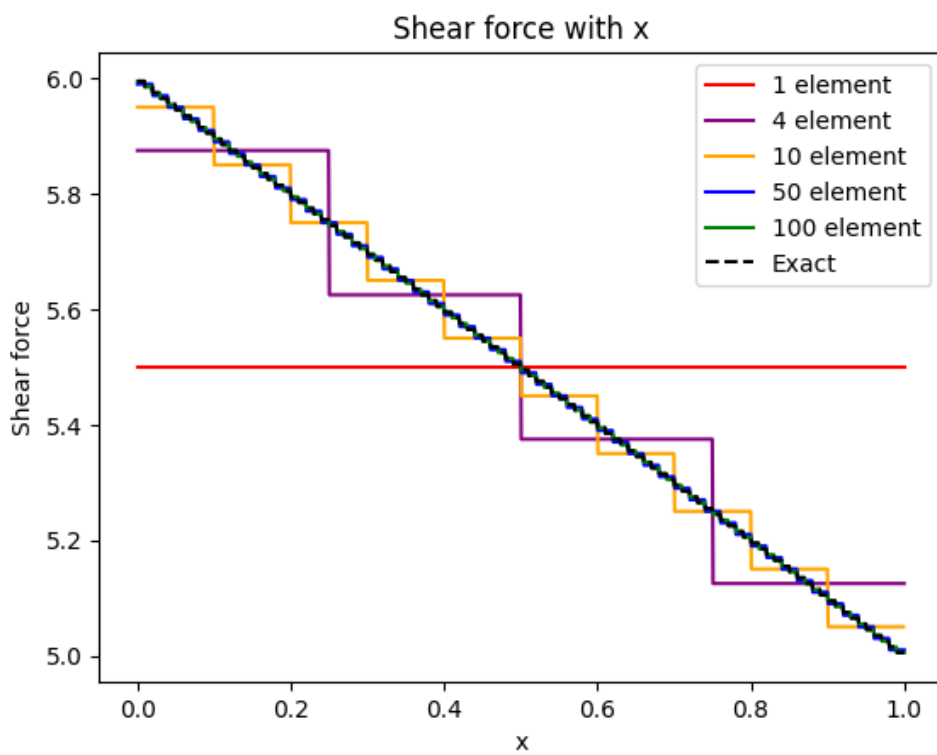
RMSE % = 0.9229651204807908 %



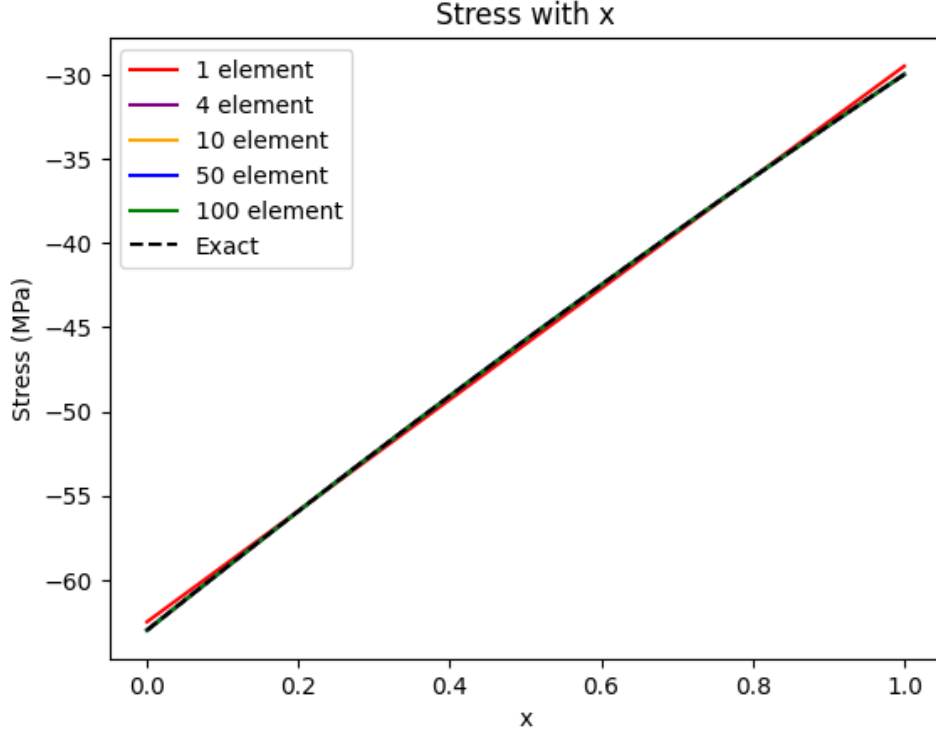
RMSE % = 0.9235163544395251 %



RMSE % = 0.9235187394198392 %



RMSE % = 0.9235188021954196 %



$$\text{RMSE \%} = 0.9235188025278858 \%$$

## 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.

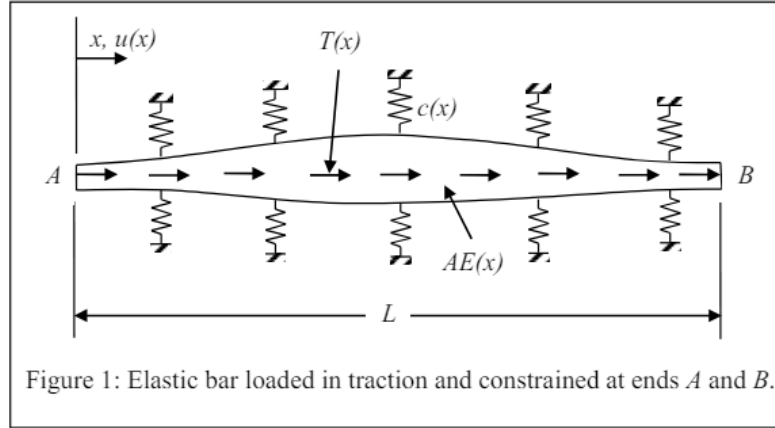


Figure 1: Elastic bar loaded in traction and constrained at ends  $A$  and  $B$ .

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  $\frac{du}{dx}|_{x=1} = 0$ . 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*

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def finite_element_solver(n_elements, degree, selection, bc_left, bc_right,
5     a_coeff, c_coeff, f_coeff):
6     def shape_function(degree, coordinates, coordinates_derivative, selection):
7         if selection == 1:
8             if degree == 1:
9                 coordinates = [[0.5, -0.5],
10                     [0.5, 0.5]]
11                 coordinates_derivative = [[-0.5],
12                     [0.5]]
13             elif degree == 2:
14                 coordinates = [[0, -0.5, 0.5],
15                     [1, 0, -1],
16                     [0, 0.5, 0.5]]
17                 coordinates_derivative = [[-0.5, 1],
18                     [0, -2],
19                     [0.5, 1]]
20             elif degree == 3:
21                 coordinates = [[-0.0625, 0.0625, 0.5625, -0.5625],
22                     [0.5625, -1.6875, -0.5625, 1.6875],
23                     [0.5625, 1.6875, -0.5625, -1.6875],
24                     [-0.0625, -0.0625, 0.5625, 0.5625]]
25                 coordinates_derivative = [[0.0625, 1.125, -1.6875],
26                     [-1.6875, -1.125, 5.0625],
27                     [1.6875, -1.125, -5.0625],
28                     [-0.0625, 1.125, 1.6875]]
29             elif degree == 4:
30                 coordinates = [[0, 0.1667, -0.1667, -0.6667, 0.6667],
31                     [0, -1.3333, 2.6667, 1.3333, -2.6667],
32                     [1, 0, -5, 0, 4],
33                     [0, 1.3333, 2.6667, -1.3333, -2.6667],
```

```

33         [0, -0.1667, -0.1667, 0.6667, 0.6667]]
34     coordinates_derivative = [[0.1667, -0.3333, -2, 2.6667],
35                               [-1.3333, 5.3333, 4, -10.6667],
36                               [0, -10, 0, 16],
37                               [1.3333, 5.3333, -4, -10.6667],
38                               [-0.1667, -0.3333, 2, 2.6667]]
39
40     else:
41         if degree == 1:
42             coordinates = [[0.5, -0.5],
43                             [0.5, 0.5]]
44             coordinates_derivative = [[-0.5],
45                                       [0.5]]
46         elif degree == 2:
47             coordinates = [[0.5, -0.5, 0, 0, 0],
48                             [-0.6124, 0, 0.6124, 0, 0],
49                             [0.5, 0.5, 0, 0, 0]]
50             coordinates_derivative = [[-0.5, 0],
51                                       [0, 1.2247],
52                                       [0.5, 0]]
53         elif degree == 3:
54             coordinates = [[0.5, -0.5, 0, 0, 0],
55                             [0, -0.7906, 0, 0.7906, 0],
56                             [-0.6124, 0, 0.6124, 0, 0],
57                             [0.5, 0.5, 0, 0, 0]]
58             coordinates_derivative = [[-0.5, 0, 0, 0],
59                                       [-0.7906, 0, 2.3717, 0],
60                                       [0, 1.2247, 0, 0],
61                                       [0.5, 0, 0, 0]]
62         elif degree == 4:
63             coordinates = [[0.5, -0.5, 0, 0, 0],
64                             [0.2338, 0, -1.4031, 0, 1.1693],
65                             [0, -0.7906, 0, 0.7906, 0],
66                             [-0.6124, 0, 0.6124, 0, 0],
67                             [0.5, 0.5, 0, 0, 0]]
68             coordinates_derivative = [[-0.5, 0, 0, 0],
69                                       [0, -2.8062, 0, 4.6771],
70                                       [-0.7906, 0, 2.3717, 0],
71                                       [0, 1.2247, 0, 0],

```

```

71         [0.5, 0, 0, 0]]
72     return coordinates, coordinates_derivative
73
74     def integration_rule(re):
75         coordinates_integration = np.array([[0.23862, 0.46791], [-0.23862,
76         0.46791], [0.66121, 0.36076],
77         [-0.66121, 0.36076], [0.93247,
78         0.17132], [-0.93247, 0.17132]])
79         s = 0
80         for i in range(0, 6):
81             su = 0
82             for j in range(0, int(re.shape[0])):
83                 su = su + (re[j] * coordinates_integration[i][0] ** j)
84                 s = s + su * coordinates_integration[i][1]
85             return s
86
87     def element_matrix(ae, c, f, p):
88         for i in range(0, p + 1):
89             for j in range(0, p + 1):
90                 ke[i][j] = (2 / element_length) * integration_rule(
91                 np.polynomial.polynomial.polymul(np.polynomial.polynomial.
92                 polymul(cod[i], cod[j]), ae[0]))
93                 ge[i][j] = (element_length / 2) * integration_rule(
94                 np.polynomial.polynomial.polymul(np.polynomial.polynomial.
95                 polymul(co[i], co[j]), c[0]))
96                 fe[i][0] = (element_length / 2) * integration_rule(
97                 np.polynomial.polynomial.polymul(co[i], f[0]))
98             return ke, ge, fe
99
100     def get_value(co, x, p, d):
101         if d == 0:
102             s = 0
103             for i in range(0, p + 1):
104                 s = s + co[i] * x ** i
105         else:
106             s = 0
107             for i in range(0, p):
108                 s = s + co[i] * x ** i

```



```

105         return s
106
107     co = np.zeros((5, 5))
108     cod = np.zeros((5, 5))
109     co, cod = shape_function(degree, co, cod, selection)
110
111     if bc_left == 2:
112         force_left = float(input("Enter force= "))
113     elif bc_left == 1:
114         displacement_left = 0
115     else:
116         spring_constant_left = float(input("Enter spring constant= "))
117         deviation_left = float(input("Enter spring deviation= "))
118
119     if bc_right == 2:
120         force_right = 0
121     elif bc_right == 1:
122         displacement_right = float(input("Enter displacement= "))
123     else:
124         spring_constant_right = float(input("Enter spring constant= "))
125         deviation_right = float(input("Enter spring deviation= "))
126
127     element_length = 1 / n_elements
128     node_location = np.zeros((n_elements, 2))
129
130     for i in range(0, n_elements):
131         for j in range(0, 2):
132             if i == j == 0:
133                 continue
134             elif j % 2 == 0:
135                 node_location[i][j] = node_location[i - 1][j + 1]
136             else:
137                 node_location[i][j] = node_location[i][j - 1] + element_length
138
139     stiffness_matrix = np.zeros((n_elements * degree + 1, n_elements * degree +
140                                 1))
141     gradient_matrix = np.zeros((n_elements * degree + 1, n_elements * degree +
142                                1))

```

```

141 force_matrix = np.zeros((n_elements * degree + 1, 1))
142 load_matrix = np.zeros((n_elements * degree + 1, 1))
143
144 for i in range(0, n_elements):
145     ke = np.zeros((degree + 1, degree + 1))
146     ge = np.zeros((degree + 1, degree + 1))
147     fe = np.zeros((degree + 1, 1))
148
149     midpoint = (node_location[i][0] + node_location[i][1]) / 2
150
151     ae_coefficient = np.array([[a_coeff[0][0] + a_coeff[0][1] * midpoint +
152     a_coeff[0][2] * midpoint ** 2,
153     a_coeff[0][1] * (element_length / 2) +
154     a_coeff[0][2] * element_length * midpoint,
155     a_coeff[0][2] * (element_length / 4)]])
156     c_coefficient = np.array([[c_coeff[0][0] + c_coeff[0][1] * midpoint +
157     c_coeff[0][2] * midpoint ** 2,
158     c_coeff[0][1] * (element_length / 2) +
159     c_coeff[0][2] * element_length * midpoint,
160     c_coeff[0][2] * (element_length / 4)]])
161     f_coefficient = np.array([[f_coeff[0][0] + f_coeff[0][1] * midpoint +
162     f_coeff[0][2] * midpoint ** 2,
163     f_coeff[0][1] * (element_length / 2) +
164     f_coeff[0][2] * element_length * midpoint,
165     f_coeff[0][2] * (element_length / 4)]])
166
167     ke, ge, fe = element_matrix(ae_coefficient, c_coefficient,
168     f_coefficient, degree)
169
170     if i == 0:
171         stiffness_matrix[:degree + 1, :degree + 1] += ke
172         gradient_matrix[:degree + 1, :degree + 1] += ge
173         force_matrix[:degree + 1, 0:1] += fe
174     else:
175         stiffness_matrix[i * degree:i * degree + degree + 1, i * degree:i *
176         degree + degree + 1] += ke
177         gradient_matrix[i * degree:i * degree + degree + 1, i * degree:i *
178         degree + degree + 1] += ge

```

```

170         force_matrix[i * degree:i * degree + degree + 1, 0:1] += fe
171
172     stiffness_plus_gradient = stiffness_matrix + gradient_matrix
173
174     if bc_left == 1:
175         for i in range(1, n_elements * degree + 1):
176             force_matrix[i] = force_matrix[i] - displacement_left *
stiffness_plus_gradient[i][0]
177         for i in range(0, n_elements * degree + 1):
178             for j in range(0, n_elements * degree + 1):
179                 if i == 0 or j == 0:
180                     stiffness_plus_gradient[i][j] = 0
181             stiffness_plus_gradient[0][0] = 1
182             force_matrix[0][0] = displacement_left
183             load_matrix[0][0] = 0
184
185     if bc_right == 1:
186         for i in range(0, n_elements * degree):
187             force_matrix[i] = force_matrix[i] - displacement_right *
stiffness_plus_gradient[i][n_elements * degree]
188         for i in range(0, n_elements * degree + 1):
189             for j in range(0, n_elements * degree + 1):
190                 if i == n_elements * degree or j == n_elements * degree:
191                     stiffness_plus_gradient[i][j] = 0
192             stiffness_plus_gradient[n_elements * degree][n_elements * degree] = 1
193             force_matrix[n_elements * degree][0] = displacement_right
194             load_matrix[n_elements * degree][0] = 0
195
196     if bc_left == 2:
197         load_matrix[0][0] = force_left
198
199     if bc_right == 2:
200         load_matrix[n_elements * degree][0] = force_right
201
202     if bc_left == 3:
203         stiffness_plus_gradient[0][0] += spring_constant_left
204         load_matrix[0][0] = spring_constant_left * deviation_left
205

```

```

206     if bc_right == 3:
207         stiffness_plus_gradient[n_elements * degree][n_elements * degree] +=
spring_constant_right
208         load_matrix[n_elements * degree][0] = spring_constant_right *
deviation_right
209
210     displacement_solution = np.linalg.inv(stiffness_plus_gradient) @ (
force_matrix + load_matrix)
211
212     x_values = np.linspace(0, 1, 1000)
213     y_values = []
214     y_derivative_values = []
215
216     for i in range(0, n_elements):
217         for j in range(0, 1000):
218             if x_values[j] >= node_location[i][0] and x_values[j] <=
node_location[i][1]:
219                 auxiliary = (2 * x_values[j] - (node_location[i][0] +
node_location[i][1])) / element_length
220                 fr = 0
221                 frd = 0
222                 b = 0
223                 for m in range(i * degree, i * degree + degree + 1):
224                     fr = fr + (displacement_solution[m][0] * get_value(co[b],
auxiliary, degree, 0))
225                     frd = frd + (displacement_solution[m][0] * get_value(cod[b
], auxiliary, degree, 1)) * (2 / element_length)
226                     b = b + 1
227                 y_values.append(fr)
228                 y_derivative_values.append(frd)
229
230     exact_values = []
231     exact_derivative_values = []
232     i = 0
233
234     while i < 1:
235         exact_values.append(-i ** 2 / 2 + (force_right + 1) * i +
displacement_left)

```

```

236     exact_derivative_values.append(-i + (force_right + 1))
237     i = i + 0.001
238
239     if np.size(y_values) < np.size(exact_values):
240         for i in range(0, (np.size(exact_values) - np.size(y_values))):
241             exact_values.pop()
242             x_values = x_values[:-1]
243     elif np.size(exact_values) < np.size(y_values):
244         for i in range(0, (np.size(y_values) - np.size(exact_values))):
245             y_values.pop()
246
247     return y_values, y_derivative_values, exact_values, exact_derivative_values
248     , x_values
249
250 number_of_elements = [1, 2, 5, 10, 100]
251 orders = [1, 2]
252 selection = 1
253 ae_coefficient = np.zeros((1, 3))
254 c_coefficient = np.zeros((1, 3))
255 f_coefficient = np.zeros((1, 3))
256 ae_order = 0
257
258 for i in range(0, ae_order + 1):
259     ae_coefficient[0][i] = 1
260
261 c_order = 0
262
263 for i in range(0, c_order + 1):
264     c_coefficient[0][i] = 0
265
266 f_order = 0
267
268 for i in range(0, f_order + 1):
269     f_coefficient[0][i] = 1
270
271 bc_left = 1
272 bc_right = 2

```

```

273
274 for order in orders:
275     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)
276     err=[exact_values[i]-exact_values[i] for i in range(len(exact_values))]
277     line1, = plt.plot(x_values, err, label="Exact solution")
278     err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
279     line2, = plt.plot(x_values, err, label="1 element")
280     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)
281     err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
282     line3, = plt.plot(x_values, err, label="2 element")
283     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)
284     err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
285     line4, = plt.plot(x_values, err, label="5 element")
286     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)
287     err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
288     line5, = plt.plot(x_values, err, label="10 element")
289     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)
290     err=[exact_values[i]-y_values[i] for i in range(len(exact_values))]
291     line6, = plt.plot(x_values, err, label="100 element")
292     plt.legend(handles=[line1, line2, line3, line4, line5, line6])
293     plt.title("Plot of error between exact solution and FEM solution")
294     plt.xlabel("x")
295     plt.ylabel("Difference in values(Error)")
296     plt.show()
297
298 for order in orders:
299     line1, = plt.plot(x_values, exact_derivative_values, label="Exact solution"
)

```

```

300     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)
301     line2, = plt.plot(x_values, y_derivative_values, label="1 element")
302     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)
303     line3, = plt.plot(x_values, y_derivative_values, label="2 element")
304     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)
305     line4, = plt.plot(x_values, y_derivative_values, label="5 element")
306     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)
307     line5, = plt.plot(x_values, y_derivative_values, label="10 element")
308     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)
309     line6, = plt.plot(x_values, y_derivative_values, label="100 element")
310     plt.legend(handles=[line1, line2, line3, line4, line5, line6])
311     plt.title("Plot of derivatives")
312     plt.xlabel("x")
313     plt.ylabel("Force")
314     plt.show()
315
316 errors = []
317 log_n = []
318 energies = []
319
320 for n in number_of_elements:
321     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)
322     error = np.log(np.linalg.norm(np.array(exact_values) - np.array(y_values)))
323     errors.append(error)
324     log_n.append(np.log(n))
325     total_energy = 0

```

```

326
327     for i in range(0, 998):
328         fx1 = y_derivative_values[i] ** 2
329         fx2 = y_derivative_values[i + 1] ** 2
330         total_energy = total_energy + 0.5 * (fx1 + fx2) * (x_values[i + 1] -
x_values[i])
331
332     energies.append(total_energy)
333
334 print(log_n, errors)
335
336 line1, = plt.plot(number_of_elements, energies, label="linear", marker=".")
337 errors = []
338 log_n = []
339 energies = []
340 exact_energies = []
341
342 for n in number_of_elements:
343     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)
344     error = np.log(np.linalg.norm(np.array(exact_values) - np.array(y_values)))
345     errors.append(error)
346     log_n.append(np.log(n))
347     total_energy = 0
348
349     for i in range(0, 998):
350         fx1 = y_derivative_values[i] ** 2
351         fx2 = y_derivative_values[i + 1] ** 2
352         total_energy = total_energy + 0.5 * (fx1 + fx2) * (x_values[i + 1] -
x_values[i])
353
354     energies.append(total_energy)
355
356     total_exact_energy = 0
357
358     for i in range(0, 998):
359         fx1 = exact_derivative_values[i] ** 2

```



```

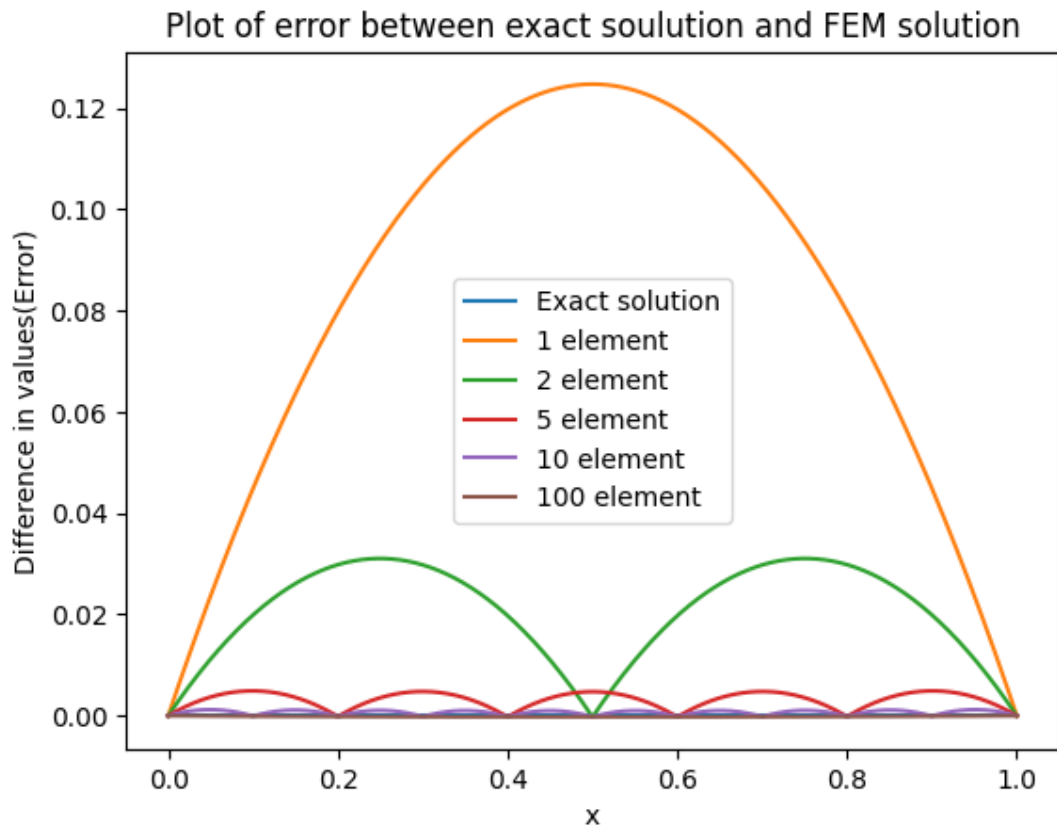
360     fx2 = exact_derivative_values[i + 1] ** 2
361     total_exact_energy = total_exact_energy + 0.5 * (fx1 + fx2) * (x_values
    [i + 1] - x_values[i])
362
363     exact_energies.append(total_exact_energy)
364
365 line2, = plt.plot(number_of_elements, energies, label="quadratic", marker=".")
366 plt.legend(handles=[line1, line2])
367 plt.xlabel("Number of elements")
368 plt.ylabel("Energy")
369 plt.title("Strain energy by approximating degree")
370 plt.show()
371
372 line1, = plt.plot(number_of_elements, energies, label="FEM solution", marker=".
    ")
373 line2, = plt.plot(number_of_elements, exact_energies, label="Exact solution",
    marker=".")
374 plt.legend(handles=[line1, line2])
375 plt.xlabel("Number of elements")
376 plt.ylabel("Energy")
377 plt.title("Strain Energy of FEM solution and Exact solution")
378 plt.show()

```

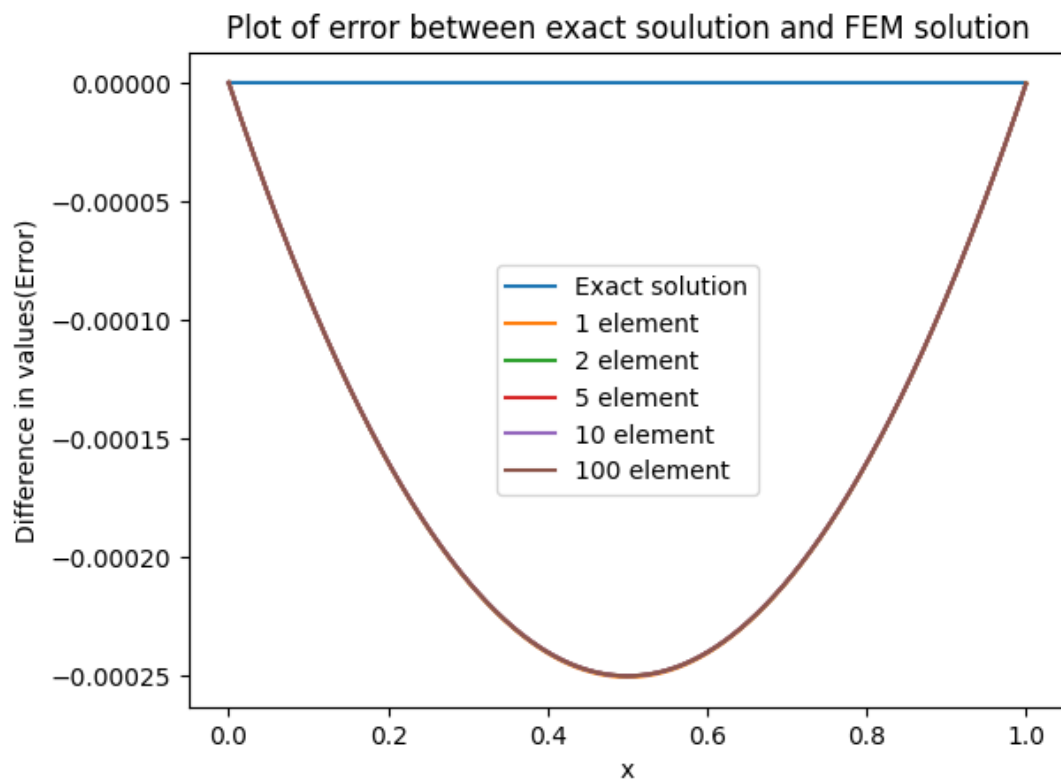
This code solves the weak form equation using finite element analysis and gives us primary and secondary variables, which we can plot to analyze the accuracy of our model with varying numbers of elements starting from 1 and going all the way to 100. We can observe that the more the number of elements 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*

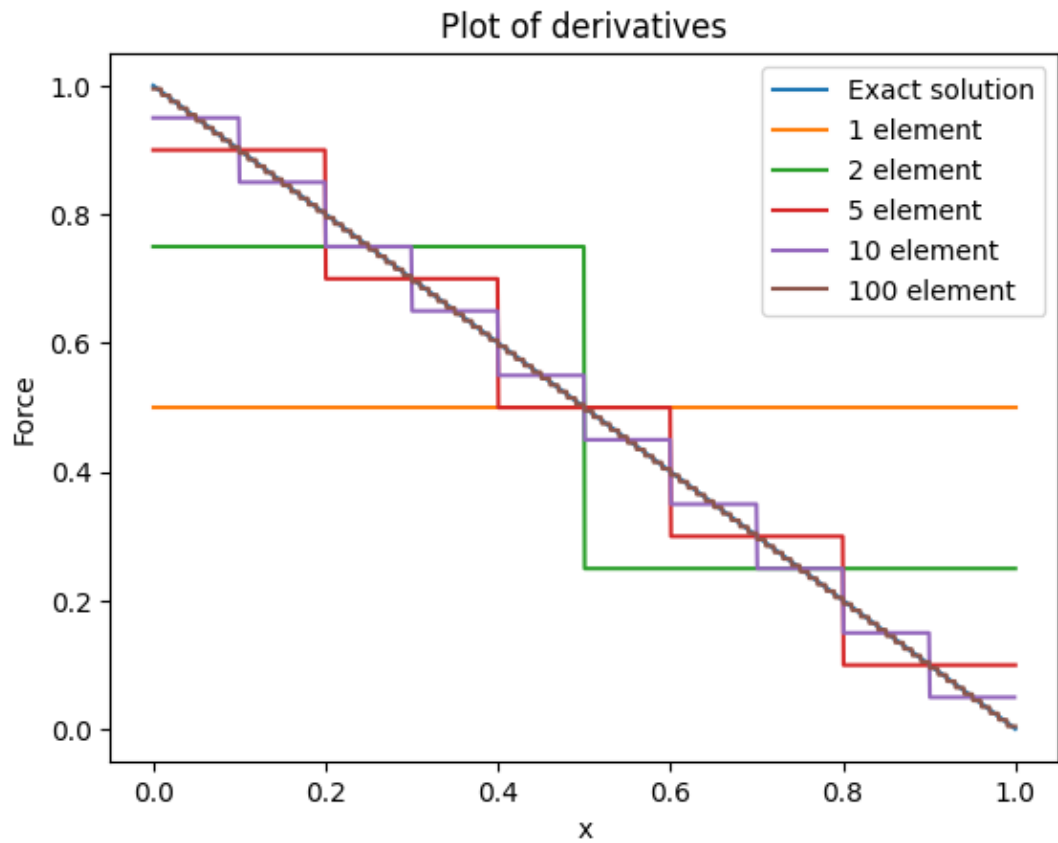
For  $AE(x) = 1$ ,  $C(x) = 0$ ,  $T(x)=1$



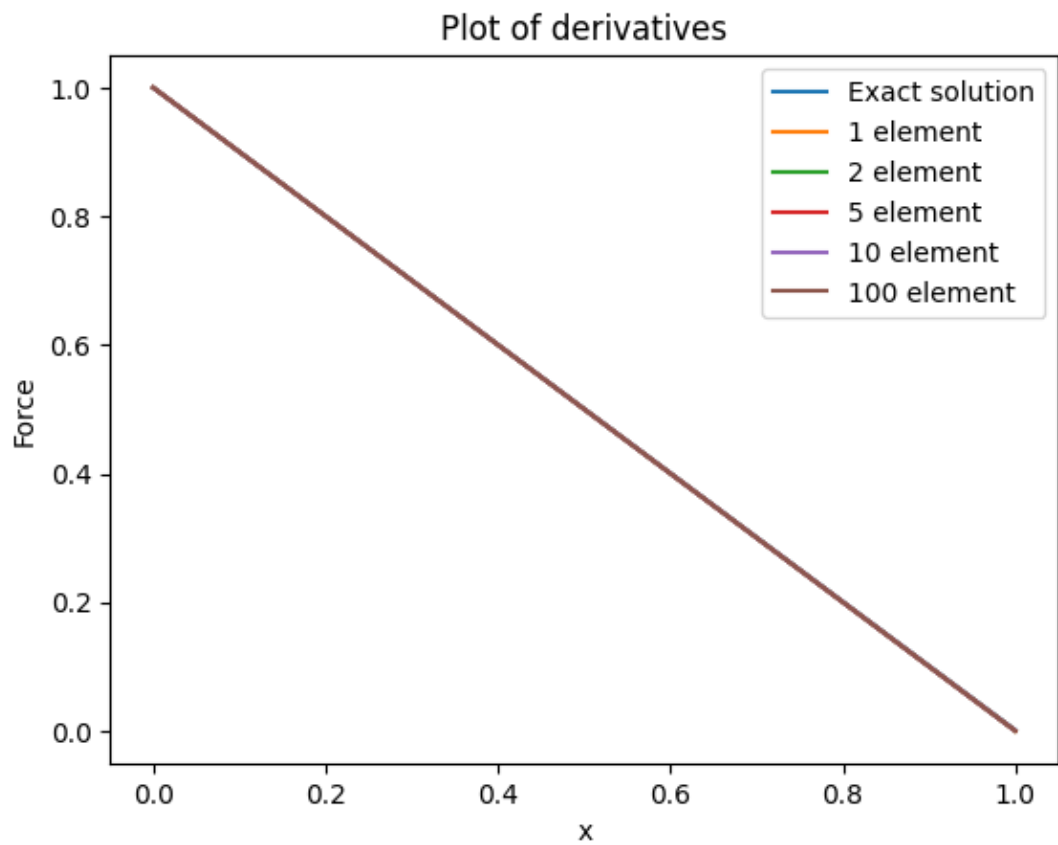
Linear Approximation



Quadratic Approximation

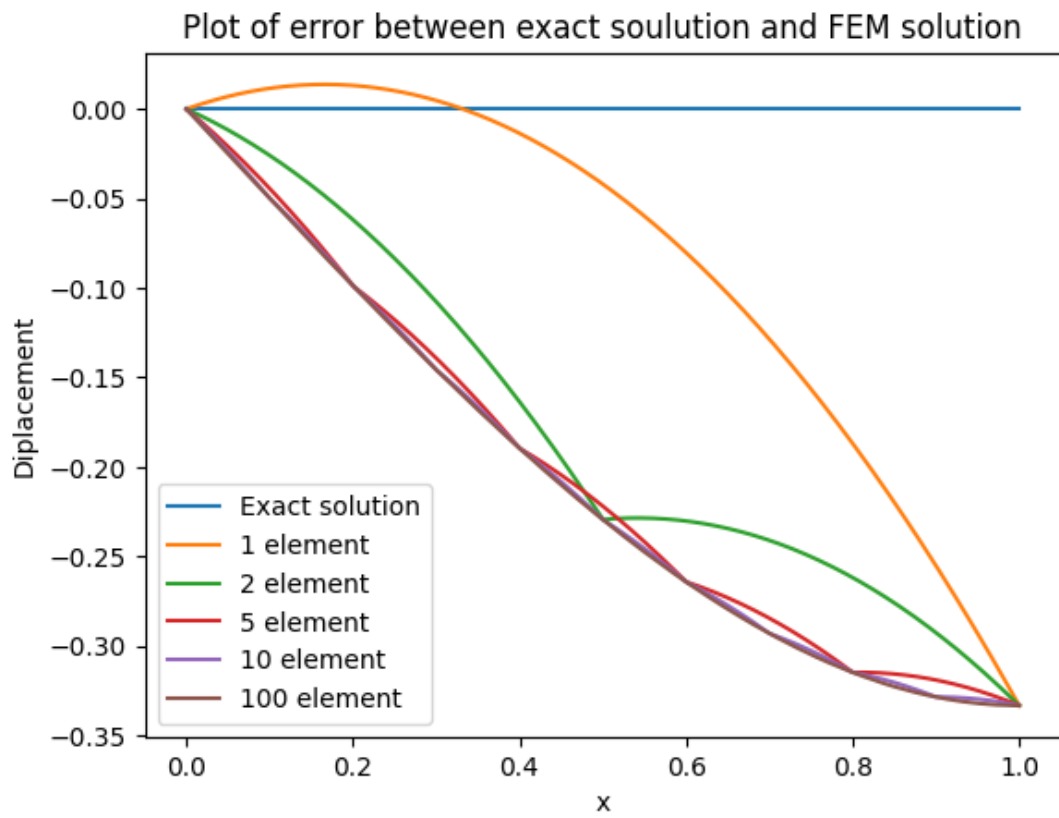


Linear Approximation

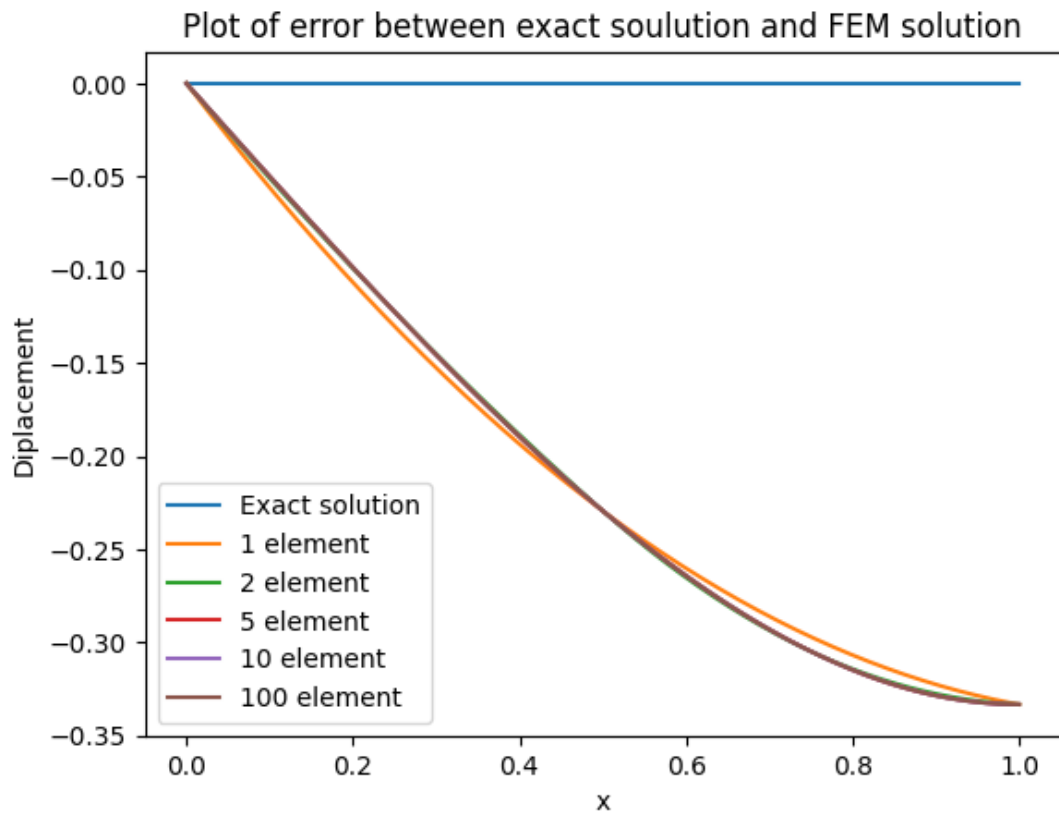


Quadratic Approximation

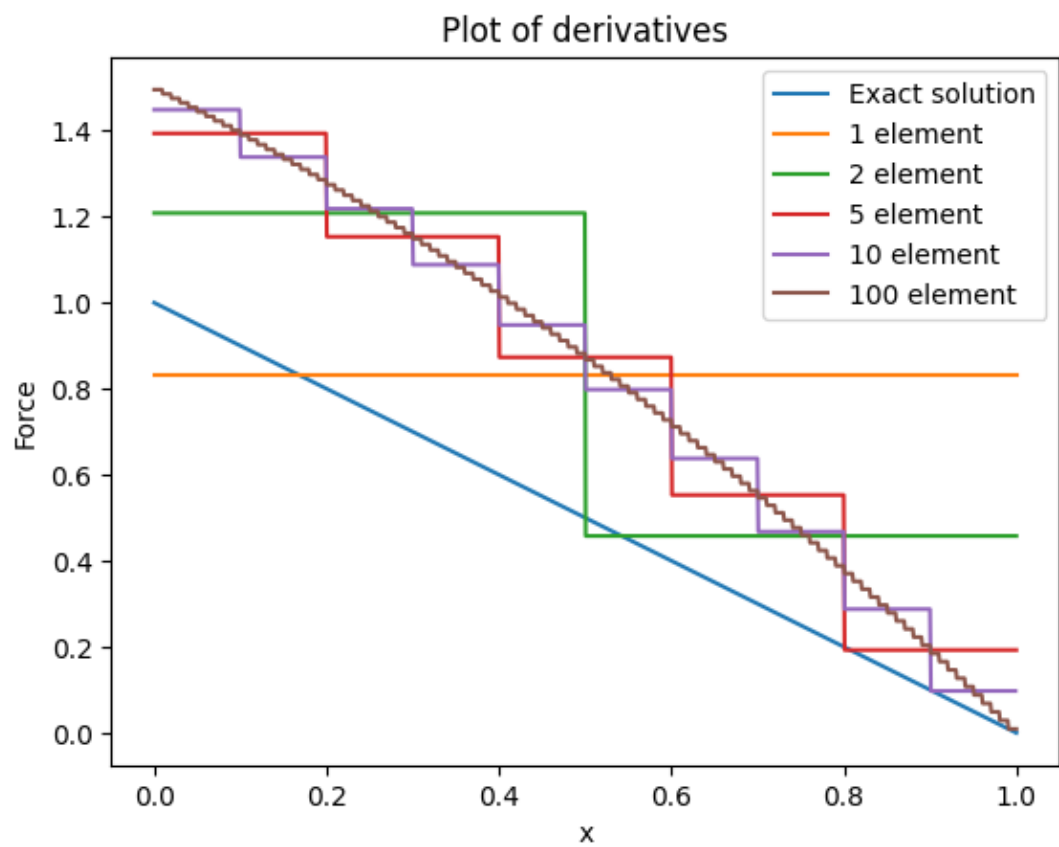
For  $AE(x) = 1$ ,  $C(x) = 0$ ,  $T(x)=x$



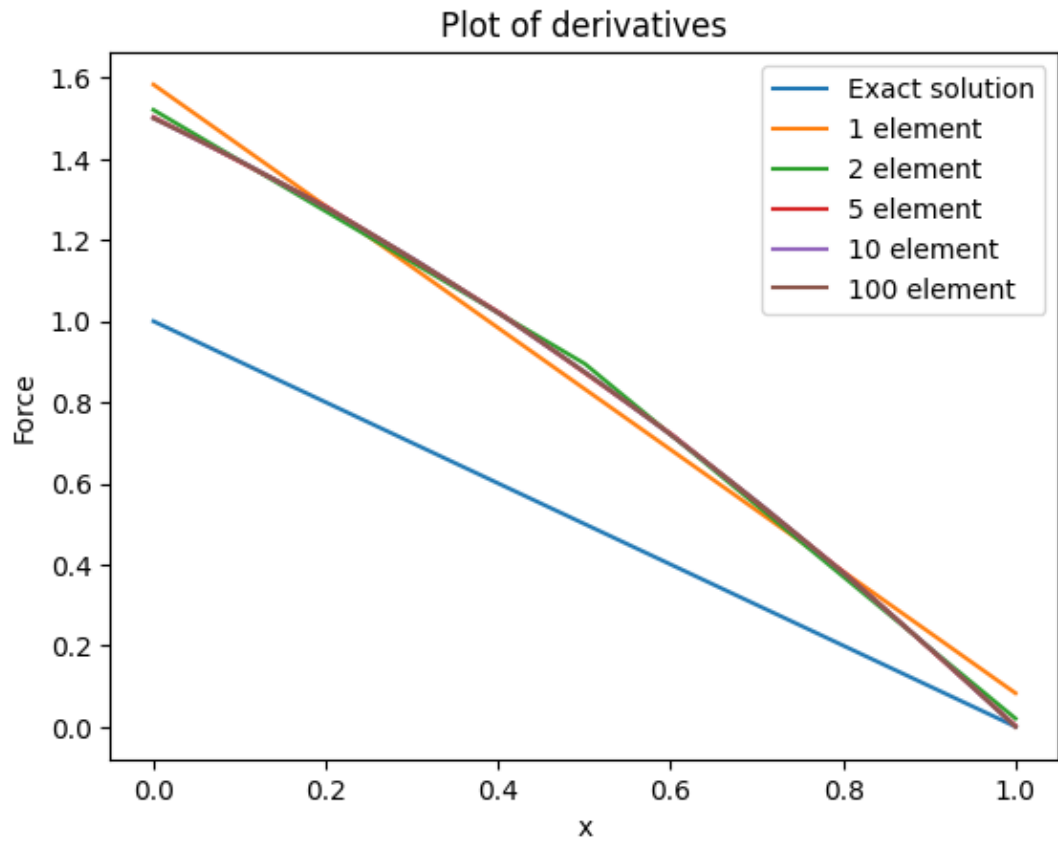
Linear Approximation



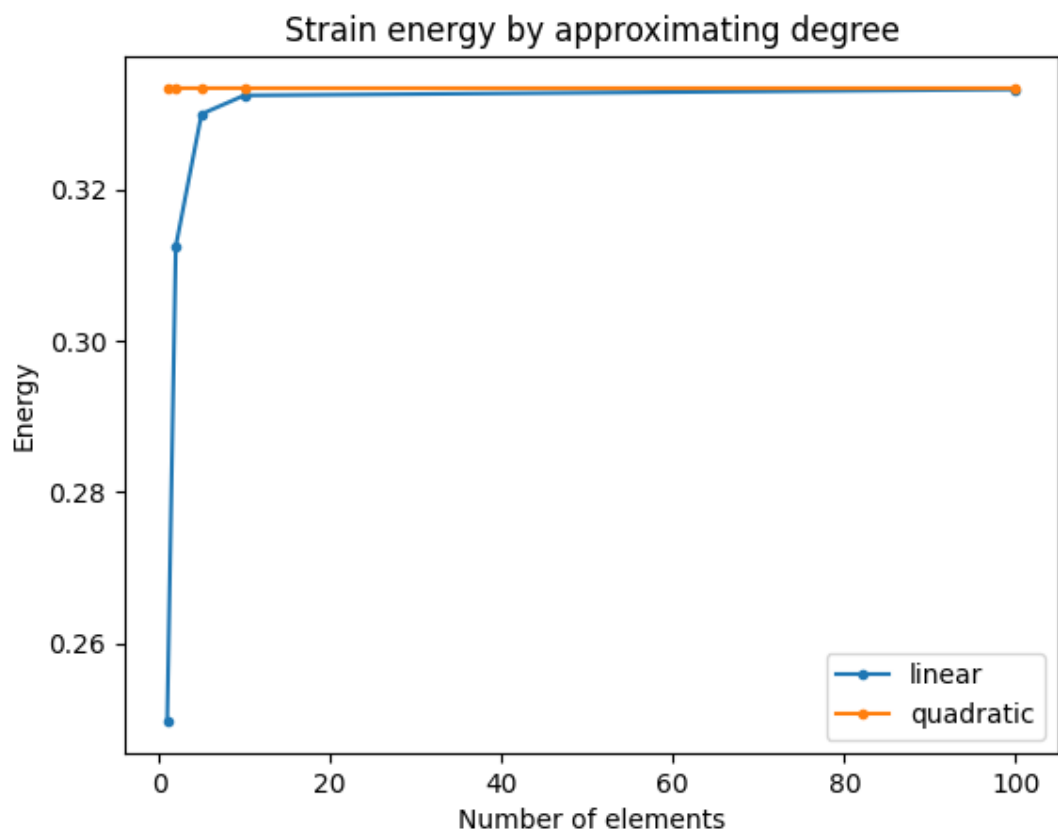
Quadratic Approximation

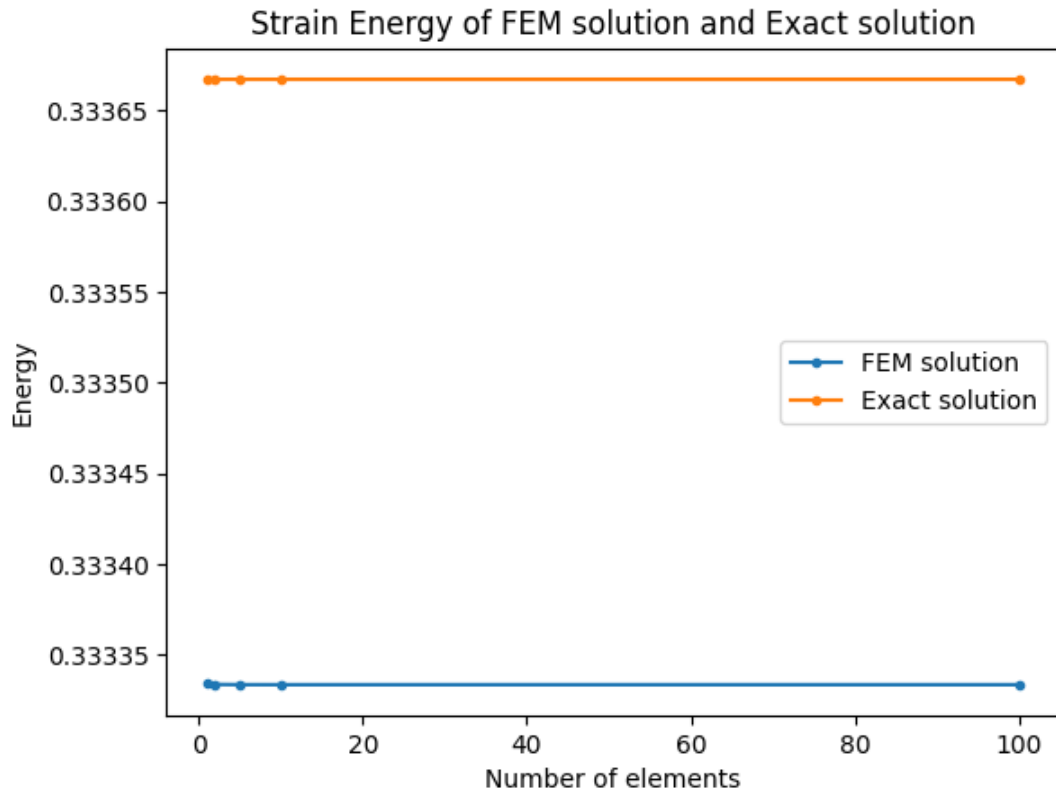


Linear Approximation



Quadratic Approximation





3. Take  $AE(x) = 1$ ,  $c(x) = 1$  and  $T(x) = 1$  with  $u(x)|_{x=0} = 0$  and  $\frac{du}{dx}|_{x=1} = 0$ . 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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 def FEM_hp(n,p,choice,bc1,bc2,aeco,cco,fco):
4     def shapeFunctions(p,co,cod,choice):
5         if(choice==1):
6             if (p==1):
7                 co=[[0.5, -0.5],
8                    [0.5,0.5]]

```

```

9         cod=[[-0.5],
10             [0.5]]
11     elif (p==2):
12         co=[[0,-0.5,0.5],
13             [1,0,-1],
14             [0,0.5,0.5]]
15         cod=[[-0.5,1],
16             [0,-2],
17             [0.5,1]]
18     elif (p==3):
19         co=[[-0.0625,0.0625,0.5625,-0.5625],
20             [0.5625,-1.6875,-0.5625,1.6875],
21             [0.5625,1.6875,-0.5625,-1.6875],
22             [-0.0625,-0.0625,0.5625,0.5625]]
23         cod=[[0.0625,1.125,-1.6875],
24             [-1.6875,-1.125,5.0625],
25             [1.6875,-1.125,-5.0625],
26             [-0.0625,1.125,1.6875]]
27     elif (p==4):
28         co=[[0,0.1667,-0.1667,-0.6667,0.6667],
29             [0,-1.3333,2.6667,1.3333,-2.6667],
30             [1,0,-5,0,4],
31             [0,1.3333,2.6667,-1.3333,-2.6667],
32             [0,-0.1667,-0.1667,0.6667,0.6667]]
33         cod=[[0.1667,-0.3333,-2,2.6667],
34             [-1.3333,5.3333,4,-10.6667],
35             [0,-10,0,16],
36             [1.3333,5.3333,-4,-10.6667],
37             [-0.1667,-0.3333,2,2.6667]]
38     else:
39         if (p==1):
40             co=[[0.5,-0.5],
41                 [0.5,0.5]]
42             cod=[[-0.5],
43                 [0.5]]
44         elif (p==2):
45             co=[[0.5,-0.5,0,0,0],
46                 [-0.6124,0,0.6124,0,0],

```



```

47         [0.5,0.5,0,0,0]]
48         cod=[[-0.5,0],
49              [0,1.2247],
50              [0.5,0]]
51     elif (p==3):
52         co=[[0.5,-0.5,0,0,0],
53            [0,-0.7906,0,0.7906,0],
54            [-0.6124,0,0.6124,0,0],
55            [0.5,0.5,0,0,0]]
56         cod=[[-0.5,0,0,0],
57            [-0.7906,0,2.3717,0],
58            [0,1.2247,0,0],
59            [0.5,0,0,0]]
60     elif (p==4):
61         co=[[0.5,-0.5,0,0,0],
62            [0.2338,0,-1.4031,0,1.1693],
63            [0,-0.7906,0,0.7906,0],
64            [-0.6124,0,0.6124,0,0],
65            [0.5,0.5,0,0,0]]
66         cod=[[-0.5,0,0,0],
67            [0,-2.8062,0,4.6771],
68            [-0.7906,0,2.3717,0],
69            [0,1.2247,0,0],
70            [0.5,0,0,0]]
71     return co,cod
72 def integrate(re):
73     coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
74     coi=np.append(coi,
75 [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
76     s=0
77     for i in range(0,6):
78         su=0
79         for j in range(0,int(re.shape[0])):
80             su=su+(re[j]*coi[i][0]**j)
81         s=s+su*coi[i][1]
82     return s
83 def elementMatrix(ae,c,f,p):
84     for i in range(0, p+1):

```

```

84         for j in range(0, p+1):
85             ke[i][j]=(2/h)*integrate(np.polynomial.polynomial.polymul(np.
polynomial.polynomial.polymul(cod[i],cod[j]),ae[0]))
86             ge[i][j]=(h/2)*integrate(np.polynomial.polynomial.polymul(np.
polynomial.polynomial.polymul(co[i],co[j]),c[0]))
87             fe[i][0]=(h/2)*integrate(np.polynomial.polynomial.polymul(co[i],f
[0]))
88         return ke,ge,fe
89     def getValue(co,x,p,d):
90         if(d==0):
91             s=0
92             for i in range(0,p+1):
93                 s=s+co[i]*x**i
94         else:
95             s=0
96             for i in range(0,p):
97                 s=s+co[i]*x**i
98         return s
99     co=np.zeros((5,5))
100     cod=np.zeros((5,5))
101     co,cod=shapeFunctions(p,co,cod,choice)
102     if(bc1==2):
103         force1=float(input("Enter force = "))
104     elif (bc1==1):
105         dis1=0
106     else:
107         spr1=float(input("Enter spring constant = "))
108         dev1=float(input("Enter spring deviation = "))
109
110     if(bc2==2):
111         force2=0
112     elif(bc2==1):
113         dis2=float(input("Enter displacement = "))
114     else:
115         spr2=float(input("Enter spring constant = "))
116         dev2=float(input("Enter spring deviation = "))
117
118     h=1/n

```

```

119     nodeLocations=np.zeros((n,2))
120     for i in range(0,n):
121         for j in range(0,2):
122             if (i==j==0):
123                 continue
124             elif (j%2==0):
125                 nodeLocations[i][j]=nodeLocations[i-1][j+1]
126             else:
127                 nodeLocations[i][j]=nodeLocations[i][j-1]+h
128
129     K=np.zeros((n*p+1,n*p+1))
130     G=np.zeros((n*p+1,n*p+1))
131     F=np.zeros((n*p+1,1))
132     Q=np.zeros((n*p+1,1))
133     for i in range(0,n):
134         ke=np.zeros((p+1,p+1))
135         ge=np.zeros((p+1,p+1))
136         fe=np.zeros((p+1,1))
137         sunod=(nodeLocations[i][0]+nodeLocations[i][1])/2
138         aecof=np.array([[aeco[0][0]+aeco[0][1]*sunod+aeco[0][2]*sunod**2,aeco
139 [0][1]*(h/2)+aeco[0][2]*h*sunod,aeco[0][2]*(h/4)]]))
140         ccof=np.array([[cco[0][0]+cco[0][1]*sunod+cco[0][2]*sunod**2,cco
141 [0][1]*(h/2)+cco[0][2]*h*sunod,cco[0][2]*(h/4)]]))
142         fcof=np.array([[fco[0][0]+fco[0][1]*sunod+fco[0][2]*sunod**2,fco
143 [0][1]*(h/2)+fco[0][2]*h*sunod,fco[0][2]*(h/4)]]))
144         ke,ge,fe=elementMatrix(aecof,ccof,fcof,p)
145
146         if (i==0):
147             K[:p+1,:p+1]+=ke
148             G[:p+1,:p+1]+=ge
149             F[:p+1,0:1]+=fe
150         else:
151             K[i*p:i*p+p+1,i*p:i*p+p+1]+=ke
152             G[i*p:i*p+p+1,i*p:i*p+p+1]+=ge
153             F[i*p:i*p+p+1,0:1]+=fe
154
155     KF=K+G
156     if (bc1==1):
157         for i in range(1,n*p+1):

```

```

154         F[i]=F[i]-dis1*KF[i][0]
155     for i in range(0,n*p+1):
156         for j in range (0,n*p+1):
157             if(i==0 or j==0):
158                 KF[i][j]=0
159
160     KF[0][0]=1
161     F[0][0]=dis1
162     Q[0][0]=0
163
164     if (bc2==1):
165         for i in range (0,n*p):
166             F[i]=F[i]-dis2*KF[i][n*p]
167         for i in range(0,n*p+1):
168             for j in range (0,n*p+1):
169                 if(i==n*p or j==n*p):
170                     KF[i][j]=0
171
172             KF[n*p][n*p]=1
173             F[n*p][0]=dis2
174             Q[n*p][0]=0
175
176     if (bc1==2):
177         Q[0][0]=force1
178
179     if (bc2==2):
180         Q[n*p][0]=force2
181
182     if (bc1==3):
183         KF[0][0]+=spr1
184         Q[0][0]=spr1*dev1
185
186     if (bc2==3):
187         KF[n*p][n*p]+=spr2
188         Q[n*p][0]=spr2*dev2
189
190
191     U=np.linalg.inv(KF)@(F+Q)
192
193
194     x=np.linspace(0,1,1000)
195     yh=[]
196     yhd=[]
197     for i in range(0,n):
198         for j in range(0,1000):
199             if(x[j]>=nodeLocations[i][0] and x[j]<=nodeLocations[i][1]):
200                 aux=(2*x[j]-(nodeLocations[i][0]+nodeLocations[i][1]))/h

```

```

192         fr=0
193         frd=0
194         b=0
195         for m in range(i*p,i*p+p+1):
196             fr=fr+(U[m][0]*getValue(co[b],aux,p,0))
197             frd=frd+(U[m][0]*getValue(cod[b],aux,p,1))*(2/h)
198             b=b+1
199         yh.append(fr)
200         yhd.append(frd)
201     ye=[]
202     yed=[]
203     i=0
204     while (i<1):
205         ye.append(((np.e**(-i))*(np.e**(i) - 1)*(-np.e**(i) + np.e**(2)))/(np.e
206         **(2) + 1))
207         yed.append((np.e**(2-i)-np.e**(i))/(np.e**(2)+1))
208         i=i+0.001
209     if (np.size(yh)<np.size(ye)):
210         for i in range(0,(np.size(ye)-np.size(yh))):
211             ye.pop()
212             yed.pop()
213             x=x[:-1]
214     elif (np.size(ye)>np.size(yh)):
215         for i in range(0,(np.size(yh)-np.size(ye))):
216             yh.pop()
217             yhd.pop()
218     return yh,yhd,ye,yed,x
219
220 def strainEnergy(ae,fx,fxd,c):
221     s=0
222     for i in range(0,998):
223         fx1=ae*fxd[i]**2+c*fx[i]**2
224         fx2=ae*fxd[i+1]**2+c*fx[i]**2
225         s=s+0.5*(fx1+fx2)*(x[i+1]-x[i])
226     return s
227
228 numElements = [1,10,20,40,80,100]

```

```

229 order = [1,2,3,4]
230 choice = 1
231 aeco = np.zeros((1,3))
232 cco = np.zeros((1,3))
233 fco = np.zeros((1,3))
234 aep = 0
235
236 for i in range (0,aep+1):
237     aeco[0][i]=1
238 cp=0
239 for i in range (0,cp+1):
240     cco[0][0]=1
241 fp=0
242 for i in range (0,fp+1):
243     fco[0][0]=1
244 bc1=1
245 bc2=2
246 for p in order:
247     yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)
248     line1, = plt.plot(x,yh, label="1 element", color='purple')
249     yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
250     line2, = plt.plot(x,yh, label="10 element", color='brown')
251     yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
252     line3, = plt.plot(x,yh, label="20 element", color='green')
253     yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
254     line4, = plt.plot(x,yh, label="40 element", color='red')
255     yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
256     line7, = plt.plot(x,yh, label="80 element", color='green')
257     yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
258     line5, = plt.plot(x,yh, label="100 element", color='blue')
259     line6, = plt.plot(x,ye, label="exact", color='black', linestyle='dashed')
260     plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])
261     plt.title("FEM with exact solution")
262     plt.xlabel("x")
263     plt.ylabel("u")
264     plt.show()
265 for p in order:
266     yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)

```

```

267 # line1, =plt.plot(x,yhd, label="1 element")
268     yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
269 # line2, =plt.plot(x,yhd, label="10 element")
270     yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
271 # line3, =plt.plot(x,yhd, label="20 element")
272     yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
273 # line4, =plt.plot(x,yhd, label="40 element")
274     yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
275 # line7, =plt.plot(x,yhd, label="80 element")
276     yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
277 # line5, =plt.plot(x,yhd, label="100 element")
278 # line6, =plt.plot(x,yed, label="exact")
279 # plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])
280 # plt.title("Plot of derivatives")
281 # plt.xlabel("x")
282 # plt.ylabel("F")
283 # plt.show()
284 error=[]
285 logElements=[]
286 energy=[]
287 energyerr=[]
288 logEnergyError=[]
289 for n in numElements :
290     yh,yhd,ye,yed,x=FEM_hp(n,1,choice,bc1,bc2,aeco,cco,fco)
291     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
292     uex=np.linalg.norm(np.array(ye))
293     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
294     error.append(er)
295     logEnergyError.append(np.log(energynorm/uex))
296     logElements.append(np.log(n))
297     strar=np.array(ye)-np.array(yh)
298     strard=np.array(yed)-np.array(yhd)
299     eg=strainEnergy(1,yhd,yh,0)
300     erreg=strainEnergy(1,strard,strar,0)
301     energyerr.append(erreg)
302     energy.append(eg)
303 print(logElements,error)
304 line1,=plt.plot(logElements,error, label="linear",marker=".", color='purple')

```

```

305 # line1,=plt.plot(logElements,energy, label="linear",marker=".", color='purple
    ')
306 # line1,=plt.plot(logElements,logEnergyError, label="linear",marker=".", color
    ='purple')
307 error=[]
308 logElements=[]
309 energy=[]
310 energyerr=[]
311 logEnergyError=[]
312 for n in numElements      :
313     yh,yhd,ye,yed,x=FEM_hp(n,2,choice,bc1,bc2,aeco,cco,fco)
314     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
315     uex=np.linalg.norm(np.array(ye))
316     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
317     error.append(er)
318     logEnergyError.append(np.log(energynorm/uex))
319     logElements.append(np.log(n))
320     strar=np.array(ye)-np.array(yh)
321     strard=np.array(yed)-np.array(yhd)
322     eg=strainEnergy(1,yhd,yh,0)
323     erreg=strainEnergy(1,strard,strar,0)
324     energyerr.append(erreg)
325     energy.append(eg)
326 print(logElements,error)
327 line2,=plt.plot(logElements,error, label="quadratic",marker=".", color='green')
328 # line2,=plt.plot(logElements,energy, label="quadratic",marker=".", color='
    green')
329 # line2,=plt.plot(logElements,logEnergyError, label="quadratic",marker=".",
    color='green')
330 error=[]
331 logElements=[]
332 energy=[]
333 energyerr=[]
334 logEnergyError=[]
335 for n in numElements      :
336     yh,yhd,ye,yed,x=FEM_hp(n,3,choice,bc1,bc2,aeco,cco,fco)
337     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
338     uex=np.linalg.norm(np.array(ye))

```



```

339     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
340     error.append(er)
341     logEnergyError.append(np.log(energynorm/uex))
342     logElements.append(np.log(n))
343     strar=np.array(ye)-np.array(yh)
344     strard=np.array(yed)-np.array(yhd)
345     eg=strainEnergy(1,yhd,yh,0)
346     erreg=strainEnergy(1,strard,strar,0)
347     energyerr.append(erreg)
348     energy.append(eg)
349 print(logElements,error)
350 line3,=plt.plot(logElements,error, label="cubic",marker=".", color='blue')
351 # line3,=plt.plot(logElements,energy, label="cubic",marker=".", color='blue')
352 # line3,=plt.plot(logElements,logEnergyError, label="cubic",marker=".", color='
    blue')
353 error=[]
354 logElements=[]
355 energy=[]
356 energyerr=[]
357 logEnergyError=[]
358 energyexact=[]
359 for n in numElements :
360     yh,yhd,ye,yed,x=FEM_hp(n,4,choice,bc1,bc2,aeco,cco,fco)
361     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
362     uex=np.linalg.norm(np.array(ye))
363     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
364     error.append(er)
365     logEnergyError.append(np.log(energynorm/uex))
366     logElements.append(np.log(n))
367     strar=np.array(ye)-np.array(yh)
368     strard=np.array(yed)-np.array(yhd)
369     eg=strainEnergy(1,yhd,yh,0)
370     energyexact.append(strainEnergy(1,yed,ye,0))
371     erreg=strainEnergy(1,strard,strar,0)
372     energyerr.append(erreg)
373     energy.append(eg)
374 print(logElements,error)
375 line4,=plt.plot(logElements,error, label="quartic",marker=".", color='orange')

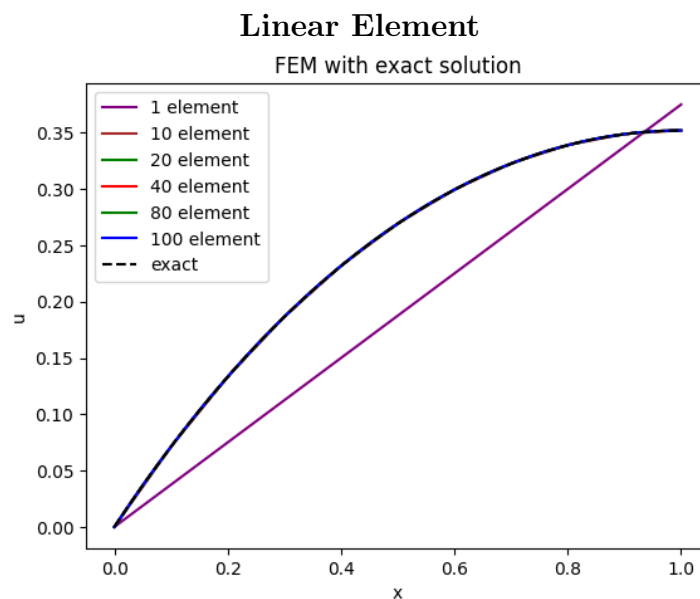
```

```

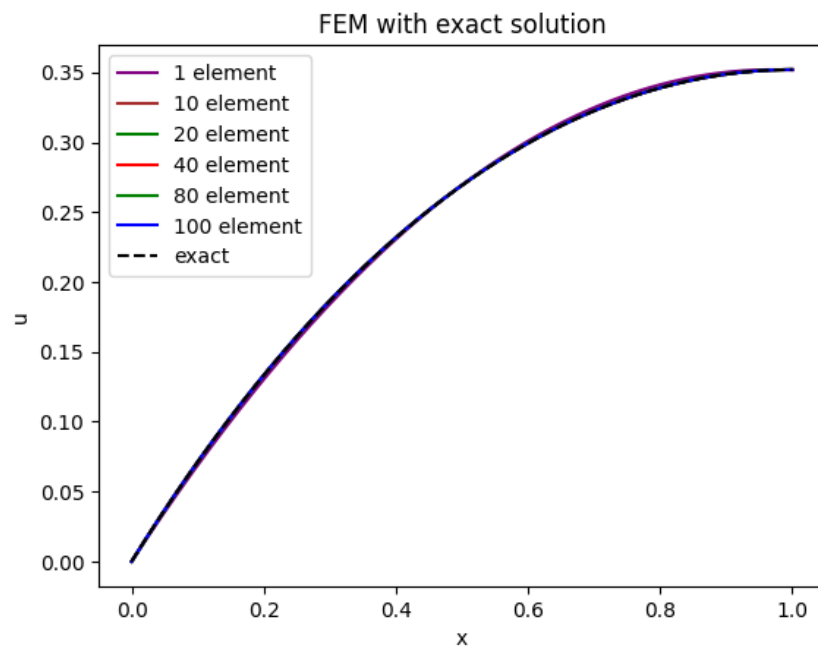
376 # line4,=plt.plot(logElements,energy, label="quartic",marker=".", color='orange
    ')
377 # line4,=plt.plot(logElements,logEnergyError, label="quartic",marker=".", color
    ='orange')
378
379 plt.legend(handles=[line1, line2, line3, line4])
380 plt.title("Error with increasing number of elements")
381 plt.ylabel("ln(error)")
382 plt.xlabel("ln(No. of elements)")
383 plt.show()
384 # plt.title("Strain energy with increasing number of elements")
385 # plt.ylabel("ln(Strain energy)")
386 # plt.xlabel("ln(No. of elements)")
387 # plt.show()
388 # plt.title("Strain energy of the error with increasing number of elements")
389 # plt.ylabel("ln(Strain energy of the error)")
390 # plt.xlabel("ln(No. of elements)")
391 # plt.show()
392 # plt.title("Relative error with increasing number of elements")
393 # plt.ylabel("ln(Relative error)")
394 # plt.xlabel("ln(No. of elements)")
395 # plt.show()

```

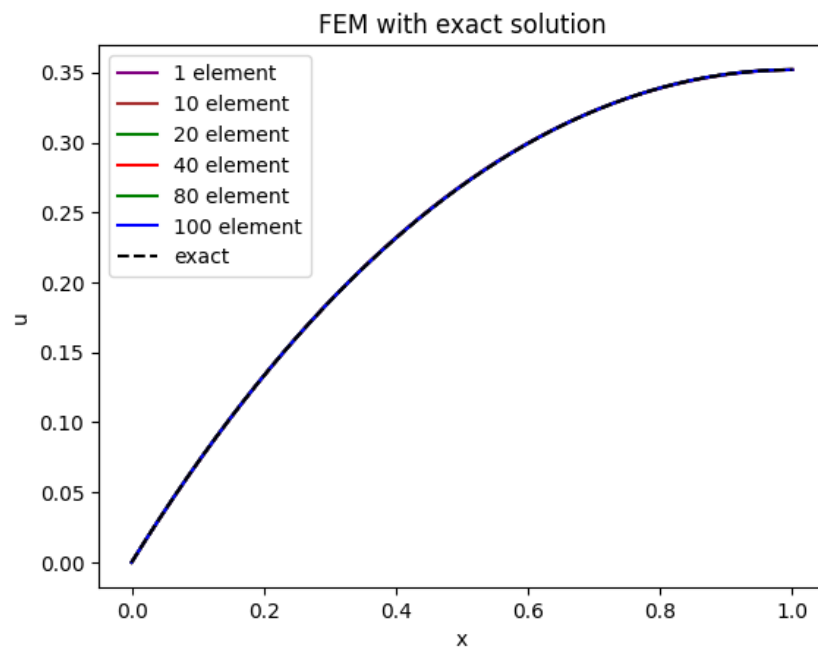
## OUTPUT PLOTS



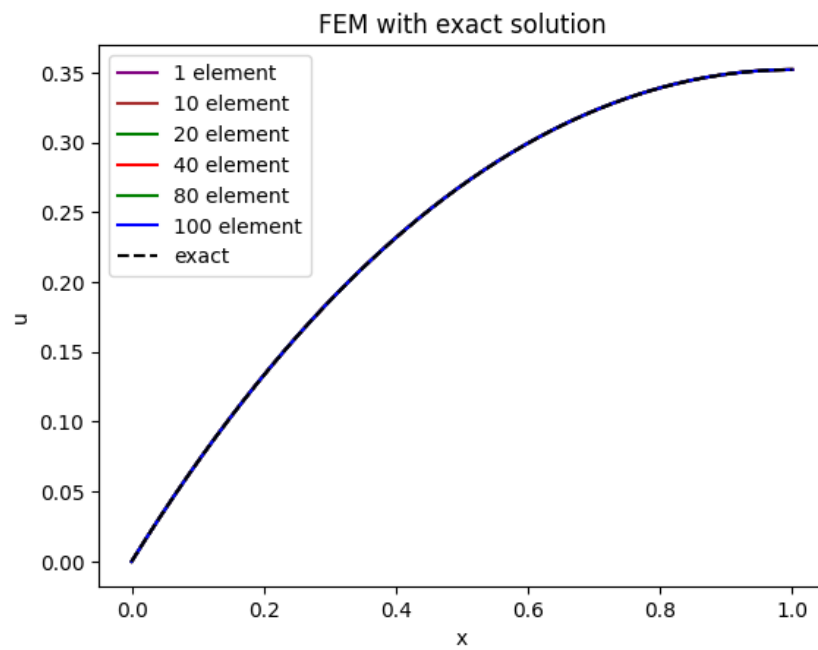
## Quadratic Element



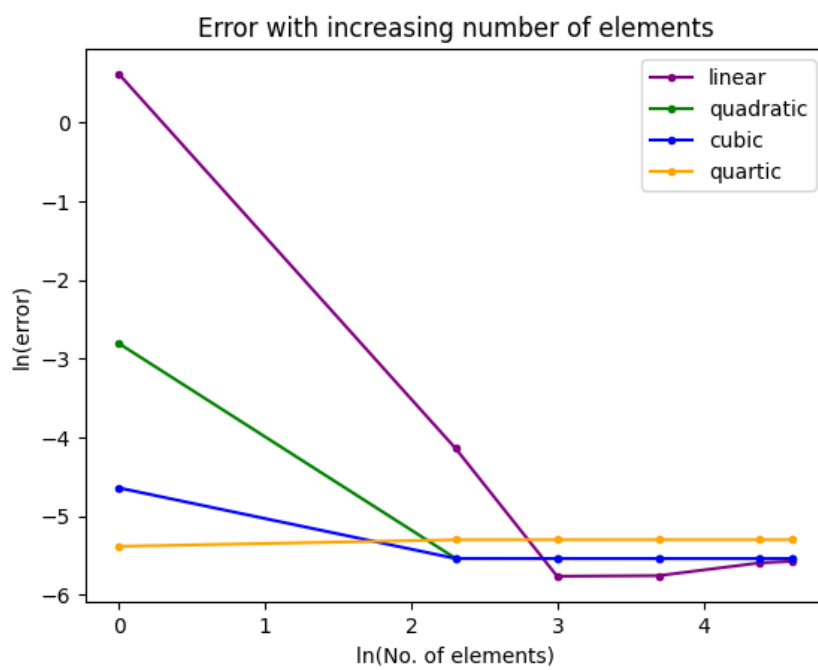
## Cubic Element



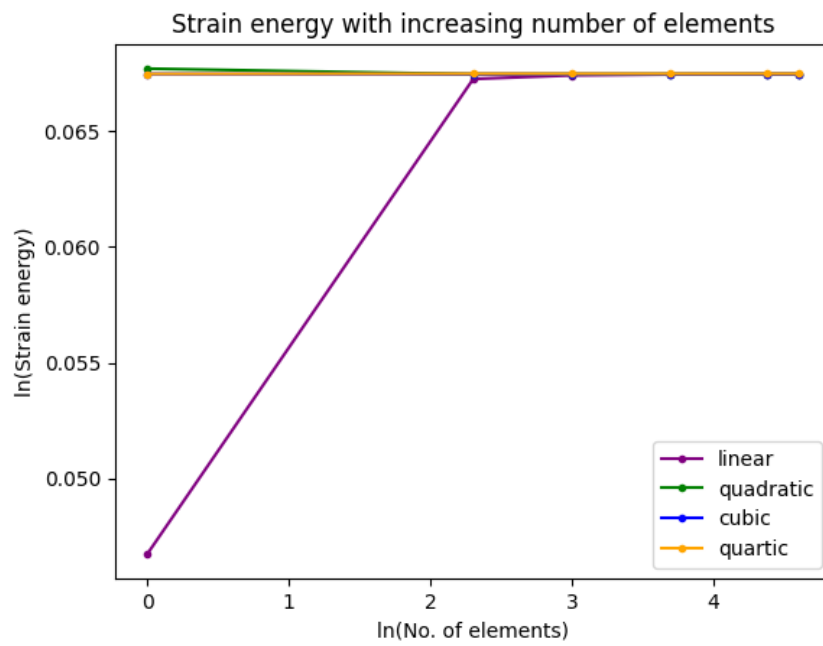
## Quartic Element



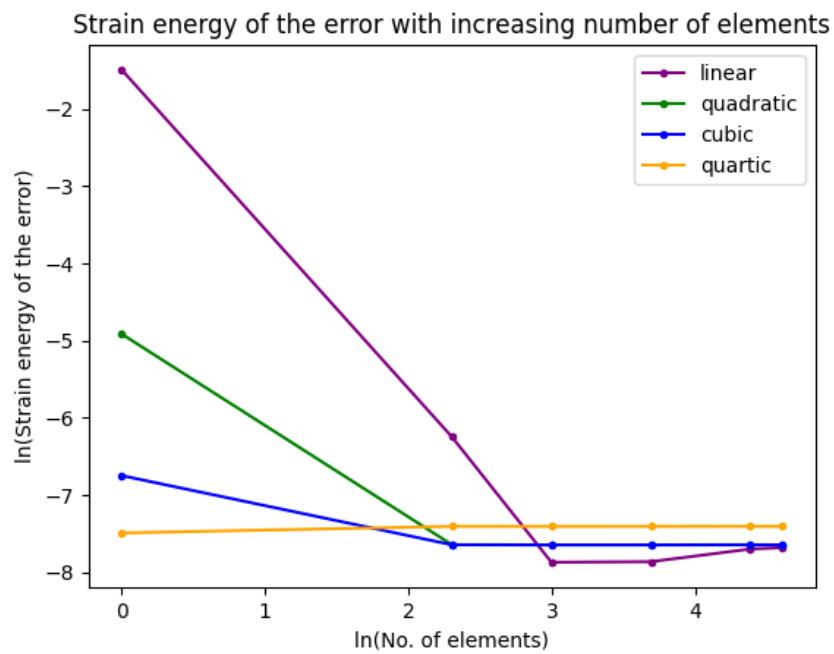
## Error



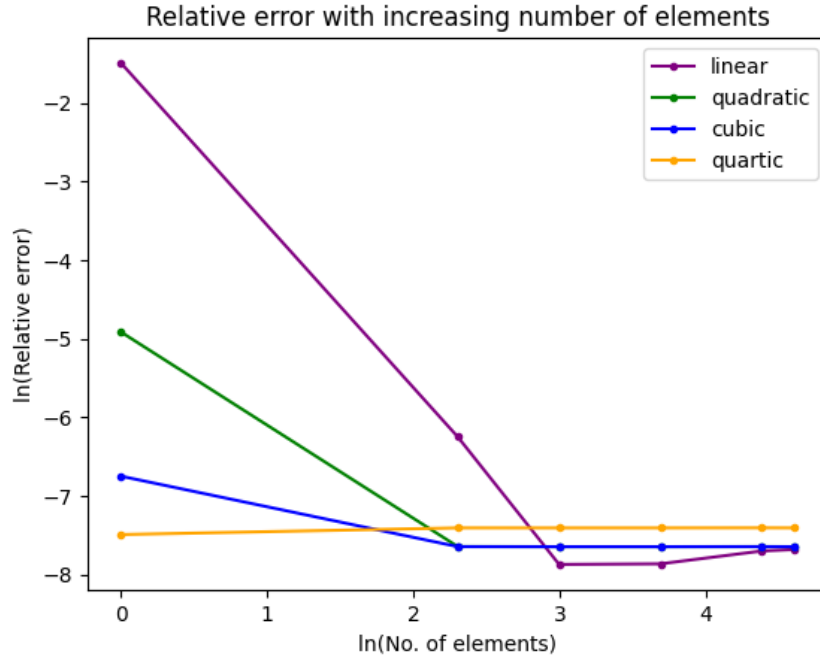
## Strain Energy



## 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*

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 def FEM_hp(n,p,choice,bc1,bc2,aeco,cco,fco):
4     def shapeFunctions(p,co,cod,choice):
5         if(choice==1):
6             if (p==1):
7                 co=[[0.5, -0.5],
8                    [0.5,0.5]]
9                 cod=[[-0.5],
10                    [0.5]]
11             elif (p==2):
12                 co=[[0, -0.5, 0.5],
13                    [1, 0, -1],
14                    [0, 0.5, 0.5]]
15                 cod=[[-0.5, 1],
16                    [0, -2],
17                    [0.5, 1]]
18             elif (p==3):
19                 co=[[-0.0625, 0.0625, 0.5625, -0.5625],
20                    [0.5625, -1.6875, -0.5625, 1.6875],
21                    [0.5625, 1.6875, -0.5625, -1.6875],
22                    [-0.0625, -0.0625, 0.5625, 0.5625]]
23                 cod=[[0.0625, 1.125, -1.6875],
24                    [-1.6875, -1.125, 5.0625],
25                    [1.6875, -1.125, -5.0625],
26                    [-0.0625, 1.125, 1.6875]]
27             elif (p==4):
28                 co=[[0, 0.1667, -0.1667, -0.6667, 0.6667],
29                    [0, -1.3333, 2.6667, 1.3333, -2.6667],

```

```

30         [1,0,-5,0,4],
31         [0,1.3333,2.6667,-1.3333,-2.6667],
32         [0,-0.1667,-0.1667,0.6667,0.6667]]
33     cod=[[0.1667,-0.3333,-2,2.6667],
34          [-1.3333,5.3333,4,-10.6667],
35          [0,-10,0,16],
36          [1.3333,5.3333,-4,-10.6667],
37          [-0.1667,-0.3333,2,2.6667]]
38     else:
39         if (p==1):
40             co=[[0.5,-0.5],
41                [0.5,0.5]]
42             cod=[[ -0.5],
43                 [0.5]]
44         elif (p==2):
45             co=[[0.5,-0.5,0,0,0],
46                [-0.6124,0,0.6124,0,0],
47                [0.5,0.5,0,0,0]]
48             cod=[[ -0.5,0],
49                 [0,1.2247],
50                 [0.5,0]]
51         elif (p==3):
52             co=[[0.5,-0.5,0,0,0],
53                [0,-0.7906,0,0.7906,0],
54                [-0.6124,0,0.6124,0,0],
55                [0.5,0.5,0,0,0]]
56             cod=[[ -0.5,0,0,0],
57                 [-0.7906,0,2.3717,0],
58                 [0,1.2247,0,0],
59                 [0.5,0,0,0]]
60         elif (p==4):
61             co=[[0.5,-0.5,0,0,0],
62                [0.2338,0,-1.4031,0,1.1693],
63                [0,-0.7906,0,0.7906,0],
64                [-0.6124,0,0.6124,0,0],
65                [0.5,0.5,0,0,0]]
66             cod=[[ -0.5,0,0,0],
67                 [0,-2.8062,0,4.6771],

```



```

68         [-0.7906,0,2.3717,0],
69         [0,1.2247,0,0],
70         [0.5,0,0,0]]
71     return co,cod
72 def integrate(re):
73     coi=np.array([[0.23862,0.46791],[-0.23862,0.46791],[0.66121,0.36076]])
74     coi=np.append(coi,
75     [[-0.66121,0.36076],[0.93247,0.17132],[-0.93247,0.17132]], axis=0)
76     s=0
77     for i in range(0,6):
78         su=0
79         for j in range(0,int(re.shape[0])):
80             su=su+(re[j]*coi[i][0]**j)
81             s=s+su*coi[i][1]
82     return s
83 def elementMatrix(ae,c,f,p):
84     for i in range(0, p+1):
85         for j in range(0, p+1):
86             ke[i][j]=(2/h)*integrate(np.polynomial.polynomial.polymul(np.
87             polynomial.polynomial.polymul(cod[i],cod[j]),ae[0]))
88             ge[i][j]=(h/2)*integrate(np.polynomial.polynomial.polymul(np.
89             polynomial.polynomial.polymul(co[i],co[j]),c[0]))
90             fe[i][0]=(h/2)*integrate(np.polynomial.polynomial.polymul(co[i],f
91             [0]))
92     return ke,ge,fe
93 def getValue(co,x,p,d):
94     if(d==0):
95         s=0
96         for i in range(0,p+1):
97             s=s+co[i]*x**i
98     else:
99         s=0
100         for i in range(0,p):
101             s=s+co[i]*x**i
102     return s
103 co=np.zeros((5,5))
104 cod=np.zeros((5,5))
105 co,cod=shapeFunctions(p,co,cod,choice)

```

```

102     if(bc1==2):
103         force1=0.3183
104     elif (bc1==1):
105         dis1=0
106     else:
107         spr1=float(input("Enter spring constant= "))
108         dev1=float(input("Enter spring deviation= "))
109
110     if(bc2==2):
111         force2=0
112     elif(bc2==1):
113         dis2=float(input("Enter displacement= "))
114     else:
115         spr2=10
116         dev2=0
117     h=1/n
118     nodeLocations=np.zeros((n,2))
119     for i in range(0,n):
120         for j in range(0,2):
121             if (i==j==0):
122                 continue
123             elif (j%2==0):
124                 nodeLocations[i][j]=nodeLocations[i-1][j+1]
125             else:
126                 nodeLocations[i][j]=nodeLocations[i][j-1]+h
127
128     K=np.zeros((n*p+1,n*p+1))
129     G=np.zeros((n*p+1,n*p+1))
130     F=np.zeros((n*p+1,1))
131     Q=np.zeros((n*p+1,1))
132     for i in range(0,n):
133         ke=np.zeros((p+1,p+1))
134         ge=np.zeros((p+1,p+1))
135         fe=np.zeros((p+1,1))
136         sunod=(nodeLocations[i][0]+nodeLocations[i][1])/2
137         aecof=np.array([[aeco[0][0]+aeco[0][1]*sunod+aeco[0][2]*sunod**2,aeco
138         [0][1]*(h/2)+aeco[0][2]*h*sunod,aeco[0][2]*(h/4)]])
139         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)]]))
139     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]]])
140     ke,ge,fe=elementMatrix(aecof,ccof,fcof,p)
141
142     if (i==0):
143         K[:p+1,:p+1]+=ke
144         G[:p+1,:p+1]+=ge
145         F[:p+1,0:1]+=fe
146     else:
147         K[i*p:i*p+p+1,i*p:i*p+p+1]+=ke
148         G[i*p:i*p+p+1,i*p:i*p+p+1]+=ge
149         F[i*p:i*p+p+1,0:1]+=fe
150     KF=K+G
151     if (bc1==1):
152         for i in range (1,n*p+1):
153             F[i]=F[i]-dis1*KF[i][0]
154         for i in range(0,n*p+1):
155             for j in range (0,n*p+1):
156                 if(i==0 or j==0):
157                     KF[i][j]=0
158     KF[0][0]=1
159     F[0][0]=dis1
160     Q[0][0]=0
161     if (bc2==1):
162         for i in range (0,n*p):
163             F[i]=F[i]-dis2*KF[i][n*p]

```

```

164         for i in range(0,n*p+1):
165             for j in range (0,n*p+1):
166                 if(i==n*p or j==n*p):
167                     KF[i][j]=0
168
169                 KF[n*p][n*p]=1
170                 F[n*p][0]=dis2
171                 Q[n*p][0]=0
172
173             if (bc1==2):
174                 Q[0][0]=force1
175
176             if (bc2==2):
177                 Q[n*p][0]=force2
178
179             if (bc1==3):
180                 KF[0][0]+=spr1
181                 Q[0][0]=spr1*dev1
182
183             if (bc2==3):
184                 KF[n*p][n*p]+=spr2
185                 Q[n*p][0]=spr2*dev2
186
187
188         U=np.linalg.inv(KF)@(F+Q)
189
190
191         x=np.linspace(0,1,1000)
192         yh=[]
193         yhd=[]
194
195         for i in range(0,n):
196             for j in range(0,1000):
197                 if(x[j]>=nodeLocations[i][0] and x[j]<=nodeLocations[i][1]):
198                     aux=(2*x[j]-(nodeLocations[i][0]+nodeLocations[i][1]))/h
199                     fr=0
200                     frd=0
201                     b=0
202
203                     for m in range(i*p,i*p+p+1):
204                         fr=fr+(U[m][0]*getValue(co[b],aux,p,0))
205                         frd=frd+(U[m][0]*getValue(cod[b],aux,p,1))*(2/h)
206                         b=b+1
207
208                     yh.append(fr)
209                     yhd.append(frd)
210
211         ye=[]
212         yed=[]

```

```

202     i=0
203     while (i<1):
204         ye.append(((np.sin(np.pi*i))/(np.pi**2))+(0.1/np.pi))
205         yed.append((np.cos(np.pi*i))/np.pi)
206         i=i+0.001
207
208     if (np.size(yh)<np.size(ye)):
209         for i in range(0,(np.size(ye)-np.size(yh))):
210             ye.pop()
211             yed.pop()
212             x=x[:-1]
213     elif (np.size(ye)<np.size(yh)):
214         for i in range(0,(np.size(yh)-np.size(ye))):
215             yh.pop()
216             yhd.pop()
217     return yh,yhd,ye,yed,x
218
219 def strainEnergy(ae,fx,fxd,c):
220     s=0
221     for i in range(0,998):
222         fx1=ae*fxd[i]**2+c*fx[i]**2
223         fx2=ae*fxd[i+1]**2+c*fx[i]**2
224         s=s+0.5*(fx1+fx2)*(x[i+1]-x[i])
225     return s
226
227 numElements=[1,10,20,40,80,100]
228 order=[1,2,3,4]
229 choice=1
230 aeco=np.zeros((1,3))
231 cco=np.zeros((1,3))
232 fco=np.zeros((1,10))
233 aeco[0][1]=1
234 cco[0][0]=0
235 fco[0][0]=0
236 fco[0][1]=3.1416
237 fco[0][2]=0
238 fco[0][3]=-5.1677
239 fco[0][5]=2.5501

```

```

240 fco[0][7]=-0.599264
241 fco[0][9]=0.08214588
242 bc1=2
243 bc2=3
244 for p in order:
245     yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)
246     line1, =plt.plot(x,yh, label="1 element", color='purple')
247     yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
248     line2, =plt.plot(x,yh, label="10 element", color='brown')
249     yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
250     line3, =plt.plot(x,yh, label="20 element", color='green')
251     yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
252     line4, =plt.plot(x,yh, label="40 element", color='red')
253     yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
254     line7, =plt.plot(x,yh, label="80 element", color='orange')
255     yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
256     line5, =plt.plot(x,yh, label="100 element", color='blue')
257     line6, =plt.plot(x,ye, label="exact", color='black', linestyle='dashed')
258     plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])
259     plt.title("Plot of FEM with exact solution")
260     plt.xlabel("x")
261     plt.ylabel("u")
262     plt.show()
263 for p in order:
264     yh,yhd,ye,yed,x=FEM_hp(1,p,choice,bc1,bc2,aeco,cco,fco)
265     # line1, =plt.plot(x,yhd, label="1 element")
266     yh,yhd,ye,yed,x=FEM_hp(10,p,choice,bc1,bc2,aeco,cco,fco)
267     # line2, =plt.plot(x,yhd, label="10 element")
268     yh,yhd,ye,yed,x=FEM_hp(20,p,choice,bc1,bc2,aeco,cco,fco)
269     # line3, =plt.plot(x,yhd, label="20 element")
270     yh,yhd,ye,yed,x=FEM_hp(40,p,choice,bc1,bc2,aeco,cco,fco)
271     # line4, =plt.plot(x,yhd, label="40 element")
272     yh,yhd,ye,yed,x=FEM_hp(80,p,choice,bc1,bc2,aeco,cco,fco)
273     # line7, =plt.plot(x,yhd, label="80 element")
274     yh,yhd,ye,yed,x=FEM_hp(100,p,choice,bc1,bc2,aeco,cco,fco)
275     # line5, =plt.plot(x,yhd, label="100 element")
276     # line6, =plt.plot(x,yed, label="exact")
277     # plt.legend(handles=[line1, line2, line3, line4, line7, line5, line6])

```

```

278 # plt.title("Plot of derivatives")
279 # plt.xlabel("x")
280 # plt.ylabel("F")
281 # plt.show()
282 error=[]
283 logElements=[]
284 energy=[]
285 energyerr=[]
286 logEnergyError=[]
287 for n in numElements:
288     yh,yhd,ye,yed,x=FEM_hp(n,1,choice,bc1,bc2,aeco,cco,fco)
289     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
290     uex=np.linalg.norm(np.array(ye))
291     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
292     error.append(er)
293     logEnergyError.append(np.log(energynorm/uex))
294     logElements.append(np.log(n))
295     strar=np.array(ye)-np.array(yh)
296     strard=np.array(yed)-np.array(yhd)
297     eg=strainEnergy(1,yhd,yh,0)
298     erreg=strainEnergy(1,strard,strar,0)
299     energyerr.append(erreg)
300     energy.append(eg)
301 print(logElements,error)
302 line1=plt.plot(logElements,error, label="linear",marker=".", color='purple')
303 # line1=plt.plot(logElements,energy, label="linear",marker=".", color='purple
    ')
304 # line1=plt.plot(logElements,energyerr, label="linear",marker=".", color='
    purple')
305 # line1=plt.plot(logElements,logEnergyError, label="linear",marker=".", color
    ='purple')
306 error=[]
307 logElements=[]
308 energy=[]
309 energyerr=[]
310 logEnergyError=[]
311 for n in numElements:
312     yh,yhd,ye,yed,x=FEM_hp(n,2,choice,bc1,bc2,aeco,cco,fco)

```

```

313     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
314     uex=np.linalg.norm(np.array(ye))
315     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
316     error.append(er)
317     logEnergyError.append(np.log(energynorm/uex))
318     logElements.append(np.log(n))
319     strar=np.array(ye)-np.array(yh)
320     strard=np.array(yed)-np.array(yhd)
321     eg=strainEnergy(1,yhd,yh,0)
322     erreg=strainEnergy(1,strard,strar,0)
323     energyerr.append(erreg)
324     energy.append(eg)
325 print(logElements,error)
326 line2=plt.plot(logElements,error, label="quadratic",marker=".", color='green')
327 # line2=plt.plot(logElements,energy, label="quadratic",marker=".", color='
    green')
328 # line2=plt.plot(logElements,energyerr, label="quadratic",marker=".", color='
    green')
329 # line2=plt.plot(logElements,logEnergyError, label="quadratic",marker=".",
    color='green')
330 error=[]
331 logElements=[]
332 energy=[]
333 energyerr=[]
334 logEnergyError=[]
335 for n in numElements:
336     yh,yhd,ye,yed,x=FEM_hp(n,3,choice,bc1,bc2,aeco,cco,fco)
337     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
338     uex=np.linalg.norm(np.array(ye))
339     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
340     error.append(er)
341     logEnergyError.append(np.log(energynorm/uex))
342     logElements.append(np.log(n))
343     strar=np.array(ye)-np.array(yh)
344     strard=np.array(yed)-np.array(yhd)
345     eg=strainEnergy(1,yhd,yh,0)
346     erreg=strainEnergy(1,strard,strar,0)
347     energyerr.append(erreg)

```



```

348     energy.append(eg)
349 print(logElements,error)
350 line3=plt.plot(logElements,error, label="cubic",marker=".", color='blue')
351 # line3=plt.plot(logElements,energy, label="cubic",marker=".", color='blue')
352 # line3=plt.plot(logElements,energyerr, label="cubic",marker=".", color='blue
    ')
353 # line3=plt.plot(logElements,logEnergyError, label="cubic",marker=".", color='
    blue')
354 error=[]
355 logElements=[]
356 energy=[]
357 energyerr=[]
358 logEnergyError=[]
359 energyexact=[]
360 for n in numElements:
361     yh,yhd,ye,yed,x=FEM_hp(n,4,choice,bc1,bc2,aeco,cco,fco)
362     er=np.log(np.linalg.norm(np.array(ye)-np.array(yh)))
363     uex=np.linalg.norm(np.array(ye))
364     energynorm=np.linalg.norm(np.array(ye)-np.array(yh))
365     error.append(er)
366     logEnergyError.append(np.log(energynorm/uex))
367     logElements.append(np.log(n))
368     strar=np.array(ye)-np.array(yh)
369     strard=np.array(yed)-np.array(yhd)
370     eg=strainEnergy(1,yhd,yh,0)
371     energyexact.append(strainEnergy(1,yed,ye,0))
372     erreg=strainEnergy(1,strard,strar,0)
373     energyerr.append(erreg)
374     energy.append(eg)
375 print(logElements,error)
376 line4=plt.plot(logElements,error, label="quartic",marker=".", color='orange')
377 # line4=plt.plot(logElements,energy, label="quartic",marker=".", color='orange
    ')
378 # line4=plt.plot(logElements,energyerr, label="quartic",marker=".", color='
    orange')
379 # line4=plt.plot(logElements,logEnergyError, label="quartic",marker=".", color
    ='orange')
380

```

```

381 plt.legend(handles=[line1, line2, line3, line4])
382 plt.title("Error with increasing number of elements")
383 plt.ylabel("ln(error)")
384 plt.xlabel("ln(No. of elements)")
385 plt.show()

386 # plt.title("Strain energy with increasing number of elements")
387 # plt.ylabel("ln(Strain energy)")
388 # plt.xlabel("ln(No. of elements)")
389 # plt.show()

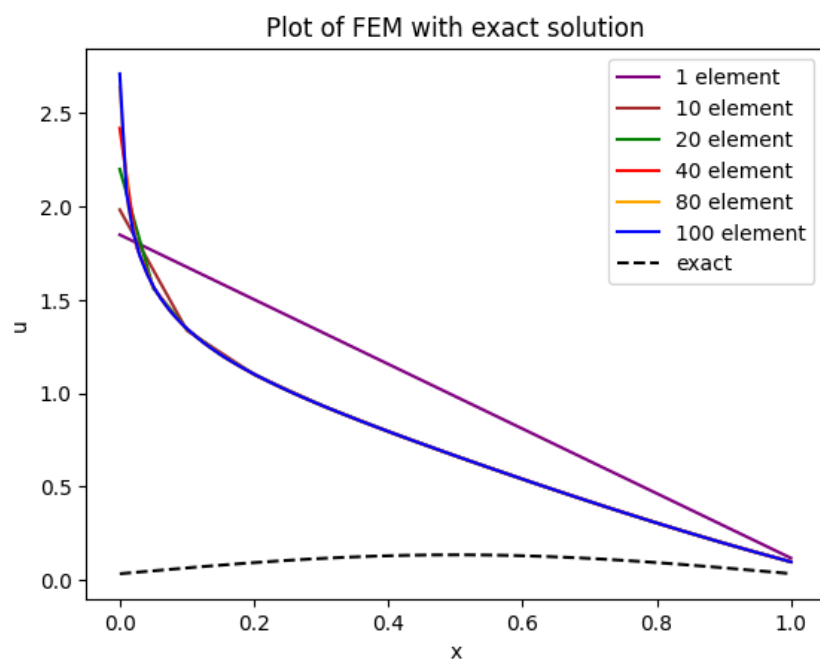
390 # plt.title("Strain energy of the error with increasing number of elements")
391 # plt.ylabel("ln(Strain energy of the error)")
392 # plt.xlabel("ln(No. of elements)")
393 # plt.show()

394 # plt.title("Relative error with increasing number of elements")
395 # plt.ylabel("ln(Relative error)")
396 # plt.xlabel("ln(No. of elements)")
397 # plt.show()

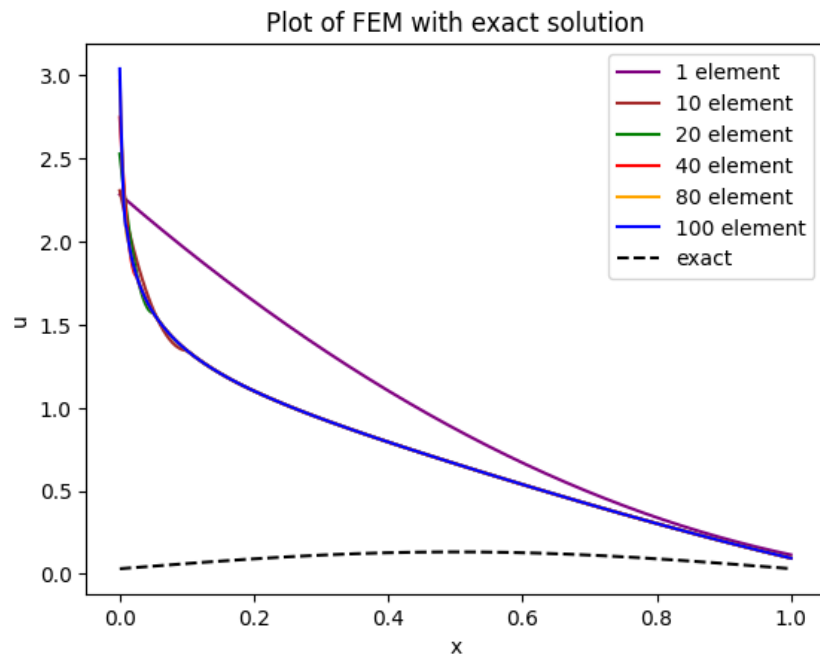
```

## OUTPUT PLOTS

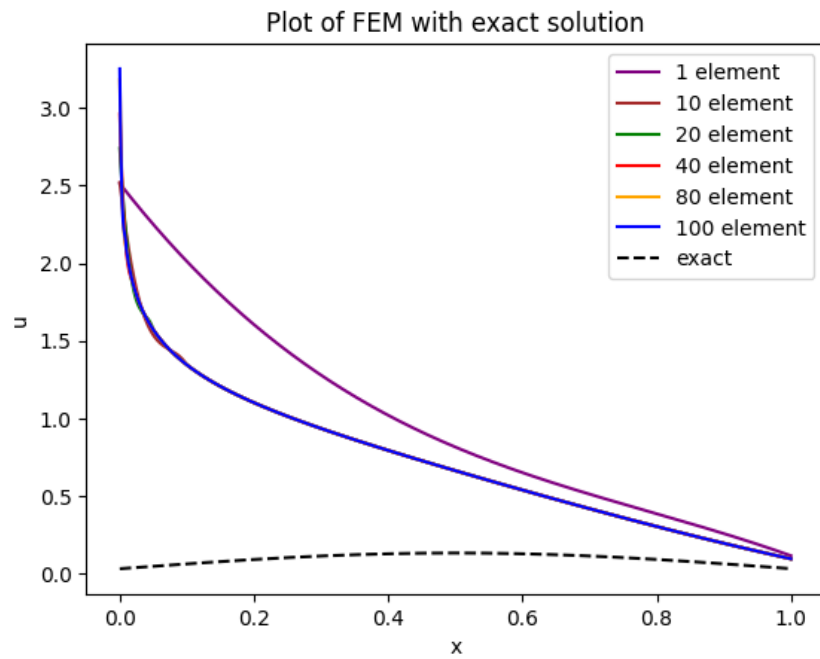
### Linear Element



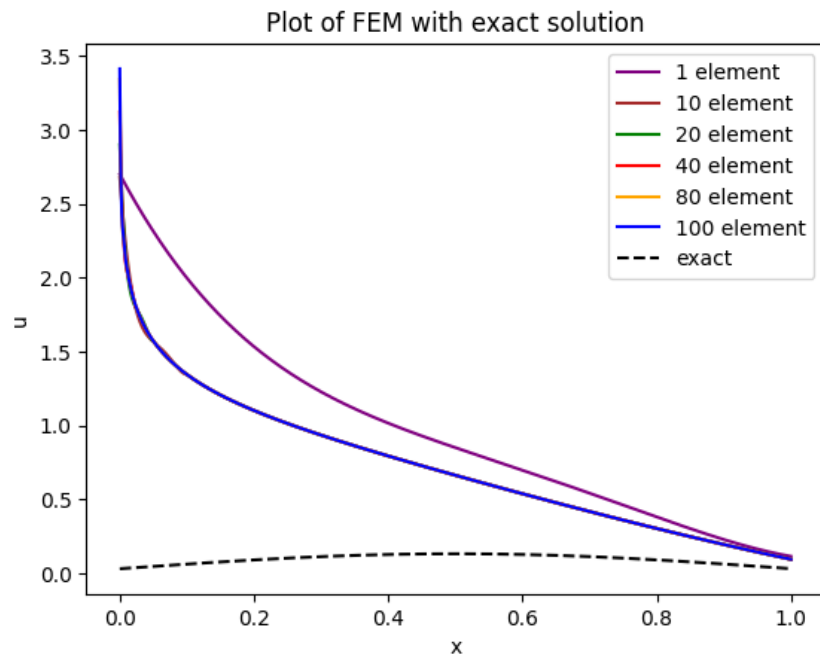
## Quadratic Element



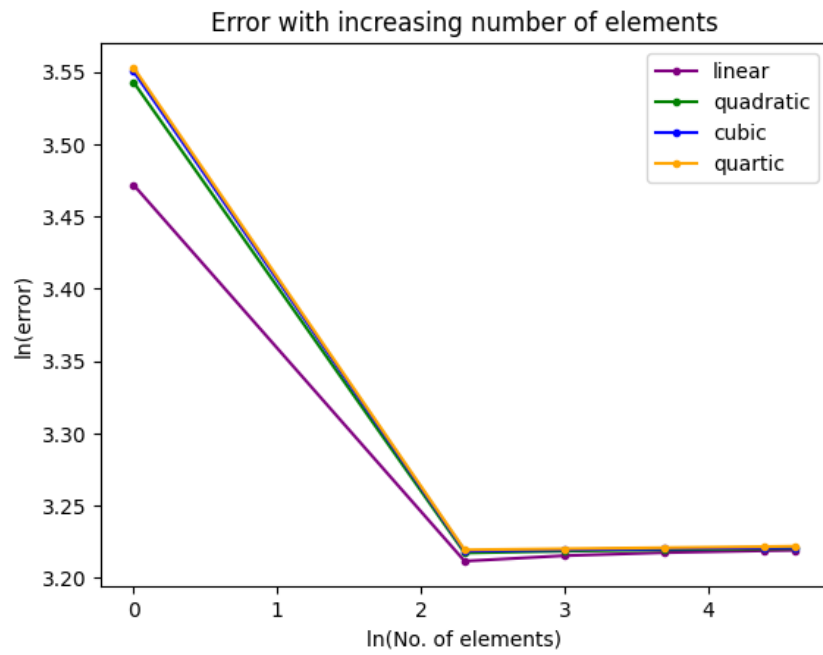
## Cubic Element



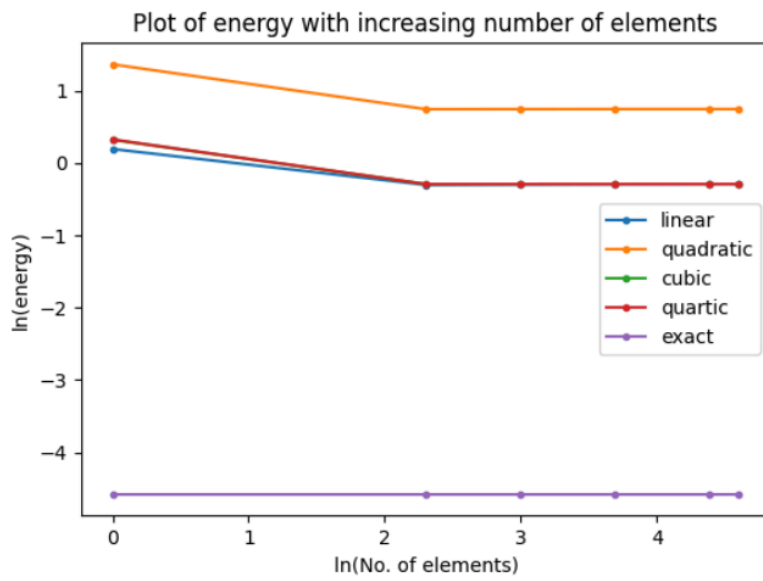
## Quartic Element



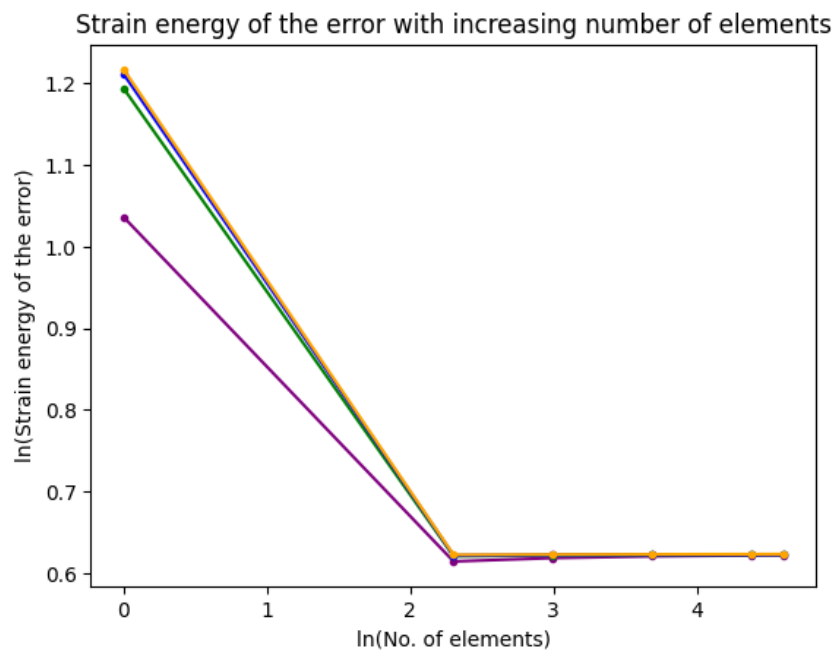
## Error



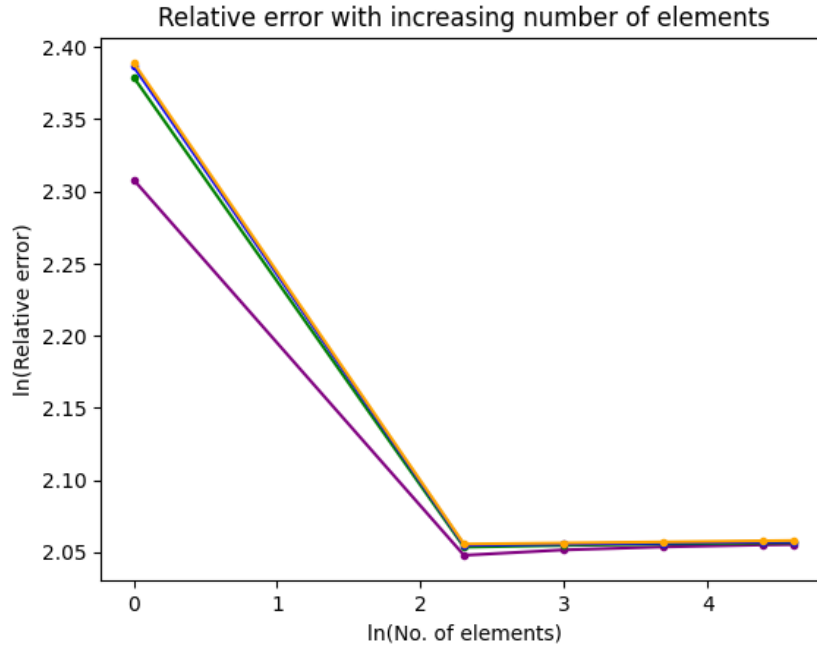
## Strain Energy



## 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.