

MAE7750_HW2

March 10, 2023

1 MAE 7750: HW 2

Hanfeng Zhai

Cornell University

1.1 Hands-on Numerical Implementation

1.1.1 Scenario 1: $x = 1.8$

Newton-Raphson Method

```
[47]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
v1_values_NR = []; v2_values_NR = []
G1_values_NR = []; G2_values_NR = []
R1_values_NR = []; R2_values_NR = []
num_iter_NR = []

v_0 = np.transpose(np.array([0.042, 0.042]))
lamb = np.arange(0.25, 10, 0.25)
P1 = 1; P2 = 0
v = v_0
x = 1.8
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1, 0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); print(G_benchmark)
# K_T = np.array([0.6*v[0]+6, -2*x*v[1]], [-1, 1]); #print("K_T", K_T)

for lamb_i in lamb:
    ii = 1
    num_iter = 0
    while True:
        # print("check lambda:", lamb_i)
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
```

```

P = np.transpose(np.array([1,0]))
G = R - lamb_i * P;
K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])
Delta_v = -np.dot(np.linalg.inv(K_T),G);
v = v + Delta_v; #print("v:",v)
abs_G = np.sqrt(G[0]**2 + G[1]**2); #print("abs G:", abs_G)
TOL = abs_G; #print("tol:", TOL); #print("<=====>")
num_iter = num_iter+1
if TOL <= 1e-4 * G_benchmark or ii > 15:
    v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
    G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
    R1_values_NR.append(R[0]);R2_values_NR.append(R[1])
    num_iter_NR.append(num_iter)
    break
ii += 1

v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
R1_values_NR.append(R[0]);R2_values_NR.append(R[1])

v1_loop_NR = np.array(v1_values_NR);v2_loop_NR = np.array(v2_values_NR)
G1_loop_NR = np.array(G1_values_NR);G2_loop_NR = np.array(G2_values_NR)
R1_loop_NR = np.array(R1_values_NR);R2_loop_NR = np.array(R2_values_NR)

```

0.2488396176

Modified Newton-Raphson Method

```

[49]: v1_values_MNR = [];v2_values_MNR = []
      G1_values_MNR = [];G2_values_MNR = []
      R1_values_MNR = [];R2_values_MNR = []
      num_iter_MNR = []

      v = v_0

      G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2);
      K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]]);
      Delta_v = []; inv_K = np.linalg.inv(K_T);
      for lamb_i in lamb:
          ii = 1
          num_iter = 0
          while True:
              R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
              P = np.transpose(np.array([1,0]))
              G = R - lamb_i * P; #print(G)
              Delta_v = -np.dot(inv_K,G);
              v = v + Delta_v;

```

```

abs_G = np.sqrt(G[0]**2 + G[1]**2);
TOL = abs_G;
num_iter = num_iter+1
if TOL <= 1e-4 * G_benchmark or ii > 15:
    v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
    G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
    R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])
    num_iter_MNR.append(num_iter)
    break
ii += 1

v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])

v1_loop_MNR = np.array(v1_values_MNR);v2_loop_MNR = np.array(v2_values_MNR)
G1_loop_MNR = np.array(G1_values_MNR);G2_loop_MNR = np.array(G2_values_MNR)
R1_loop_MNR = np.array(R1_values_MNR);R2_loop_MNR = np.array(R2_values_MNR)

```

Modified Newton-Raphson Method with Line Search

```

[50]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

x=1.8
lamb = np.arange(0.25, 10, 0.25)

v1_values_MNRLS = [];v2_values_MNRLS = []
G1_values_MNRLS = [];G2_values_MNRLS = []
R1_values_MNRLS = [];R2_values_MNRLS = []
num_iter_MNRLS = []
v_0 = np.transpose(np.array([0,0]))
v = v_0
R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2);
K_T = np.array([[0.6*v[0]+6, -2*x*v[1]], [-1, 1]]);
Delta_v = []; inv_K = np.linalg.inv(K_T);

alpha_0 = 0; alpha_1 = 1; alpha = []

def g_func(alpha, Delta_v, lamb, vv):
    R = np.transpose(np.array([0.2*(vv[0] + alpha*Delta_v[0])**3 - x*(vv[1] +
    ↪alpha*Delta_v[1])**2 + 6*(vv[0] + alpha*Delta_v[0]),\

```

```

(vv[1] + alpha*Delta_v[1])-(vv[0] +
→alpha*Delta_v[0]))))
    P = np.transpose(np.array([1,0])); G = R - lamb * P;
→#print("R,G,P,alpha,Delta_v:",R,G,P,alpha,Delta_v)
    output = np.dot(Delta_v, G); #print("g function:",output)
    return output

for lamb_i in lamb:
    ii = 1
    alpha_prev = alpha_0; alpha = alpha_1

    Delta_v = -np.dot(inv_K, G0)
    num_iter = 0

    while True:
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]]));
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; Delta_v = -np.dot(inv_K,G);
        denom = g_func(alpha, Delta_v, lamb_i, v) - g_func(alpha_prev, Delta_v,
→lamb_i, v)
        interval = g_func(alpha, Delta_v, lamb_i, v) * ( (alpha - alpha_prev) /
→denom )
        alpha_prev = alpha
        alpha = alpha - interval
        v = v + alpha * Delta_v
        num_iter = num_iter + 1
        if np.abs(g_func(alpha, Delta_v, lamb_i, v)) > 0.5 * np.abs(g_func(0,
→Delta_v, lamb_i, v)) or ii > 5:
            v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
            G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
            R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])
            num_iter_MNRLS.append(num_iter); #print(np.abs(g_func(alpha,
→Delta_v, lamb_i, v)), "/", 0.5 * np.abs(g_func(0, Delta_v, lamb_i, v)))
            break
        ii += 1

    v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
    G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
    R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])

v1_loop_MNRLS = np.array(v1_values_MNRLS);v2_loop_MNRLS = np.
→array(v2_values_MNRLS)
G1_loop_MNRLS = np.array(G1_values_MNRLS);G2_loop_MNRLS = np.
→array(G2_values_MNRLS)

```

```
R1_loop_MNRLS = np.array(R1_values_MNRLS);R2_loop_MNRLS = np.
→array(R2_values_MNRLS)
```

BFGS Method

```
[52]: import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
import matplotlib as mpl

v1_values_BFGS = [];v2_values_BFGS = []
G1_values_BFGS = [];G2_values_BFGS = []
R1_values_BFGS = [];R2_values_BFGS = []
num_iter_BFGS = []

v_0 = np.transpose(np.array([0,0]))
lamb = np.arange(0.5, 10, 0.25)
v = v_0
x = 1.8
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2)
P = np.transpose(np.array([1,0]))

left_temp_ab = np.ones([2,2]);
right_temp_ba = np.ones([2,2]);

for lamb_i in lamb:
    ii = 1; K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])
    num_iter = 0
    # We need to initialize the BFGS with Newton-Raphson
    H_prev = np.linalg.inv(K_T);
    R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
    G = R - lamb_i * P;
    Delta_v_NR = -np.dot(np.linalg.inv(K_T),G);
    v_prev = v; v = v + Delta_v_NR;
    R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
    G_prev = G; G = R - lamb_i * P;
    while True:
        w = v - v_prev; g = G - G_prev;

        a_temp_1 = np.matmul(np.transpose(g),w);
        a_temp_2 = 1/a_temp_1;
        a = a_temp_2*w;
```

```

b_temp_1 = - np.matmul(np.transpose(w), g);
b_temp_2 = np.matmul(np.transpose(w), G_prev);
b_temp_3 = b_temp_1/b_temp_2;
b_temp_4 = np.sqrt(b_temp_3);
b_temp_5 = b_temp_4 * G_prev
b = - (g - b_temp_5);

left_temp_ab[0][0] = a[0]*b[0]; left_temp_ab[0][1] = a[0]*b[1];
left_temp_ab[1][0] = a[1]*b[0]; left_temp_ab[1][1] = a[1]*b[1];
left_temp = np.eye(2)+left_temp_ab;

right_temp_ba[0][0] = b[0]*a[0];right_temp_ba[0][1] = b[0]*a[1];
right_temp_ba[1][0] = b[1]*a[0];right_temp_ba[1][1] = b[1]*a[1];
right_temp = np.eye(2)+right_temp_ba;

H = left_temp @ H_prev @ right_temp; H_prev = H;

Delta_v_BFGS = -np.matmul(H, G)
v_prev = v; v = v + Delta_v_BFGS
R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); G = R - lamb_i * P
abs_G = np.sqrt(G[0]**2 + G[1]**2);
TOL = abs_G;num_iter = num_iter+1
if TOL <= 1e-4 * G_benchmark or ii > 15:
    v1_values_BFGS.append(v[0]); v2_values_BFGS.append(v[1])
    G1_values_BFGS.append(G[0]); G2_values_BFGS.append(G[1])
    R1_values_BFGS.append(R[0]); R2_values_BFGS.append(R[1])
    num_iter_BFGS.append(num_iter)
    break
ii += 1

v1_values_BFGS.append(v[0]);v2_values_BFGS.append(v[1])
G1_values_BFGS.append(G[0]);G2_values_BFGS.append(G[1])
R1_values_BFGS.append(R[0]);R2_values_BFGS.append(R[1])

v1_loop_BFGS = np.array(v1_values_BFGS);v2_loop_BFGS = np.array(v2_values_BFGS)
G1_loop_BFGS = np.array(G1_values_BFGS);G2_loop_BFGS = np.array(G2_values_BFGS)
R1_loop_BFGS = np.array(R1_values_BFGS);R2_loop_BFGS = np.array(R2_values_BFGS)

```

Analytical Solution

```

[44]: import sympy as sym

v_analy, lamb_i= sym.symbols('v_analy lamb_i')

eq = sym.Eq(0.2*v_analy**3 - 1.8*v_analy**2 + 6*v_analy - lamb_i, 0)
v_analy_solns = []

```

```

lamb = np.arange(0, 10, 0.25)
for lamb_i_val in lamb:
    solns = sym.solve(eq.subs(lamb_i, lamb_i_val), v_analy)
    solns = solns[0]; #print(solns)
    v_analy_solns.append(solns)

v_analy_solns = np.array(v_analy_solns)

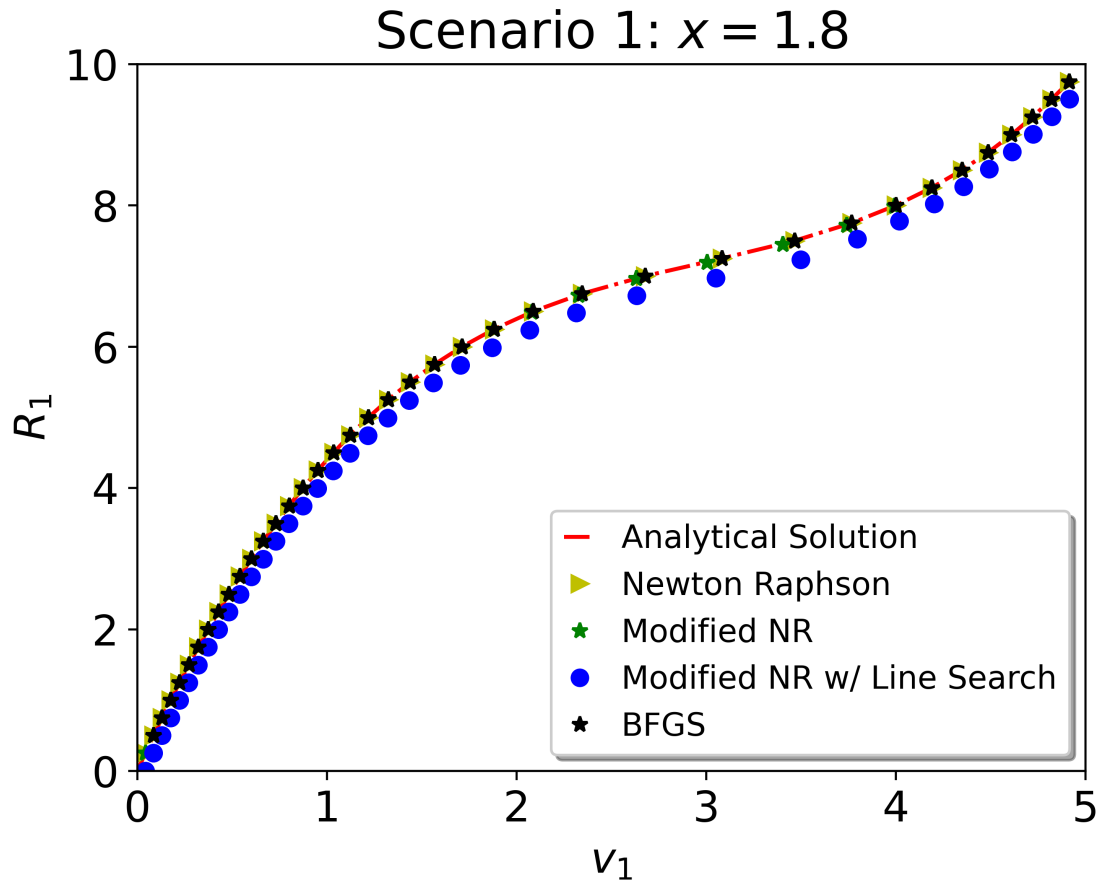
R_analy = 0.2*v_analy_solns**3 - 1.8*v_analy_solns**2 + 6*v_analy_solns

```

```

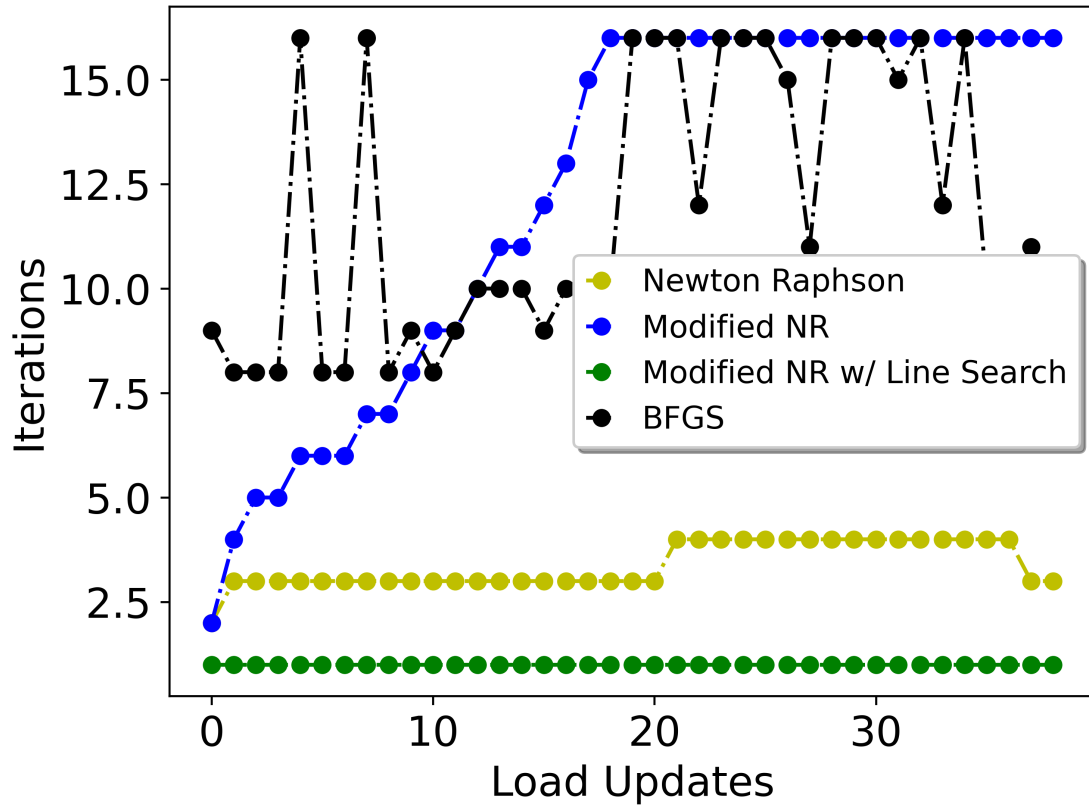
[53]: import matplotlib as mpl
plt.plot(v_analy_solns,R_analy,'r-.',label='Analytical Solution')
plt.plot(v1_loop_NR,R1_loop_NR,'y>',label='Newton Raphson')
plt.plot(v1_loop_MNR,R1_loop_MNR,'g*',label='Modified NR')
plt.plot(v1_loop_MNRLS,R1_loop_MNRLS,'bo',label='Modified NR w/ Line Search')
plt.plot(v1_loop_BFGS,R1_loop_BFGS,'k*',label='BFGS')
plt.xlabel("$v_1$")
plt.ylabel("$R_1$")
plt.ylim([0,10])
plt.xlim([0,5])
plt.title("Scenario 1: $x = 1.8$")
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
plt.show()
plt.figure(figsize=(5, 3)); mpl.rcParams.update({'font.size': 16})

```



<Figure size 2500x1500 with 0 Axes>

```
[56]: plt.plot(num_iter_NR, 'yo-.', label='Newton Raphson')
plt.plot(num_iter_MNR, 'bo-.', label='Modified NR')
plt.plot(num_iter_MNRLS, 'go-.', label='Modified NR w/ Line Search')
plt.plot(num_iter_BFGS, 'ko-.', label='BFGS')
plt.xlabel("Load Updates")
plt.ylabel("Iterations")
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
```

1.1.2 Scenario 2: $x = 2.1$

Newton-Raphson Method

```
[25]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
v1_values_NR = [];v2_values_NR = []
G1_values_NR = [];G2_values_NR = []
R1_values_NR = [];R2_values_NR = []
num_iter_NR = []

v_0 = np.transpose(np.array([0.042,0.042]))
lamb = np.arange(0.25, 10, 0.25)
v = v_0
x = 2.1
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); print(G_benchmark)
```

```

for lamb_i in lamb:
    ii = 1
    num_iter = 0
    while True:
        # print("check lambda:", lamb_i)
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; #print("G value:", G)
        K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])
        Delta_v = -np.dot(np.linalg.inv(K_T),G); #print("Delta v:", Delta_v)#-np.
→dot(np.linalg.inv(K_T), G);
        # print("K_T:", K_T); print("inverse K_T:", np.linalg.inv(K_T))
        v = v + Delta_v; #print("v:",v)
        abs_G = np.sqrt(G[0]**2 + G[1]**2); #print("abs G:", abs_G)
        TOL = abs_G; #print("tol:", TOL); #print("<=====>")
        num_iter = num_iter+1
        if TOL <= 1e-4 * G_benchmark or ii > 15:
            v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
            G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
            R1_values_NR.append(R[0]);R2_values_NR.append(R[1])
            num_iter_NR.append(num_iter)
            break
        ii += 1

    v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
    G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
    R1_values_NR.append(R[0]);R2_values_NR.append(R[1])

v1_loop_NR = np.array(v1_values_NR);v2_loop_NR = np.array(v2_values_NR)
G1_loop_NR = np.array(G1_values_NR);G2_loop_NR = np.array(G2_values_NR)
R1_loop_NR = np.array(R1_values_NR);R2_loop_NR = np.array(R2_values_NR)

```

0.2483104176

Modified Newton-Raphson Method

```

[26]: import numpy as np

v1_values_MNR = [];v2_values_MNR = []
G1_values_MNR = [];G2_values_MNR = []
R1_values_MNR = [];R2_values_MNR = []
num_iter_MNR = []
x = 2.1
v_0 = np.transpose(np.array([0.042,0.042]))
lamb = np.arange(0.25, 10, 0.25)
v = v_0

```

```

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); #print("tolerance:",
→G_benchmark*1e-4)

K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]]);
Delta_v = []; inv_K = np.linalg.inv(K_T); #print("inv K:", inv_K)
for lamb_i in lamb:
    ii = 1
    num_iter = 0
    while True:
        # print("check lambda:", lamb_i)
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; #print("G value:", G, "/", "inv_K:", inv_K)
        Delta_v = -np.dot(inv_K,G);
        v = v + Delta_v; #print("v value:", v, "/", "Delta v:", Delta_v)
        abs_G = np.sqrt(G[0]**2 + G[1]**2);
        TOL = abs_G; #print("=====", TOL, "=====")
        num_iter = num_iter + 1
        if TOL <= 1e-4 * G_benchmark or ii > 15:
            v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
            G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
            R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])
            num_iter_NR.append(num_iter)
            break
        ii += 1

    v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
    G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
    R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])

v1_loop_MNR = np.array(v1_values_MNR);v2_loop_MNR = np.array(v2_values_MNR)
G1_loop_MNR = np.array(G1_values_MNR);G2_loop_MNR = np.array(G2_values_MNR)
R1_loop_MNR = np.array(R1_values_MNR);R2_loop_MNR = np.array(R2_values_MNR)

```

Modified Newton-Raphson Method with Line Search

```

[27]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

lamb = np.arange(0.25, 10, 0.25)

```

```

v1_values_MNRLS = [];v2_values_MNRLS = []
G1_values_MNRLS = [];G2_values_MNRLS = []
R1_values_MNRLS = [];R2_values_MNRLS = []
num_iter_MNRLS = []
v_0 = np.transpose(np.array([0,0]))
v = v_0
R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2);
K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]]);
Delta_v = []; inv_K = np.linalg.inv(K_T);

alpha_0 = 0; alpha_1 = 1; alpha = []

def g_func(alpha, Delta_v, lamb, vv):
    R = np.transpose(np.array([0.2*(vv[0] + alpha*Delta_v[0])**3 - x*(vv[1] +
→alpha*Delta_v[1])**2 + 6*(vv[0] + alpha*Delta_v[0]),\
                                (vv[1] + alpha*Delta_v[1])-(vv[0] +
→alpha*Delta_v[0])]))
    P = np.transpose(np.array([1,0])); G = R - lamb * P;
→#print("R,G,P,alpha,Delta_v:",R,G,P,alpha,Delta_v)
    output = np.dot(Delta_v, G); #print("g function:",output)
    return output

for lamb_i in lamb:
    ii = 1
    alpha_prev = alpha_0; alpha = alpha_1
    Delta_v = -np.dot(inv_K, G0)
    num_iter = 0

    while True:
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]]));
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; Delta_v = -np.dot(inv_K,G);
        denom = g_func(alpha, Delta_v, lamb_i, v) - g_func(alpha_prev, Delta_v,
→lamb_i, v)
        interval = g_func(alpha, Delta_v, lamb_i, v) * ( (alpha - alpha_prev) /
→denom )

        alpha_prev = alpha
        alpha = alpha - interval
        v = v + alpha * Delta_v
        num_iter = num_iter+1

```

```

        if np.abs(g_func(alpha, Delta_v, lamb_i, v)) > 0.5 * np.abs(g_func(0,
→Delta_v, lamb_i, v)) or ii > 5:
            v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
            G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
            R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])
            num_iter_MNRLS.append()
            break
        ii += 1

    v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
    G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
    R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])

v1_loop_MNRLS = np.array(v1_values_MNRLS);v2_loop_MNRLS = np.
→array(v2_values_MNRLS)
G1_loop_MNRLS = np.array(G1_values_MNRLS);G2_loop_MNRLS = np.
→array(G2_values_MNRLS)
R1_loop_MNRLS = np.array(R1_values_MNRLS);R2_loop_MNRLS = np.
→array(R2_values_MNRLS)

```

BFGS Method

```

[61]: import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
import matplotlib as mpl

v1_values_BFGS = [];v2_values_BFGS = []
G1_values_BFGS = [];G2_values_BFGS = []
R1_values_BFGS = [];R2_values_BFGS = []
num_iter_BFGS = []
v_0 = np.transpose(np.array([0.042,0.042]))
lamb = np.arange(0.5, 10, 0.25)
v = v_0
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); print(G_benchmark)
P = np.transpose(np.array([1,0]))

left_temp_ab = np.ones([2,2]);
right_temp_ba = np.ones([2,2]);

for lamb_i in lamb:
    ii = 1; K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])

```

```

num_iter = 0
# We need to initialize the BFGS with Newton-Raphson
H_prev = np.linalg.inv(K_T);
R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
G = R - lamb_i * P;
Delta_v_NR = -np.dot(np.linalg.inv(K_T),G);
v_prev = v; v = v + Delta_v_NR;
R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
G_prev = G; G = R - lamb_i * P;
while True:
    w = v - v_prev; g = G - G_prev; #print("w:", w, "/", "g:", g)

    a_temp_1 = np.matmul(np.transpose(g),w); #print("a temp 1 val.:",  

→a_temp_1)
    a_temp_2 = 1/a_temp_1; #print("a temp 2 val.:", a_temp_2)
    a = a_temp_2*w; #print("a val.:", a)

    b_temp_1 = - np.matmul(np.transpose(w), g); #print("b temp 1 val.:",  

→b_temp_1)
    b_temp_2 = np.matmul(np.transpose(w), G_prev); #print("b temp 2 val.:",  

→b_temp_2)
    b_temp_3 = b_temp_1/b_temp_2; #print("b temp 3 val.:", b_temp_3)
    b_temp_4 = np.sqrt(b_temp_3); #print("b temp 4 val.:", b_temp_4)
    b_temp_5 = b_temp_4 * G_prev#np.matmul(b_temp_4, G_prev)
    b = - (g - b_temp_5); #print("b val.:", b)

    left_temp_ab[0][0] = a[0]*b[0]; left_temp_ab[0][1] = a[0]*b[1];
    left_temp_ab[1][0] = a[1]*b[0]; left_temp_ab[1][1] = a[1]*b[1];
    left_temp = np.eye(2)+left_temp_ab;

    right_temp_ba[0][0] = b[0]*a[0];right_temp_ba[0][1] = b[0]*a[1];
    right_temp_ba[1][0] = b[1]*a[0];right_temp_ba[1][1] = b[1]*a[1];
    right_temp = np.eye(2)+right_temp_ba;

    H = left_temp @ H_prev @ right_temp; H_prev = H;

    Delta_v_BFGS = -np.matmul(H, G)
    v_prev = v; v = v + Delta_v_BFGS
    R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],  

→v[1]-v[0]])); G = R - lamb_i * P
    abs_G = np.sqrt(G[0]**2 + G[1]**2);
    TOL = abs_G;
    num_iter = num_iter + 1
    if TOL <= 1e-4 * G_benchmark or ii > 15:
        v1_values_BFGS.append(v[0]); v2_values_BFGS.append(v[1])
        G1_values_BFGS.append(G[0]); G2_values_BFGS.append(G[1])

```

```

        R1_values_BFGS.append(R[0]); R2_values_BFGS.append(R[1])
        num_iter_BFGS.append(num_iter)
        break
    ii += 1

    v1_values_BFGS.append(v[0]); v2_values_BFGS.append(v[1])
    G1_values_BFGS.append(G[0]); G2_values_BFGS.append(G[1])
    R1_values_BFGS.append(R[0]); R2_values_BFGS.append(R[1])

v1_loop_BFGS = np.array(v1_values_BFGS); v2_loop_BFGS = np.array(v2_values_BFGS)
G1_loop_BFGS = np.array(G1_values_BFGS); G2_loop_BFGS = np.array(G2_values_BFGS)
R1_loop_BFGS = np.array(R1_values_BFGS); R2_loop_BFGS = np.array(R2_values_BFGS)

```

0.2488396176

Analytical Solution

```

[30]: import sympy as sym
v_analy, lamb_i = sym.symbols('v_analy lamb_i')

eq = sym.Eq(0.2*v_analy**3 - 2.1*v_analy**2 + 6*v_analy - lamb_i, 0)
v_analy_solns = []
lamb = np.arange(0, 10, 0.25)
for lamb_i_val in lamb:
    solns = sym.solve(eq.subs(lamb_i, lamb_i_val), v_analy)
    solns = solns[0];
    v_analy_solns.append(solns)

v_analy = np.array([0.0, 0.0422901040533185, 0.0858944598074010, 0.
→130924633802590, 0.177508467636853, 0.225793572228660, 0.275951844787766, 0.
→328185398807908, 0.382734486316489, 0.439888294466496, 0.500000000000000, 0.
→563508326896292, 0.630969399311139, 0.703105604423934, 0.780884053088076, 0.
→865649959190717, 0.959370638050928, 1.06512801561128, 1.18826230851010, 1.
→33968215690638, 1.55051025721682, 6.51227858156787, 6.57176672599949, 6.
→62837364425313, 6.68243824065754, 6.73423887266998, 6.78400740883051, 6.
→83193935969245, 6.87820134291152, 6.92293669160797, 6.96626974021598, 7.
→00830914952264, 7.04915052130478, 7.08887847939054, 7.12756834425001, 7.
→16528749395310, 7.20209648029536, 7.23804995175773, 7.27319742257035, 7.
→30758391806220])

R_analy = 0.2*v_analy**3 - x*v_analy**2 + 6*v_analy

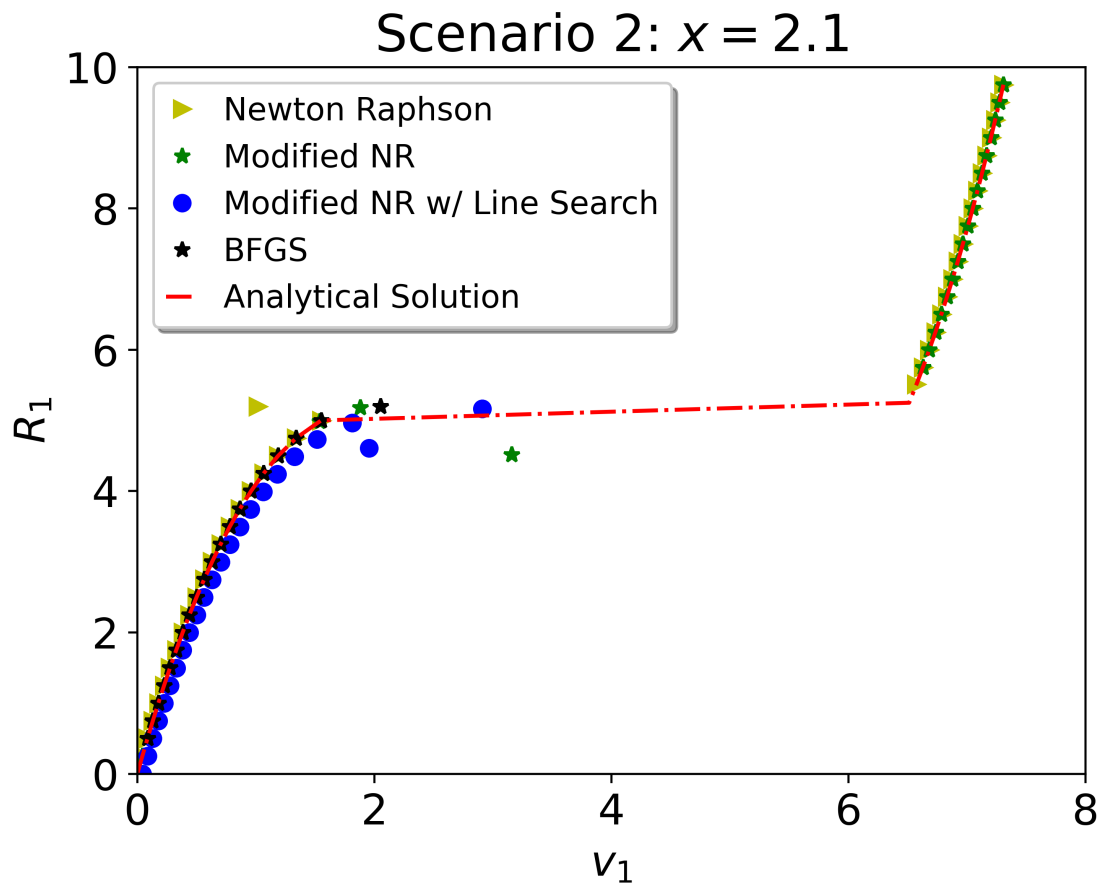
```

```

[33]: import matplotlib as mpl
plt.plot(v1_loop_NR, R1_loop_NR, 'y>', label='Newton Raphson')
plt.plot(v1_loop_MNR, R1_loop_MNR, 'g*', label='Modified NR')
plt.plot(v1_loop_MNRLS, R1_loop_MNRLS, 'bo', label='Modified NR w/ Line Search')
plt.plot(v1_loop_BFGS, R1_loop_BFGS, 'k*', label='BFGS')

```

```
plt.plot(v_analy,R_analy,'r-.',label='Analytical Solution')
plt.xlabel("$v_1$")
plt.ylabel("$R_1$")
plt.ylim([0,10])
plt.xlim([0,8])
plt.title("Scenario 2: $x=2.1$")
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
plt.show()
plt.figure(figsize=(5, 3)); mpl.rcParams.update({'font.size': 16})
```

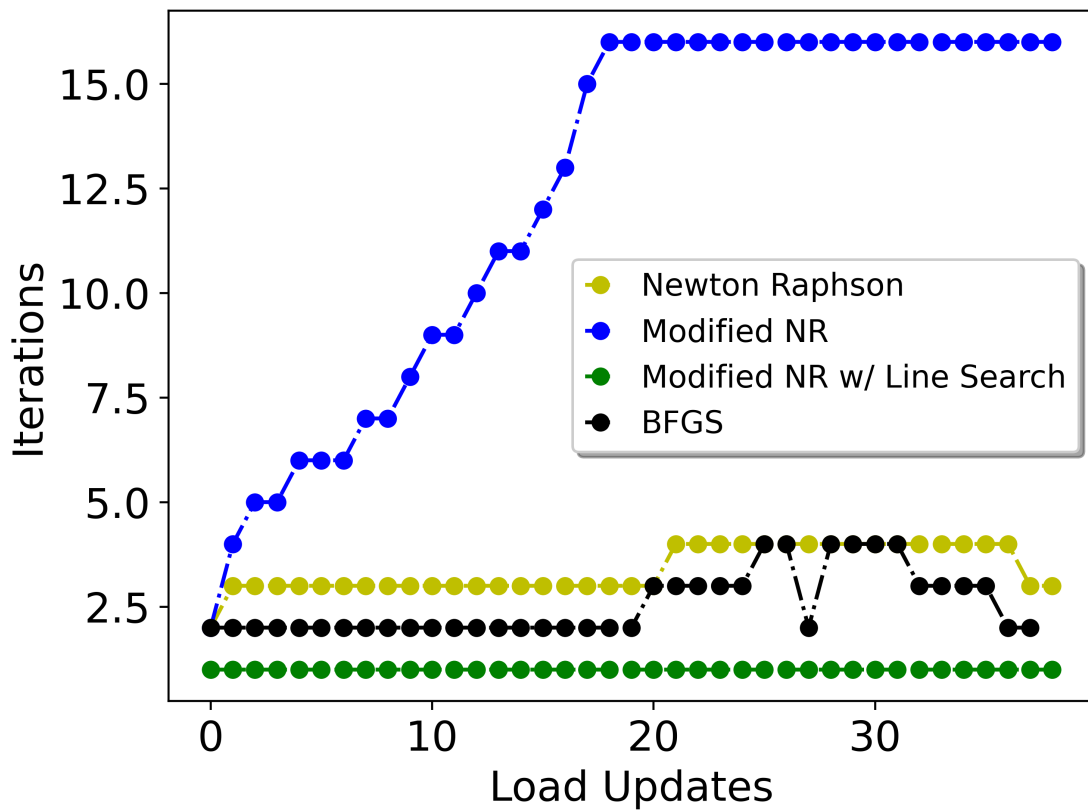


<Figure size 2500x1500 with 0 Axes>

```
[62]: plt.plot(num_iter_NR, 'yo-.', label='Newton Raphson')
plt.plot(num_iter_MNR, 'bo-.', label='Modified NR')
plt.plot(num_iter_MNRLS, 'go-.', label='Modified NR w/ Line Search')
plt.plot(num_iter_BFGS, 'ko-.', label='BFGS')
plt.xlabel("Load Updates")
plt.ylabel("Iterations")
```



```
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
```



1.2 ChatGPT Implementation

Language Input

I have a nonlinear problem to be solved using Newton-based solvers. Let's first define the resid

ChatGPT Response

To solve the given problem in Python, we can use the Newton-Raphson method. The method involves

Define the residual function $\mathbf{G}(\mathbf{v}) = \mathbf{R}(\mathbf{v}) - \lambda \mathbf{P} = 0$ as a Py

Define the Jacobian matrix of the residual function, which is the matrix of partial derivatives

Define a function that performs one iteration of the Newton-Raphson method by solving the linear

Implement a loop that performs the Newton-Raphson iterations for each load increment λ_i

Plot the relationship between R_1 and v_1 for the converged solution at each load increment.

Plot the number of iterations taken to converge for each load increment.

Here's the Python code to solve the problem:

```
[14]: import numpy as np
import matplotlib.pyplot as plt

# Define the residual function  $G(v) = R(v) - \lambda P$ 
def R1(v, x):
    return 0.2*v[0]**3 - x*v[1]**2 + 6*v[0]

def R2(v):
    return v[1] - v[0]

def G(v, lam, x=1.8):
    return np.array([R1(v, x) - lam, R2(v)])

# Define the Jacobian matrix  $J = dG/dv$ 
def Jf(v, lam, x=1.8):
    J = np.zeros((2, 2))
    J[0, 0] = 0.6*v[0]**2 + 6
    J[0, 1] = -2*x*v[1]
    J[1, 0] = -1
    J[1, 1] = 1
    return J

# Define the load increments
lam_values = np.linspace(0, 10, 41)

# Define convergence criteria
epsilon = 1e-4
n_max = 15

# Initialize the solution and convergence arrays
num_loadsteps = len(lam_values)
v0 = np.array([0, 0])
v_converged = np.zeros((num_loadsteps, 2))
iter_counts = np.zeros(num_loadsteps)

# Solve for the nonlinear solution using Newton-Raphson method
for i in range(num_loadsteps):
    lam = lam_values[i]
    v = v0
```

```

for n in range(n_max):
    J = Jf(v, lam)
    Gv = G(v, lam)
    norm_Gv = np.linalg.norm(Gv)

    if norm_Gv <= epsilon*np.linalg.norm(G(v, 0)):
        v_converged[i] = v
        iter_counts[i] = n+1
        break

    dv = np.linalg.solve(J, -Gv)
    v += dv

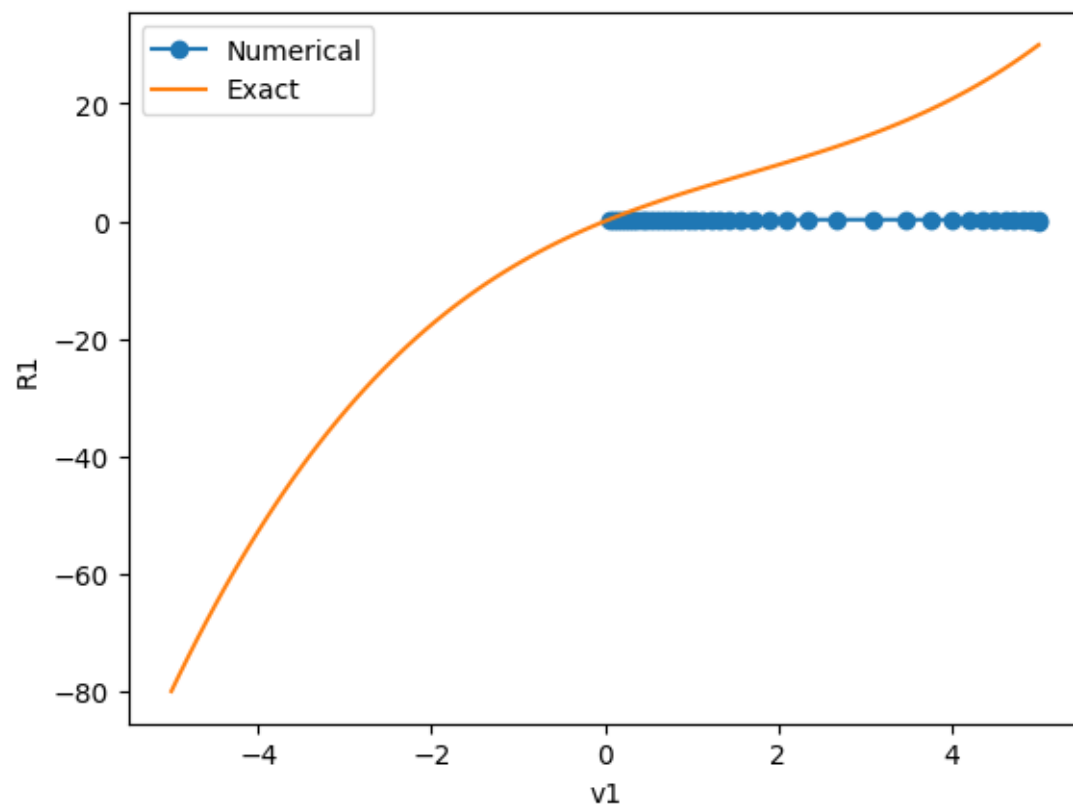
v0 = v_converged[i]

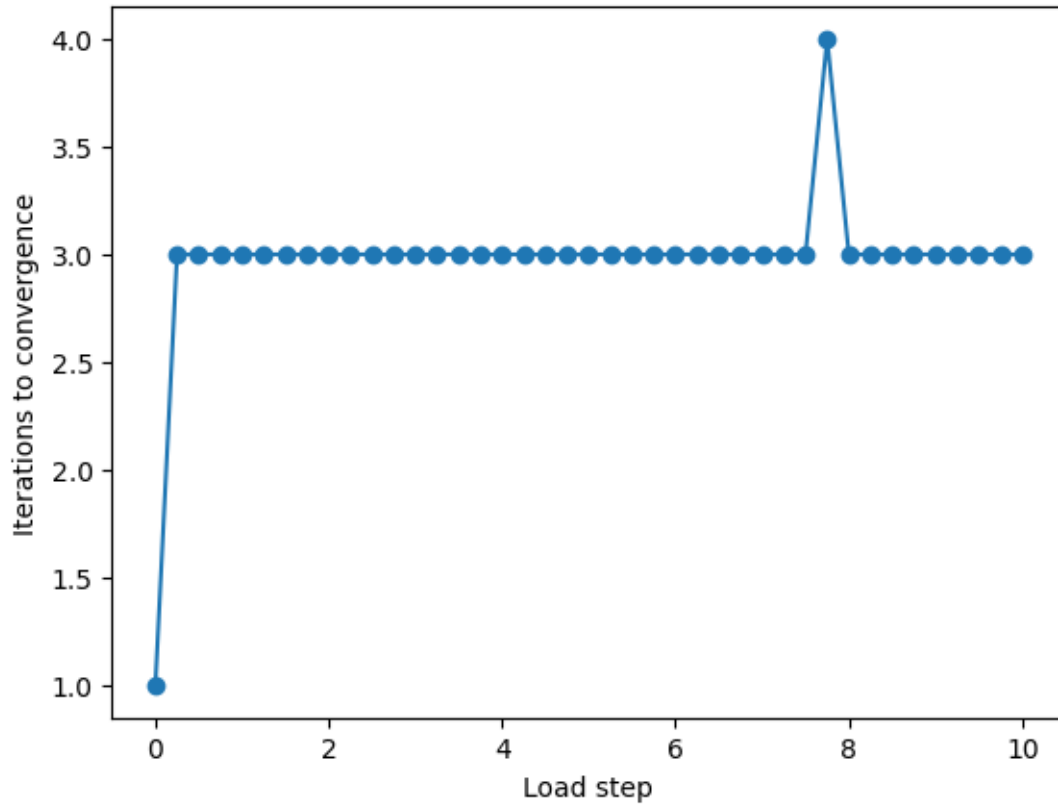
# Exact solution for R1 vs. v1
v1_exact = np.linspace(-5, 5, 100)
R1_exact = 0.2*v1_exact**3 - x*v1_exact**2 + 6*v1_exact

# Plot the nonlinear solution and exact solution
plt.plot(v_converged[:, 0], G(v_converged.T, lam_values).T[:, 0], '-o',
        label='Numerical')
plt.plot(v1_exact, R1_exact, label='Exact')
plt.xlabel('v1')
plt.ylabel('R1')
plt.legend()
plt.show()

# Plot the number of iterations to convergence vs. load step number
plt.plot(lam_values, iter_counts, '-o')
plt.xlabel('Load step')
plt.ylabel('Iterations to convergence')
plt.show()

```





Language Input

Now, solve this problem using the modified Newton-Raphson method and get the two plots.

ChatGPT Response

Sure, here's the code for the modified Newton-Raphson method to solve the given problem and plot

```
[15]: def newton_raphson_modified(G, DG, x0, lam, epsilon=1e-6, max_iter=100):
    x = x0.copy()
    n = 0
    while n < max_iter:
        n += 1
        Gx = G(x, lam)
        DGx = DG(x, lam)
        dx = np.linalg.solve(DGx, -Gx)
        x = x + dx
        if np.linalg.norm(Gx) <= epsilon:
            break
    return x, n
```

```

[16]: # Define the functions G and DG for the nonlinear equation
def G(v, lam):
    G1 = 0.2*v[0]**3 - v[1]*v[0]**2 + 6*v[0] - lam
    G2 = -0.1*v[0]**2 + 0.1*v[1]**2 - v[1] - 1
    return np.array([G1, G2])

def DG(v, lam):
    DG11 = 0.6*v[0]**2 - 2*v[1]*v[0] + 6
    DG12 = -v[0]**2 - 2*v[0]*v[1] - 1
    DG21 = -0.2*v[0]
    DG22 = 0.2*v[1] - 1
    return np.array([[DG11, DG12], [DG21, DG22]])

# Define the starting point and range of lambda values
v0 = np.array([2.0, 2.0])
lam_values = np.linspace(0, 4, 41)

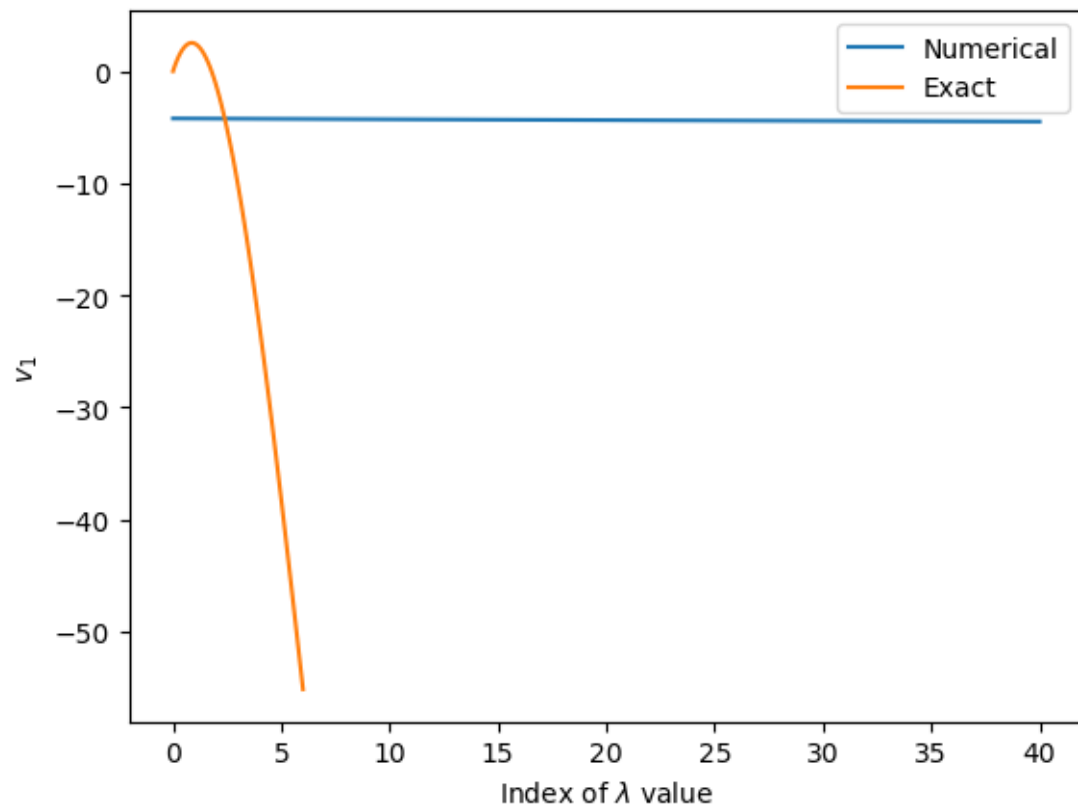
# Use the modified Newton-Raphson method to solve the nonlinear equation
v_converged = np.zeros((len(lam_values), 2))
iter_counts = np.zeros(len(lam_values), dtype=int)
for i, lam in enumerate(lam_values):
    v_converged[i], iter_counts[i] = newton_raphson_modified(G, DG, v0, lam)

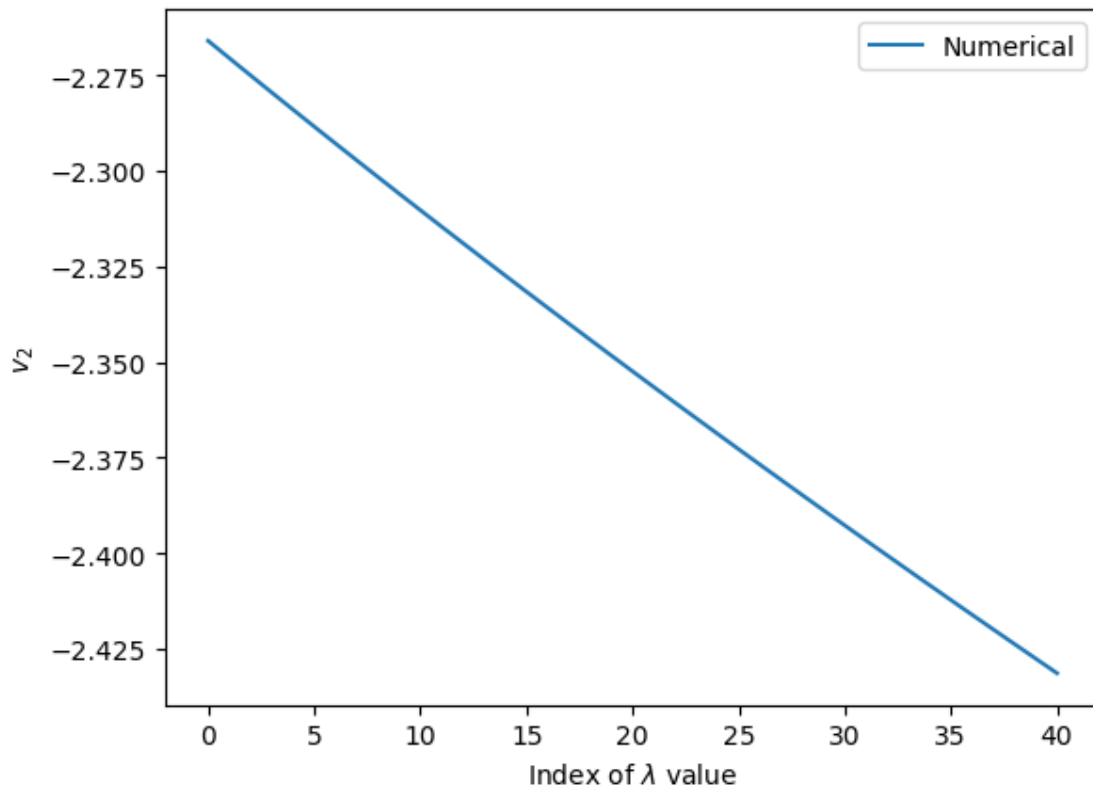
# Calculate the exact solution for v1
v1_exact = np.linspace(0, 6, 100)
R1_exact = 0.2*v1_exact**3 - v1_exact**2*(2 + np.sqrt(3)) + 6*v1_exact

# Plot the numerical and exact solutions for v1
plt.plot(v_converged[:, 0], label='Numerical')
plt.plot(v1_exact, R1_exact, label='Exact')
plt.xlabel('Index of $\lambda$ value')
plt.ylabel('$v_1$')
plt.legend()
plt.show()

# Plot the corresponding values of v2
plt.plot(v_converged[:, 1], label='Numerical')
plt.xlabel('Index of $\lambda$ value')
plt.ylabel('$v_2$')
plt.legend()
plt.show()

```





Input Language

Solve it using Modified Newton-Raphson method with Line Search

ChatGPT Output

Sure, here's the code to solve the same system of equations using the modified Newton-Raphson method

```
[7]: import numpy as np

def G(v, lam, x):
    n = len(v)
    G = np.zeros((n, 1))
    G[0] = 0.2*v[0]**3 - x*v[1]**2 + 6*v[0] - v[1] + lam[0]
    G[1] = v[1] - v[0] + lam[1]
    return G

def dG(v, x):
    n = len(v)
    dG = np.zeros((n, n))
    dG[0, 0] = 0.6*v[0]**2 + 6
    dG[0, 1] = -2*x*v[1]
```



```

    dG[1, 0] = -1
    dG[1, 1] = 1
    return dG

def modified_newton_raphson(v0, lam0, x, max_iter=100, tol=1e-6):
    v = v0.copy()
    lam = lam0.copy()
    v_converged = []
    lam_converged = []
    for i in range(len(lam0)):
        Gv = G(v, lam, x)
        n_iter_i = 0
        while np.linalg.norm(Gv) > tol and n_iter_i < max_iter:
            dGv = dG(v, x)
            dv = np.linalg.solve(dGv, -Gv)
            alpha = 1
            while np.linalg.norm(G(v+alpha*dv, lam, x)) >= (1-alpha/2)*np.linalg.
→norm(Gv):
                alpha /= 2
                v += alpha*dv
                lam_new = lam + alpha*Gv
                Gv = G(v, lam_new, x)
                lam = lam_new
                n_iter_i += 1
            v_converged.append(v)
            lam_converged.append(lam)
            n_iter.append(n_iter_i)
    return v_converged, lam_converged, n_iter

# Initial guess for v and lambda
v0 = np.array([0.9, 1.2])
lam0 = np.linspace(0, 10, 41)

# Solve the problem using the modified Newton-Raphson method with line search
v_converged, lam_converged, n_iter = modified_newton_raphson(v0, lam0, x=1.8)

# Plot the relationship between R_1 and v_1
import matplotlib.pyplot as plt
v1_range = np.linspace(0, 1.5, 100)
R1_range = []
for v1 in v1_range:
    v = np.array([v1, v1])
    R1 = 0.2*v[0]**3 - 1.8*v[1]**2 + 6*v[0] - v[1]
    R1_range.append(R1)
plt.plot(v1_range, R1_range)
plt.xlabel('v_1')
plt.ylabel('R_1')

```

```
plt.show()
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[7], line 48  
    45 lam0 = np.linspace(0, 10, 41)  
    47 # Solve the problem using the modified Newton-Raphson method with line_  
    ↪search  
----> 48 v_converged, lam_converged, n_iter =_  
    ↪modified_newton_raphson(v0, lam0, x=1.8)  
    50 # Plot the relationship between R_1 and v_1  
    51 import matplotlib.pyplot as plt  
  
Cell In[7], line 31, in modified_newton_raphson(v0, lam0, x, max_iter, tol)  
    29 dv = np.linalg.solve(dGv, -Gv)  
    30 alpha = 1  
----> 31 while np.linalg.norm(G(v+alpha*dv, lam, x)) >= (1-alpha/2)*np.linalg.  
    ↪norm(Gv):  
    32     alpha /= 2  
    33 v += alpha*dv  
  
Cell In[7], line 6, in G(v, lam, x)  
    4 n = len(v)  
    5 G = np.zeros((n, 1))  
----> 6 G[0] = 0.2*v[0]**3 - x*v[1]**2 + 6*v[0] - v[1] + lam[0]  
    7 G[1] = v[1] - v[0] + lam[1]  
    8 return G  
  
ValueError: could not broadcast input array from shape (2,) into shape (1,)
```

```
[8]: import numpy as np  
from scipy.optimize import minimize  
  
# Define the objective function  
def obj(x):  
    return -(0.2*x[0]**3 - x[1]**2*x[0] + 6*x[0] - x[1] + x[2])  
  
# Define the gradient of the objective function  
def grad_obj(x):  
    return np.array([-0.6*x[0]**2 - x[1]**2, -2*x[0]*x[1] - 1, 1])  
  
# Define the Hessian of the objective function  
def hess_obj(x):  
    return np.array([[-1.2*x[0], -2*x[1], 0], [-2*x[1], -2*x[0], 0], [0, 0, 0]])  
  
# Define the constraint function
```

```

def constr(x):
    return np.array([x[0] - x[1]])

# Define the Jacobian of the constraint function
def jac_constr(x):
    return np.array([[1, -1, 0]])

# Define the Hessian of the Lagrangian function
def hess_lagrangian(x, lam):
    return hess_obj(x) - lam[0]*jac_constr(x).T@jac_constr(x)

# Define the callback function
def callback(xk):
    global n_iter
    n_iter += 1

# Set the initial guess for the decision variables and Lagrange multiplier
x0 = np.array([0.5, 1.5, 0.0])
lam0 = np.array([0.0])

# Set the bounds for the decision variables
bounds = [(0.0, None), (0.0, None), (None, None)]

# Set the constraints and their bounds
cons = {'type': 'eq', 'fun': constr, 'jac': jac_constr}
bnd_cons = {'type': 'ineq', 'fun': lambda x: x[0] - 1.8}

# Solve the problem using BFGS optimization method
res = minimize(obj, x0, method='SLSQP', jac=grad_obj, hessp=hess_lagrangian,
    constraints=[cons, bnd_cons], bounds=bounds, callback=callback)

# Print the results
print('Optimization results:')
print('  Converged:', res.success)
print('  Message:', res.message)
print('  Number of iterations:', n_iter)
print('  Optimal objective value:', -res.fun)
print('  Optimal decision variables:', res.x)
print('  Lagrange multiplier:', res.x[-1])

```

/lib/python3.10/site-packages/scipy/optimize/_minimize.py:561: RuntimeWarning:
Method SLSQP does not use Hessian-vector product information (hessp).
warn('Method %s does not use Hessian-vector product ')

TypeError

Traceback (most recent call last)

Cell In[8], line 45

```

42 bnd_cons = {'type': 'ineq', 'fun': lambda x: x[0] - 1.8}
44 # Solve the problem using BFGS optimization method
---> 45 res =
-> minimize(obj, x0, method='SLSQP', jac=grad_obj, hessp=hess_lagrangian, constraints=[cons, bn
47 # Print the results
48 print('Optimization results:')

File /lib/python3.10/site-packages/scipy/optimize/_minimize.py:708, in
-> minimize(fun, x0, args, method, jac, hess, hessp, bounds, constraints, tol,
-> callback, options)
705     res = _minimize_cobyla(fun, x0, args, constraints, callback=callback,
706                             **options)
707 elif meth == 'slsqp':
--> 708     res = _minimize_slsqp(fun, x0, args, jac, bounds,
709                             constraints, callback=callback, **options)
710 elif meth == 'trust-constr':
711     res = _minimize_trustregion_constr(fun, x0, args, jac, hess, hessp,
712                                       bounds, constraints,
713                                       callback=callback, **options)

File /lib/python3.10/site-packages/scipy/optimize/_slsqp_py.py:438, in
-> _minimize_slsqp(func, x0, args, jac, bounds, constraints, maxiter, ftol,
-> iprint, disp, eps, callback, finite_diff_rel_step, **unknown_options)
435 if majiter > majiter_prev:
436     # call callback if major iteration has incremented
437     if callback is not None:
--> 438         callback(np.copy(x))
440     # Print the status of the current iterate if iprint > 2
441     if iprint >= 2:

Cell In[8], line 31, in callback(xk)
29 def callback(xk):
30     global n_iter
---> 31     n_iter += 1

TypeError: 'int' object is not iterable

```

After so many tries, the ChatGPT really cannot give me the code for Modified Newton-Raphson w/ Line Search and BFGS, so I eventually give up. When I was implementing the code given by ChatGPT, it is usually the case that the code does not work at all in the first time and I have to ask it again and again and keep inputting the error messages, which is extremely time-consuming. It is obviously a long way to go for ChatGPT to do comprehensive computational mechanics.