In [ ]:
```
import newton_and_quasinewton_script as newt
import numpy as np
import matplotlib.pyplot as plt
import math
from numpy.linalg import inv
from numpy.linalg import norm
```

Homework 6

Problem 1)

(i)

In [2]:
```
def F1(x):
    return np.array([x[0]**2+x[1]**2-4,np.exp(x[0])+x[1]-1])
def J1(x):
    return np.array([[2*x[0],2*x[1]],[np.exp(x[0]),1]])
x01=np.array([1.0,1.0])
```

In [3]:
```
r1,rn1,nf1,nj1 = newt.lazy_newton_method_nd(F1,J1,x01,1e-10,100)
```

```
C:\Users\xman7\AppData\Local\Temp\ipykernel_22968\1970405929.py:2: RuntimeWarning: o
verflow encountered in exp
  return np.array([x[0]**2+x[1]**2-4,np.exp(x[0])+x[1]-1])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[3], line 1
----> 1 r1,rn1,nf1,nj1 = newt.lazy_newton_method_nd(F1,J1,x01,1e-10,100)

File c:\Users\xman7\OneDrive - UCB-O365\Documents\APPM4600\Homework\Homework6\newton
_and_quasinewton_script.py:90, in lazy_newton_method_nd(f, Jf, x0, tol, nmax, verb)
     88 if verb:
     89     print("Fn = ",Fn)
---> 90 pn = -lu_solve((lu, piv), Fn); #We use lu solve instead of pn = -np.linalg.s
olve(Jn,Fn);
     91 xn = xn + pn;
     92 npn = np.linalg.norm(pn); #size of Newton step

File c:\Users\xman7\AppData\Local\Programs\Python\Python311\Lib\site-packages\scipy
\linalg\_decomp_lu.py:171, in lu_solve(lu_and_piv, b, trans, overwrite_b, check_fini
te)
    169 (lu, piv) = lu_and_piv
    170 if check_finite:
--> 171     b1 = asarray_chkfinite(b)
    172 else:
    173     b1 = asarray(b)

File c:\Users\xman7\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy
\lib\function_base.py:630, in asarray_chkfinite(a, dtype, order)
    628 a = asarray(a, dtype=dtype, order=order)
    629 if a.dtype.char in typecodes['AllFloat'] and not np.isfinite(a).all():
--> 630     raise ValueError(
    631         "array must not contain infs or NaNs")
    632 return a

ValueError: array must not contain infs or NaNs
```

This process diverges, and therefore has terrible performance.

In [ ]:
```
r2,rn2,nf2= newt.broyden_method_nd(F1,J1,x01,1e-10,100)
```

In [ ]:
```
print(r2)
print(len(rn2))
```

```
[ 1.00416874 -1.72963729]
22
```

(ii)

In [ ]:
```
x02=np.array([1,-1])
r12,rn12,nf12,nj12 = newt.lazy_newton_method_nd(F1,J1,x02,1e-10,100)
```

In [ ]:
```
print(r12)
print(len(rn12))
```

```
[ 1.00416874 -1.72963729]
38
```

In [ ]:
```
r22,rn22,nf22= newt.broyden_method_nd(F1,J1,x02,1e-10,100)
```

In [ ]:
```
print(r22)
print(len(rn22))
```

```
[ 1.00416874 -1.72963729]
15
```

(iii)

In [ ]:
```
x03=np.array([0,0])
```

In [ ]:
```
r13,rn13,nf13,nj13 = newt.lazy_newton_method_nd(F1,J1,x03,1e-10,100, verb = True)
```

```
lu =  [[1. 1.]
 [0. 0.]]
piv =  [1 1]
|--n--|----xn----|---|f(xn)|---|
|--0--|0.0000000|4.000000000000|
Fn =  [-4.  0.]
|--1--|inf|inf|
Fn =  [inf inf]
```

```
c:\Users\xman7\OneDrive - UCB-O365\Documents\APPM4600\Homework\Homework6\newton_and_quasinewton_script.py:70: LinAlgWarning: Diagonal number 2 is exactly zero. Singular matrix.
  lu, piv = lu_factor(Jn);
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 r13,rn13,nf13,nj13 = newt.lazy_newton_method_nd(F1,J1,x03,1e-10,100, verb = 
True)

File c:\Users\xman7\OneDrive - UCB-O365\Documents\APPM4600\Homework\Homework6\newton
_and_quasinewton_script.py:90, in lazy_newton_method_nd(f, Jf, x0, tol, nmax, verb)
     88 if verb:
     89     print("Fn = ",Fn)
---> 90 pn = -lu_solve((lu, piv), Fn); #We use lu solve instead of pn = -np.linalg.s
olve(Jn,Fn);
     91 xn = xn + pn;
     92 npn = np.linalg.norm(pn); #size of Newton step

File c:\Users\xman7\AppData\Local\Programs\Python\Python311\Lib\site-packages\scipy
\linalg\_decomp_lu.py:171, in lu_solve(lu_and_piv, b, trans, overwrite_b, check_fini
te)
    169 (lu, piv) = lu_and_piv
    170 if check_finite:
--> 171     b1 = asarray_chkfinite(b)
    172 else:
    173     b1 = asarray(b)

File c:\Users\xman7\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy
\lib\function_base.py:630, in asarray_chkfinite(a, dtype, order)
    628 a = asarray(a, dtype=dtype, order=order)
    629 if a.dtype.char in typecodes['AllFloat'] and not np.isfinite(a).all():
--> 630     raise ValueError(
    631         "array must not contain infs or NaNs")
    632 return a

ValueError: array must not contain infs or NaNs
```

The lazy newton method diverges again.

In [ ]:
```
r23,rn23,nf23= newt.broyden_method_nd(F1,J1,x03,1e-10,100)
```

In [ ]:
```
print(r23)
print(len(rn23))
```

```
[-1.81626407  0.8373678 ]
22
```

This X_0 gives a new root for the system using broyden compared to the other two initial conditions.

The Broyden method is much better in this case than the lazy newton method, but neither is better than newtons original in terms of number of iterations.

Problem 2

In [19]:
```
def F2(x):

    F = np.zeros(3)
    F[0] = x[0] +math.cos(x[0]*x[1]*x[2])-1.
    F[1] = (1.-x[0])**(0.25) + x[1] +0.05*x[2]**2 -0.15*x[2]-1
    F[2] = -x[0]**2-0.1*x[1]**2 +0.01*x[1]+x[2] -1
    return F

def J2(x):

    J =np.array([[1.+x[1]*x[2]*math.sin(x[0]*x[1]*x[2]),x[0]*x[2]*math.sin(x[0]*x[1
          [-0.25*(1-x[0])**(-0.75),1,0.1*x[2]-0.15],
          [-2*x[0],-0.2*x[1]+0.01,1]])
    return J

def G2(x):

    F = F2(x)
    G = F[0]**2 + F[1]**2 + F[2]**2
    return G

def gradG2(x):


    gradG = np.transpose(J2(x)).dot(F2(x))
    return gradG

x21 = np.array([0,0,0])
```

In [27]:
```
def newton_method_nd(f,Jf,x0,tol,nmax):


    xn = x0
    rn = x0
    Fn = f(xn)
    n=0
    nf=1; nJ=0
    normPn=1

    while normPn>tol and n<=nmax:

        Jn = Jf(xn)
        nJ+=1


        pn = -np.linalg.solve(Jn,Fn)
        xn = xn + pn
        normPn = np.linalg.norm(pn)

        n+=1
        rn = np.vstack((rn,xn))
        Fn = f(xn)
        nf+=1

    r=xn
```

```
    return (r,rn,nf,nJ);

def SteepestDescent(x,tol,Nmax):
    i=0
    while  i < Nmax+1:
        g1 = G2(x)
        z = gradG2(x)
        z0 = norm(z)


        z = z/z0

        alpha3 = 1
        dif_vec = x - alpha3*z
        g3 = G2(dif_vec)

        while g3>=g1:
            alpha3 = alpha3/2
            dif_vec = x - alpha3*z
            g3 = G2(dif_vec)

        if alpha3<tol:


            return [x,g1,i]

        alpha2 = alpha3/2
        dif_vec = x - alpha2*z
        g2 = G2(dif_vec)

        h1 = (g2 - g1)/alpha2
        h2 = (g3-g2)/(alpha3-alpha2)
        h3 = (h2-h1)/alpha3

        alpha0 = 0.5*(alpha2 - h1/h3)
        dif_vec = x - alpha0*z
        g0 = G2(dif_vec)

        if g0<=g3:
            alpha = alpha0
            gval = g0

        else:
            alpha = alpha3
            gval =g3

        x = x - alpha*z

        if abs(gval - g1)<tol:

            return [x,gval,i]

        i+=1


    return [x,g1,i]
```

In [ ]:
```
r3,rn3,nf3,nj3 = newton_method_nd(F2,J2,x21,1e-6,100)
```

In [9]:
```
print(r3)
print(len(rn3))
```

```
[0.  0.1 1. ]
5
```

In [28]:
```
r31, g, i= SteepestDescent(x21,1e-6,100)
```

In [29]:
```
print(r31)
print(i)
```

```
[-6.27724957e-05  9.99685854e-02  9.99984493e-01]
5
```

In [30]:
```
r32,g2,i2 = SteepestDescent(x21,5e-2,100)
r33,rn31,nf31,nj31 = newton_method_nd(F2,J2,r31, 1e-6,100)
```

In [33]:
```
print("Steepest Descent output:",r32)
print("Newton output:",r33)
print("total number of steps:",i2+len(rn31))
```

```
Steepest Descent output: [-0.02001828  0.09005646  0.99526475]
Newton output: [-3.57108742e-17  1.00000000e-01  1.00000000e+00]
total number of steps: 4
```

This result is technically better than the steepest descent or newton methods, although not by much in this example. This is better because steepest descent works quickly to bring the guess into the basin of convergence of newtons method, then newtons method works quickly to converge, converging quadratically.

In [ ]: