# Problem 1

Minimize $f(x_1, x_2) = x_1^2 + x_2^2 + x_1x_2 - 3x_1$ using conjugate gradient algorithm.
$x^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

(i) We known that $f(x) = \frac{1}{2}X^TQX - X^Tb$ where $Q = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ and b = $\begin{bmatrix} 3 \\ 0 \end{bmatrix}$

$f(x) = \frac{1}{2}X^T \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} X - X^T \begin{bmatrix} 3 \\ 0 \end{bmatrix}$

$\therefore$g(x$^{(k)}$) $= \nabla f(x^{(k)} = Qx^{(k)} - b = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} X - \begin{bmatrix} 3 \\ 0 \end{bmatrix}$

Set $d^{(0)} = -g^{(0)}$

$g^{(0)} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} -3 \\ 0 \end{bmatrix}$

$$\alpha^{(0)} = -\frac{g^{(0)}d^{(0)}}{d^{(0)}Qd^{(0)}} = -\frac{\begin{bmatrix} -3 & 0 \end{bmatrix} \begin{bmatrix} -3 \\ 0 \end{bmatrix}}{\begin{bmatrix} 3 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \end{bmatrix}} = 0.5$$

$x^{(1)} = x^{(0)} + \alpha^{(0)}d^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0.5 \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 0 \end{bmatrix}$

$g^{(1)} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} - \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1.5 \end{bmatrix}$

$$\beta^{(0)} = \frac{g^{(1)}Qd^{(0)}}{d^{(0)}Qd^{(0)}} = -\frac{\begin{bmatrix} 0 & 1.5 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} -3 \\ 0 \end{bmatrix}}{\begin{bmatrix} 3 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \end{bmatrix}} = 0.25$$

$d^{(1)} = -g^{(1)} + \beta^{(0)}d^{(0)} = -\begin{bmatrix} 0 \\ 1.5 \end{bmatrix} + 0.25 \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.75 \\ -1.5 \end{bmatrix}$

$$\alpha^{(1)} = -\frac{g^{(1)}d^{(1)}}{d^{(1)}Qd^{(1)}} = -\frac{\begin{bmatrix} 0 & 1.5 \end{bmatrix} \begin{bmatrix} 0.75 \\ -1.5 \end{bmatrix}}{\begin{bmatrix} 0.75 & -1.5 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0.75 \\ -1.5 \end{bmatrix}} = 0.666$$

$x^{(2)} = x^{(1)} + \alpha^{(1)}d^{(1)} = \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} + 0.666 \begin{bmatrix} 0.75 \\ -1.5 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix} = x^*$

Since Conjugate gradient finds minimizer for quadratic in 2 steps

FONC - $\nabla f(x^{(2)}) = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \end{bmatrix} - \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ - Satisfied

SOSC $F(x^{(2)}) = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$

All leading principal minors are greater than zero

$\Delta_1 = 2 > 0$

$\Delta_2 = det(Q) = 4 - 1 = 3 > 0$ - Satisfied

Thus, $x^*$ is a strictly local minimizer

# Problem 2,3,4

Use your line search algorithm that you developed in hw02 to minimize
$f(x_1, x_2) = (x_2 - x_1)^4 + 12x_1x_2 - x_1 + x_2 - 3$
using
2. Rank One Algorithm
3. The DFP Algorithm
4. The BFGS Algorithm
Here is a snippet of the code for working out the minimization using all the algorithms listed above.

```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov  2 10:59:51 2019

@author: sarde
"""
#Problem 1
import numpy as np
import matplotlib.pyplot as plt
#from mpl_toolkits import mplot3d

def Bracketing(f, alphInit, epsi):
    """
    Does bracketing to set up initial interval for line search
    inputs - function, initial parameter, epsilon
    outputs - initial interval
    """
    x0 = alphInit
    x1 = x0+epsi
    x2 = x0+ 2*epsi
    n = r = 1
    if(f(x0)>f(x1) and f(x1)<f(x2)):
        a0 = x0
        b0 = x2
        return a0,b0
    else:
        while not (f(x0)>f(x1) and f(x1)<f(x2)):
            if((f(x0)>f(x1) and f(x1)>f(x2))):
                x0 = x1
                x1 = x2
                x2 = x2 +(2**(n+1)) * epsi
                n+=1
            elif((f(x0)<f(x1) and f(x1)<f(x2))):
                x2 = x1
                x1 = x0
                x0 = x0 - (2**r) * epsi
                r+=1
        a0 = x0
        b0 = x2
        return a0,b0

def goldSec(a0,b0,fcn):
    """
    Reduces uncertainty range of the interval found out by bracketing
    alpha can be picked from the reduced uncertainty interval
    inputs - bracketed interval, function
    outputs - reduced uncertainty interval
    """
    N = np.ceil(np.log(0.001/np.linalg.norm(b0-a0))/np.log(.6180))
    rho = .382
    a = a0
    b = b0
    s = a+rho*(b-a)
```

```
    t = a + (1-rho) * (b-a)
    f1 = fcn(s)
    f2 = fcn(t)
    for i in range(1,int(N+1)):
        if(fcn(s)<fcn(t)):
            b = t
            t = s
            s = a+rho*(b-a)
            f2 = f1
            f1 = fcn(s)
        elif(fcn(s)>fcn(t)):
            a = s
            s = t
            t = a+(1-rho)*(b-a)
            f1 = f2
            f2 = fcn(t)
        else:
            break
    return a,b

def RankOne(X_init, f):
    """
    Basic rank one update for estimating the Inverse hessian
    Hk
    Initial Hk chosen to be 10,2; 2, 10 - pd and symmetric
    inputs - Initial array of X, function f
    outputs - updated array of X
    """
    normx = 1
    H = []
    H.append(np.array([[1,0],[0,1]]))
    #if gradient of initial point is zero STOP
    if grad(X_init[0][0], X_init[0][1]) == (0,0):
        return X_init
    else:
        Hk = H[-1]  #set Hk as last element in H[]
        gradk = grad(X_init[0][0],X_init[0][1]) #find kth gradient
        d = -Hk.dot(gradk)  #set direction according to kth gradient and Hk
        fAlph = lambda alpha: (f(X_init[-1][0] + alpha*d[0], X_init[-1][0] \
                                 + alpha*d[1])) #find a fAlph
        al1, al2 = Bracketing(fAlph, 0, 1) #find optimum alpha interval
        alpa, alpb = goldSec(al1, al2, fAlph)
        alphak = (alpa+alpb)/2  #find optimum alpha
        X_init.append(X_init[-1]+np.multiply(alphak,d)) #find second point
        iters = 1
        #iterate till consecutive points are within a small value epsilon
        while normx > epsilon and grad(X_init[-1][0],X_init[-1][1]) and iters < max_iterations:
            #update Hk as the last element in the H estimates list
            Hk = H[-1]
            gradk = grad(X_init[-1][0],X_init[-1][1]) #find gradient of latest pt
            dk = -Hk.dot(gradk) #find the direction from Hk and gk
            #line search over x_alpha*d
            fAlph = lambda alphak: (f(X_init[-1][0] + alphak*dk[0], \
                                      X_init[-1][1] + alphak*dk[1]))
            #bracketing and golden section to find optimum alpha
            al1, al2 = Bracketing(fAlph, 0, 1)
            alpa, alpb = goldSec(al1, al2, fAlph)
            alphak = (alpa+alpb)/2
            #append updated point to the list of points
            X_init.append(X_init[iters]+np.multiply(alphak,dk))
            deltaXk = alphak*dk
            deltaGk = np.subtract(grad(X_init[-1][0],X_init[-1][1]),gradk)
            #Rank one update of Hk
            H.append(np.array(Hk+(((deltaXk-Hk.dot(deltaGk)).dot\
                                       (np.transpose((deltaXk-Hk.dot(deltaGk)))) )\
```

```
                         /(np.transpose(deltaGk).dot(deltaXk-Hk.dot(deltaGk))))))
            normx = np.linalg.norm(X_init[-1]-X_init[-2])
            #if np.all(np.linalg.eigvals(H[-1]) > 0):
            #    print('The estimated Hk+1 is not pd, descent not guaranteed')
            #    break
            #else:
            #    continue
            iters+=1
        return X_init

def DFP(X_init, f):
    """
    DFP update for estimating the Inverse hessian
    Initial Hk chosen to be 10,2; 2, 10 - pd and symmetric
    inputs - Initial array of X, function f
    outputs - updated array of X
    Apart from Hk+1 updation the rest of the steps for this algorithm are
    similar to that in Rank one update
    """
    normx = 1
    H = []
    H.append(np.array([[1,0],[0,1]]))
    if grad(X_init[0][0], X_init[0][1]) == (0,0):
        return X_init
    else:
        Hk = H[-1]
        gradk = grad(X_init[0][0],X_init[0][1])
        d = -Hk.dot(gradk)
        fAlph = lambda alpha: (f(X_init[-1][0] + alpha*d[0], X_init[-1][0] + \
                                 alpha*d[1]))
        al1, al2 = Bracketing(fAlph, 0, 1)
        alpa, alpb = goldSec(al1, al2, fAlph)
        alphak = (alpa+alpb)/2
        X_init.append(X_init[-1]+np.multiply(alphak,d))
        iters = 1
        while normx > epsilon and iters < max_iterations:
            Hk = H[-1]
            gradk = grad(X_init[-1][0],X_init[-1][1])
            dk = -Hk.dot(gradk)
            fAlph = lambda alphak: (f(X_init[-1][0] + alphak*dk[0], \
                                     X_init[-1][1] + alphak*dk[1]))
            al1, al2 = Bracketing(fAlph, 0, 1)
            alpa, alpb = goldSec(al1, al2, fAlph)
            alphak = (alpa+alpb)/2
            X_init.append(X_init[iters]+np.multiply(alphak,dk))
            deltaXk = alphak*dk
            deltaGk = np.subtract(grad(X_init[-1][0],X_init[-1][1]),gradk)
            #DFP update formula for Hk+1
            H.append(np.array(Hk+(deltaXk.dot(np.transpose(deltaXk))/\
                                  (np.transpose(deltaXk).dot(deltaGk))) -\
                              (np.transpose((Hk.dot(deltaGk))).dot(Hk.dot(deltaGk)))/ \
                              ((np.transpose(deltaGk).dot(Hk)).dot(deltaGk))))
            normx = np.linalg.norm(X_init[-1]-X_init[-2])
            iters+=1
        return X_init

def BFGS(X_init, f):
    """
    BFGS update for estimating the Inverse hessian
    Hk
    Initial Hk chosen to be 10,2; 2, 10 - pd and symmetric
    inputs - Initial array of X, function f
    outputs - updated array of X
    Apart from Hk+1 updation the rest of the steps for this algorithm are
    similar to that in Rank one update
```

```
    """
    normx = 1
    H = []
    H.append(np.array([[1,0],[0,1]]))
    if grad(X_init[0][0], X_init[0][1]) == (0,0):
        return X_init
    else:
        Hk = H[-1]
        gradk = grad(X_init[0][0],X_init[0][1])
        d = -Hk.dot(gradk)
        fAlph = lambda alpha: (f(X_init[-1][0] + alpha*d[0], X_init[-1][0] \
                                 + alpha*d[1]))
        al1, al2 = Bracketing(fAlph, 0,3)
        alpa, alpb = goldSec(al1, al2, fAlph)
        alphak = (alpa+alpb)/2
        X_init.append(X_init[-1]+np.multiply(alphak,d))
        iters = 1
        while normx > epsilon and iters < max_iterations:
            Hk = H[-1]
            gradk = grad(X_init[-1][0],X_init[-1][1])
            dk = -Hk.dot(gradk)
            fAlph = lambda alphak: (f(X_init[-1][0] + alphak*dk[0], \
                                      X_init[-1][1] + alphak*dk[1]))
            al1, al2 = Bracketing(fAlph, 0, 3)
            alpa, alpb = goldSec(al1, al2, fAlph)
            alphak = (alpa+alpb)/2
            X_init.append(X_init[iters]+np.multiply(alphak,dk))
            deltaXk = alphak*dk
            deltaGk = np.subtract(grad(X_init[-1][0],X_init[-1][1]),gradk)
            #BFGS update formula for Hk+1
            H.append(np.array(Hk+np.array(1+(((np.transpose(deltaGk)).dot(Hk))\
                                              .dot(deltaGk))/(np.transpose(deltaGk)\
                                                  .dot(deltaXk)))\
                                  .dot(((deltaXk.dot(np.transpose(deltaXk)))\
                                        /(np.transpose(deltaGk)).dot(deltaXk)))\
                                  -((((Hk.dot(deltaGk)).dot(np.transpose(deltaXk)))+ \
                                    np.transpose(((Hk.dot(deltaGk)).\
                                                  dot(np.transpose(deltaXk)))))\
                                    /((np.transpose(deltaGk)).dot(deltaXk)))))
            normx = np.linalg.norm(X_init[-1]-X_init[-2])
            iters+=1
        return X_init

def plotSeq(X_init, meth_eg):
    """
    Plot the sequence of Xk on level sets of f
    and save the level sets and points in a .csv file
    """
    yplt = xplt = np.arange(-1, 1, 0.025)
    Xpt, Ypt = np.meshgrid(xplt, yplt)
    Z = f(Xpt,Ypt)
    fig, ax = plt.subplots(num = None, figsize = (8,6), dpi = 90, facecolor = 'w', edgecolor =
    'k')
#     plt.figure()
    CS = ax.contour(Xpt, Ypt, Z)
    ax.clabel(CS, inline=1, fontsize=10)
    ax.set_title('Sequence of points')
    plx = []
    ply = []
    fval = []
    for i in range(len(X_init)):
        plx.append(X_init[i][0])
        ply.append(X_init[i][1])
        fval.append(f(plx[-1], ply[-1]))
    plt.plot(plx, ply , 'b-^')
```

```python
    if meth_eg == 1:
        np.savetxt('RankOneSeqPtsEg1.csv', np.column_stack((plx,ply,fval)), \
                      delimiter = ",", fmt = '%s')
    elif meth_eg == 2:
        np.savetxt('RankOneSeqPtsEg2.csv', np.column_stack((plx,ply,fval)), \
                      delimiter = ",", fmt = '%s')
    elif meth_eg == 3:
        np.savetxt('DFPSeqPtsEg1.csv', np.column_stack((plx,ply,fval)), \
                      delimiter = ",", fmt = '%s')
    elif meth_eg == 4:
        np.savetxt('DFPSeqPtsEg2.csv', np.column_stack((plx,ply,fval)), \
                      delimiter = ",", fmt = '%s')
    elif meth_eg == 5:
        np.savetxt('BFGSSeqPtsEg1.csv', np.column_stack((plx,ply,fval)), \
                      delimiter = ",", fmt = '%s')
    elif meth_eg == 6:
        np.savetxt('BFGSSeqPtsEg2.csv', np.column_stack((plx,ply,fval)), \
                      delimiter = ",", fmt = '%s')
#     plt.quiver(plx[:-1], ply[:-1], scale_units='xy', angles='xy', scale=1)

if __name__ == "__main__":
    #define function in terms of x1 x2
    f = lambda x1, x2: ((x2-x1)**4 + 12*x1*x2 - x1 + x2 - 3)
    #supply gradients
    grad = lambda x1, x2: ([(12*x2 + 4*(x1-x2)**3 - 1),(12*x1 - 4*(x1-x2)**3 + 1)])
    #dy = lambda x1, x2:
    epsilon = 0.01
    max_iterations = 1000

    X_init0 = []
    X_init0.append(np.array([0.55,0.7]))
    X_init0 = RankOne(X_init0,f)
    plotSeq(X_init0,1)

    X_init1 = []
    X_init1.append(np.array([-0.9,-0.5]))
    X_init1 = RankOne(X_init1,f)
    plotSeq(X_init1,2)

    X_init2 = []
    X_init2.append(np.array([0.55,0.7]))
    X_init2 = DFP(X_init2,f)
    plotSeq(X_init2,3)

    X_init3 = []
    X_init3.append(np.array([-0.9,-0.5]))
    X_init3 = DFP(X_init3,f)
    plotSeq(X_init3,4)

    X_init4 = []
    X_init4.append(np.array([.55,0.7]))
    X_init4 = BFGS(X_init4,f)
    plotSeq(X_init4,5)

    X_init5 = []
    X_init5.append(np.array([-0.9,-0.5]))
    X_init5 = BFGS(X_init5,f)
    plotSeq(X_init5,6)
```

Code 1: Minimization using Quasi Newton methods

The methods listed above use an estimated Hessian inverse to avoid the calculation of the Hessian inverse as we do in the Newtons method. The algorithm to calculate the $H_k$ for every method is different.

$\boxed{H_{k+1} = H_k + \Delta H_k}$

1. For rank one update update for $H_k$.

$$H_{k+1} = H_k + \frac{(\Delta x^{(k)} - H_k \Delta g^{(k)})(\Delta x^{(k)} - H_k \Delta g^{(k)})^T}{\Delta g^{(k)}(\Delta x^{(k)} - H_k \Delta g^{(k)})}$$

But for such a case, the $H_{k+1}$ may not come up to be positive definite, which won't guarantee descent.
The solution for this is rank 2 update.
We use the DFP algorithm to solve the problem.

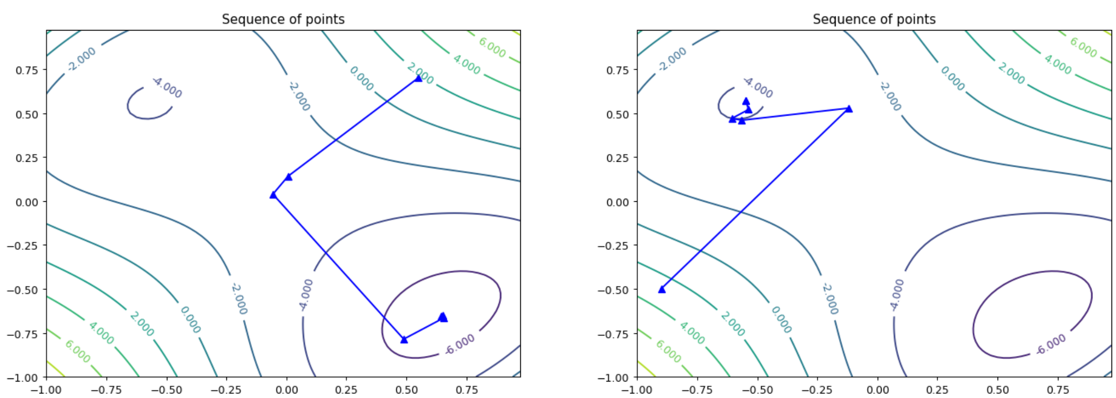
2. The DFP algorithm update for $H_k$.

$$H_{k+1} = H_k + \frac{[\Delta x^{(k)}][\Delta x^{(k)T}]}{\Delta x^{(k)T} \Delta g^{(k)}} - \frac{[H_k \Delta g^{(k)}][H_k \Delta g^{(k)}]^T}{\Delta g^{(k)T} H_k \Delta g^{(k)}}$$

However, DFP algorithm tends to get stuck for larger problems.
Hence, we use the BFGS algorithm which uses the concept of duality or complements.

3. THE BFGS algorithm update for $H_k$.

$$H_{k+1} = H_k + (1 + \frac{[\Delta g^{(k)}][\Delta x^{(k)T}]}{\Delta g^{(k)T} \Delta x^{(k)}}) \frac{[\Delta x_{(k)}][\Delta x_{(k)}^T]}{\Delta g_{(k)}^T \Delta x_{(k)}} - \frac{[H_k \Delta g^{(k)} \Delta x^{(k)T}][H_k \Delta g^k \Delta x^{(k)T}]^T}{\Delta g^{(k)T} \Delta x^{(k)}}$$



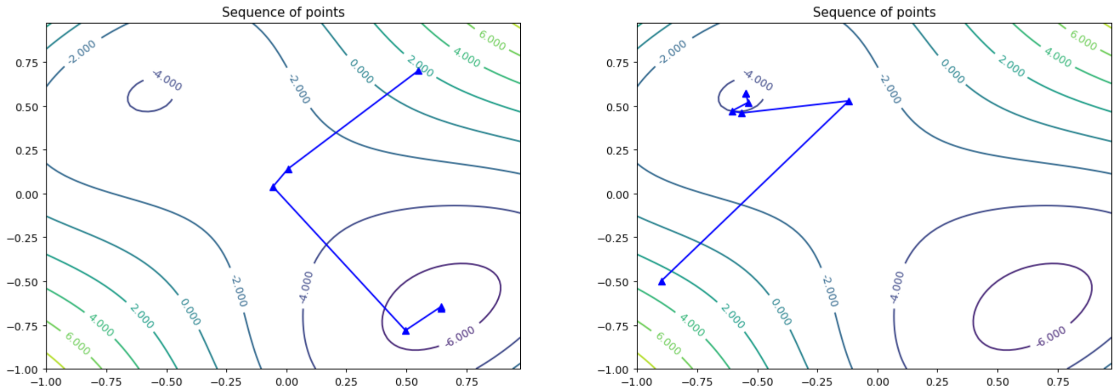

(a) Initial point = 0.55,0.7

(b) Initial point = -0.9,-0.5

Figure 1: Rank One update Sequence of points

| X | Y | f |
|---|---|---|
| 0.55 | 0.7 | 1.77050625 |
| 0.007297347 | 0.14061915 | -2.854048497 |
| -0.055853 | 0.038430448 | -2.931395001 |
| 0.489168063 | -0.785604421 | -6.245512717 |
| 0.656829996 | -0.664435843 | -6.510702465 |
| 0.645434583 | -0.656197846 | -6.513552996 |
| 0.651010288 | -0.651238225 | -6.513890669 |

| X | Y | f |
|---|---|---|
| -0.9 | -0.5 | 2.8256 |
| -0.118517765 | 0.527903314 | -2.929762817 |
| -0.563848054 | 0.459001504 | -3.988256862 |
| -0.605154588 | 0.469356235 | -4.000846138 |
| -0.535930493 | 0.520003333 | -4.04507726 |
| -0.548092856 | 0.572709684 | -4.067940243 |
| -0.548244319 | 0.56842678 | -4.068074801 |

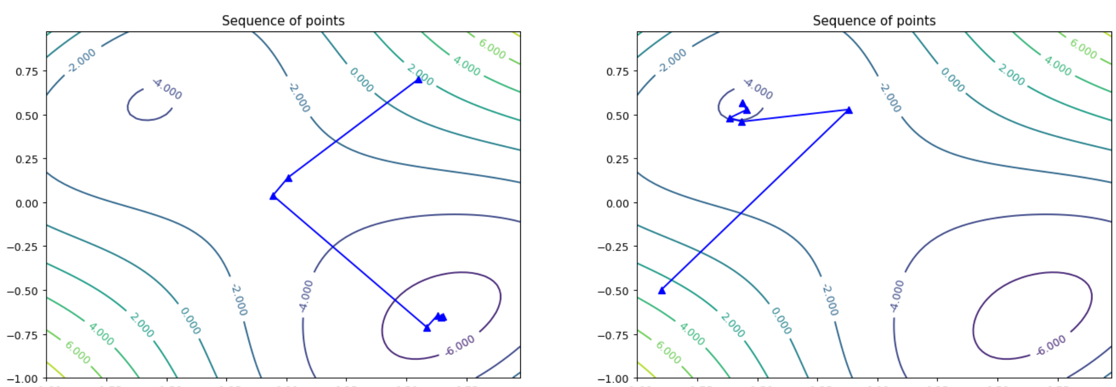



(a) Initial point = 0.55,0.7 (b) Initial point = -0.9,-0.5

Figure 2: DFP Update sequence of points

| X | Y | f |
|---|---|---|
| 0.55 | 0.7 | 1.77050625 |
| 0.007297347 | 0.14061915 | -2.854048497 |
| -0.055853 | 0.038430448 | -2.931395001 |
| 0.496503118 | -0.780528182 | -6.26790961 |
| 0.643706433 | -0.646228255 | -6.513042073 |
| 0.646654561 | -0.653090272 | -6.513772218 |

| X | Y | f |
|---|---|---|
| -0.9 | -0.5 | 2.8256 |
| -0.118517765 | 0.527903314 | -2.929762817 |
| -0.563848054 | 0.459001504 | -3.988256862 |
| -0.605803284 | 0.470129456 | -4.001628743 |
| -0.537312144 | 0.5188848 | -4.044984078 |
| -0.548157809 | 0.572324826 | -4.067975854 |
| -0.548221159 | 0.568744919 | -4.068071161 |




(a) Initial point = 0.55,0.7 (b) Initial point = -0.9,-0.5

Figure 3: BFGS update sequence of points

| X | Y | f |
|---|---|---|
| 0.55 | 0.7 | 1.77050625 |
| 0.007297347 | 0.14061915 | -2.854048497 |
| -0.055853 | 0.038430448 | -2.931395001 |
| 0.584996509 | -0.711453832 | -6.465793308 |
| 0.63028041 | -0.643055696 | -6.508112203 |
| 0.645878768 | -0.655488904 | -6.513625979 |
| 0.650956631 | -0.651310305 | -6.513890075 |

| X | Y | f |
|---|---|---|
| -0.9 | -0.5 | 2.8256 |
| -0.118517765 | 0.527903314 | -2.929762817 |
| -0.563848054 | 0.459001504 | -3.988256862 |
| -0.613883243 | 0.480348338 | -4.010672297 |
| -0.544150016 | 0.526406849 | -4.053247804 |
| -0.561990744 | 0.567589651 | -4.070127997 |
| -0.56190924 | 0.567366709 | -4.070128505 |

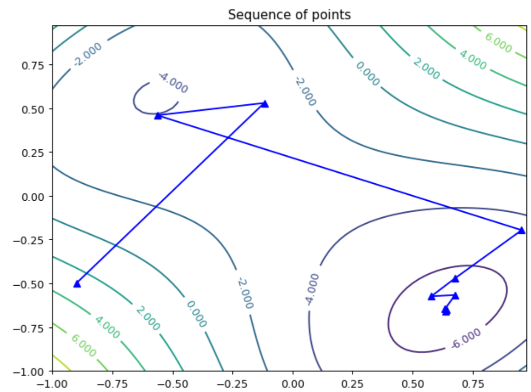



Figure 4: High value of alpha - leading towards global minima

We can see that it takes approximately 4-6 iterations to reach near the minima.
Also, the initial value of $H_k$ and $\alpha$ also decides how fast/ slow the algorithm will reach the minima.
My observation is keeping a high value of initial $H_k$ or $\alpha$ will make the algorithm take large steps and may lead it to the global minima at times.
However, for high values of alpha/ Hk, we can lose positive definiteness of the matrix

# Problem 5

We know that $F = ma$
Here we know that the vectors for F and a are given

$$F = \begin{bmatrix}1\\2\\3\end{bmatrix} \; a = \begin{bmatrix}3\\5\\6\end{bmatrix}$$

As F = ma

$$m^* = \begin{bmatrix}a^T & a\end{bmatrix}^{-1} a^T F$$

$$= \begin{bmatrix}\begin{bmatrix}3 & 5 & 6\end{bmatrix} & \begin{bmatrix}3\\5\\6\end{bmatrix}\end{bmatrix}^{-1} \begin{bmatrix}3 & 5 & 6\end{bmatrix} \begin{bmatrix}1\\2\\3\end{bmatrix}$$

$$= [70]^{-1}[31] = 31/70kg$$

# Problem 6

We know that $p_x = ap_y + bp_z$
From the given data we can formulate the equations which take the matrix form as follows

$$\begin{bmatrix}1 & 2\\1 & 1\\3 & 2\end{bmatrix} \begin{bmatrix}a\\b\end{bmatrix} = \begin{bmatrix}6\\4\\5\end{bmatrix}$$

Thus, $\begin{bmatrix}a\\b\end{bmatrix}^* = \begin{bmatrix}p_{xy}^T & p_{xy}\end{bmatrix}^{-1} p_{xy}^T p_z = \begin{bmatrix}\begin{bmatrix}1 & 1 & 3\\2 & 1 & 2\end{bmatrix} & \begin{bmatrix}1 & 2\\1 & 1\\3 & 2\end{bmatrix}\end{bmatrix}^{-1} \begin{bmatrix}1 & 1 & 3\\2 & 1 & 2\end{bmatrix} \begin{bmatrix}6\\4\\5\end{bmatrix} \therefore \begin{bmatrix}a\\b\end{bmatrix}^* = \begin{bmatrix}-0.5\\3.3889\end{bmatrix}$