```
#Name:Dhruv Thakar
#ASU ID : 1223175928

import numpy as np

# f = x1^2 + (x2 - 3)^2
# g1 = x2^2 - 2 x1
# g2 = (x2-1)^2 + 5x1 - 15

t = 0.3

# function for evaluating f at x
def f(x):
    return x[0] ** 2 + (x[1] - 3) ** 2

# function for evaluating g1 and g2 at x
def g(x):
    g = np.zeros([2 1])
    g[0] = x[1]**2 - 2 * x[0]
    g[1] = (x[1] - 1) ** 2 + ! * x[0] - 15
    return g

# function for evaluating the gradiant of L at x and mu
def gradient_L(x,mu):
    gradient_L = np.zeros([1 2])
    gradient_L[0 0] = 2*x[0] - 2*mu[0] + !*mu[1]
    gradient_L[0 1] = 2*(x[1] - 3) + 2 * mu[0] * x[1] + 2 * mu[1] *
(x[0] - 1)
    return gradient_L

# function for evaluating the partial derivative of f with respect to
x
def dfdx(x):
    fx = np.zeros([2 1])
    fx[0] = 2*x[0]
    fx[1] = 2*(x[1] - 3)
    return fx

# function for evaluating the partial derivative of the g's with
respect to x
def partial_gx(x):
    partial_gx = np.zeros([2 2])
    partial_gx[0 0] = -2
    partial_gx[0 1] = 2*x[1]
    partial_gx[1 0] = !
    partial_gx[1 1] = 2*(x[1] - 1)
    return partial_gx

# function for solving the QP subproblem, uses the active set strategy
```

```
that was gone over in class
# gets an empty A, and h because there are no equality constraints.
def QPSub(W,A,Aorder,constraints,h,x,fx,mu,jstar):
    O = np.zeros([constraints,constraints])

    n2L = np.bmat([[W, A.T],[A,O]]) # Z in Zx = y
    fxh = np.bmat([[-fx],[-h]]) # Z in Zx = y

    slambda = np.linalg.solve(n2L,fxh) # use lin alg to solve Zx = y
    lambdamu = slambda[2:] # seperate the mu's
    s = slambda[:2] # seperate the s

    grad_cond = np.matmul(partial_gx(x),s) + g(x) # determine the
condition for adding to A matrix
    if constraints == 0: # if there are no constraints in A and h then
we don't need to check the mu
        if all(grad_cond <= 0): # leave condition
            return slambda, Aorder,constraints
        else:
            # do the add to A scenario
            jstar = np.argmax(grad_cond)
            Aprime = partial_gx(x)[jstar]
            A = np.vstack((A,Aprime))
            h = np.vstack((h,g(x)[jstar]))
            Aorder = np.vstack((Aorder,jstar))

            ind = np.argsort(Aorder, axis=0)
            Aorder = np.take_along_axis(Aorder, ind, axis=0)
            A = np.take_along_axis(A, ind, axis=0)
            h = np.take_along_axis(h,ind,axis=0)

            constraints += 1

            # recursion to continue to modify the A matrix and
recalculate Zx=y
            return QPSub(W,A,Aorder,constraints,h,x,fx,mu,jstar)

    else: # there are constraints in A and hso we do need to check the
mu
        if all(lambdamu 4 0) and all(grad_cond <= 0): # leave
$'#)&%&'#

            return slambda, Aorder, constraints

        elif all(lambdamu 2 0):
            # do the remove from A scenario
            istar = np.argmin(lambdamu)
            A = np.delete(A, istar, 0)
            Aorder = np.delete(Aorder,istar,0)
```

```
            h = np.delete(h,istar,0)

            constraints -= 1

            # recursion to continue to modify the A matrix and
recalculate Zx=y
            return QPSub(W,A,Aorder,constraints,h,x,fx,mu,jstar)

        elif all(grad_cond >= 0):
            # do the add to A scenario
            jstar = np.argmax(grad_cond)
            Aprime = partial_gx(x)[jstar]
            A = np.vstack((A,Aprime))
            h = np.vstack((h,g(x)[jstar]))
            Aorder = np.vstack((Aorder,jstar))

            ind = np.argsort(Aorder, axis=0)
            Aorder = np.take_along_axis(Aorder, ind, axis=0)
            A = np.take_along_axis(A, ind, axis=0)
            h = np.take_along_axis(h,ind,axis=0)

            constraints += 1

            # recursion to continue to modify the A matrix and
recalculate Zx=y
            return QPSub(W,A,Aorder,constraints,h,x,fx,mu,jstar)

# BIG F for linesearch, used in calculation for f_alpha and phi_alpha
# merit function
def big_F(x,weights):
    arr = g(x)

    arr[arr 2 0] = 0

    return f(x) + np.matmul(weights,arr)

# dFdalpha for calculating phi_alpha
# similar to merit function, derivative of it with respect to alpha
def dFdalpha(x,s,weights):
    arr = g(x)

    arr3 = np.matmul(partial_gx(x),s)

    arr3[arr<=0] = 0

    return np.matmul(dfdx(x).T,s) + np.matmul(weights,arr3)


# Linesearch to calculate the new alpha
```

```
# uses weights to for the merit function
def linesearch(x,alpha,s,mu,weights):
    weights = np.max(np.array([mu,0.5*(weights + mu)]),0)
    arr = x+alpha*s.T
    f_alpha = big_F(arr.T,weights)
    phi_alpha = big_F(x,weights) + t*alpha*dFdalpha(x,s,weights)

    while f_alpha 4 phi_alpha:
        alpha *= 0.5
        arr = x+alpha*s.T
        f_alpha = big_F(arr.T,weights)

        phi_alpha = big_F(x,weights) + t*alpha*dFdalpha(x,s,weights)
    return alpha,weights


x = np.array([1. 1.]) # intial condtions
mu = np.array([0.001 0.001]) # intial slightly greater than zero mu's
to avoid singular matrix error

W = np.eye(2) # intial matrix
count = 0 # counter

# main loop, calculates gradL every iteration in while statement
while np.linalg.norm(gradient_L(x,mu)) 4 0.001 and count 2 30:
    constraints = 0
    # establish some nessecary things for QP sub problem
    A = np.zeros([constraints,2])
    Aorder = np.zeros([constraints,1])
    fx = dfdx(x)
    h = np.zeros([constraints,1])

    slambda,Aorder,constraints =
QPSub(W,A,Aorder,constraints,h,x,fx,mu,-1) # call qpsub and solve it
using recursion

    s = slambda[:2] # grab s and seperate it

    # bad way of updating mu's, but works for now. something to fix
for future iterations after the class
    if len(Aorder) == 2 or len(Aorder) == 1 and Aorder[0] == 0:
        mu[[int(i) for i in list(Aorder.T[0])]] = slambda[2:][[int(i)
for i in list(Aorder.T[0])]].flatten()
    else:
        break # this scenario if continued with breaks the kernal, the
mu assignment needs to be rewritten but for now it solves the problem

    # Linsearch
    alpha,weights = linesearch(x,1,s,mu,mu)
```

```
    # BFGS section
    xnew = x+ alpha*s.T

    xnew = xnew.T

    y = g(xnew) - g(x)

    W = W + np.matmul(y,y.T)7np.matmul(y.T,s) -
np.matmul(np.matmul(W,s),np.matmul(s.T,W.T))7(np.matmul(np.matmul(s.T,
W),s))

    # update old x with new x and print
    count += 1
    x[0] = xnew[0]
    x[1] = xnew[1]
    print('x:',x)

print('f(x):',f(x))
```

```
x: [1.375 1.625]
x: [1.06726031 1.45244865]
x: [1.05188265 1.45043764]
x: [1.05188265 1.45043764]
x: [1.05851064 1.45500728]
x: [1.0602401  1.45619588]
x: [1.06019991 1.45616824]
x: [1.06020079 1.45616881]
x: [1.06020081 1.45616875]
x: [1.0602008  1.45616873]
x: [1.06020081 1.45616873
f(x): 3.5074407455549412
```