TPG4155 Computer Methods in Engineering
Exercise 1

Problem 1

A company is looking at a reservoir geometry and has decided that the best locations to place CO2 injectors named A, B, C, D and E are at the coordinates (10, 10), (30, 50), (16.667, 29), (0.555, 29.888) and (22.2221, 49.988), respectively, in the xy-plane. The company also needs a central platform to run the injectors from. It is anticipated that during an average week the total volume pumped through each injector pipeline will be 10, 18, 20, 14, and 25 (arbitrary units) for injectors A, B, C, D, and E, respectively. Longer pipelines require more powerful compressors in order to compensate for friction loss in the pipes. The company therefore wants to minimize the total distance-volume (distance from platform to injector multiplied by volume).

a) To accomplish this, where in the xy-plane should the platform be located? Use the steepest descent method. Visually verify that your solution is correct by plotting the function.

Solution:

This lengthy word problem can be distilled into the problem of minimizing the distance between the central platform and each injector, weighted by the average weekly volume. The distance between points on a two-dimensional plane can easily be found by making a sketch and using the Pythagorean theorem. With the platform at (x, y), the distance to a given injector at (x_i, y_i) is $\sqrt{(x - x_i)^2 + (y - y_i)^2}$. Thus the equation to be minimized is the following:

$$\begin{aligned} f(x,y) = &\, 10\sqrt{(x - A(1))^2 + (y - A(2))^2)} \\ &+ 18\sqrt{(x - B(1))^2 + (y - B(2))^2)} \\ &+ 20\sqrt{(x - C(1))^2 + (y - C(2))^2)} \\ &+ 14\sqrt{(x - D(1))^2 + (y - D(2))^2)} \\ &+ 25\sqrt{(x - E(1))^2 + (y - E(2))^2)}, \end{aligned} \tag{1}$$

where $A(1), A(2), B(1)$, etc. are the x- and y-coordinates of the different injectors. In order to use the steepest decent method, we also need the gradient of $f(x, y)$. That means we have to

differentiate $f(x,y)$ in both x- and y-directions:

$$\begin{aligned}\frac{\partial f}{\partial x} &= 10\frac{(x-A(1))}{\sqrt{(x-A(1))^2+(y-A(2))^2)}} \\ &+ 18\frac{(x-B(1))}{\sqrt{(x-B(1))^2+(y-B(2))^2)}} \\ &+ 20\frac{(x-C(1))}{\sqrt{(x-C(1))^2+(y-C(2))^2)}} \\ &+ 14\frac{(x-D(1))}{\sqrt{(x-D(1))^2+(y-D(2))^2)}} \\ &+ 25\frac{(x-E(1))}{\sqrt{(x-E(1))^2+(y-E(2))^2)}},\end{aligned} \tag{2}$$

$$\begin{aligned}\frac{\partial f}{\partial y} &= 10\frac{(y-A(2))}{\sqrt{(x-A(1))^2+(y-A(2))^2)}} \\ &+ 18\frac{(y-B(2))}{\sqrt{(x-B(1))^2+(y-B(2))^2)}} \\ &+ 20\frac{(y-C(2))}{\sqrt{(x-C(1))^2+(y-C(2))^2)}} \\ &+ 14\frac{(y-D(2))}{\sqrt{(x-D(1))^2+(y-D(2))^2)}} \\ &+ 25\frac{(y-E(2))}{\sqrt{(x-E(1))^2+(y-E(2))^2)}}.\end{aligned} \tag{3}$$

To use the steepest decent method, we then need to define the objective function and its gradient, as

```
locInjVol=np.array([[10,10,10],[30,50,18],[16.667,29,20],[0.555,29.888,14],[22.2221,49.988,25]]

def distvol(locPlat,locInj):
    totDistVol=0.0
    for inj in locInj:
        totDistVol+=inj[-1]*np.sqrt(np.sum((locPlat[:]-inj[:-1])**2))
    return totDistVol

def gradDistVol(locPlat,locInj):
    grad=np.array([0.0,0.0])
    for dir in [0,1]:
        for inj in locInj:
            grad[dir]+=inj[-1]*(locPlat[dir]-inj[dir])/np.sqrt(np.sum((locPlat[:]-inj[:-1])**2)
    return grad
```

We then run the following code

```
locPlat=np.array([0.0,0.0])

#Max number of steps
nn=1000
#Constant for step length
```

```
gamma=0.1
#Convergence criteria
eps=1E-6
diff=2*eps
ii=0
while ii<nn and diff>eps:
    ii+=1
    nextLocPlat=locPlat-gamma*gradDistVol(locPlat,locInjVol)
    diff=np.sqrt(np.sum((locPlat-nextLocPlat)**2))
    locPlat=np.copy(nextLocPlat)

print(locPlat,ii,diff)
```

This gives the optimal position $P = [19.812363, 41.123202]$. *Solution found after 236 iterations. Figure 1 shows the objective function* $f(x,y)$*, injector well locations and platform location found by optimization. Notice that the platform is indeed in the minimum of the function.*

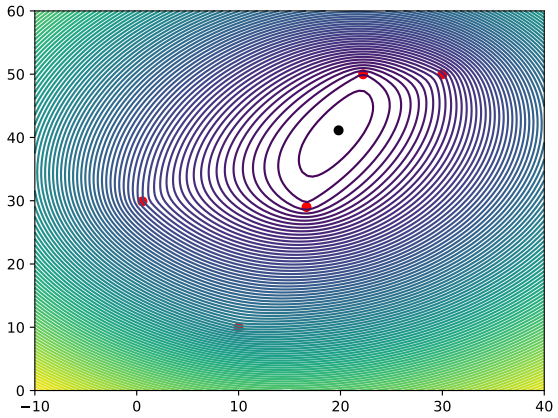

Figure 1: Objective function used in minimization, injector wells (red) and platform feeding the wells found in the optimization process (black).

b) Use the Nelder-Mead algorithm to solve the same problem. Compare the two solutions (number of iterations, number of objective function evaluations, run time, accuracy, etc.).

Solution:
We then run the following code

```
def hObjectiveFunction(fx,fy):
    return -distvol(np.array([fx,fy]),locInjVol)

#Coefficients
fAlpha=1
fGamma=1
fRho=0.5
fSigma=0.5

def updateFuncVal(aaSimplex):
    for ii in range(0,3):
        aaSimplex[ii,2]=hObjectiveFunction(aaSimplex[ii,0],aaSimplex[ii,1])
    return aaSimplex
```

```
def updateFuncValSingle(aPoint):
    aPoint[2]=hObjectiveFunction(aPoint[0],aPoint[1])
    return aPoint

def sortPoints(aaSimplex):
    iChange=1
    while iChange>0:
        iChange=0
        if aaSimplex[0,2]<aaSimplex[1,2]:
            aPointTemp=np.copy(aaSimplex[0,:])
            aaSimplex[0,:]=np.copy(aaSimplex[1,:])
            aaSimplex[1,:]=np.copy(aPointTemp)
            iChange+=1
        if aaSimplex[1,2]<aaSimplex[2,2]:
            aPointTemp=np.copy(aaSimplex[1,:])
            aaSimplex[1,:]=np.copy(aaSimplex[2,:])
            aaSimplex[2,:]=np.copy(aPointTemp)
            iChange+=1
    return aaSimplex

def hNedlerMead(aaSimplex):
    #Calculate centroid
    aCentroid=np.array([np.sum(aaSimplex[:-1,0])/2.0,np.sum(aaSimplex[:-1,1])/2.0,0.0])
    aCentroid=updateFuncValSingle(aCentroid)
    #Find reflection point
    aReflection=aCentroid+fAlpha*(aCentroid-aaSimplex[-1,:])
    aReflection=updateFuncValSingle(aReflection)
    if aReflection[2]>aaSimplex[1,2]:
        if aReflection[2]<aaSimplex[0,2]:
            #Replace the worst point with the reflection point
            aaSimplex[2,:]=np.copy(aReflection)
        else:
            #Find expansion point
            aExpansion=aReflection+fGamma*(aReflection-aCentroid)
            aExpansion=updateFuncValSingle(aExpansion)
            if aExpansion[2] > aReflection[2]:
                aaSimplex[2,:]=np.copy(aExpansion)
            else:
                aaSimplex[2,:]=np.copy(aReflection)
    else:
        #Calculate contraction point inside or outside
        if aReflection[2]>aaSimplex[2,2]:
            #Check contration outside
            aContraction=aCentroid+fRho*(aReflection-aCentroid)
            aContraction=updateFuncValSingle(aContraction)
        else:
            #Check contration inside
            aContraction=aCentroid+fRho*(aaSimplex[2,:]-aCentroid)
            aContraction=updateFuncValSingle(aContraction)
        if (aReflection[2]>aaSimplex[2,2] and ...
            aContraction[2]>aReflection[2]) or ...
            (aReflection[2]≤aaSimplex[2,2] and aContraction[2]>aaSimplex[2,2]):
            aaSimplex[2,:]=np.copy(aContraction)
        #Shrink
        else:
            aaSimplex[:,:]=aaSimplex[0,:]+fSigma*(aaSimplex[:,:]-aaSimplex[0,:])
    aaSimplex=updateFuncVal(aaSimplex)
    return aaSimplex
```

```
#Define the initial simplex
aaSimplex=np.zeros((3,3))
aaSimplex[0,:-1]=[0.0,0.0]
aaSimplex[1,:-1]=[20.0,0.0]
aaSimplex[2,:-1]=[0.0,20.0]

aaSimplex=updateFuncVal(aaSimplex)
aaSimplex=sortPoints(aaSimplex)

#Set convergence criteria
fEps=1E-3
iMaxIt=200
fMaxError=np.sqrt((aaSimplex[0,2]-aaSimplex[2,2])**2)
fError=fMaxError

ii=0
while ii<iMaxIt and fError>fEps:
    ii+=1
    aaSimplex=hNedlerMead(aaSimplex)
    sortPoints(aaSimplex)
    fError=np.sqrt((aaSimplex[0,2]-aaSimplex[2,2])**2)
    print('Iteration ',ii,' Error: ',fError,' Objective of best point: ...
        ',aaSimplex[0,2])

print(aaSimplex[0,:])
```

We observe that the code converge in 22 iterations, and with a very similar result as in the previous problem.