Computer Methods in Engineering
Exercise on pattern search

All the exercises in derivative-free optimization will be based on the same problems. The problems will be described in this exercise, and the different methods will be applied in the following exercises. This means that you will probably prefer to re-use your code from one exercise to the next. You are of course free to do so, but you can also choose to write new code for each exercise.

There will be underlying problems to solve, one regarding optimal location of a platform for CO_2 injection, and one regarding optimal location of wind turbines. The problems will be described in the following.

Problem 1

A company is looking at a reservoir geometry and has decided that the best locations to place CO_2 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

```
    # Objective function for the CO2 injection problem
    # Want to minimize distance to injection points weighted by injection ...
        volume
    def fDistVol(self, afLocPlat):
        fTotDistVol = 0.0
        for afInj in self.aafLocInjVol:
            fTotDistVol += afInj[-1] * ...
                self.np.sqrt(self.np.sum((afLocPlat[:] - afInj[:-1]) ** 2))
        return fTotDistVol

    # Gradient of the objective function for the CO2 injection problem
    def afGradDistVol(self, afLocPlat):
        afGrad = self.np.array([0.0, 0.0])
        for iDir in [0, 1]:
            for afInj in self.aafLocInjVol:
                afGrad[iDir] += afInj[-1] * (afLocPlat[iDir] - afInj[iDir]) ...
                    / self.np.sqrt(self.np.sum((afLocPlat[:] - afInj[:-1]) ...
                    ** 2))
        return afGrad
```

We then run the following code

```
    def gradientDescent(self, afStartPoint, afGradient, fStepSize=0.1, ...
        fEps=1E-6, iMaxIt=1000):
        afCurPoint = self.np.copy(afStartPoint)
        afNextPoint = self.np.copy(afStartPoint)
        diff = 2 * fEps
        ii = 0
        while ii < iMaxIt and diff > fEps:
            ii += 1
            afNextPoint = afCurPoint - fStepSize * afGradient(afCurPoint)
            diff = self.np.sqrt(self.np.sum((afCurPoint - afNextPoint) ** 2))
            afCurPoint = self.np.copy(afNextPoint)
        return afCurPoint, ii, diff

    def plotGradientDescent(self, fObjectiveFunc, afGradient,aafLocations, ...
        plotName):
        x = self.np.linspace(-10, 40, 100)
        y = self.np.linspace(0, 60, 100)
        aafXv, aafYv = self.np.meshgrid(x, y)
        afXr = self.np.ravel(aafXv)
        afYr = self.np.ravel(aafYv)
        afZ = []
        for jj in range(len(afXr)):
            afZ.append(fObjectiveFunc(self.np.array([afXr[jj], afYr[jj]])))
        afZr = self.np.asarray(afZ)
        aafZv = afZr.reshape(aafXv.shape)
        afLocPoint = self.np.array([0.0, 0.0])
        afLocPoint, ii, diff = self.gradientDescent(afLocPoint, afGradient, ...
            fStepSize=0.1, fEps=1E-6, iMaxIt=1000)
        print('Gradient descent results: ',afLocPoint, ii, diff, ...
            fObjectiveFunc(afLocPoint))
        fig = plt.figure()
        plt.contour(aafXv, aafYv, aafZv, 100)
        plt.scatter(afLocPoint[0], afLocPoint[1], color='k')
        plt.scatter(aafLocations[:, 0], aafLocations[:, 1], color='r')
        plt.savefig(plotName, bbox_inches='tight')

    def plotCO2GradientDescent(self):
        self.plotGradientDescent(self.fDistVol, self.afGradDistVol, ...
            self.aafLocInjVol, 'platLocCO2.pdf')
```

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, where we have an objective value of* 1361.

b) Use the pattern search 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 patternSearch(self, afStartPoint, fObjectiveFunc,fStepSize=1.0, ...
        fRedStepSize=0.5,fCutStepSize=0.01, iMaxIt=1000):
        afCurPoint = self.np.copy(afStartPoint)
        afNextPoint = self.np.copy(afStartPoint)
```

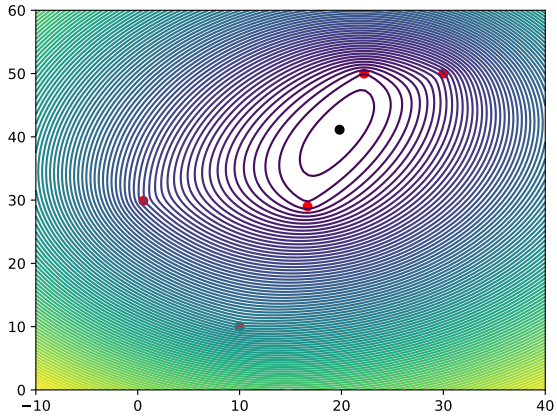

Figure 1: Objective function used in minimization, injector wells (red) and platform feeding the wells found in the optimization process (black).

```
        ii = 0
        while ii < iMaxIt and fStepSize > fCutStepSize:
            ii += 1
            fCurVal = fObjectiveFunc(afCurPoint)
            afTestPoints = self.np.array([afCurPoint + ...
                self.np.array([fStepSize, 0.0]),
                                          afCurPoint + ...
                                              self.np.array([-fStepSize, 0.0]),
                                          afCurPoint + self.np.array([0.0, ...
                                              fStepSize]),
                                          afCurPoint + self.np.array([0.0, ...
                                              -fStepSize])])
            fBestVal = fCurVal
            afBestPoint = self.np.copy(afCurPoint)
            for afTest in afTestPoints:
                fTestVal = fObjectiveFunc(afTest)
                if fTestVal < fBestVal:
                    fBestVal = fTestVal
                    afBestPoint = self.np.copy(afTest)
            if fBestVal < fCurVal:
                afNextPoint = self.np.copy(afBestPoint)
            else:
                fStepSize *= fRedStepSize
                afNextPoint = self.np.copy(afCurPoint)
            afCurPoint = self.np.copy(afNextPoint)
        return afCurPoint, ii

    def runCO2PatternSearch(self):
        afLocPoint = self.np.array([0.0, 0.0])
        afLocPoint, ii = self.patternSearch(afLocPoint, self.fDistVol, ...
            fStepSize=1.0, fRedStepSize=0.5, fCutStepSize=0.01, iMaxIt=1000)
        print('CO2 Pattern search: ', afLocPoint, ii, ...
            self.fDistVol(afLocPoint))
```

We observe that the code converge in 74 iterations, and with a very similar result as in the previous problem.

Problem 2

A company have an offshore wind-farm. They currently have 5 wind turbines, all located

inside their license which we assume to be inside the area $[0-20, 0-20]$ in the xy-plane, where the unit is in kilometers. Assume that the existing wind turbines have coordinates (2.5,3.5), (17.2,3.2), (4.2,16.8), (16.2,18.4) and (11.1,9.8). The company is looking at placing an additional wind turbine (a sixth wind turbine in addition to the existing ones).

The wind power is given by the function

$$\text{erf}\left(\frac{1}{5}\sqrt{\sum_i (x_w - x_i)^2 + (y_w - x_i)^2}\right) \tag{4}$$

where erf is the error function, i runs over all existing wind turbine locations (x_i, y_i), and (x_w, y_w) is the location of the suggested new wind turbine location.

The company is also given a bonus for placing turbines away from the license border, given as

$$\text{erf}\left(\text{abs}(x_w - 0.0)\right) + \text{erf}\left(\text{abs}(x_w - 20.0)\right) + \text{erf}\left(\text{abs}(y_w - 0.0)\right) + \text{erf}\left(\text{abs}(y_w - 20.0)\right) \tag{5}$$

The objective is then to maximize the sum of the two equations above.

a) Use the pattern search method to find the optimal wind turbine location for an additional wind turbine (a sixth wind turbine in addition to the existing ones).

Solution:

The objective function for the wind turbine location problem is given as

```
    # Objective function for the wind farm location problem
    # Want to maximize distance to other turbines and boundaries
    def fWindFunc(self, afNewLoc):
        from scipy.special import erf
        fTotDistPen = 0.0
        for afTurbine in self.aafTurbineLocations:
            fTotDistPen += erf(0.2 * self.np.sqrt(self.np.sum((afTurbine[:] ...
                - afNewLoc[:]) ** 2)))
        fTotDistPen += erf(self.np.abs(afNewLoc[0] - 0.0))
        fTotDistPen += erf(self.np.abs(afNewLoc[0] - 20.0))
        fTotDistPen += erf(self.np.abs(afNewLoc[1] - 0.0))
        fTotDistPen += erf(self.np.abs(afNewLoc[1] - 20.0))
        # We want to maximize distance, so return negative value to use ...
            minimization algorithms
        return -fTotDistPen
```

The problem can be solved in python by the following script:

```
    def runWindPatternSearch(self, afLocPoint):
        afLocPoint = self.np.array([10.0, 10.0])
        afLocPoint, ii = self.patternSearch(afLocPoint, self.fWindFunc, ...
            fStepSize=5.0, fRedStepSize=0.5, fCutStepSize=0.1, iMaxIt=1000)
        print('Wind turbine Pattern search: ', afLocPoint, ii, ...
            self.fWindFunc(afLocPoint))
        self.plotWindPatternSearch(afLocPoint)
```

Depending on the starting location and velocities, this can give different solutions. One solution is shown in Figure 2, that was obtained with starting location $(10, 10)$*, and which used 12 iterations to end at the location* $(2.03, 10.31)$*, which gives a maximum value of* 8.878*.*

The plotting is done by the following code:

```
    def plotWindPatternSearch(self, afLocPoint):
        x = self.np.linspace(0.0, 20.0, 100)
        y = self.np.linspace(0.0, 20.0, 100)
        aafXv, aafYv = self.np.meshgrid(x, y)
        afXr = self.np.ravel(aafXv)
        afYr = self.np.ravel(aafYv)
        afZ = []
        for jj in range(len(afXr)):
            afZ.append(self.fWindFunc(self.np.array([afXr[jj], afYr[jj]])))
        afZr = self.np.asarray(afZ)
        aafZv = afZr.reshape(aafXv.shape)
        fig = plt.figure()
        plt.contourf(aafXv, aafYv, aafZv, 20)
        plt.scatter(self.aafTurbineLocations[:, 0], ...
            self.aafTurbineLocations[:, 1], color='r')
        plt.scatter(afLocPoint[0], afLocPoint[1], color='k')
        plt.savefig('windPatternSearch.pdf', bbox_inches='tight')
```

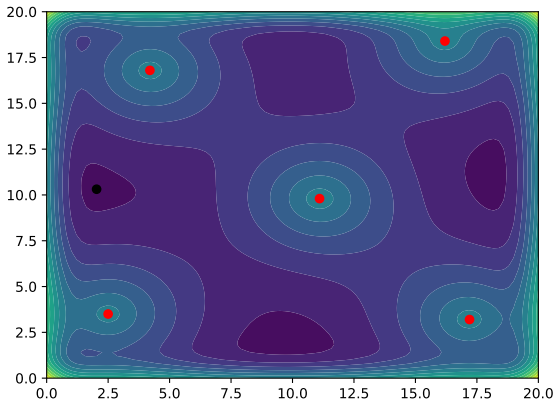

Figure 2: Objective function used in maximization, existing wind turbines in red and optimal location of new turbine in black.

b) What happens when you re-run your algorithm - do you get the same result?

Solution:

The pattern search algorithm will give the same result when re-run, as it is a deterministic algorithm. However, the result can depend on the starting location. A good strategy is to run the algorithm several times with different starting locations to find the global optimum. In our problem we have several local optima, and the algorithm can get stuck in one of these.