Computer Methods in Engineering
Exercise on the Nelder-Mead algorithm

We will now continue with the same two problems as we had in the exercise for the pattern search algorithm.

Problem 1
Use the Nelder-Mead algorithm to find where in the xy-plane the platform for the CO_2 injection should be located?
Solution:
We use the same objective function as in the previous exercise, and implement the Nelder-Mead algorithm in Python. The implementation is shown below:

```
    def aafUpdateFuncVal(self, aafSimplex, fObjectiveFunction):
        for ii in range(3):
            aafSimplex[ii, 2] = fObjectiveFunction(aafSimplex[ii, :-1])
        return aafSimplex

    def afUpdateFuncValSingle(self, afPoint, fObjectiveFunction):
        afPoint[2] = fObjectiveFunction(afPoint[:-1])
        return afPoint

    def aafSortPoints(self, aafSimplex):
        iChange = 1
        while iChange > 0:
            iChange = 0
            if aafSimplex[0, 2] > aafSimplex[1, 2]:
                afPointTemp = self.np.copy(aafSimplex[0, :])
                aafSimplex[0, :] = self.np.copy(aafSimplex[1, :])
                aafSimplex[1, :] = self.np.copy(afPointTemp)
                iChange += 1
            if aafSimplex[1, 2] > aafSimplex[2, 2]:
                afPointTemp = self.np.copy(aafSimplex[1, :])
                aafSimplex[1, :] = self.np.copy(aafSimplex[2, :])
                aafSimplex[2, :] = self.np.copy(afPointTemp)
                iChange += 1
        return aafSimplex

    def haafNelderMead(self, aafSimplex,fObjectiveFunction):
        fAlpha = 1
        fGamma = 1
        fRho = 0.5
        fSigma = 0.5
        afCentroid = self.np.array([self.np.sum(aafSimplex[:-1, 0]) / 2.0, ...
            self.np.sum(aafSimplex[:-1, 1]) / 2.0, 0.0])
        afCentroid = self.afUpdateFuncValSingle(afCentroid,fObjectiveFunction)
        afReflection = afCentroid + fAlpha * (afCentroid - aafSimplex[-1, :])
        afReflection = ...
            self.afUpdateFuncValSingle(afReflection,fObjectiveFunction)
        if afReflection[2] < aafSimplex[1, 2]:
            if afReflection[2] > aafSimplex[0, 2]:
                aafSimplex[2, :] = self.np.copy(afReflection)
            else:
                afExpansion = afReflection + fGamma * (afReflection - ...
                    afCentroid)
                afExpansion = ...
                    self.afUpdateFuncValSingle(afExpansion,fObjectiveFunction)
                if afExpansion[2] < afReflection[2]:
                    aafSimplex[2, :] = self.np.copy(afExpansion)
                else:
                    aafSimplex[2, :] = self.np.copy(afReflection)
        else:
            if afReflection[2] < aafSimplex[2, 2]:
                afContraction = afCentroid + fRho * (afReflection - afCentroid)
                afContraction = ...
                    self.afUpdateFuncValSingle(afContraction,fObjectiveFunction)
            else:
                afContraction = afCentroid + fRho * (aafSimplex[2, :] - ...
                    afCentroid)
                afContraction = ...
                    self.afUpdateFuncValSingle(afContraction,fObjectiveFunction)
            if (afReflection[2] < aafSimplex[2, 2] and afContraction[2] < ...
                afReflection[2]) or (afReflection[2] ≥ aafSimplex[2, 2] and ...
                afContraction[2] < aafSimplex[2, 2]):
                aafSimplex[2, :] = self.np.copy(afContraction)
            else:
                aafSimplex[:, :] = aafSimplex[0, :] + fSigma * ...
                    (aafSimplex[:, :] - aafSimplex[0, :])
        aafSimplex = self.aafUpdateFuncVal(aafSimplex,fObjectiveFunction)
        return aafSimplex

    def runNelderMead(self,fObjectiveFunction,aafSimplex):
        aafSimplex = self.aafUpdateFuncVal(aafSimplex,fObjectiveFunction)
        aafSimplex = self.aafSortPoints(aafSimplex)
        fEps = 1E-3
        iMaxIt = 200
        fMaxError = self.np.sqrt((aafSimplex[0, 2] - aafSimplex[2, 2]) ** 2)
        fError = fMaxError
        ii = 0
        while ii < iMaxIt and fError > fEps:
            ii += 1
            aafSimplex = self.haafNelderMead(aafSimplex,fObjectiveFunction)
            self.aafSortPoints(aafSimplex)
            fError = self.np.sqrt((aafSimplex[0, 2] - aafSimplex[2, 2]) ** 2)
            #print('Iteration ', ii, ' Error: ', fError, ' Objective of ...
                best point: ', aafSimplex[0, 2])
        print('Nelder-Mead: Iteration ', ii, ' Error: ', fError, ' ...
            Objective of best point: ', aafSimplex[0, 2])
        print('Best point: ', aafSimplex[0, :])

    def runCO2NelderMead(self):
```

We then run the following code

```
    def runCO2NelderMead(self):
        aafSimplex = self.np.zeros((3, 3))
        aafSimplex[0, :-1] = [0.0, 0.0]
        aafSimplex[1, :-1] = [20.0, 0.0]
        aafSimplex[2, :-1] = [0.0, 20.0]
        self.runNelderMead(self.fDistVol,aafSimplex)
```

This gives a result very similar to the previous one, with the optimal position $P = [19.801, 41.13]$. *Solution found after 22 iterations, which is significantly faster than the 74 iterations for the patter search algorithm.*

Problem 2

Now, use the Nelder-Mead algorithm to find the optimal location for an additional wind turbine. The objective function is the same as in the previous exercise.
Solution:
The problem can be solve in python by the following script, in combination with the implementation of the Nelder-Mead algorithm above:

```
    def runWindNelderMead(self):
        aafSimplex = self.np.zeros((3, 3))
        aafSimplex[0, :-1] = [15.0, 15.0]
        aafSimplex[1, :-1] = [15.0, 10.0]
        aafSimplex[2, :-1] = [10.0, 15.0]
        self.runNelderMead(self.fWindFunc,aafSimplex)
```

Running this code gives the location $(10.29, 17.89)$, *with a maximum value of* 8.802. *The solution was found after 12 iterations, which is equal to the 12 iterations needed by the pattern search algorithm. However, we see that the Nelder-Mead algorithm gives a slightly worse solution, as it ends up in another local optimum than the pattern search algorithm. This shows that both algorithms can get stuck in local optima, and that it is a good idea to run the algorithms several times with different starting locations to find the global optimum.*