Computer Methods in Engineerings
Exercise on gradient descent

Problem 1
Consider the one dimensional function

$$f(x) = 2x^3 + 2x^2 - 4x \quad \text{for } x \in [-1, 1]. \tag{1}$$

a) Find the roots of $f(x)$ and check your result using the numpy `roots()` function. *Hint:* The `roots()` function take the polynomial coefficients as a vector input, e.g. for $h(x) = 2x^2 - 1$ you would write: `np.roots([2.0,0.0,-1])`.

Solution:

```
''' Solution to problem 1 in exercise on gradient descent
'''

import numpy as np
import matplotlib.pyplot as plt
from math import *

#Function to find root
def f(x) :
  rval = 2*x**3+2*x**2-4*x
  return rval

#Find root :
coeff = np.array([2.0,2.0,-4.0,0.0])
r = np.roots(coeff)
print(r)

dx=0.01
xv = np.arange(-3,2.0,dx)
ff = f(xv)
plt.plot(xv,ff)
plt.xlim(-3.0,2.0)
plt.ylim(-3.0,5.0)
plt.plot(r[0],f(r[0]),'ro',label='first root' )
plt.plot(r[1],f(r[1]),'bo',label='second root')
plt.plot(r[2],f(r[2]),'go',label='third root')
plt.plot([-3,2], [0,0])
plt.legend()
plt.xlabel('x')
plt.ylabel('f(x)')
plt.savefig('root.pdf')
```

```
plt.show()



def gradf(x):
  return 6*x**2+4*x-4

#Error tolerance
eps = 1e-4

#Maximum number of iterations
maxIt = 100

#Initial point
xx=0.75
itr=0
diff=2*eps

while(itr<(maxIt-1) and diff>eps):
    xxnew=xx-f(xx)/gradf(xx)
    diff=np.abs(xxnew-xx)
    xx=xxnew
    print(itr,xx)
    itr+=1

print('Root: ',xx, '\n Number of iterations: ',itr)

def ddf(x):
  return 12*x+4

#Initial point
xx=-0.25
itr=0
diff=2*eps

while(itr<(maxIt-1) and diff>eps):
    xxnew=xx-gradf(xx)/ddf(xx)
    diff=np.abs(xxnew-xx)
    xx=xxnew
    print(itr,xx)
    itr+=1

print('Extremum: ',xx, '\n Number of iterations: ',itr)


plt.plot(xv,ff)
plt.xlim(-3.0,2.0)
plt.ylim(-3.0,5.0)
plt.plot(xx,f(xx),'ro',label='extremum' )
plt.plot([-3,2], [0,0])
plt.legend()
plt.xlabel('x')
plt.ylabel('f(x)')
plt.savefig('extremum.pdf')
plt.show()
```

b) Plot $f(x)$ in python using `plot()`. Use an appropriate sampling interval Δx on the

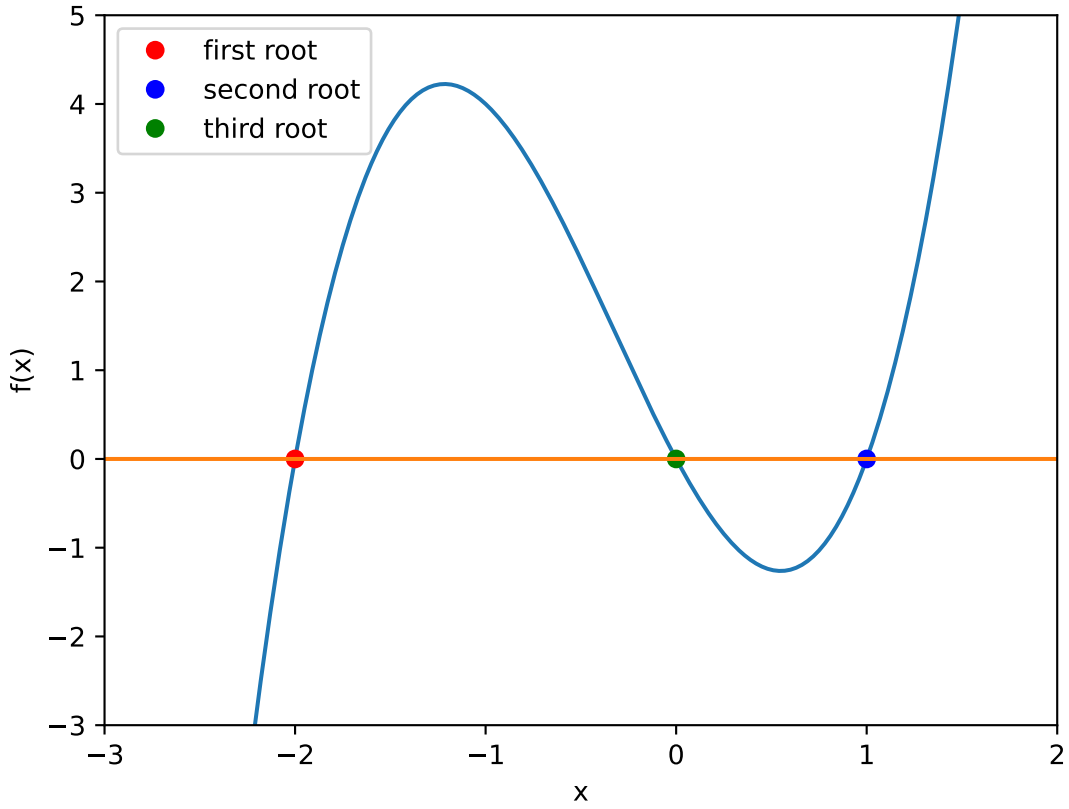

Figure 1: Roots of the function $f(x) = 2x(x - 1.0)(x + 1.0)$

.

x-axis. Give your plot a title and label the axes. Indicate the roots with a red dot and blue dot. Plot also the line $y = 0$

Solution:
See figure 1.

c) Use Netwon's method to find a root of the function f. Suggested starting-points are 0.5 and 0.75. As there are several roots, explain why you get the root you get when using the Netwon method.

Solution:
See the code in the solution of problem 1a.

d) Use Netwon's method to find a maximum or minimum value of the function f. Suggested starting-points are somewhere in the range $[-0.5, 0.0]$. Explain why you get the solution you get when using the Netwon method and your chosen starting-point.

Solution:
See the code in the solution of problem 1a. See figure 2.

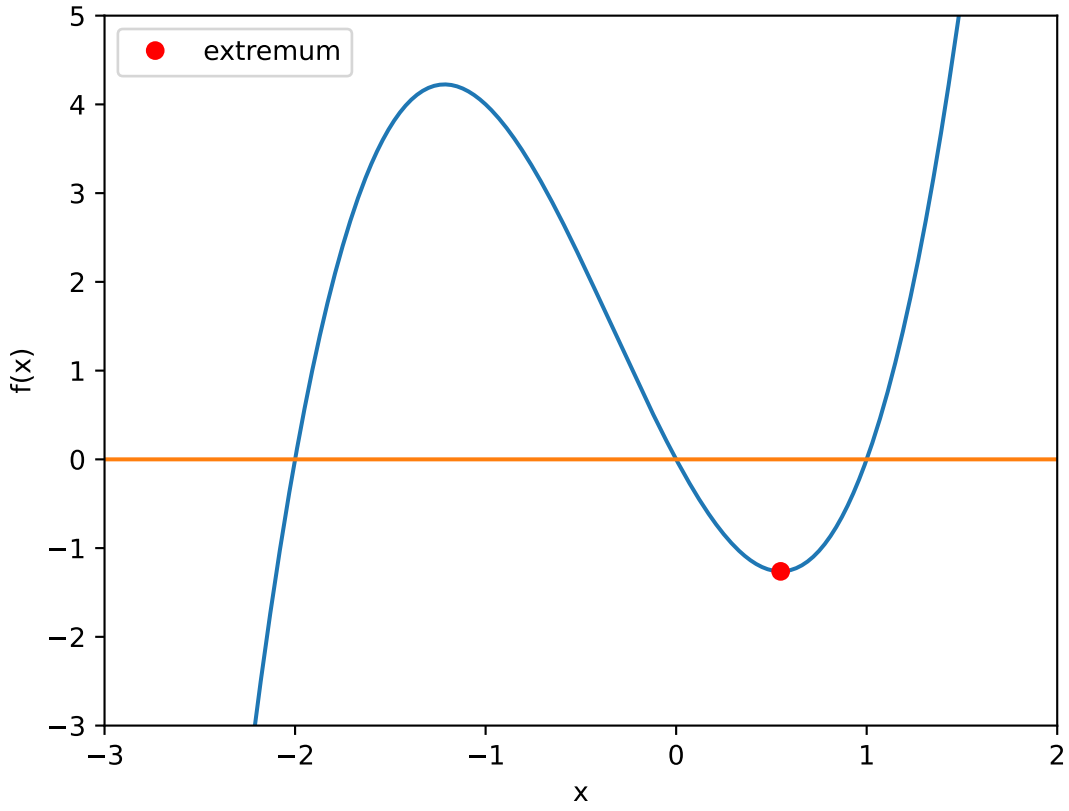

Figure 2: Extremum of the function $f(x)$

.

Problem 2

We will now continue with the exercise from the bracket search, where we assumed we were going to build a pipeline to transport CO2 from a capture site to an offshore storage site. A code for calculating the profit from the pipeline as a function of the radius of the pipe was given in the previous exercise.

a) Create a code to find the gradient of the profit with respect to radius.

Solution:
We can calculate the derivatives as follows:

```
    def derCostPipe(self,radius):
        return 2*m.pi*radius*self.length*self.materialPrice

    def derCostUse(self,radius):
        derDeltaP=-4*8*self.viscosity* ...
            self.length*self.flowRate/(m.pi*radius**5)
        return derDeltaP*self.pricePerPascalperCubed

    def derProfit(self,radius,time):
        return -self.derCostUse(radius)* self.flowRate*time - ...
            self.derCostPipe(radius)
```

b) Use the derivate to make a more efficent bracket search. Compare the number of iterations with the barcket search conducted in the previous exercise.

Solution:
We can replace the test values α and β by the derivative, as shown in the following code:

```
    def bracketDer(self,time):
        from random import random
        #Maximum number of iterations
        self.iN = 100

        #Error tolerance
        self.feps = 1e-3

        #Storing values during run for later plotting
        self.afA=np.zeros(self.iN)
        self.afB=np.zeros(self.iN)

        #Starting values for our interval [a,b] and internal points ...
            alpha and beta
        self.afA[0] = 0.1
        self.afB[0] = 0.4

        # This is an iterative algorithm, and we iterate over ii
        ii=0
        while(ii<self.iN and (self.afB[ii]-self.afA[ii])>self.feps):
            fTestRadius=(self.afA[ii]+self.afB[ii])/2
            # If derivative is positive change b value
            if self.derProfit(fTestRadius,time)>0:
                self.afA[ii+1]=fTestRadius
                self.afB[ii+1]=self.afB[ii]
            else:
```

```
                self.afB[ii+1]=fTestRadius
                self.afA[ii+1]=self.afA[ii]
            ii+=1
        #print('Number of object function evaluations: ',2*ii)
        return (self.afA[ii]+self.afB[ii])/2, ...
            self.profit((self.afA[ii]+self.afB[ii])/2,time), ii
```

When running this code, the optimum was found after 9 iterations. A couple of tests with the old bracket search code resulted in between 14 and 21 iterations.

c) Create a code where you do a gradient descent to find the optimum.

Solution:
We can add a code similar to this to our class:

```
    def gradientDescent(self, time, initRadius=0.3):
        fRadius = initRadius
        fOldRadius = initRadius*1.2  # Ensure the loop starts
        fStepSize=1e-9
        self.iN=100
        self.eps=1e-3
        ii=0
        while ii < self.iN and abs(fRadius-fOldRadius)>self.eps:
            grad = self.derProfit(fRadius, time)
            fOldRadius = fRadius
            fRadius = fRadius + fStepSize * grad
            ii+=1
        return fRadius, self.profit(fRadius, time), ii+1
```

When running this code, the optimum was found after 11 iterations, a similar number to the bracket method. However, this code is very dependent on the step-size used.