Computational Labs:

17/02/2022

Kevin Grainger

20333346

Lab 1: Finding Minima of Functions

1) INTRODUCTION

The aim of this lab was to implement and investigate numerical methods (Bi-section method & The Newton Raphson Method) used to find points of interest on given functions (namely; roots, maxima and minima). We aimed to graphically illustrate the mechanisms involved in each method, investigate the accuracy and efficiency of each method and the relationship between these values, and finally to apply the Newton Raphson Method to a Potential Energy Function. The lab consisted of three exercises:

(1.1) Exercise 1: Finding the roots of a parabolic function (f(x)) using the bi-section method

- Principal: This method relies on the choosing of two points along a function, one below the x-axis and one above (so that the root of the function can be said to lie in this interval). This interval is halved along the x-axis (.ie (x_1-x_3)/2). This new midpoint is called x_2, if f(x_2) is positive we know the root is between x_1 & x_2 , but between x_2 & x_3 if f(x_2) is negative. With many iterations of this process the root can be found approximately.
- In order to investigate the accuracy and efficiency of this method, the tolerance was graphed against the number of steps (iterations) taken to reach the tolerance. It was found that the variables were inversely proportional.

(1.2) Exercise 2: Finding the Roots of Functions using the Newton Raphson Method.

- Principal: This method is based on a simplified version of the taylor expansion.
  $f(x + h) = 0$
  $f(x + h) = f(a) + hf'(a)$+......(1)
  $a + h = a - f(a)/f'(a) \quad (2) \quad -> when\, Eq\,(1) = 0$
- Using Equation 2 a iterative function that finds for what value x does f(x)=0. Ie. The root.
- This method was used to find both roots of the same parabolic function as in Exercise 1 and the mechanism was highlighted with points marking each iteration.

(1.3) Exercise 3:

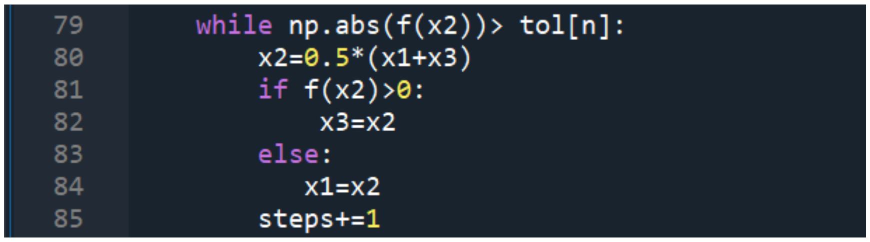
- This Exercise was an application of the methods outlined in Exercise 2.
- I used the Newton Raphson method to find the minima of a force field between the Sodium and Chlorine atoms in a salt molecule.

- The interaction potential was graphed along with its derivative, which corresponds to the Force.
- An error in my program led to a slightly inaccurate value for this exercise. I believe the problem is in the while loop used but I was unable to rectify this. Yet the value I reached was close to the Potential energy minima, but the right techniques were used.

2) Methodology

2.1 Exercise 1 (Bisection Method)

1) Firstly I defined my function:
   $y = x^2 + 3x - 5$
   Using np.arange I created the range in which I wanted my function to be plotted over. I defined three variables; x_1, x_2, and x_3 . These will serve as detailed in (Sec 1.1). The tolerance (ie. How far the numerical value must be from the true value) was then set to 0.0001.
2) Next the function that will perform the bisections is defined. The program calculates the point x_2 and continually reiterates this calculation until the point x_2 is within the tolerance from 0 (0.0001). Each bisection adds one to the value steps (the number of steps taken to yield a value).

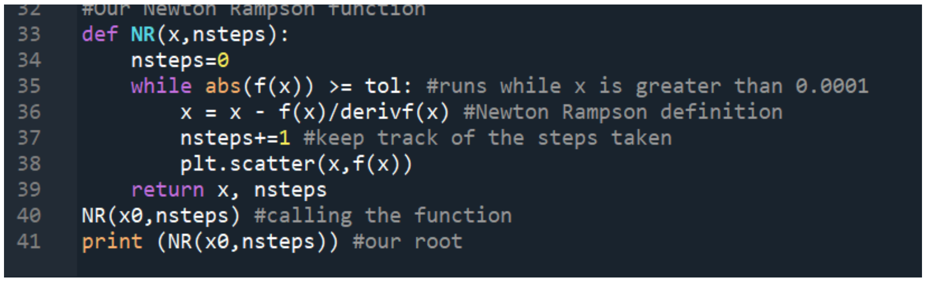


```
    while np.abs(f(x2))> tol[n]:
        x2=0.5*(x1+x3)
        if f(x2)>0:
            x3=x2
        else:
           x1=x2
        steps+=1
```

3) We print the approximate root and the number of steps taken to find it.
4) By switching our starting x_3 so that it is mirrored on the other side of the parabola I found the second root of the function.
5) After letting this function run, we want to graphically represent the process as stated in the aims. I did this by first graphing the function f(x) and the line x=0. Then every bisection was programmed to leave a dot on the graph.
6) For the final piece of analysis a second while loop is added which runs the above function but with varying tolerances. This function produces varying numbers of steps for given tolerances, from this I graphed log 10 of the tolerance against the value ‘steps’. This gives us a graphical representation of how the number of steps needed changes as with the required accuracy of the answer.

2.2 Exercise 2- Newton Raphson Method:

1) Firstly the function from Exercise 1 is carried over and it’s the derivative is calculated analytically. These form functions.
2) Using the Equations 1 & 2 outlined in the intro I formed an iterative function to constantly update the value x.

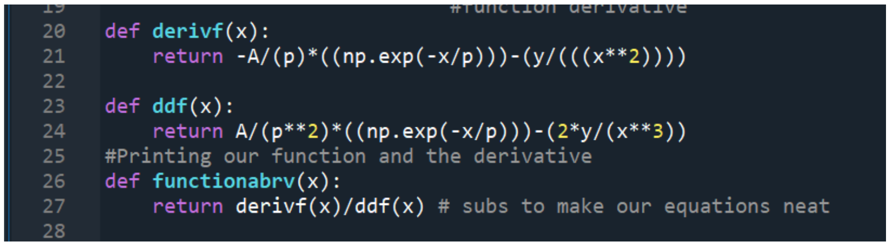


```
#Our Newton Rampson function
def NR(x,nsteps):
    nsteps=0
    while abs(f(x)) >= tol: #runs while x is greater than 0.0001
        x = x - f(x)/derivf(x) #Newton Rampson definition
        nsteps+=1 #keep track of the steps taken
        plt.scatter(x,f(x))
    return x, nsteps
NR(x0,nsteps) #calling the function
print (NR(x0,nsteps)) #our root
```

3) From this we can identify the root of the function and print it as above.

2.3 Finding the Minima Roots of a Potential Energy Function

1) For this exercise the first and second derivatives of the potential energy function needed to be calculated analytically .
   Said PE function:
   $V(x) = Ae^{-x/p} - e^{2}/4\pi\varepsilon x$
2) We define the Potential energy function and its derivatives as python functions. As such:

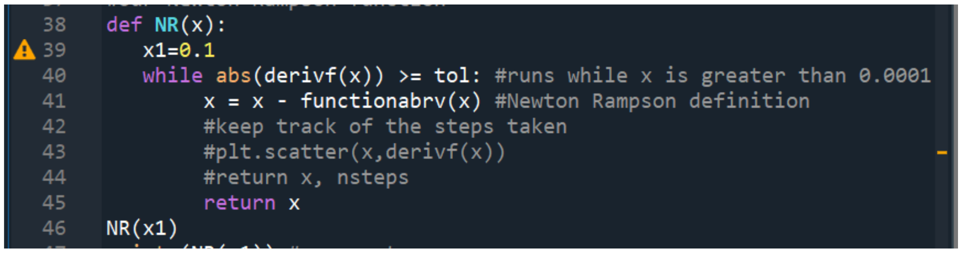


```
                 #function derivative
def derivf(x):
    return -A/(p)*((np.exp(-x/p)))-(y/(((x**2))))

def ddf(x):
    return A/(p**2)*((np.exp(-x/p)))-(2*y/(x**3))
#Printing our function and the derivative
def functionabrv(x):
    return derivf(x)/ddf(x) # subs to make our equations neat

```

3) Taking the same iterative Newton Raphson function used in exercise 2 we simply replace the parabolic function with the new defined in the first step.
4) The aim of the exercise is to find where the force produced by the atoms is least. Given this we are looking for the minimum of the First derivative, (ie. the Force)
   $dV(x)/dx$ = Force
5) The Newton Raphson function is arranged as such



```
def NR(x):
   x1=0.1
   while abs(derivf(x)) >= tol: #runs while x is greater than 0.0001
        x = x - functionabrv(x) #Newton Rampson definition
        #keep track of the steps taken
        #plt.scatter(x,derivf(x))
        #return x, nsteps
        return x
NR(x1)
```

6) The x1 value is our desired root. This represents the position in the field under least force.
7) This point is plugged into the Potential Energy function to find the corresponding energy of that position.

3) Results

<u>3.1) Exercise 1 (Bisection Method)</u>

i) Graph of F(x), F(0), x=0, & x_2 (bisections)

From this we can see every iteration of the value x_2 , the graph visualizes the while loop as it approaches the function root.

We can see the dots progressively approaching the function root.

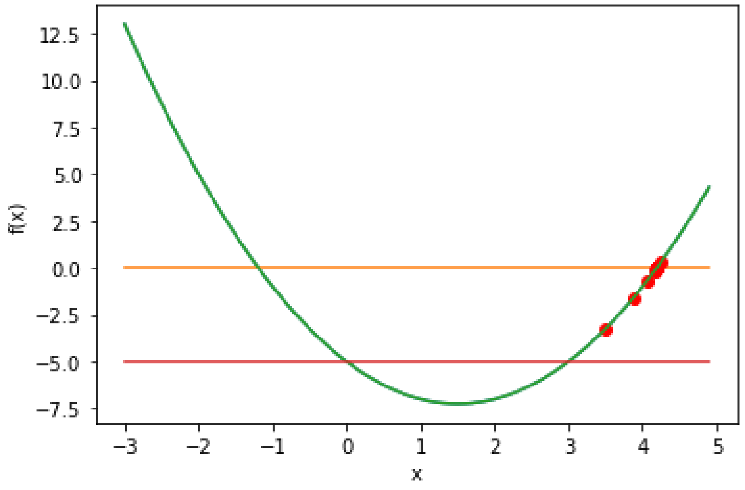
- F(X) is Green
- F(x)=0 is Orange
- F(0) is in Red

<u>Root 1:</u>

Starting Coordinates:

X_1 = 2

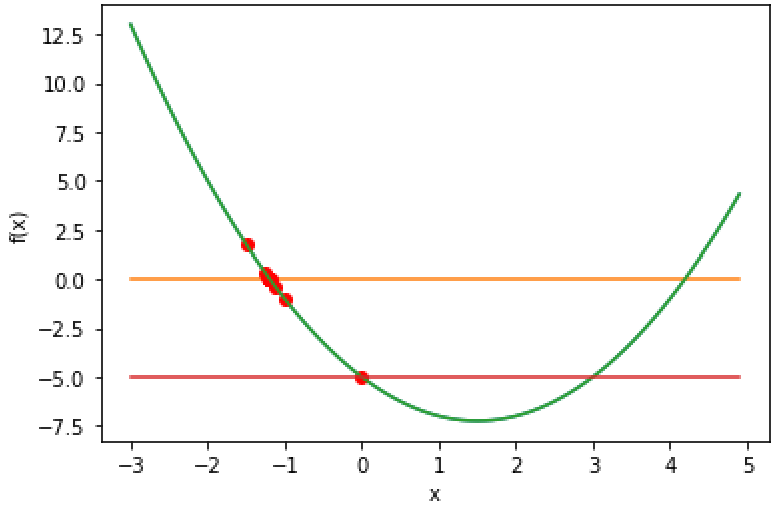
X_3 = 5



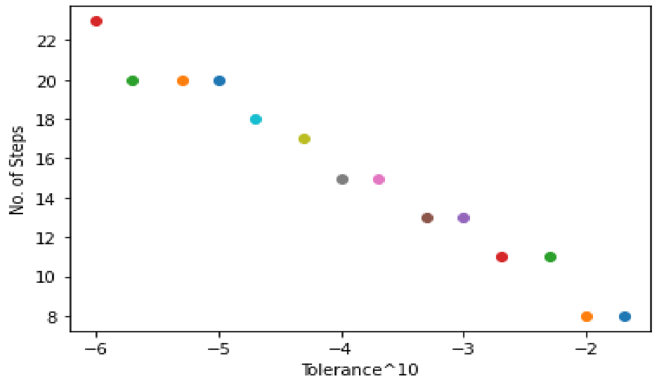
<u>F(x) :</u> 7.556471973657608e-05 ≈ 0

Root : *x=* ***4.192596435546875***
Steps Taken : ***15***
Analytical Value : 4.1925824035673

Root 2 :
$X_1 = 2$
$X_3 = -2$



F(X): *-8.877739310264587e-05 ≈ 0*
Root : *x=* ***-1.19256591796875***
Steps taken: ***16***
Analytical Solution: -1.1925824035673
Number of Steps vs. Tolerance
We see an inversely proportional relationship between the tolerance value and the number of steps the program takes. The result is a sort of step function.



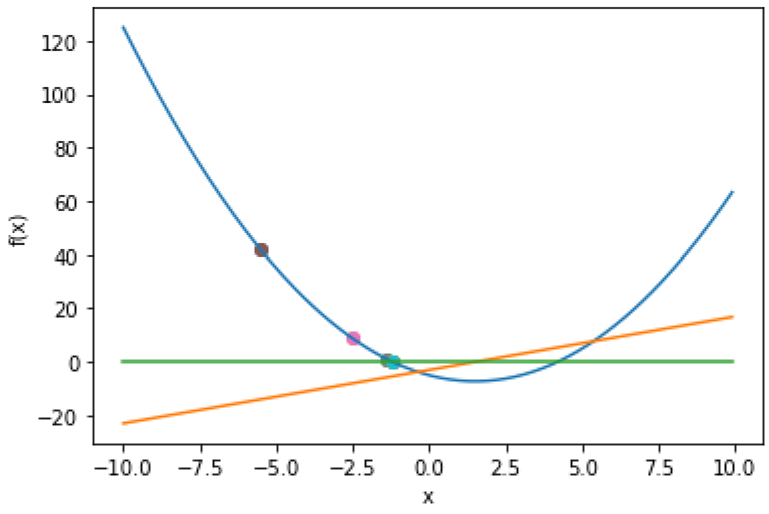
<u>3.2 Exercise 2 (Newton Raphson)</u>

Here we have F(x) in blue.

F(x)=0 is Green.

F'(x) is Orange

<u>First Root:</u>

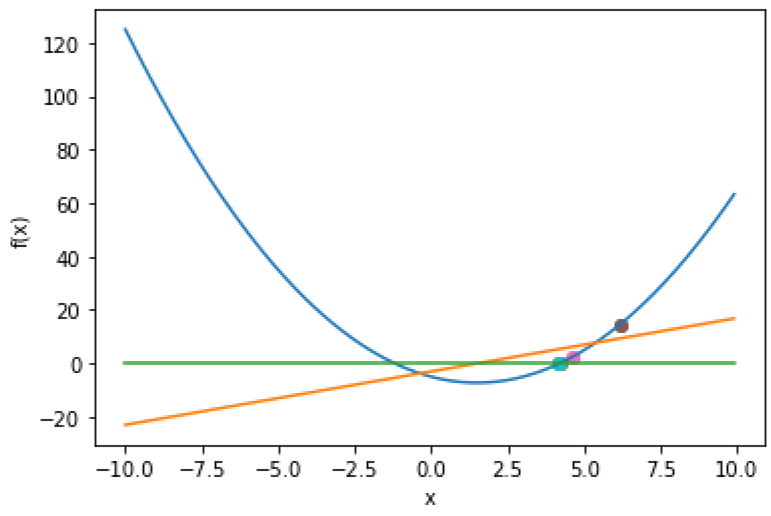


Root: -1.1925953510752247

Steps: 5

Analytical Solution: -1.1925824035673

<u>Second Root:</u>



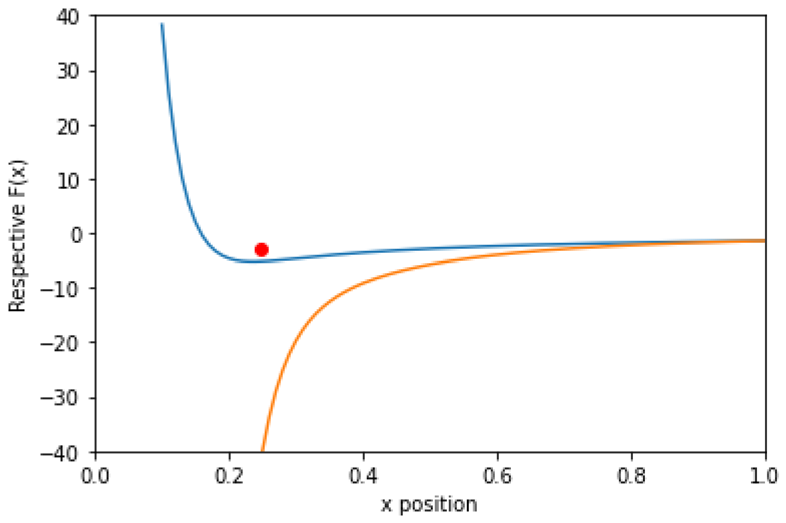
Root: 4.19258240768227
Steps: 5
Analytical Value : 4.1925824035673

We see the Newton Raphson Method is three times as efficient as the bisection method.

(3.3) Exercise 3: Minima of PE Function

Here we have:

- V(x) in Orange (Potential Energy)
- V'(x) in Orange
- The dot marks the potential Energy at the minimum of V'(x)



As you can see the dot doesn't sit on the V(x) line, meaning there is an error in the value. But the dot does sit close to the minimum potential energy.

Conclusions:

Exercise 1:

The bisection method functioned well. I was able to accurately find both roots of the parabolic function. This method would not work well for functions of higher indexes , especially given it's inefficiency .
I found that this method was only 33% as efficient as the Newton raphson method.
The relationship between the tolerance and the number of steps taken by the program was found to be linearly inversely proportional. The graph resembled a step function

Exercise 2

The Newton Raphson Method turned out to be a much easier and more efficient method of finding the minima of functions.
It was 3 times as efficient as the previous method.
I successfully found both roots of the function with the same degree of accuracy as with the Bisection Method.

Exercise 3

This exercise was an application of the methods investigated in exercise 2.
The potential energy and force functions were graphed successfully.
Using the Newton Raphson Method a value for the minima of V'(x) was found. Although by looking at the graph we see the value was not entirely accurate. This is due to some coding error. Regardless, the program ran and a value close to the correct value was found.

Code:

Ex 1

```
"""

import numpy as np
import matplotlib.pylab as plt
x=np.arange(-3,5,0.1)

def f(x):
    return x**2-x*3-5

print (f(x))
plt.plot(x,f(x))
plt.plot(x,0.0*x+0.0*f(0.0))
plt.xlabel("x")
plt.ylabel("f(x)")
x1=2
x3=-2
x2=0


tol=0.0001
nsteps=0
while abs(f(x2))>tol:
   x2=0.5*(x1+x3)
   if f(x2)>0:
        x3=x2
   else :
        x1=x2
   nsteps+=1
   plt.plot(x2,f(x2),'ro')
#create another while loop and add an else if so that
#it cannot produce the same root as before
#plt.show()

plt.plot(x,f(x))
plt.plot(x,0.0*x+f(0.0))
plt.show()
print (f(x2))
print (x2)
print (nsteps)

'''

'
def nsteps(T):
    steps=0
    x4=1
    x5=1
    x6=5
    while abs(f(x5))>T:
        x5=0.5*(x4+x6)
        if f(x5)>0:
            x6=x5
        else:
           x4=x5
        steps+=1
    return steps



'''

'''
tol = [0.02, 0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002, 0.0001, 0.00005, 0.00002, 0.00001,0.000005, 0.000002, 0.000001]
n = 0
steps = 0
#By nesting while loops we can get step values for each tolerance value in the array
x2=0.5*(x1+x3)
while n <14:
    while np.abs(f(x2))> tol[n]:
        x2=0.5*(x1+x3)
        if f(x2)>0:
            x3=x2
        else:
           x1=x2
        steps+=1
    plt.scatter(np.log10(tol[n]),steps)
    n += 1
plt.xlabel("Tolerance^10")
plt.ylabel("No. of Steps")
plt.show()

'''
```

Ex 2

```
import numpy as np
import matplotlib.pylab as plt
x=np.arange(-10,10,0.1)
#our x range
#our function
def f(x):
    return x**2-x*3-5
#function derivative
def derivf(x):
    return 2*x-3
#Printing our function and the derivative
print (f(x))
plt.plot(x,f(x))
plt.plot(x,derivf(x))
plt.plot(x,0*f(x))
plt.xlabel('x')
plt.ylabel('f(x)')
nsteps=0
tol=0.0001 #Tolerance
x0=10 # This chose a number far from the root so that the graph clearly
#shows the mechanism
#Our Newton Rampson function
def NR(x,nsteps):
    nsteps=0
    while abs(f(x)) >= tol: #runs while x is greater than 0.0001
        x = x - f(x)/derivf(x) #Newton Rampson definition
        nsteps+=1 #keep track of the steps taken
        plt.scatter(x,f(x))
    return x, nsteps
NR(x0,nsteps) #calling the function
print (NR(x0,nsteps)) #our root
plt.show()


tol = [0.02, 0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002, 0.0001, 0.00005, 0.00002, 0.00001,0.000005, 0.000002, 0.000001]
n = 0
steps = 0

def nsteps(x,nsteps):
    n = 0
    while n <14:
        while np.abs(f(x))> tol[n]:
            x = x - f(x)/derivf(x)
            nsteps+=1 #keep track of the steps taken
            plt.scatter(x,f(x))
            return x, nsteps
        plt.scatter(np.log10(tol[n]),steps)
        n += 1
nsteps(x,nsteps)
plt.xlabel("Tolerance^10")
plt.ylabel("No. of Steps")
plt.show()
```

Ex 3

```
import numpy as np
import matplotlib.pylab as plt
x=np.arange(0.1,5,0.01)
#our x range
#our function
A=1090
p=0.033
y=1.44
def f(x):
    return A*(np.exp(-x/p))-(y/(x))

                                    #function derivative
def derivf(x):
    return -A/(p)*((np.exp(-x/p)))-(y/(((x**2))))

def ddf(x):
    return A/(p**2)*((np.exp(-x/p)))-(2*y/(x**3))
#Printing our function and the derivative
def functionabrv(x):
    return derivf(x)/ddf(x) # subs to make our equations neat

print (f(x))
plt.plot(x,f(x))
plt.plot(x,derivf(x))
plt.ylim(-40,40)
plt.xlim(0,1)
tol=0.0000000001 #Tolerance
x1=0.5 #random number in range
#shows the mechanism
#Our Newton Rampson function
def NR(x):
   x1=0.1
   while abs(derivf(x)) >= tol: #runs while x is greater than 0.0001
        x = x - functionabrv(x) #Newton Rampson definition
        #keep track of the steps taken
        #plt.scatter(x,derivf(x))
        #return x, nsteps
        return x
NR(x1)
print (NR(x1)) #our root
print (f(x1))
#plt.scatter(x1,f(x1),color='red')
plt.plot(NR(x1),f(x1),'ro')
plt.xlabel('x position')
plt.ylabel('Respective F(x)')
plt.show()
```