Laboratory 1
Finding minima of functions

Conor Kirby

October 9th, 2023

1 Abstract

In this lab we are using Python to investigate the minima of functions using two different methods, the bisection method and the Newton-Raphson method. Using our findings in multiple separate experiments we conclude that the Newton-Raphson method is much more efficient as the bisection method has a linear rate of convergence whereas the Newton-Raphson method has a quadratic rate of convergence.

Using these methods, we explore the concept of potential energy and its relationship to stable equilibrium. Initially we focus on the potential energy of a harmonic oscillator $V(x) = \frac{1}{2} k x^2$, attempting to identify the states of stable equilibrium, corresponding to the minima of V(x); when the force is equal to zero:

$$F(x) = -\frac{dV(x)}{dx}.$$

Finally, we apply what we have learned to investigate ionic interactions between two ions such as a sodium cation and a chlorine anion using the potential energy formula:

$$V(x) = Ae^{-\frac{x}{p}} - \frac{e^2}{4\pi\varepsilon_0 x}$$

Where p = 0.033nm, $\frac{e^2}{4\pi\varepsilon_0} = 1.44$ eV nm, A= 1090 eV are the numerical values needed. We unsuccessfully attempted to find the stable equilibrium point of the system, ie. when V'(x) = 0 meaning F should equal zero. However, it was found that the force is equal to 218.6843 N. This may be due to error in the python code however we still learned the significance of the stable equilibrium point for understanding a complex system. Our x value for the minimum of the potential, and hence the

equilibrium separation distance for the ions is 0.1578 nm which provides valuable insight for understanding this ionic interaction.

2 Introduction

In classical mechanics, the potential energy V(x), plays a crucial role in understanding the physical properties of a system. It varies with position relative to many parts of the system. A widely known example of this is the potential energy of a harmonic oscillator V(x) = ½ k x2.

The force acting on a particle along the x-axis is the negative gradient of the potential energy function; $F(x) = -\frac{dV(x)}{dx}$. For the harmonic oscillator potential, $F(x) = -kx$, is described by Hooke's law.

In this experiment we investigate the stable and unstable equilibrium points of potential energy functions. At extrema of the potential energy function the force acting on a particle is zero. The equilibrium point is unstable if the system is displaced an arbitrarily small distance from the equilibrium state, the forces of the system cause it to move even farther away. The equilibrium point is stable if the system is displaced an arbitrarily small distance from the equilibrium state, the forces of the system cause it to move back to the original equilibrium position. Points of inflection also represent points of unstable equilibrium. Identifying the stable equilibrium points is key in understanding the behaviour of a particle or a system of particles. In our case when the slope of V(x) is zero, and it's second derivative is positive.

In this investigation we use the bisection method and the Newton-Raphson method to find roots of functions. The bisection method involves narrowing down an interval around a root until the root is found within some tolerance. The Newton-Raphson method uses the Taylor series expansion to iteratively find improved estimates for the roots.

3 Methodology

This lab tests our numerical analysis skills and our ability to understand complex systems. Lucky for us, python is fantastic for understanding and visualising data. In this case we will be using Spyder in Anaconda as the programming environment and using numpy and matplotlib.pyplot [3] libraries for mathematical expressions and plotting.

Below is an example of python scripting. In this case we are plotting a parabolic curve and a line.

```python
#importing libraries for later use
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-5.0, 5.0, 0.2)
a = 1
b = 1
c = -6

#plotting f(x) and our line
plt.plot(x, a * x * x + b * x + c)
plt.plot(x, 0.0 * x)
plt.show()
```

Fig 3.1: Python code demonstration for plotting a function [1]

Using Functions in Python

Functions are an extremely useful tool in python and will aid us massively in this experiment. A lot of functions are predefined, for example the sin(x), cos(x), and tan(x) functions, however we must define our own functions to help solve a particular problem. All python functions start with the command ***"def"*** followed by the function name, an argument in brackets, and a colon.

```python
def square(n):
    """Returns the square of a number."""
    squared = n ** 2
    print "%d squared is %d." % (n, squared)
    return squared
```

Fig 3.2: Defining a function which returns the square of a number [2]

3.1 Finding the Roots of a Parabolic Function using the Bisection Method

We start off by defining a parabolic function with two real roots and plotting it with the line f(x) = 0. We then define the variables x_1, x_2, and x_3 to values such that $f(x_1) < 0$ and $f(x_2) > 0$. x_2 will be the midpoint of x_1 and x_3 and we keep redefining the variables in a 'for loop' until x_2 is within a certain tolerance of the root.

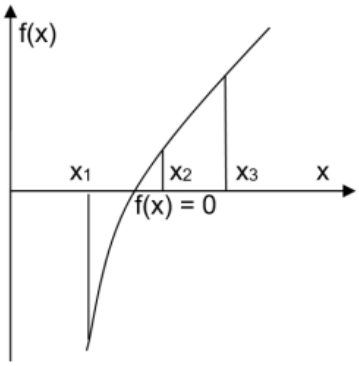

Fig 3.3: The Bisection Method [1]

An aim of this part of the experiment is to test the convergence of the bisection method. We must create a variable called nsteps which increments every time the for loop runs through until the root is found. We vary the tolerance and plot the number of steps vs the –log of the tolerance to visualise the data.

3.2 Finding the Roots of a Parabolic Function using the Newton-Raphson Method

The Newton-Raphson method relies on a Taylor Series Expansion about a specific point. This method updates its initial guess using the derivative of the according to the following equation:

$$x_1 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

Where x_1 is the current estimate of the root. We now, similarly to the last exercise, define another parabolic function with real roots and plot it using plt.plot(). However, this time instead of plotting f(x) = 0, we define another function which is the derivative of f(x) and plot it with the function.

We must initialise our guess x_1 to 1, as our current estimate of the root, and use a while loop to keep updating the estimate using the Newton-Raphson rule until $|f(x_1)|$ is within a certain tolerance.

```
tol_list = []
nsteps_list = []
exp_array = np.arange(-40.0, -2.0, 1)

for exp in exp_array:
    tol = 10**exp
    tol_list.append(tol)

for tol in tol_list:
    x1 = -2
    while abs(f(x1))>tol:
        x1 = x1 - f(x1)/f1(x1)
print(x1)
```

Fig 3.4: Using the Newton-Raphson Method to find the root

Similarly, to the last section we need to test the convergence behaviour by counting the number of iterations. We create a list to store the number of steps for each tolerance value and use the same method as the last section to count the number of iterations.

Plot the number of steps vs the –logarithm of the tolerances to visualise the data and make an analysis of the convergence behaviour.

3.3. Finding the Minima of the Potential Energy Function for an Ionic Interaction

The potential energy for our ionic interaction can be expressed as:

$$V(x) = Ae^{-\frac{x}{p}} - \frac{e^2}{4\pi\varepsilon_0 x}$$

The goal is to find the value of x for the minimum of this function which corresponds to the equilibrium separation of the ions.

Create a code to define the function above using np.exp() from the numpy library for the exponential term. Plot the function over a range of x values from 0.01 to 1.00.

Create a new function for the force, calculating $\frac{dV(x)}{dt}$ by hand. Plot this function on the same graph. Then calculate $\frac{d^2V(x)}{dx^2}$ and plot. Below should be the equations for V'(x) and V''(x) :

$$V'(x) = -\frac{Ae^{-\frac{x}{p}}}{p} + \frac{e^2}{4\pi\varepsilon_0 x^2} \qquad V''(x) = \frac{Ae^{-\frac{x}{p}}}{p^2} - \frac{2e^2}{4\pi\varepsilon_0 x^3}$$

Use the Newton-Raphson method to determine the minima of V(x) and it's corresponding value of x. Ensure to check your plot to verify that your value is reasonable.

4 Results

4.1 Exercise 1

We started off testing the efficiency of the bisection method vs the Newton-Raphson method in determining the roots of a function. The curve $x^2 + 4x + 3$ was used to test the bisection method. The bisection method successfully gave us our roots consistently to the desired accuracy.

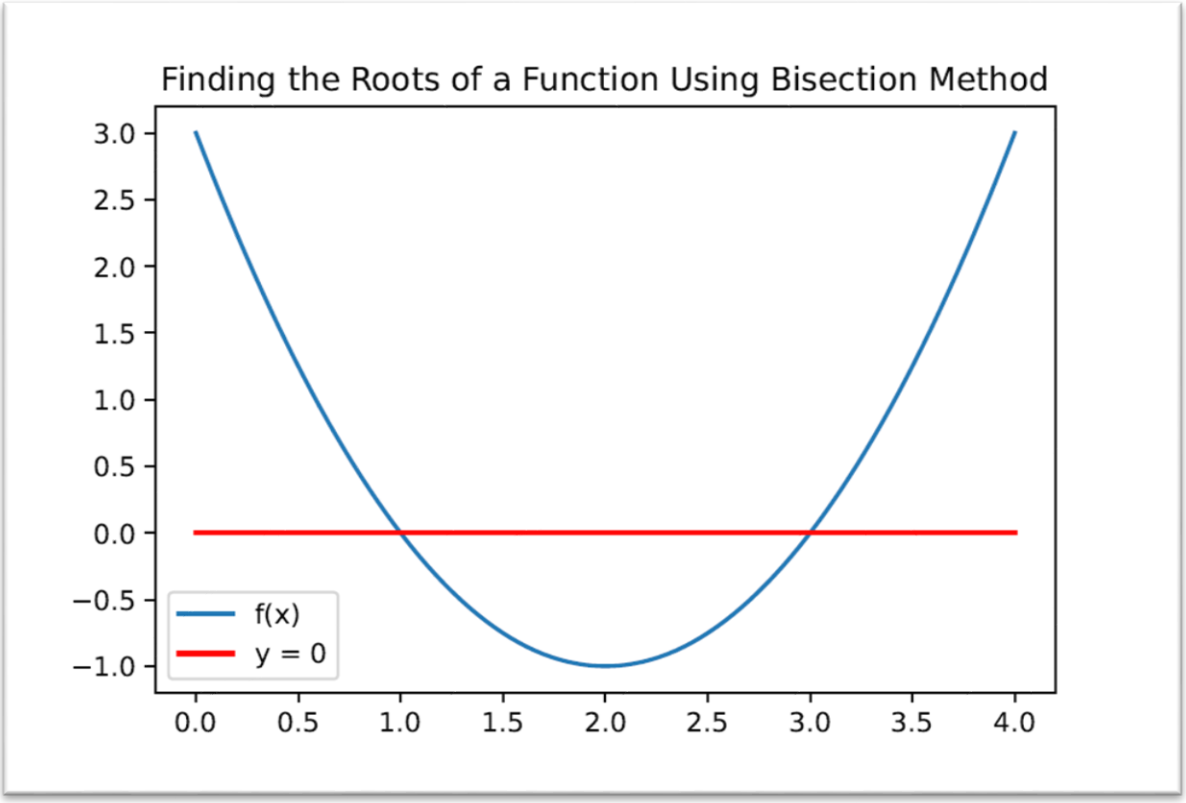

Fig 4.1: Finding the roots of a function using the bisection method

This is the table of data that was collected when investigating the convergence of the bisection method. Because of the way the bisection method functions, as the tolerance lowers the number of iterations needed increases linearly. This makes it slightly inefficient in comparison to the Newton-Raphson Method which I will talk about next. Here is the graph proving the linear relation between the tolerance and the number of iterations.

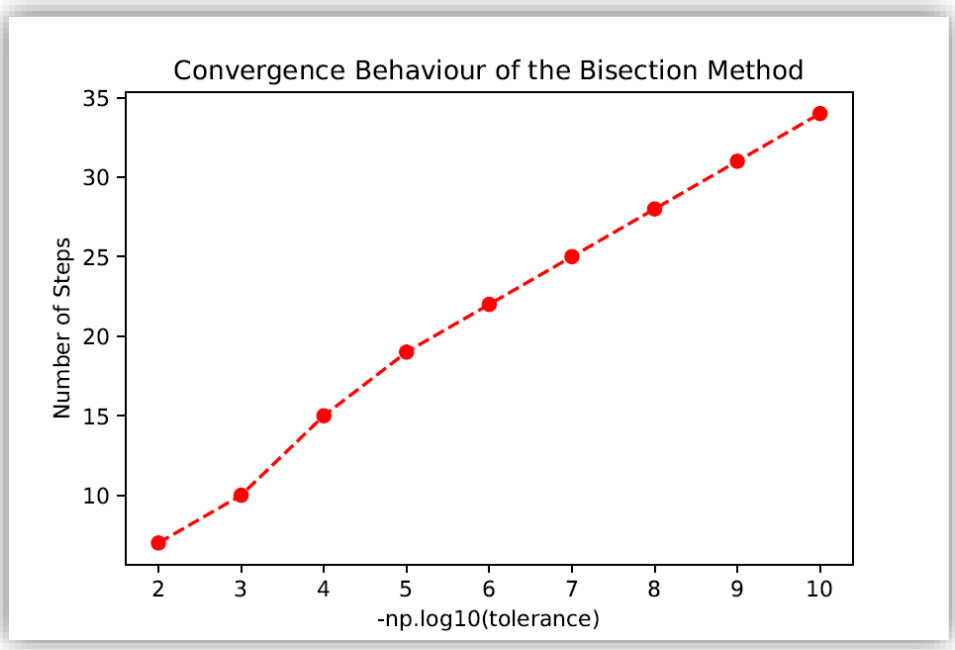

Fig 4.2: Graph to test the Convergence of the Bisection Method

4.2 Exercise 2

However, with further investigation we could conclude that the Newton-Raphson Method provided an accurate value for the root with a shorter rate of convergence, taking less steps than the bisection method because the initial guess is already reasonably close to the root, therefore requiring fewer iterations to find our root.

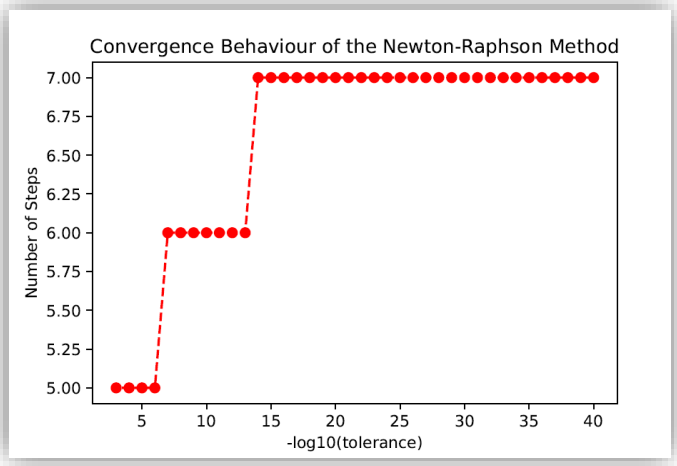

Fig 4.3: Testing the convergence behaviour of the Newton-Raphson method

Convergence of N-R method	
Tolerance	Numbers of iterations needed
10^{-2}	5
10^{-3}	5
...	...
10^{-5}	5
10^{-6}	6
...	...
10^{-12}	6
10^{-13}	7
...	...
10^{-40}	7

Convergence of bisection method	
Tolerance	Numbers of iterations needed
10^{-2}	7
10^{-3}	10
10^{-4}	15
...	...
10^{-10}	34

Fig 4.4: Comparing tables for the convergence behaviours of the Newton-Raphson and the bisection methods

It's clear from the data that the Newton-Raphson Method is much more efficient than the bisection method.

4.3 Exercise 3

Putting our newfound methods to use we attempted to find the equilibrium point of the given system. Unfortunately, due to a possible mistake in the code we could not extract our desired result. Using the Newton-Raphson Method we attempted to find the minimum of the potential energy, however when the root was determined we found that the force did not equal zero. Meaning that the point we found was not a minimum of the potential and therefore not the point of stable equilibrium. The reason for trying to find the stable equilibrium for this system is because it is a crucial parameter for understanding the chemical interaction. Our x value for the minimum of the potential, and hence the equilibrium separation distance for the ions is 0.1578 nm.

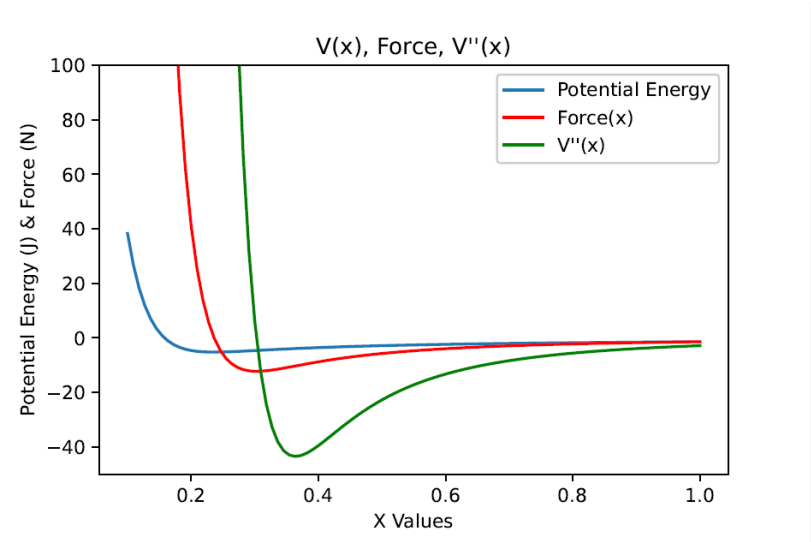

Fig 4.5: Graphing the potential function, force, and the second derivative of the potential

5 Conclusions

In this experiment, we explored the ideas of potential energy, stable and unstable equilibriums, and numerical methods for finding the roots and minima of functions. Starting off with the well-known example of the potential of a harmonic oscillator and then applying our learnings to an example of the ionic interactions between a sodium and a chlorine ion.

We used these numerical methods to investigate these ionic interactions, potentials and states of equilibrium. Through Spyder we used Python 3.0 to help calculate and visualise the results.

Using these methods throughout the experiment we gained valuable insight into the use of both the bisection and Newton-Raphson Method, and also the significance of stable equilibrium in understanding the inner workings of complex physical systems. Our final value for the equilibrium separation distance of the ions was 0.1578 nm which gives extremely useful information for understanding the ionic interaction.

6 Bibliography

[1]: Trinity College Dublin SF Computational Lab Manual

[2]: CodeAcademy - https://www.codecademy.com/learn

[3]: Matplotlib- https://matplotlib.org/

7 Appendix

Exercise 1:

```
"""
Created on Wed Sep 27 09:08:50 2023

@author: Conor Kirby
"""
import numpy as np
import matplotlib.pyplot as plt

#Excercise 1: Choosing Parabola

def f(x):
    a = 1
    b = -4
    c = 3
    return a*x*x + b*x + c

x = np.linspace(0, 4, 100)
```

```
plt.plot(x, f(x))
plt.plot(x, 0.0 * x, color = 'red')
plt.title("Finding the Roots of a Function Using Bisection Method")
plt.legend(["f(x)", "y = 0"])
#plt.savefig('C:/Users/Conor Kirby/OneDrive/Desktop/f.plot1.pdf')
plt.show()
#finding root 1
nsteps_list = np.array([])
tol_list = np.array([0.01, 0.001, 0.0001, 0.00001, 0.000001, 0.0000001, 0.00000001, 0.000000001,
0.0000000001])

for tol in (tol_list):
   print(tol)
   x1 = -1
   x3 = 2.5
   x2 = 0.5*(x1 + x3)
   print(nsteps_list)
    #number of steps
   nsteps = 0
   while abs(f(x2)) > tol: #tolerance of 0.0001
      x2 = 0.5*(x1 + x3)
      nsteps = nsteps + 1
      if f(x2) < 0:
         #plt.plot(x2, f(x2), '--o')
         x3 = x2
      elif f(x2) > 0:
         #plt.plot(x2, f(x2), '--o')
         x1 = x2
      else:
         print(str(x2) + " is the root.")
   print(nsteps)
   nsteps_list = np.append(nsteps_list, nsteps)
print(nsteps_list)
plt.plot(-np.log10(tol_list), nsteps_list, '--ro')
plt.ylabel("Number of Steps")
plt.xlabel("-np.log10(tolerance)")
plt.title('Convergence Behaviour of the Bisection Method')
#plt.savefig('C:/Users/Conor Kirby/OneDrive/Desktop/f.bisectionmethod.pdf')
plt.show()
print(x2)
```

Exercise 2:

```
"""
Created on Wed Oct  4 10:11:50 2023

@author: Conor Kirby
"""
import numpy as np
import matplotlib.pyplot as plt
```

```
def f(x):
    a = 1
    b = -4
    c = 3
    return a*x*x + b*x + c

def f1(x):
    a = 1
    b = -4
    return 2*a*x + b


x1 = -2
x = np.linspace(-2, 6, 100)
plt.plot(x, f(x))
plt.plot(x, f1(x))
plt.plot(x1, f1(x1), "ro")
plt.title("Finding the Roots of a Function Using the Newton-Raphson Method")
plt.legend(["f(x)", "The Derivative of f(x)"])
#plt.savefig('C:/Users/Conor Kirby/OneDrive/Desktop/f.plot2.pdf')
plt.show()


tol_list = []
nsteps_list = []
exp_array = np.arange(-40.0, -2.0, 1)

for exp in exp_array:
    tol = 10**exp
    tol_list.append(tol)
#print(tol_list)
for tol in tol_list:
    x1 = -2
    nsteps = 0
    while abs(f(x1))>tol:
        x1 = x1 - f(x1)/f1(x1)
        nsteps += 1
    #print(nsteps)
    nsteps_list.append(nsteps)
print(nsteps_list)
plt.plot(-np.log10(tol_list), nsteps_list, '--ro')
plt.ylabel("Number of Steps")
plt.xlabel("-log10(tolerance)")
plt.title("Convergence Behaviour of the Newton-Raphson Method")
#print(nsteps) #much more efficient than the bisection method
print(x1)
print(f(x1))
```

Exercise 3:

```
# -*- coding: utf-8 -*-
"""
```

```
Created on Wed Oct  4 11:24:18 2023

@author: Conor Kirby
"""

import numpy as np
import matplotlib.pyplot as plt

k = 1.44
A = 1090
p = 0.033

def V(x):
    return A*np.exp(-x/p) -k/x

x = np.linspace(0.1, 1.00, 100)
plt.ylim(-50, 100)
plt.plot(x, V(x))

def F(x):
    return -(A*np.exp(-x/p)*(-1/p) + k/(x**2))

plt.plot(x, F(x), "r")
plt.show

# V1 = A*np.exp(-x/p)*(-1/p) + k/(x**2)
def V11(x):
    return A*np.exp(-x/p)*(1/(p**2)) - 2*k/(x**3)

plt.plot(x, V11(x), "g")
plt.title("V(x), Force, V''(x)")
plt.xlabel("X Values")
plt.ylabel("Potential Energy (J) & Force (N)")
plt.legend(["Potential Energy", "Force(x)", "V''(x)"], facecolor='white', framealpha=1, labelcolor = 'black')
#plt.savefig('C:/Users/Conor Kirby/OneDrive/Desktop/f_vfv11.pdf')
plt.show()



tol_list = []
nsteps_list = []
exp_array = np.arange(-10.0, -1.0, 0.25)

for exp in exp_array:
    tol = 10**exp
    tol_list.append(tol)
#print(tol_list)

for tol in tol_list:
    x1 = -2
    nsteps = 0
    while abs(V(x1))>tol:
```

```
        x1 = x1 - V(x1)/-F(x1)
        nsteps += 1
    #print(nsteps)
    nsteps_list.append(nsteps)

print(x1) #min = 0.15783985885829135
print(nsteps_list)
plt.plot(-np.log10(tol_list), nsteps_list, '--mo')
plt.ylabel("Number of Steps")
plt.xlabel("-log10(tolerance)")
plt.title("Newton-Raphson Method for Ionic Interaction Potentials Exp")
plt.savefig('C:/Users/Conor Kirby/OneDrive/Desktop/f_nrmforexercise3.pdf')
plt.show()


print(F(x1)) # Value of F(x) at min of V(x) = 218.6843.... electrostatic intermolecular force??
print(V11(x1))
```