Laboratory 1: Finding minima of functions

ismisebrendan

February 21, 2023

1 Introduction

The aim of this lab was to find the minima of function using two computational methods, the bisection method, and the Newton-Raphson method and apply the latter of these methods to the problem of determining the minimum of a potential energy function for the ionic interaction of two ions.

The bisection method is used to find the roots of functions. It begins by selecting an interval $[x_1, x_3]$ with a root of the function, $x = r$ falling somewhere on that interval, i.e. $f(x_1) < 0$ and $f(x_3) > 0$ or $f(x_1) > 0$ and $f(x_3) < 0$ (this can be done simply by 'eyeballing' the graph of the function).

Say for this example that $f(x_1) < 0$ and $f(x_3) > 0$. First the midpoint of x_1 and x_3, x_2 is calculated.

$$x_2 = \frac{x_1 + x_3}{2} \tag{1.1}$$

It is now determined if $f(x_2)$ is greater than or less than 0. If $f(x_2) < 0$ then x_2 lies between x_1 and the r so x_1 is replaced by x_2, and similarly if $f(x_2) > 0$ x_3 is replaced by x_2. The process is then iterated over until the value of $|f(x_2)|$ is less than some predetermined tolerance value. The value of x_2 is then taken as the value of r, the root of the function.

This is of course very tedious to do by hand, however it is very easy for a computer to do this, assuming of course that the initial values of x_1 and x_3 are inputted manually, and the tolerance value is attainable by the computer.

The Newton-Raphson method can also be used to find roots of functions in a similar way. An initial estimate of the root r, x_1, of the function $f(x)$ is first obtained (perhaps by visual inspection of the graph of the function). If $f(x_1) = 0$ then simply $r = x_1$. In the likely event that $f(x_1) \neq 0$ we must find a better estimate for r.

This can be done by linearly approximating $f(x)$ with its tangent line at x_1. The x-intercept of this line, x_2 could be considered a better approximation of r. x_2 is not treated the same way as x_1 above, if $f(x_2) = 0$ then $r = x_2$. If $f(x_2) \neq 0$ we find the tangent line to f at x_2, take its x-intercept as x_3 and repeat this process until x_n approaches r.[1]

The formula for this can be derived from the formula for a line.

$$y - y_1 = m(x - x_1) \tag{1.2}$$

In the first iteration $y_1 = f(x_1)$, $x_1 = x_1$, and $m = f'(x_1)$.

$$y - f(x_1) = f'(x_1)(x - x_1) \tag{1.3}$$

When the tangent line crosses the x-axis $y = 0$ and $x = x_2$.

$$-f(x_1) = f'(x_1)(x_2 - x_1)$$

$$\implies x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{1.4}$$

This can be generalised for x_{n+1} as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.5}$$

This can be implemented in a programming language such as python to find the roots of almost any function.

The Newton-Raphson method does have some difficulties with some functions however. If $f'(x_n) = 0$ for any n then Eq. 1.5 involves division by zero, so x_{n+1} cannot be generated. There are some functions also for which it will never find a solution, such as $f(x) = x^{1/3}$. [1]

The interaction potential between two singly charged ions is given by the below equation, representing the Pauli repulsion of the electron clouds at short range, and the long range electrostatic attraction between ions.

$$V(x) = Ae^{-\frac{x}{p}} - \frac{e^2}{4\pi\epsilon_0 x} \tag{1.6}$$

The x value at which $V(x)$ is the equilibrium bond length of the ions. This also corresponds to the separation of the ions at which the force between them, $F(x)$, is zero. This is because $F(x)$ is equal to the negative of the derivative of the potential energy with respect to position.

$$V'(x) = -\frac{A}{p}e^{-\frac{x}{p}} + \frac{e^2}{4\pi\epsilon_0 x^2} \tag{1.7}$$

$$F(x) = -V'(x) = \frac{A}{p}e^{-\frac{x}{p}} - \frac{e^2}{4\pi\epsilon_0 x^2} \tag{1.8}$$

The x position of the minimum of the potential energy can therefore be found by finding the x value corresponding to the zero of the force.

The aims of the lab were achieved through the implementation of the bisection method in python, with Eq. 1.1 used to calculate the new points for this method. The Newton-Raphson method was also implemented in python, with Eq. 1.5 used to find new estimates for the value of the root. The Newton-Raphson method was also used to find the minimum of the potential energy function for ionic interactions by using the force and its derivative in Eq. 1.5.

2 Methodology

2.1 Bisection Method

The code for this section can be found in *Appendix A: Code, 5.1.1 Bisection Method* below.

The function $f(x) = 2x^2 + 5x - 10$ was chosen and initialised as a function in python and plotted using the matplotlib.pyplot library. Through visual inspection values for x_1 and x_3 of -1 and 2 respectively were chosen. It was checked that when inputted into the function these x-values gave points below and above the x-axis respectively.

From Eq. 1.1 a new value, x_2, was generated and the point $(x_2, f(x_2))$ was plotted. It was determined whether $f(x_2)$ was greater or less than 0. If $x_2 > 0$ then x_3 was redefined to be equal to x_2 and if $x_2 < 0$ then x_1 was redefined to be equal to x_2.

This was repeated in a while loop until $|f(x_2)|$ was less than the desired tolerance value (initially 0.0001). The value of x_2 (the root) and $f(x_2)$ were then printed, and the fact that it was a correct root was confirmed.

This was repeated with different initial values of x_1 and x_3, -3 and -5 respectively to find the other root of the function.

The number of steps taken for this method to find the root for different tolerance values was found by repeatedly finding the root for different tolerance values greater than or equal to 10^{-15} through the use of a while loop. The number of steps taken and the corresponding tolerance value were recorded in numpy arrays and a graph of the number of steps versus the base 10 logarithm of the tolerance was plotted.

2.2 Newton-Raphson Method

The code for this section can be found in *Appendix A: Code, 5.1.2 Newton-Raphson Method* below.

Again, the function $f(x) = 2x^2 + 5x - 10$ was chosen and initialised as a function in python and plotted as was its derivative. They were both plotted using the matplotlib.pyplot library. The initial value of x_1 was chosen to be $x_1 = 1$. Eq. 1.5 was used to find the first estimate of the root and update the variable x_1 accordingly. The point $(x_1, f(x_1))$ was then plotted.

Using a while loop this was repeated until $|f(x_1)|$ was less than the desired tolerance value (initially 0.0001). The value of x_1 (the root) and $f(x_1)$ were then printed, and the fact that it was a correct root was confirmed.

This was repeated with x_1 initialised to -4 to find the other root of the function.

The number of steps taken for this method to find the root for different tolerance values was found by repeatedly finding the root for different tolerance values greater than or equal to 10^{-15} through the use of a while loop. The number of steps taken and the corresponding tolerance value were recorded in numpy arrays and a graph of the number of steps versus the base 10 logarithm of the tolerance was plotted.

2.3 Comparison of Newton-Raphson Method and Bisection Method

The code for this section can be found in *Appendix A: Code, 5.1.3 Comparison of Newton-Raphson Method and Bisection Method* below.

In order to create a graph of the number of steps taken to find the root against the tolerance of the root for the two methods on the same axes a new file was created. The function and its derviative were initialised in python.

The final steps of both 2.1 and 2.2 were copied into this file. Arrays were created to store the values of the tolerance and the number of steps taken for each method to reach the root. The root of the function was found for different tolerance values greater than or equal to 10^{-15} through the use of a while loop.

The number of steps taken for each method and the corresponding tolerance value were recorded in numpy arrays and a graph of the number of steps versus the base 10 logarithm of the tolerance was plotted for the two methods.

2.4 Ionic Interaction Potentials

The code for this section can be found in *Appendix A: Code, 5.1.4 Ionic Interaction Potentials* below.

The potential energy function $V(x)$, Eq. 1.6 was defined and plotted, as was the force between the ions, $F(x)$ as shown in Eq. 1.8. The derivative of the force, $F'(x)$ was also calculated and initialised as a function.

$$V''(x) = \frac{A}{p^2} e^{-\frac{x}{p}} - \frac{e^2}{2\pi\epsilon_0 x^3} \tag{2.1}$$

$$F'(x) = -V''(x) = -\frac{A}{p^2} e^{-\frac{x}{p}} + \frac{e^2}{2\pi\epsilon_0 x^3} \tag{2.2}$$

As $F(x) = -V'(x)$ and $F'(x) = -V''(x)$,

$$\frac{V'(x_n)}{V''(x_n)} = \frac{-F(x_n)}{-F'(x_n)} = \frac{F(x_n)}{F'(x_n)} \tag{2.3}$$

As such the Newton-Raphson method can be used to find the zero of the derivative of the potential energy function using the force and its derivative rather than the first and second derivatives of the potential energy function.

The Newton-Raphson method was implemented with a tolerance value of 10^{-10} to find the x-value of the zero of the force and therefore the minimum of the potential energy function.

3 Results

3.1 Bisection Method

The initial midpoint of x_1 and x_3, x_2 is shown in Fig. 3.1 below.

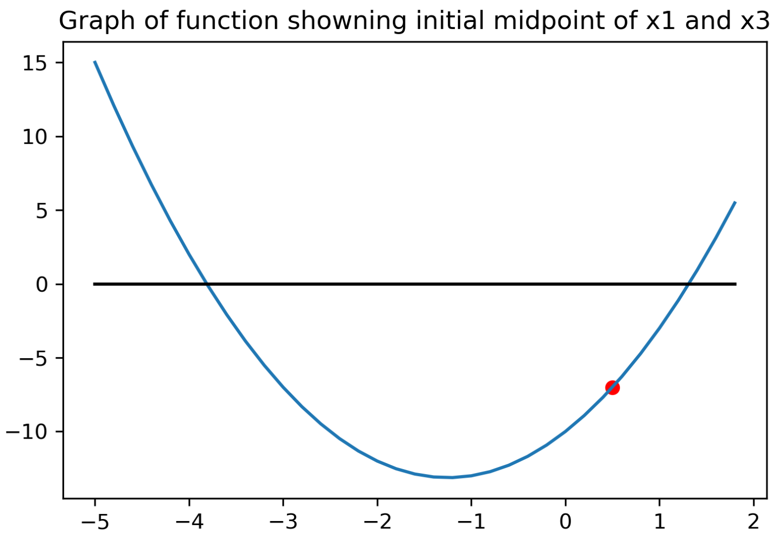

Figure 3.1: The graph showing the initial value of x_2.

After a number of iterations with two different initial x_1 and x_3 values the two roots of the function were found with a tolerance of 0.0001. They are show in Fig. 3.1.

The roots of the function were found to be $x_{2_a} = -3.8117(1)$ and $x_{2_b} = 1.3117(1)$ respectively. This is in agreement with the roots determined algebraically from the quadratic formula;

$$x_{2_{a,b}} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$= \frac{-5 \pm \sqrt{5^2 - 4(2)(-10)}}{2(2)}$$

$$x_{2_a} = -\frac{5 + \sqrt{105}}{4} = -3.8117 \tag{3.1}$$

$$x_{2_b} = \frac{-5 + \sqrt{105}}{4} = 1.3117 \tag{3.2}$$

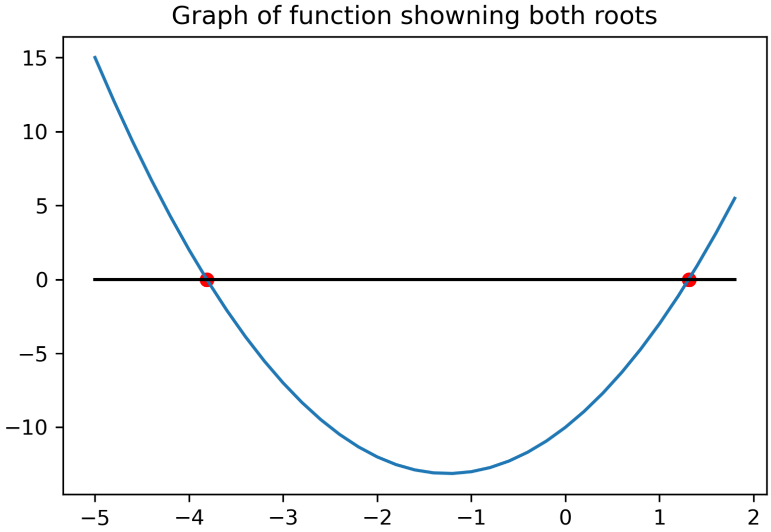

Figure 3.2: The graph showing the two roots of the function $f(x) = 2x^2 + 5x - 10$ as determined by the bisection method.

The number of steps required to calculate the root of the function was seen to increase as the tolerance of the root decreased (or the strictness of the root increases). The relationship between the tolerance of the root (tol) and the number of steps taken to calculate the root (nsteps) is shown in figure 3.1, a selection of these results is shown in Table 3.1 also.

Convergence of bisection method	
tol	nsteps
10^{-1}	7
5×10^{-2}	8
2.5×10^{-3}	10
...	...
4.54747×10^{-14}	48
2.27374×10^{-14}	48
1.13687×10^{-14}	48

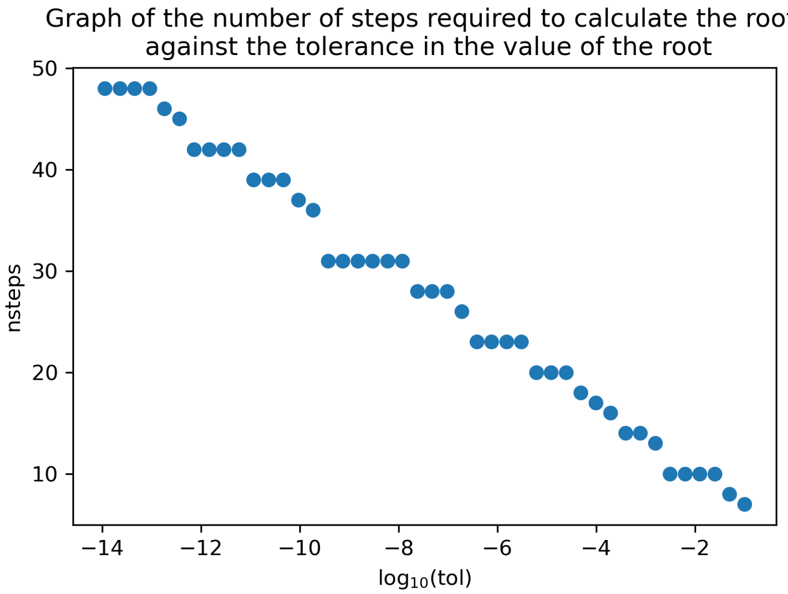

Figure 3.3: The graph of $log_{10}(tol)$ against nsteps.

3.2 Newton-Raphson Method

The initial estimation of x_1 is shown in Fig. 3.2 below.

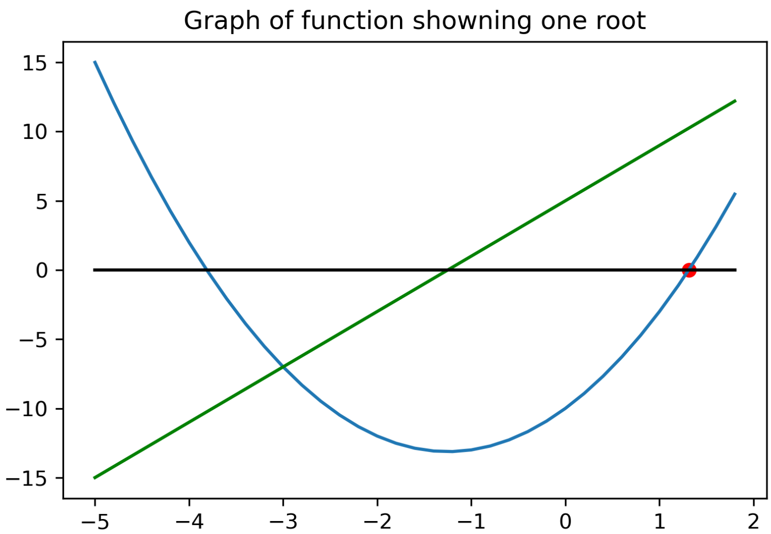

Figure 3.4: The graph showing the initial value of x_1.

After a number of iterations with two different initial x_1 values the two roots of the function were found with a tolerance of 0.0001. They are show in Fig. 3.2.

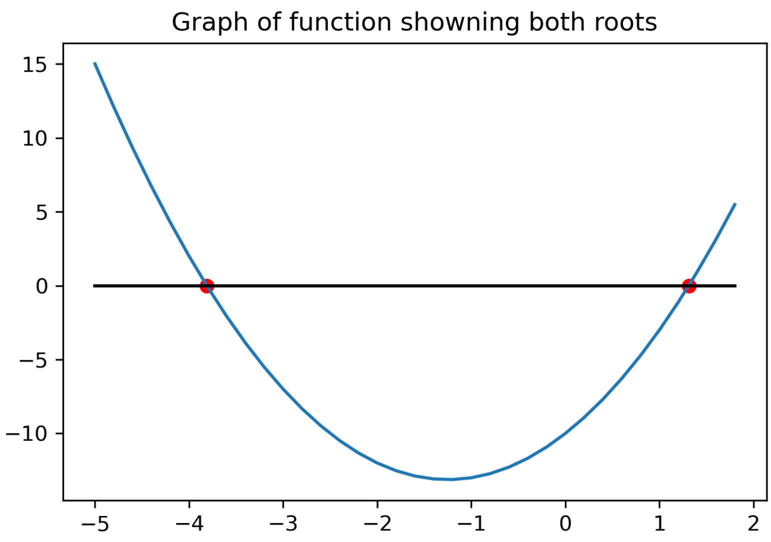

Figure 3.5: The graph showing the two roots of the function $f(x) = 2x^2 + 5x - 10$ as determined by the Newton-Raphson method.

The roots of the function were found to be $x_{1_a} = -3.8117(1)$ and $x_{1_b} = 1.3117(1)$ respectively. This is in agreement with the roots determined algebraically from the quadratic formula as shown in Eq. 3.1 and Eq. 3.2 above and by the bisection method.

The number of steps required to calculate the root of the function was seen to increase as the tolerance of the root decreased (or the strictness of the root increases). The relationship between the tolerance of the root (tol) and the number of steps taken to calculate the root (nsteps) is shown in Fig. 3.2, a selection of these results is shown in Table 3.2 also.

Convergence of Newton-Raphson method	
tol	nsteps
10^{-1}	2
5×10^{-2}	2
2.5×10^{-3}	2
...	...
4.54747×10^{-14}	4
2.27374×10^{-14}	4
1.13687×10^{-14}	4

From examination of the data it is seen that the root calculated is functionally indistinguishable from the algebraically determined root after the tolerance decreases to below approximately 2×10^{-8}. This appears to be the greatest accuracy achievable with python. This is also approximately the point at which the graph in Fig. 3.2 plateaus at 4 steps.

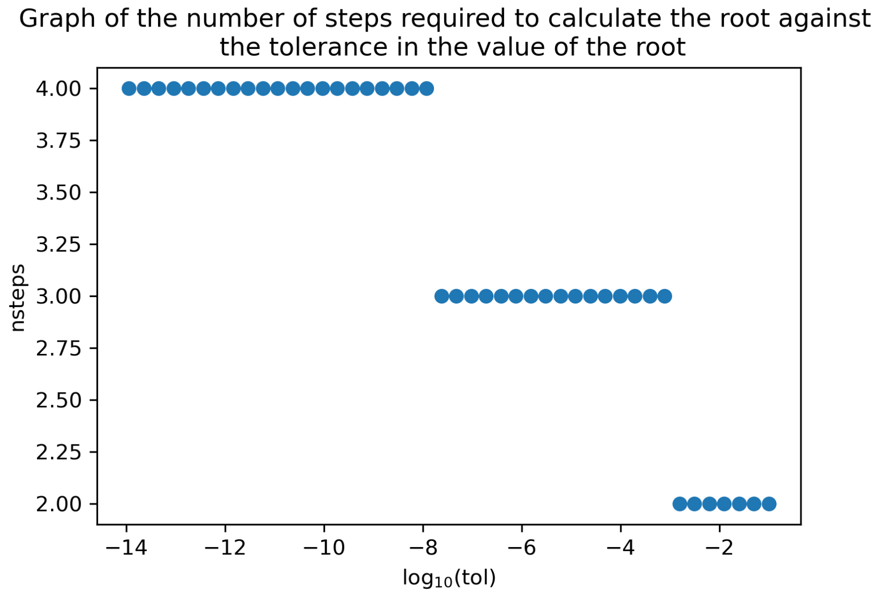

Figure 3.6: The graph of $log_{10}(tol)$ against nsteps.

3.3 Comparison of Newton-Raphson Method and Bisection Method

The graphs of the relationships of tol and nsteps for both methods is shown in Fig. 3.3.

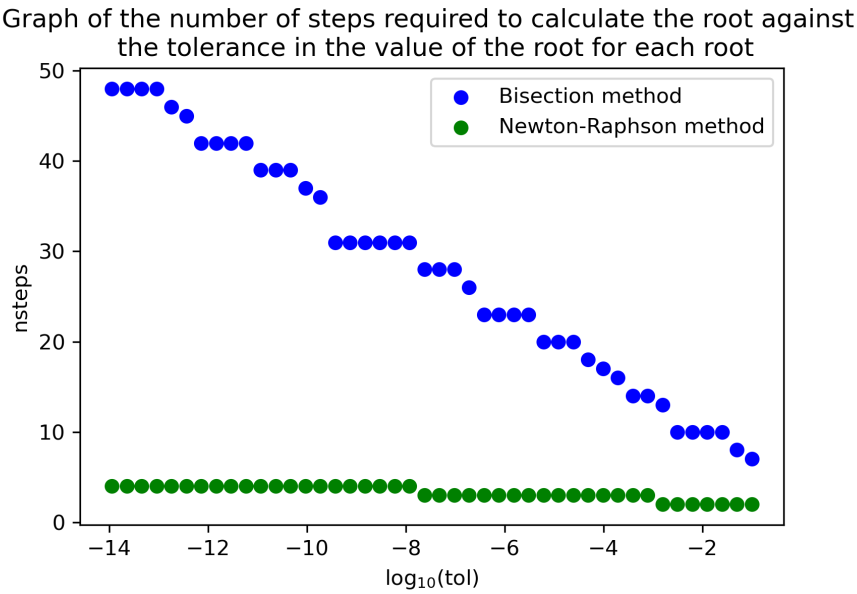

Figure 3.7: The graph of $log_{10}(tol)$ against nsteps for both methods.

As can be seen clearly from the graph the Newton-Raphson method is vastly more efficient than the bisection method, in some cases calculating the same root in less than 10% needed to calculate it with the bisection method.

3.4 Ionic Interaction Potentials

The graph showing the potential energy (V) of the ions as a function of distance (x) and the force (F) on said ions, as well as the minimum value of the potential energy is shown in Fig. 3.4.

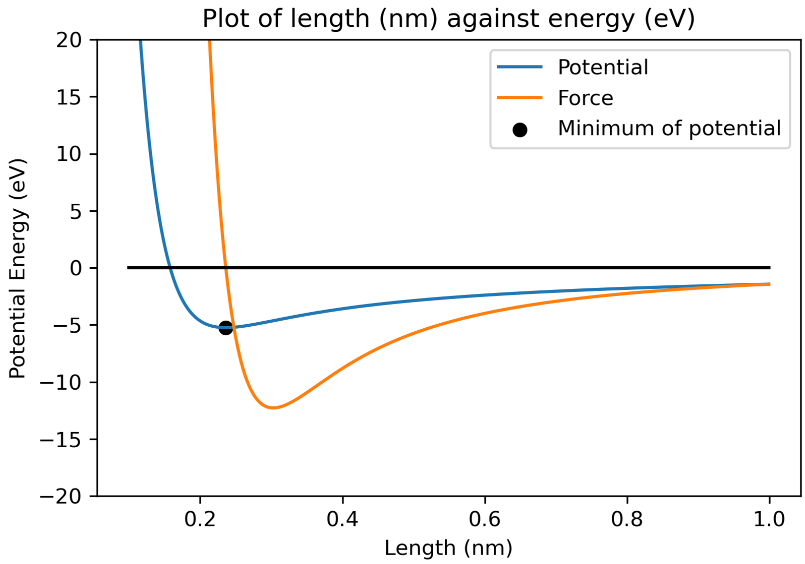

Figure 3.8: The V and F for the ions against x, also showing the minimum value of V(x) as calculated by the Newton-Raphson method.

The minimum value of V(x) and its corresponding x-value, as calculated by the Newton-Raphson method through finding the zero-value of F(x) was found to be as shown in Table 3.4.

Minimum of V(x)	
x (nm)	V(x) (eV)
0.2360538484(1)	-5.2474891185(1)

The expressions for V'(x) and V"(x) are given in Eq. 1.7 and Eq. 2.1 respectively.

4 Conclusions

As can be clearly seen from the graph in Fig. 3.3 the Newton-Raphson method is much more efficient than the bisection method for finding the roots of functions. While the bisection method does work and is relatively easy to implement in python if it was desired for a large number of roots to be found by the program it would likely cause it to run much slower and for much longer than is ideally desired.

Due to its efficiency the Newton-Raphson method is generally a much better option when the root of a function is desired to be calculated. Its main drawbacks are that it requires the function to be differentiable in order to work, if at any point the derivative is equal to zero it causes division by zero, and there are some functions for which it simply does not work. However in the case that it does work it can be up to ten times faster at calculating roots than the bisection method.

Of course to fully test this statement the same code should be run for multiple different functions, both parabolic and other, more exotic, functions. However it seems clear that, at least in this special case, but likely in others as well, the Newton-Raphson method is much more efficient than the bisection method.

The Newton-Raphson method was used to successfully find the x value of the minimum of V(x), and the corresponding value also.

5 Appendices

5.1 Appendix A: Code

5.1.1 Bisection Method

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 24 16:41:41 2023

@author: wattersb
"""

import numpy as np
import matplotlib.pyplot as plt

# COEFFICIENTS OF THE PARABOLA
a = 2
b = 5
c = -10

# DEFINE PARABOLIC FUNCTION
def f(x):
    return a*x**2 + b*x + c

# GENERATE x POINTS
x = np.arange(-5.0, 2.0, 0.2)

# DEFINE THE TOLERANCE OF THE ROOT
tol = 0.0001

# PLOT THE FUNCTION AND X-AXIS
plt.plot(x, f(x))
```

```
plt.plot(x, 0.0*x, color='black')

# GENERATE INITIAL VALUES OF x1 AND x2
x1 = -1      # f(x1) < 0
x3 = 2       # f(x3) > 0

# CHECK INITIAL X VALUES
if f(x1) < 0:
    print("f(x1)<0 -> good")
else:
    print("f(x1)>0 -> not good")

if f(x3) > 0:
    print("f(x3)>0 -> good")
else:
    print("f(x3)<0 -> not good")

# DEFINE x2 AS THE MIDPOINT OF x1 AND x3
x2 = 0.5 * (x1 + x3)

plt.scatter(x2, f(x2), color='r') # plot the first guess at the root

# GRAPH OF THE INITIAL GUESS - Part 6
plt.title("Graph of function showning initial midpoint of x1 and x3")
plt.savefig("1.06Graph.png", dpi=300)
plt.show()

####################
#                  #
#    First Root    #
#                  #
####################

# ITERATE TO FIND ROOT
while np.abs(f(x2)) > tol:
    # DEFINE X2 AS THE MIDPOINT
    x2 = 0.5 * (x1 + x3)

    # UPDATE X1 OR X3 TO X2 AND SAVE NEW VALUE
    if f(x2) < 0: # f(x1) < 0 always
        x1 = x2
    elif f(x2) > 0: # f(x3) > 0 always
        x3 = x2

# OUTPUT BEST CALCULATION OF THE ROOT, THE VALUE OF THE FUNCTION HERE, AND ITS ABSOLUTE VALUE
print("x2 =", x2)
print("f(x2) =", f(x2))
print("|f(x2)| =", np.abs(f(x2)))

# GRAPH OF THE ROOT - Part 7
plt.scatter(x2, f(x2), color='r')   # plot the root
```

```
plt.plot(x, f(x))                      # plot the function
plt.plot(x, 0.0*x, color='black')    # plot the x axis
plt.title("Graph of function shownng one root")
plt.savefig("1.07Graph.png", dpi=300)
plt.show()

#####################
#                   #
#    Second Root    #
#                   #
#####################

print("---Second Root---")

# GENERATE INITIAL VALUES OF x1 AND x2
x1a = -3       # f(x1) < 0
x3a = -5       # f(x3) > 0

# CHECK INITIAL X VALUES
if f(x1a) < 0:
    print("f(x1)<0 -> good")
else:
    print("f(x1)>0 -> not good")

if f(x3a) > 0:
    print("f(x3)>0 -> good")
else:
    print("f(x3)<0 -> not good")

# DEFINE X2 AS THE MIDPOINT
x2a = 0.5 * (x1a + x3a)

# ITERATE TO FIND ROOT
while np.abs(f(x2a)) > tol:
    # DEFINE X2 AS THE MIDPOINT
    x2a = 0.5 * (x1a + x3a)

    # UPDATE X1 OR X3 TO X2 AND OUTPUT NEW VALUES
    if f(x2a) < 0:
        x1a = x2a
    elif f(x2a) > 0:
        x3a = x2a

# OUTPUT BEST CALCULATION OF THE ROOT, THE VALUE OF THE FUNCTION HERE, AND ITS ABSOLUTE VALUE
print("x2 =", x2a)
print("f(x2) =", f(x2a))
print("|f(x2)| =", np.abs(f(x2a)))

# GRAPH OF THE ROOT - Part 8
plt.scatter(x2a, f(x2a), color='r') # plot the root
plt.plot(x, f(x))                      # plot the function
```

```
plt.plot(x, 0.0*x, color='black')   # plot the x axis
plt.title("Graph of function showning another root")
plt.show()

# GRAPH OF BOTH ROOTS
#plot the roots
plt.scatter(x2, f(x2), color='r')   # plot the root
plt.scatter(x2a, f(x2a), color='r') # plot the other root
plt.plot(x, f(x))                   # plot the function
plt.plot(x, 0.0*x, color='black')   # plot the x axis
plt.title("Graph of function showning both roots")
plt.savefig("1.08Graph.png", dpi=300)
plt.show()

########################################################
#                                                      #
#    Dependence of number of steps on the tolerance    #
#                                                      #
########################################################

steps_list = np.array([])   # array of the number of steps
tol_list = np.array([])     # array of the tolerance values

# INITIAL TOLERANCE OF THE ROOT
tol = 0.1

while tol >= 10e-15:

    # INTITIALISE X VALUES
    x1 = -1
    x3 = 2
    x2 = 0.5 * (x1 + x3)

    nsteps = 0      # counter for number of steps to find root

    # ITERATE TO FIND THE ROOT
    while np.abs(f(x2)) > tol:
        # DEFINE X2 AS THE MIDPOINT
        x2 = 0.5 * (x1 + x3)

        # UPDATE X1 OR X3 TO X2 AND OUTPUT NEW VALUES
        if f(x2) < 0:
            x1 = x2
        elif f(x2) > 0:
            x3 = x2
        nsteps += 1

    # APPEND TO THE ARRAYS
    steps_list = np.append(steps_list, nsteps)
    tol_list = np.append(tol_list, tol)

    # DECREASE THE TOLERANCE FOR NEXT TIME
```

```
    tol = tol/2

# PLOT OF THE NUMBER OF STEPS TAKEN AGAINST THE LOG BASE 10 OF THE
   TOLERANCE
plt.scatter(np.log10(tol_list), steps_list)
plt.xlabel("log$_{10}$(tol)")
plt.ylabel("nsteps")
plt.title("Graph of the number of steps required to calculate the root
   \n against the tolerance in the value of the root")
plt.savefig("1.10Graph.png", dpi=300)
plt.show()
```

5.1.2 Newton-Raphson Method

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 31 14:55:17 2023

@author: wattersb
"""

import numpy as np
import matplotlib.pyplot as plt

# COEFFICIENTS OF THE PARABOLA
a = 2
b = 5
c = -10

# DEFINE PARABOLIC FUNCTION
def f(x):
    return a*x**2 + b*x + c

# DEFINE DERIVATIVE
def df(x):        # x position, h
    return 2*a*x + b

# GENERATE x POINTS
x = np.arange(-5.0, 2.0, 0.2)

# DEFINE THE TOLERANCE OF THE ROOT
tol = 0.0001

# PLOT THE FUNCTION, ITS DERIVATIVE AND X-AXIS
plt.plot(x, f(x))
plt.plot(x, df(x), color='g')
plt.plot(x, 0.0*x, color='black')


# INITITAL GUESS OF ROOT
x1 = 1  # initital estimate of the root
x1 = x1 - f(x1)/df(x1)
plt.scatter(x1, f(x1), color='r')
```

```python
plt.title("Graph of function showning initial guess of root")
plt.show()


####################
#                  #
#    First Root    #
#                  #
####################

# ITERATE TO FIND ROOT
while np.abs(f(x1)) > tol:
    x1 = x1 - f(x1)/df(x1)

# GRAPH OF THE ROOT
plt.plot(x, f(x))
plt.plot(x, df(x), color='g')
plt.plot(x, 0.0*x, color='black')
plt.scatter(x1, f(x1), color='r')
plt.title("Graph of function showning one root")
plt.savefig("2Graph01.png", dpi=300)
plt.show()

# OUTPUT BEST CALCULATION OF THE ROOT, THE VALUE OF THE FUNCTION HERE,
   AND ITS ABSOLUTE VALUE
print("x1 =", x1)
print("f(x1) =", f(x1))
print("|f(x1)| =", np.abs(f(x1)))


#####################
#                   #
#    Second Root    #
#                   #
#####################

x2 = -4  # initital estimate of the second root

# ITERATE TO FIND ROOT
while np.abs(f(x2)) > tol:
    x2 = x2 - f(x2)/df(x2)

# GRAPH OF THE ROOT
plt.plot(x, f(x))
plt.plot(x, df(x), color='g')
plt.plot(x, 0.0*x, color='black')
plt.scatter(x2, f(x2), color='r')
plt.title("Graph of function showning another root")
plt.savefig("2Graph02.png", dpi=300)
plt.show()

print("x2 =", x2)
```

```
print("f(x2) =", f(x2))
print("|f(x2)| =", np.abs(f(x2)))


# GRAPH BOTH ROOTS
plt.plot(x, f(x))
plt.plot(x, df(x), color='g')
plt.plot(x, 0.0*x, color='black')
plt.scatter(x1, f(x1), color='r')
plt.scatter(x2, f(x2), color='r')
plt.title("Graph of function showning both roots")
plt.savefig("2Graph03.png", dpi=300)
plt.show()


########################################################
#                                                      #
#     Dependence of number of steps on the tolerance    #
#                                                      #
########################################################

steps_list = np.array([])   # array of the number of steps
tol_list = np.array([])     # array of the tolerance values
root_list = np.array([])    # array of the roots calculated

# INITIAL TOLERANCE OF THE ROOT
tol = 0.1    # tolerance of root

while tol >= 10e-15:

    # INTITIALISE X VALUE
    x1 = 1

    nsteps = 0      # counter for number of steps to find root

    # ITERATE TO FIND THE ROOT
    while np.abs(f(x1)) > tol:
        x1 = x1 - f(x1)/df(x1)
        nsteps += 1

    # APPEND TO THE ARRAYS
    steps_list = np.append(steps_list, nsteps)
    tol_list = np.append(tol_list, tol)
    root_list = np.append(root_list, x1)

    # DECREASE THE TOLERANCE FOR NEXT TIME
    tol = tol/2

# PLOT OF THE NUMBER OF STEPS TAKEN AGAINST THE LOG BASE 10 OF THE TOLERANCE
plt.scatter(np.log10(tol_list), steps_list)
plt.xlabel("log$_{10}$(tol)")
```

```
plt.ylabel("nsteps")
plt.title("Graph of the number of steps required to calculate the root
    against \n the tolerance in the value of the root")
plt.savefig("2Graph04.png", dpi=300)
plt.show()
```

5.1.3 Comparison of Newton-Raphson Method and Bisection Method

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 31 15:22:21 2023

@author: wattersb
"""

import numpy as np
import matplotlib.pyplot as plt

# COEFFICIENTS OF THE PARABOLA
a = 2
b = 5
c = -10

# DEFINE PARABOLIC FUNCTION
def f(x):
    return a*x**2 + b*x + c

# DEFINE DERIVATIVE
def df(x):        # x position, h
    return 2*a*x + b

# GENERATE x POINTS
x = np.arange(-5.0, 2.0, 0.2)


# INITIALISE ARRAYS
steps_list_nr = np.array([])           # array of the number of steps
    for Newton-Raphson Method
steps_list_bisection = np.array([])    # array of the number of steps
    for Bisection Method
tol_list = np.array([])                # array of the tolerance values

# INITIAL TOLERANCE OF THE ROOT
tol = 0.1

while tol >= 10e-15:

    ##########################
    #                        #
    #    Bisection Method    #
    #                        #
    ##########################

```

```
    # INTITIALISE X VALUES
    x1 = -1
    x3 = 2
    x2 = 0.5 * (x1 + x3)

    nsteps_bisecton = 0      # counter for number of steps to find root

    # ITERATE TO FIND THE ROOT
    while np.abs(f(x2)) > tol:
        # DEFINE X2 AS THE MIDPOINT
        x2 = 0.5 * (x1 + x3)

        # UPDATE X1 OR X3 TO X2 AND OUTPUT NEW VALUES
        if f(x2) < 0:
            x1 = x2
        elif f(x2) > 0:
            x3 = x2
        nsteps_bisecton += 1

    ###############################
    #                             #
    #    Newton-Raphson Method    #
    #                             #
    ###############################
    # INTITIALISE X VALUE
    x1 = 1

    nsteps_nr = 0      # counter for number of steps to find root

    # ITERATE TO FIND THE ROOT
    while np.abs(f(x1)) > tol:
        x1 = x1 - f(x1)/df(x1)
        nsteps_nr += 1

    # APPEND TO THE ARRAYS
    steps_list_bisection = np.append(steps_list_bisection,
    nsteps_bisecton)
    steps_list_nr = np.append(steps_list_nr, nsteps_nr)
    tol_list = np.append(tol_list, tol)

    # DECREASE THE TOLERANCE FOR NEXT TIME
    tol = tol/2

# GRAPH THE TWO METHODS AGAINST EACH OTHER
plt.scatter(np.log10(tol_list), steps_list_bisection, color='b', label=
    "Bisection method")
plt.scatter(np.log10(tol_list), steps_list_nr, color='g', label="Newton
    -Raphson method")
plt.legend()

plt.xlabel("log$_{10}$(tol)")
plt.ylabel("nsteps")
```

```python
plt.title("Graph of the number of steps required to calculate the root
    against \n the tolerance in the value of the root for each root")
plt.savefig("nrVbisection.png", dpi=300)
plt.show()
```

5.1.4 Ionic Interaction Potentials

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 31 15:39:03 2023

@author: wattersb
"""

import numpy as np
import matplotlib.pyplot as plt

# VALUES OF FUNCTION
k = 1.44    # eV nm,   k = e^2/(4*pi*epsilon0)
A = 1090    # eV
p = 0.033   # nm

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

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

# DERIVATIVE OF FORCE
def dF(x):
    return -A/(p**2) * np.exp(-x/p) + 2*k/(x**3)

###########################
#                         #
#    FIND MINIMUM OF V    #
#                         #
###########################

# FIND ROOT OF F
tol = 10e-10
x1 = 0.2    # initial guess

while np.abs(F(x1)) > tol:
    x1 = x1 - F(x1)/dF(x1)

# PLOT GRAPHS
x = np.arange(0.1, 1.0, 0.001)

plt.plot(x, V(x), label="Potential")
plt.plot(x, F(x), label="Force")
plt.scatter(x1, V(x1), color='black', label="Minimum of potential")
```

```python
plt.plot(x, np.zeros(x.size), color="black")

plt.title("Plot of length (nm) against energy (eV)")
plt.ylabel("Potential Energy (eV)")
plt.xlabel("Length (nm)")
plt.legend()

plt.ylim([-20,20])
plt.savefig("ionicGraph.png", dpi=300)
plt.show()

# PRINT DATA
print("Minimum of V is at x =", x1, "nm")
print("Value of V at this point is", V(x1), "eV")
print("Value of F at this point is", F(x1), "eV/nm")
```

References

[1] H. Anton, I. Bivens, S. Davis, *Calculus Late Transcendentals*, 10th ed. Wiley, 2013.