Finding Minima of Functions

Valentina Watanabe

February 2024

Abstract

Two different methods for finding the roots of a function were examined. First, the bisection method was used. This consisted of finding the square root by setting an interval between two initial values and narrowing it down through the evaluation of the middle value. The second method was the Newton-Raphson method, which worked through the evaluation of an initial valued X using the formula $X = X + \frac{f(X)}{f'(X)}$. It was found that the second method worked more efficiently given an appropriate initial X and continuous function, but the first method is always applicable. Finally, the Newton-Raphson method was used to find the minimum separation value between two ions Na+ and Cl-. This value was found to be 0.23605384841577942 nm.

1 Introduction

This computational laboratory focused on three different alternatives for finding the root values and minima of a function. We used Python and the Spyder platform to develop this. The exercises involve implementing the bisection method, the Newton-Raphson method, and using this to find the minimum distance between two ions.

2 Experimental Method

2.1 Bisection Method

The Bisection Method for finding the roots of a continuous function works by first setting an interval. The method works by repeatedly narrowing down the interval until the root is obtained. The code written takes two inputs from the user, X_1 and X_2, after showing the graph of the set parabolic function and its formula. The midpoint is calculated using the formula. The user is then given the value of $f(X_1)$,$f(X_2)$, $f(X_3)$ with their respective x-values. The first root that the program looks for is the one that has a positive asymptote, so if $f(X_1)$ < 0 or $f(X_3) > 0$, the user is asked to change the input for an appropriate value. If is not equal to zero, the graph with the parabola is plotted with the calculated middle point, and then the user has to keep inserting numbers until results in the root. A while loop is used where the root is only found after the absolute value of is not greater than 0.0001. Every time the loop runs, the variable nsteps increases by 1, as it counts the number of steps it took the user to find the root. After the first root is found, a graph of the function with the highlighted root and a second plot of nsteps against the log of the tolerance are shown. The same process is repeated for the second root, with the changed parameters f(X_1) > 0 and $f(X_3) < 0$.

2.2 Newton-Raphson Method

This method consists of using the formula $X = X + \frac{f(X)}{f'(X)}$, for an initial value X, to find the root. A while loop is created so the X value is constantly updated until the absolute value of $f(X)$ is less than the tolerance variable, which indicates the root of the function. As in the previous method, the tolerance variable is initially set to 0.0001. To find the first root, the value of X_1 is set to 1. To find the second root, the value of X_2 can be set to any which corresponds to the decreasing part of the function (negative asymptote).

2.3 Minima Roots of Potential Energy Function

The third part of the computational lab consists of finding the minimum x value for the function that represents the interaction potential between two ions, $V(x) = Ae^(-x/p) - \frac{e^2}{4\pi\epsilon_0 x}$. The method to do this was the Newton-Raphson, using the equation x = x - $\frac{F(x)}{V''(x)}$. The function of force used was F(x) = -V'(x). The other functions relevant to this were the first and second derivatives of the V(x) function, $V'(x) = \frac{-A}{p}e^{\frac{-x}{p}} + \frac{e^2}{4\pi\epsilon_0 x^2}$ and $V'(x) = \frac{A}{p^2}e^{\frac{-x}{p}} + \frac{2e^2}{4\pi\epsilon_0 x^3}$ respectively. The initial estimate for x was 0.19, which was found using the plot of V(x) against a range for x of (0.01, 1).

3 Experimental Results

3.1 Bisection Method

The function used was $f(x) = x^2 - 4x + 3$, with their respective roots at x = 1 and x = 3. The first root that the program prompted the user to find is x = 3, and the x = 1 is the second. Figure 1 and Figure 2 are the respective graphs that result from the process of finding root 1. Figure 3 and Figure 4 are from finding the second root.

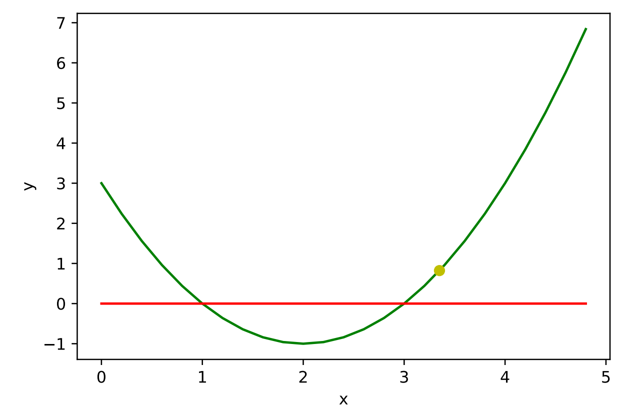

Figure 1: x vs y, with X_1

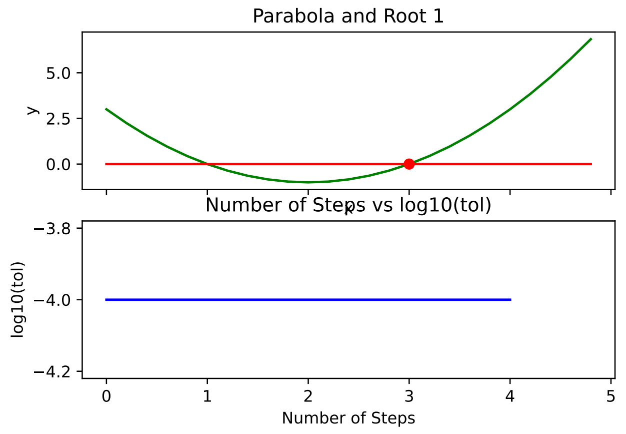

Figure 2: Parabola and Root 1 and nsteps vs. log(tol)

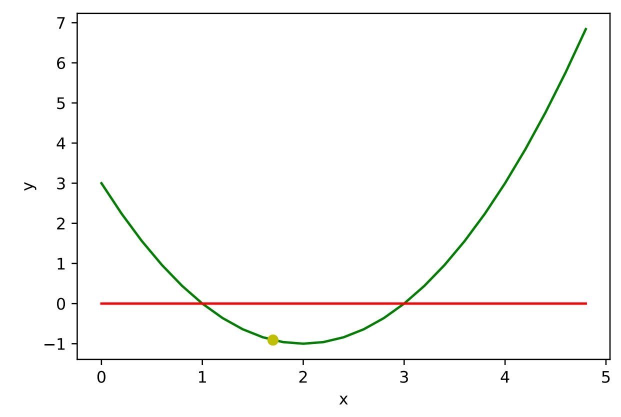

Figure 3: x vs y, with X_2

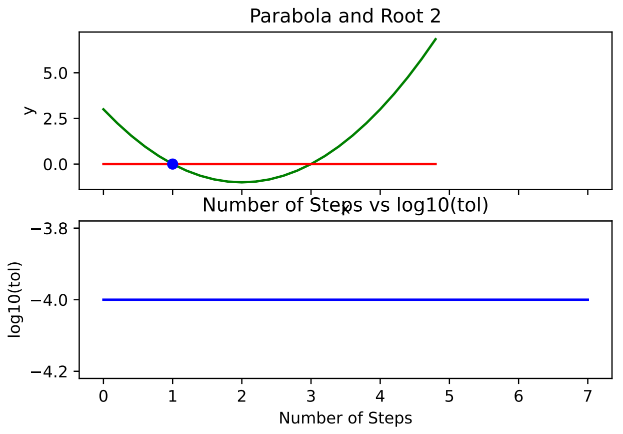

Figure 4: Parabola and Root 2 and nsteps vs. log(tol)

3.2 Newton-Raphson Method

The function used for this second exercise was changed to $f(x) = x^2 + 3x - 10$ to make the process less straightforward, as the roots of the previous values were equal to the initial values that were indicated to be set for X. Setting the tolerance to 0.00000000001 and X_2 = - 4 allowed the most accurate values for the roots to be found. In Figure 5 we have a graph of the function with its roots, and in Figure 6 we have the graphs of the number of steps vs. the tolerance for each root.

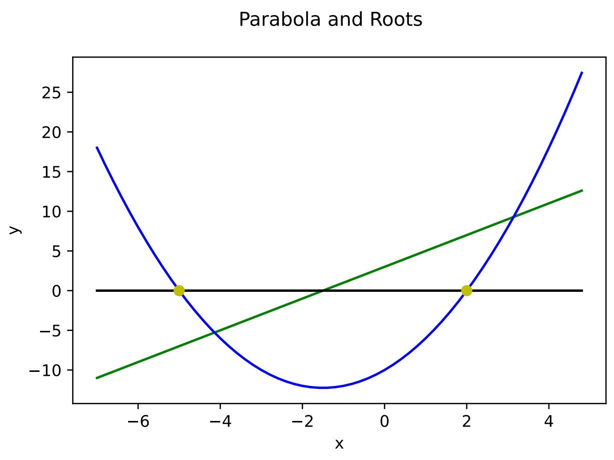

Figure 5: Parabola and roots

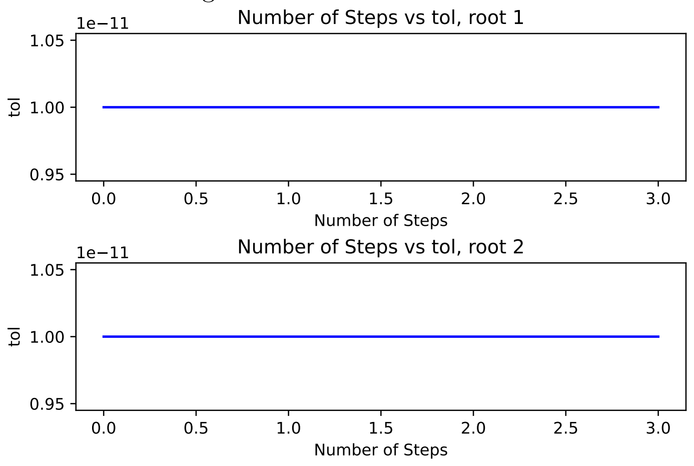

Figure 6: nsteps vs. tol for Root 1 and Root 2

Comparing this to the bisection method, we see that the convergence happens at a much higher rate. It also needs a specific good initial guess for the root we want to find, as it may find a different one otherwise. Another point of difference is that Newton's method doesn't work if the derivative is zero or approaches zero. The bisection method is assured to find the roots, but although faster, the Newton-Raphson method is not.

3.3 Minima Roots of Potential Energy Function

The value found for x was found using Figure 7, V(x) vs. x. The value of x where the interaction potential between the two ions was at a minimum results equal to 0.23605384841577942 nm. Figure 8 is the graph of the force F(x) vs x.

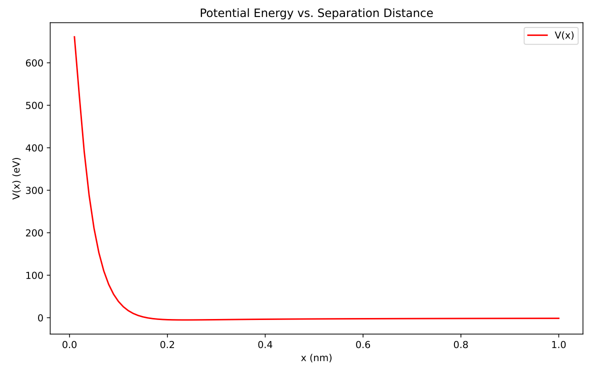

Figure 7: V(x) vs. X

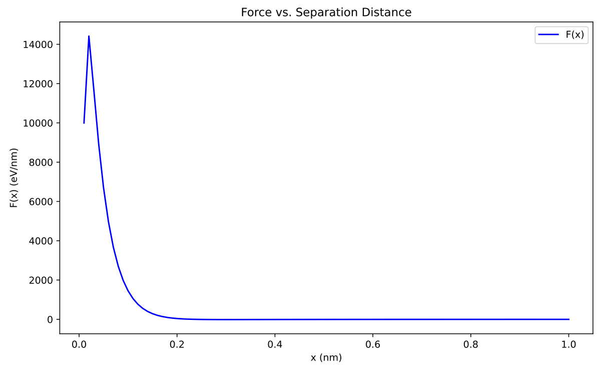

Figure 8: F(x) vs. x

4 Conclusion

This lab aimed to successfully work through two different methods of finding valuable information from a function. While reliable for all types of functions, the bisection method was shown to have more iterations than Newton's method. We then applied the best method, the Newton-Raphson, to find the distance at which the sodium chloride bond is at a minimum. This calculated value was 0.23605384841577942 nm.

5 Appendix

```
Exercise 1:
import numpy as np
import matplotlib.pylab as plt
def parabola(x):
    a = 1
    b = -4
    c = 3
    y = a*x**2 + b*x + c
    return y
x = np.arange(0,5,0.2)
# print(x)
y= parabola(x)
# print(y)
plt.plot(x,y,"g")
plt.plot(x,0.0*x,"r")
plt.xlabel('x')
plt.ylabel('y')
plt.show()

#FIND ROOT 1
print("f(x) = x^2 - 4x + 3")
print("Find root 1, with positive asymptote")
# valores x1 y x3 en input
x1 = float(input("X1: "))
x3 = float(input("X3: "))

# respectivos valores de y de x1 y x3

while parabola(x1) > 0:
    print ("f(x1) > 0, choose another value.")
    x1 = float(input("new X1: "))
while parabola(x3) < 0:
    print ("f(x3) < 0, choose another value.")
    x3 = float(input("new X3: "))
```

```
x2 = 0.5 * (x1 + x3)

# valores de y
print("f(" + str((x1)) + ")= " + str(parabola(x1)))
print("f(" + str((x2)) + ")= " + str(parabola(x2)))
print("f(" + str((x3)) + ")= " + str(parabola(x3)))

x = np.arange(0,5,0.2)
y= parabola(x)
plt.plot(x,y,"g")
plt.plot(x,0.0*x,"r")
plt.plot(x2, parabola(x2), "yo")
plt.xlabel('x')
plt.ylabel('y')
plt.savefig('Ex. 1 Root 1 part 1.pdf')
plt.show()

# updated values de x1 y x3
if parabola(x2) > 0 or parabola(x2) < 0 :
    choose = input(" f(x2) >= 0, change x1 or x3? ")
    if choose == "x1":
        x1 = float(input("choose new x1: "))
    else:
        x3 = float(input("choose new x3: "))
    x2 = 0.5 * (x1 + x3)
    print("f(" + str((x1)) + ")= " + str(parabola(x1)))
    print("f(" + str((x2)) + ")= " + str(parabola(x2)))
    print("f(" + str((x3)) + ")= " + str(parabola(x3)))

tol = 0.0001
nsteps = 0
while abs(parabola(x2)) > tol:
    print("Not the root. Choose new x1 and x2 values.")
    x1 = float(input("choose new x1: "))
    x3 = float(input("choose new x3: "))
    x2 = 0.5 * (x1 + x3)
    while parabola(x1) > 0:
        print ("f(x1) > 0, choose another value.")
        x1 = float(input("new X1: "))
    while parabola(x3) < 0:
        print ("f(x3) < 0, choose another value.")
        x3 = float(input("new X3: "))
    nsteps += 1
    print("f(" + str((x1)) + ")= " + str(parabola(x1)))
    print("f(" + str((x2)) + ")= " + str(parabola(x2)))
    print("f(" + str((x3)) + ")= " + str(parabola(x3)))
print(f"Root found after {nsteps} steps.")
print("Root at x =", x2)
print("f(x) =", parabola(x2))

# Plot a graph of nsteps versus the logarithm of tol
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
ax1.plot(x, parabola(x), "g")
ax1.plot(x, 0.0 * x, "r")
ax1.plot(x2, parabola(x2), 'ro', label='Root 1')
ax1.set_ylabel('y')
ax1.set_xlabel('x')
ax1.set_title("Parabola and Root 1")

ax2.plot(range(nsteps + 1), [np.log10(tol)] * (nsteps + 1), "b")
ax2.set_xlabel("Number of Steps")
ax2.set_ylabel("log10(tol)")
ax2.set_title("Number of Steps vs log10(tol)")
fig.savefig('full_figure1.pdf')

plt.legend()
plt.show()

# FIND ROOT 2

print("f(x) = x^2 - 4x + 3")
print("Find root 2")
# valores x1 y x3 en input
x11 = float(input("X1: "))
x33 = float(input("X3: "))

# respectivos valores de y de x1 y x3

while parabola(x11) > 0:
    print ("f(x1) > 0, choose another value.")
    x11 = float(input("new X1: "))
while parabola(x3) < 0:
    print ("f(x3) < 0, choose another value.")
    x33 = float(input("new X3: "))
x22 = 0.5 * (x11 + x33)

# valores de y
print("f(" + str((x11)) + ")= " + str(parabola(x11)))
print("f(" + str((x22)) + ")= " + str(parabola(x22)))
print("f(" + str((x33)) + ")= " + str(parabola(x33)))
```

```
x = np.arange(0,5,0.2)
y= parabola(x)
plt.plot(x,y,"g")
plt.plot(x,0.0*x,"r")
plt.plot(x22, parabola(x22), "yo")
plt.xlabel('x')
plt.ylabel('y')
plt.savefig('Ex. 1 Root 2 part 1.pdf')
plt.show()

# updated values de x1 y x3
if parabola(x22) > 0 or parabola(x22) < 0 :
    choose = input(" f(x2) >= 0, change x1 or x3? ")
    if choose == "x1":
        x11 = float(input("choose new x1: "))
    else:
        x33 = float(input("choose new x3: "))
    x22 = 0.5 * (x11 + x33)
    print("f(" + str((x11)) + ")= " + str(parabola(x11)))
    print("f(" + str((x22)) + ")= " + str(parabola(x22)))
    print("f(" + str((x33)) + ")= " + str(parabola(x33)))

tol = 0.0001
nsteps = 0
while abs(parabola(x22)) > tol:
    print("Not the root. Choose new x1 and x2 values.")
    x11 = float(input("choose new x1: "))
    x33 = float(input("choose new x3: "))
    x22 = 0.5 * (x11 + x33)
    while parabola(x11) > 0:
        print ("f(x1) > 0, choose another value.")
        x11 = float(input("new X1: "))
    while parabola(x33) < 0:
        print ("f(x3) < 0, choose another value.")
        x33 = float(input("new X3: "))
    nsteps += 1
    print("f(" + str((x11)) + ")= " + str(parabola(x11)))
    print("f(" + str((x22)) + ")= " + str(parabola(x22)))
    print("f(" + str((x33)) + ")= " + str(parabola(x33)))

print(f"Root found after {nsteps} steps.")
print("Root at x =", x22)
print("f(x) =", parabola(x22))

# Plot a graph of nsteps versus the logarithm of tol
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
```

```
ax1.plot(x, parabola(x), "g")
ax1.plot(x, 0.0 * x, "r")
ax1.plot(x22, parabola(x22), "bo")
ax1.set_ylabel('y')
ax1.set_xlabel('x')
ax1.set_title("Parabola and Root 2")

ax2.plot(range(nsteps + 1), [np.log10(tol)] * (nsteps + 1), "b")
ax2.set_xlabel("Number of Steps")
ax2.set_ylabel("log10(tol)")
ax2.set_title("Number of Steps vs log10(tol)")
fig.savefig('full_figure2.pdf')

plt.legend()
plt.show()
```

Exercise 2:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 12 14:41:37 2024

@author: watanabv
"""
import numpy as np
import matplotlib.pylab as plt
a = 1
b = 3
c = -10
def parabola(x):
    y = a*x**2 + b*x + c
    return y
def derivative(x):
    yprime=  2*a*x + b
    return yprime

x1 = 1

x1 = x1 - (parabola(x1) / derivative(x1))

tol = 0.00000000001
nsteps1 = 0
while abs(parabola(x1)) > tol:

    x1 = x1 - (parabola(x1) / derivative(x1))
    nsteps1 += 1

print("Root 1: x1 = " + str(x1))
print("f(x1) = " + str(parabola(x1)))
x2 = -4
x2 = x2 - (parabola(x2)/derivative(x2))
nsteps2 = 0
while abs(parabola(x2)) > tol:
    x2 = x2 - (parabola(x2) / derivative(x2))
    nsteps2 += 1

print("Root 2: x2 = " + str(x2))
print("f(x2) = " + str(parabola(x2)))

x = np.arange(-7,5,0.2)
y= parabola(x)
yprime = derivative(x)

x = np.arange(-7,5,0.2)
y= parabola(x)
yprime = derivative(x)
plt.plot(x,yprime,"g")
plt.plot(x,y,"b")
plt.plot(x,0.0*x,"k")
plt.plot(x1, parabola(x1), "yo")
plt.plot(x2, parabola(x2),"yo")
plt.suptitle("Parabola and Roots")
plt.xlabel("x")
plt.ylabel("y")
plt.savefig('Ex 2 plot 1.pdf')
plt.show()

fig, (ax1, ax2) = plt.subplots(2, 1, layout = "constrained")
ax1.plot(range(nsteps1), [tol]*nsteps1, "b")
ax1.set_xlabel("Number of Steps")
ax1.set_ylabel("tol")
ax1.set_title("Number of Steps vs tol, root 1")

ax2.plot(range(nsteps2), [tol]*nsteps2, "b")
ax2.set_xlabel("Number of Steps")
ax2.set_ylabel("tol")
ax2.set_title("Number of Steps vs tol, root 2")
fig.savefig('Ex2 plot2.pdf')

plt.show()

Exercise 3:
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
Created on Sun Feb 13 20:13:09 2024
@author: vwitch
"""
import numpy as np
import matplotlib.pyplot as plt
# Constants
e = 1.44
A = 1090
p = 0.033
# function to evaluate V(x)
def V(x):
    return A * np.exp(-x / p) - (e/x)

# func to evaluate the force F(x)
def force(x):
    return (A/p)*(np.exp(-x/p))-(e/(x**2))

# func to evaluate the second derivative of V(x)
def second_derivative_v(x):
    return  (A / p**2) * np.exp(-x / p) - (2*e/(x**3))

# Range of x values
x_values = np.linspace(0.01, 1.00, 100)

# v plot
plt.figure(figsize=(10, 6))
plt.plot(x_values, V(x_values), 'r', label='V(x)')
plt.xlabel('x (nm)')
plt.ylabel('V(x) (eV)')
plt.title('Potential Energy vs. Separation Distance')
plt.legend()
plt.savefig('Ex.3 fig 1.pdf')
plt.show()
# force plot
plt.figure(figsize=(10, 6))
plt.plot(x_values, force(x_values), 'b', label='F(x)')
plt.xlabel('x (nm)')
plt.ylabel('F(x) (eV/nm)')
plt.title('Force vs. Separation Distance')
plt.legend()
plt.savefig('Ex.3 fig 2.pdf')
plt.show()
x1 = 0.19
#print(force(x1))
while np.abs(force(x1)) > 0.0000000001:
    x1 = x1 - (force(x1)/second_derivative_v(x1))
print(x1)
```