A Study of Newton-Raphson Approximation and its Efficiency Versus the Bisection Method

David Lawton

4th October 2023

Contents

1 Abstract

The main objective of this computational experiment, to show the efficiency of the Newton-Raphson method versus the bisection method, was successful. For the same function, from the same points, the N-R approximation reached a similarly accurate, if not more accurate results in a much shorter number of steps.

Its usefulness in context was also successfully shown, as it was used to approximate the equilibrium distance of two ions from the equation of their potential energy.

2 Introduction

In this introduction I will explain the basic theory of the experiment, and outline what was required and tested.

Firstly I will introduce the quadratic equation on which I compared the efficiency of the methods

$$-2x^2 + 12x - 6. \tag{1}$$

The first section of the experiment involved using the bisection method to approximate the roots of the above function. The bisection method can be outlined by the equation

$$x_2 = \frac{x_3 - x_1}{2}, \tag{2}$$

which is used with a combination of 'while' and 'if' functions over a countable number of steps to find an approximate value of the roots.

The second section involved using Newton-Raphson method to approximate roots for the same equation. The Newton-Raphson method for approximation can be outlined using one equation.

$$x_{i+1} = x_i - \frac{\mathrm{f}(x_i)}{\mathrm{f}'(x_i)} \tag{3}$$

This equation will always find an approximate root from any x-value, for any $\mathrm{f}(x)$, with accuracy increasing with the number of steps taken.
The final section used the Newton-Raphson Approximation to approximate the equilibrium distance between two oppositely charged ions, Na^+ and Cl^-, whose potentials due to each other is dictated by the equation below.

$$\mathrm{V}(x) = A(\exp[-x/p]) - \frac{(e^-)^2}{4\pi\varepsilon_0 x} \tag{4}$$

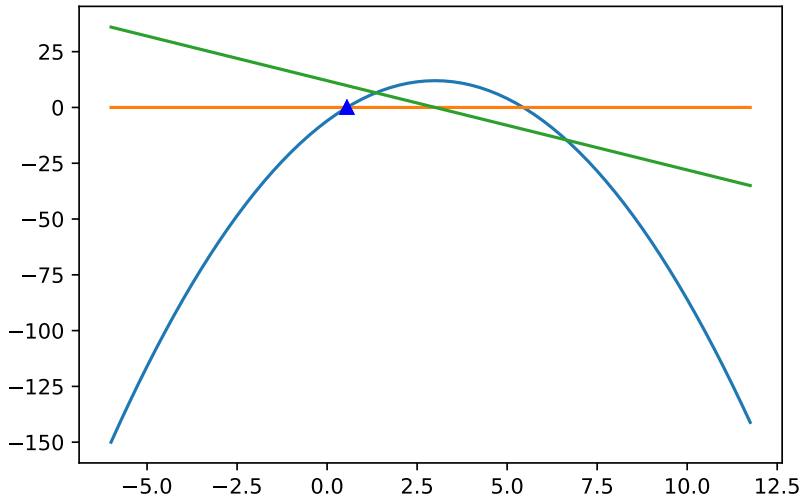

Figure 1: A graph of the function used in the experiment, its derivative and the approximated roots.

3 Methodology

3.1 Method 1

For the first section of the experiment, the method involved choosing two points (x_1, x_3) for each root, which were on either side of the root and the y-axis. Then, depending on whether the function of the halfway point, x_2 from Eq. 2, is greater or less than zero, the point on that side of the axis was set to the value of x_2.

This process is repeated inside a while loop untill a value of f(x_2) within a very small, predetermined, range of the x-axis is produced. this entire process is run inside a for loop so that the entire process is run for all values of tolerance. This is done so that efficiency can be compared to the second method. An example of a graph resulting from Method 1 is Figure 1.

3.1.1 Method 1 Code

```
for c in tollist:
    nsteps1 = 0
    while quadf(x2) != 0:
        nsteps1 += 1

        if quadf(x2) > 0:
            x3 = (x2)
            x2 = 0.5 * (x1 + x3)

        elif quadf(x2) < 0:
            x1 = x2
            x2 = 0.5 * (x1 + x3)

        if np.sqrt((quadf(x2))**2) < c:
            nsteps1list.append(nsteps1)
            root1list.append(x2)
            y1list.append(quadf(x2))
            break
    x1 = a
    x3 = b
```

3.1.2 Method 1 Efficiency

The efficiency of method 1 was measured by using a variable *nsteps1* which was used as a counter inside the while loop to keep track of the number of steps required. It was then added to a list of values, one for each tolerance and graphed against the tolerance as shown in Figure 2. As seen in Figure 2, it took between 15 and 40 steps to find a sufficiently accurate root, with steps increasing with required accuracy.

3.2 Method 2

The next section of the experiment dealt with Newton-Raphson Approximation on the same function used in method 1, which is shown in Figure 1. This method makes use of Eq. 3, which was introduced earlier. This function defines the next step of the function as an x-value, which is always closer to the root, due to the nature of the equation, which is based on the Taylor series expansion of a function.

$$f(x+h) = f(x) + hf'(x) + \mathrm{O}(h^2) \tag{5}$$

This method was done by continuously redefining a variable x_1 with respect to itself:

$$x_1 = x_1 - \frac{\mathrm{f}(x_1)}{\mathrm{f}'(x_1)} \tag{6}$$

where $\mathrm{f}(x_1)$ is defined as Eq. 1.

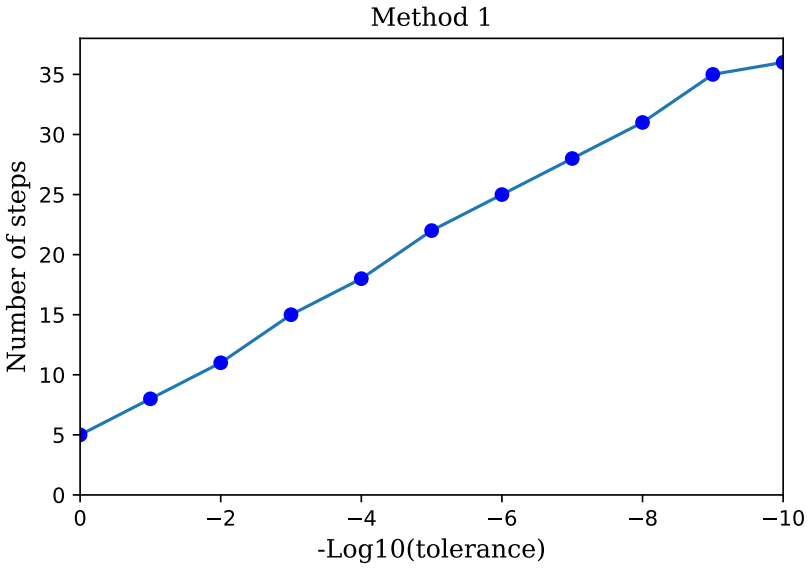

Figure 2: A graph of the steps taken against the logarithm of the tolerance for Method 1.

3.2.1 Method 2 Code

```
for c in tol:
        nsteps1 = 0
        nsteps2 = 0
        x1 = d1
        while abs(quadf(x1)) > c:
                x1 = x1 - (quadf(x1)/quadfderiv(x1))
                nsteps1 += 1
        roots.append(x1)
        Nsteps1.append(nsteps1)
```

3.2.2 Method 2 Efficiency

Similar to the first method, the efficiency of the Newton-Raphson approximation method was measured by using a variable, *nsteps2* which is appended to a list, and then resetted to 0, to count the steps required for each tolerance value. The values of *nsteps2* are then plotted against tolerance as in Figure 3.

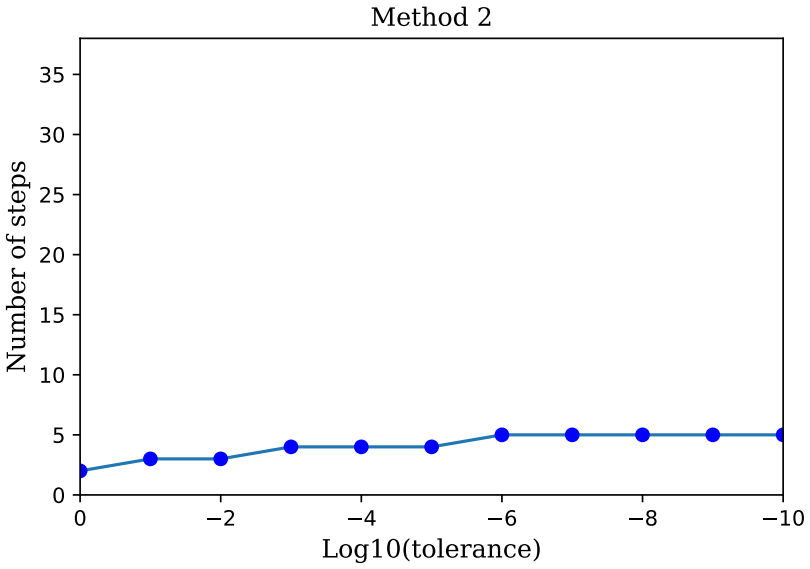

Figure 3: A graph of the steps taken against the logarithm of the tolerance for Method 2

3.3 Using Method 2 on a function V(x)

In the final section of this experiment, we used the Newton-Raphson approximation method laid out in (3.2) to estimate the equilibrium distance between two charged ions, Na^+ and Cl^-, with the function shown in Figure 4.

This was done by first manually finding the first and second derivatives of V(x), V$'(x)$ and V$''(x)$. Then defining them as functions in the code. This is because the equilibrium of V(x) is the point at which V$'(x) = 0$. Therefore the aim of our process is to find the root of V$'(x)$.

Next a while loop was created inside a loop $\forall$ values of tolerance. This loop approximated values for the roots of V$'(x)$, therefore approximating x-values for the equilibrium value of x for V(x), as in Figure 5.

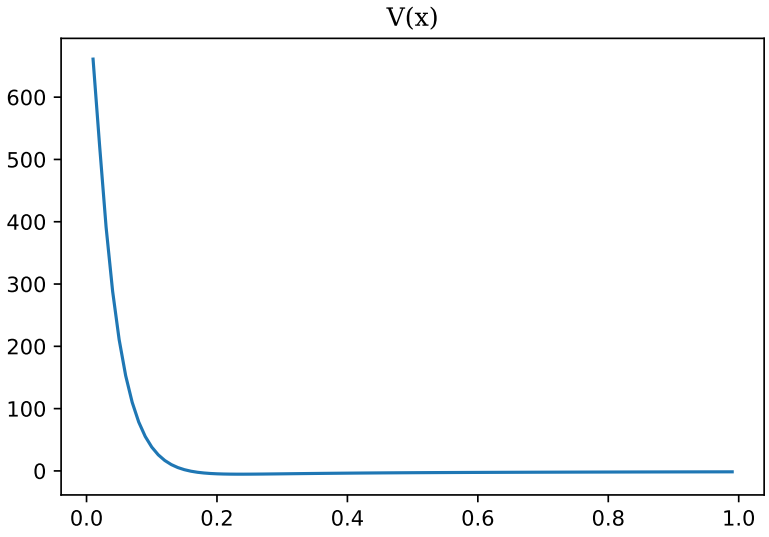

Figure 4: Graph of potential energy V(x) between two charged ions

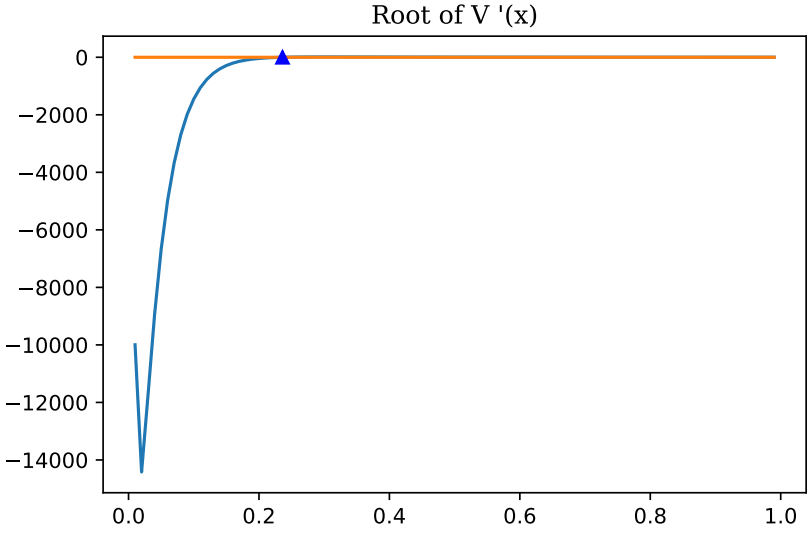

Figure 5: Graph of V'(x) and its root

4 Results

The results of the experiment were succesful in using and showing both the bisection and Newton-Raphson methods for approximation, as shown in Figure 1 and Figure 5. It was also succesful in graphing the number of steps taken for each tolerance value in each experiment. Below is a table of representative data for Method 1:

Tolerance	x-Value	y-Value	No. of Steps
10^-10	0.5505102572242322	7.260592127522614e-11	36
10^-6	0.5505103066557355	4.844004424242598e-07	25
10^-4	0.5505171352820144	6.73909059409894e-05	18
10^-2	0.5513912336117843	0.008630218333561146	11
1	0.6026303346803168	0.5052373756099806	5

It is clear from both this table and Figure 2 that as accuracy improves, the number of steps increases significantly.

In comparison the results of the Newton-Raphson method, shown in Figure 3 and the table below, take **significantly** less steps, and these steps increase at a **significantly** lower rate.

Tolerance	x-Value	y-Value	No. of Steps
10^-10	0.5505102572168157	-6.128431095930864e-14	5
10^-6	0.5505102572168157	-6.128431095930864e-14	5
10^-4	0.5505100825886116	-1.7110001007125675e-06	4
10^-2	0.5495851501312228	-0.009065872914844064	3
1	0.48225806451612896	-0.6780489073881384	2

It can also be seen above that the y-values of the Newton-Raphson approximation are much closer to a zero value.
For the results of the third section, accurate values for the single root of V(x) were found and are displayed in the table below, along with the value of V'(x) at the corresponding x.

Tolerance	x-Value	V'(x)-value
10^-10	0.23605384841577945	3.552713678800501e-15
10^-6	0.23605384833382692	-4.623427685146453e-08
10^-4	0.23605384833382692	-4.623427685146453e-08
10^-2	0.2360517473864578	-0.0011853614945245283
1	0.23571655310011616	-0.191484214390659

It is important to remember for this part that V'(x) is equal to the negative of the force being applied to the particle. At the roots of V'(x), it follows that the force equals zero, as it should at the equilibrium point.

5 Conclusion

To conclude, it is clear that the Newton-Raphson method for approximation is both very accurate and very efficient, and is clearly a powerful computational tool for both maths and physics. As for how it compares to the bisection method for approximation, it is undeniably **significantly** more accurate **and** more efficient. In the example used the Newton-Raphson method was more accurate for nearly every tolerance value.
It was both fast and efficient in its approximation of a quite complex function V(x).

6 Bibliography

2023. 2nd year Physics Lab Manual. School of Physics, Trinity College Dublin, Dublin 2.

7 Appendix

Some slight edits were made to some arbitrary lists, file locations and comments so that the code would fit on the report.

7.1 Method 1 Code(complete)

```
import numpy as np
import matplotlib.pyplot as plt

x=np.arange(-6.0, 12, 0.25)

a = -2
b = 12
c = -6

x1 = -2.0
x3 = 2.5
x5 = 7.0

def quadf(x): return a*(x**2) + b*x + c

y1 = quadf(x1)
y3 = quadf(x3)
y5 = quadf(x5)

if y1 < 0 and y3 > 0 and y5 < 0:
    print ('values correctly initialised')
```

```
#Here I define my arrays and list used in the function.
nsteps1list = []
nsteps2list = []
tollist = [10**-10, 10**-9, 10**-8, ... , 10**-3, 10**-2, 10**-1, 1]
root1list = []
y1list = []
root2list = []
y2list =[]

#Below I define the functions which return
# the required roots, lists + arrays.
def root1tol(a, b):
    x1 = a
    x3 = b
    x2 = 0.5 * (x1 + x3)


    for c in tollist:
        nsteps1 = 0
        while quadf(x2) != 0:
            nsteps1 += 1

            if quadf(x2) > 0:
                x3 = (x2)
                x2 = 0.5 * (x1 + x3)

            elif quadf(x2) < 0:
                x1 = x2
                x2 = 0.5 * (x1 + x3)

            if np.sqrt((quadf(x2))**2) < c:
                nsteps1list.append(nsteps1)
                root1list.append(x2)
                y1list.append(quadf(x2))
                break
        x1 = a
        x3 = b

def root2tol(a, b):
    x3 = a
    x5 = b
    x4 = 0.5 * (x3 + x5)
    nsteps2tol = 0
    for c in tollist:
        while quadf(x4) != 0:
```

```
            nsteps2tol += 1

            if quadf(x4) > 0:
                x3 = (x4)
                x4 = 0.5 * (x3 + x5)

            elif quadf(x4) < 0:
                x5 = x4
                x4 = 0.5 * (x3 + x5)

            if abs(quadf(x4)) < c:
                nsteps2list.append(nsteps2tol)
                root2list.append(x4)
                y2list.append(quadf(x4))
                break
        x3 = a
        x5 = b

root1tol(-2.0, 2.5)
print(root1list)
print(y1list)
print(tollist)
print(nsteps1list)
root1 = root1list[0]
print (root1)

root2 = root2tol(2.5, 7)
root2 = root2list[0]
print (root2)

#The below plot shows the quadratic function
plt.plot(x, a * x * x + b * x + c)
plt.plot(x, 0.0 * x)
plt.plot(root1, quadf(root1), 'go')
plt.plot(root2, quadf(root2), 'go')
plt.savefig('/home/dj-lawton/.../quadratic_graph_Ex1.pdf')
plt.show()

#The below plots show the relation between
# the accuracy and the no. of steps.
font = {'fontname':'serif', 'size':12}
plt.figure()
plt.ylim(0, 38)
plt.xlim(0, -10)
plt.title('Method 1', fontdict=font)
```

```
plt.xlabel("-Log10(tolerance)", fontdict=font)
plt.ylabel("Number of steps", fontdict=font)
plt.plot(np.log10(tollist), nsteps1list)
plt.plot(np.log10(tollist), nsteps1list, 'bo')
plt.savefig('/home/dj-lawton/.../nstepstolgraph.pdf')
plt.show()
plt.close()
```

7.2 Method 2 Code(complete)

```
import numpy as np
import matplotlib.pylab as plt


x=np.arange(-6.0, 12, 0.25)

a = -2
b = 12
c = -6

def quadf(x): return a*(x**2) + b*x + c
def quadfderiv(x): return 2 * a * x + b


x1 = -2
x2 = 7

def init(e1, e2):
    if quadf(e1)*quadf(e2)>0:
        print("values correctly initialised")
    else:
        print('pick new values')


tol =[10**-10, 10**-9, 10**-8, ... , 10**-3, 10**-2, 10**-1, 1]
roots1=[]
Nsteps1=[]
Nsteps2=[]
yvalues=[]
roots2=[]
def NRapprox(d1, d2):

    for c in tol:
        nsteps1 = 0
        nsteps2 = 0
```

```
        x1 = d1
        while abs(quadf(x1)) > c:
            x1 = x1 - (quadf(x1)/quadfderiv(x1))
            nsteps1 += 1
        roots1.append(x1)
        Nsteps1.append(nsteps1)
        yvalues.append(quadf(x1))
        #secondroot approximation starts on other side of maxima
        x2 = d2
        while abs(quadf(x2)) > c:
            x2 = x2 - (quadf(x2)/quadfderiv(x2))
            nsteps2 += 1
        roots2.append(x2)
        Nsteps2.append(nsteps2)

init(x1, x2)
NRapprox(-2, 7)
print(roots1)
print(Nsteps1)
print(yvalues)
#The below plot shows the quadratic function
plt.figure()
plt.plot(x, quadf(x))
plt.plot(x, 0.0 * x)
plt.plot(roots1[0], quadf(roots1[0]), 'b^')
plt.plot(roots1[1], quadf(roots1[1]), 'b^')
plt.plot(x, quadfderiv(x))
plt.savefig('/home/dj-lawton/.../quadratic_graph_Ex2.pdf')
plt.show()
plt.close()


font = {'fontname':'serif', 'size':12}
plt.figure()
plt.ylim(0, 38)
plt.xlim(0, -10)
plt.title('Method 2', fontdict=font)
plt.xlabel('Log10(tolerance)', fontdict=font)
plt.ylabel('Number of steps', fontdict=font)
plt.plot(np.log10(tol), Nsteps1)
plt.plot(np.log10(tol), Nsteps1, 'bo')
plt.savefig('/home/dj-lawton/.../Tol_vs_steps_Ex2.pdf')
plt.show()
plt.close()
```

7.3 V(x) Exercise Code(complete)

```
import numpy as np
import matplotlib.pyplot as plt

A = 1090.0
b = 1.44
p = 0.033

x = np.arange(0.01, 1, 0.01)

def V(x):
    return A*((np.e)**(-x/p))-(b/x)

#to find min/max of func, find roots of derivative
#Vdot = -F
def Vdot(x):
    return (-A/p)*((np.e)**(-x/p))+(b/(x**2))

def Vdoubledot(x):
    return (A*((np.e)**(-x/p)))/(p**2)-(2*(b/(x**3)))

x1 = 0.04


tol = [10**-10, 10**-9, 10**-8,... , 10**-3, 10**-2, 10**-1, 1]
roots=[]
Nsteps1=[]
equilofV=[]

def NRapprox(d1):

    for c in tol:
        nsteps1 = 0
        x1 = d1
        while abs(Vdot(x1)) > c:
            x1 = x1 - (Vdot(x1)/Vdoubledot(x1))
            nsteps1 += 1
        roots.append(x1)
        Nsteps1.append(nsteps1)
        equilofV.append(Vdot(x1))

    #The value of the force at the min. value of V(x) is 0,
    #as it is the point where the attractive and
    #repulsive forces are equal.
    #from observation, there is no second root.
```

```
NRapprox(0.04)
print(roots)
print(Nsteps1)
print(equilofV)



font = {'fontname':'serif', 'size':12}
plt.figure()
plt.title('Root of V \'(x)', fontdict = font)
plt.plot(x, Vdot(x))
plt.plot(x, 0.0 * x)
plt.plot(roots[0], Vdot(roots[0]), 'b^')
plt.savefig('/home/dj-lawton/.../V\'(x)_graph_Ex3.pdf')
plt.show()
plt.close()


plt.figure()
plt.title('V(x)', fontdict = font)
plt.plot(x, V(x))
plt.savefig('/home/dj-lawton/.../V(x)_graph_Ex3.pdf')
plt.show()
plt.close()

plt.figure()
plt.title('V \'\'(x)', fontdict = font)
plt.plot(x, Vdoubledot(x))
plt.savefig('/home/dj-lawton/.../V\'\'(x)_graph_Ex3.pdf')
plt.show()
plt.close()
```