II. Root Finding using Newton's Method

Kelsey K, Kailey T, Valerie K

April 27, 2024

Abstract

Contents

1 Introduction

Newton's Method is a way to find the roots of a non-linear function. He used a laborious algebra procedure to first showcase this method, surprisingly the 'Father of Calculus' did not derive nor see this as a calculus process (Ympa 1995). This procedure is also known as the Newton-Raphson Method as it was later revised (CalcWorkshop). This method is significant as it is an iterative process that allows for the answers to non-linear equations to be very precise and accurate. This process uses tangent lines to find the root of an equation. First, choose an x value then take the derivative of the function and plug the x value in. Keep going until there is convergence shown by $x_{n+1} = x_n$. When this happens, x_{n+1} is the root.

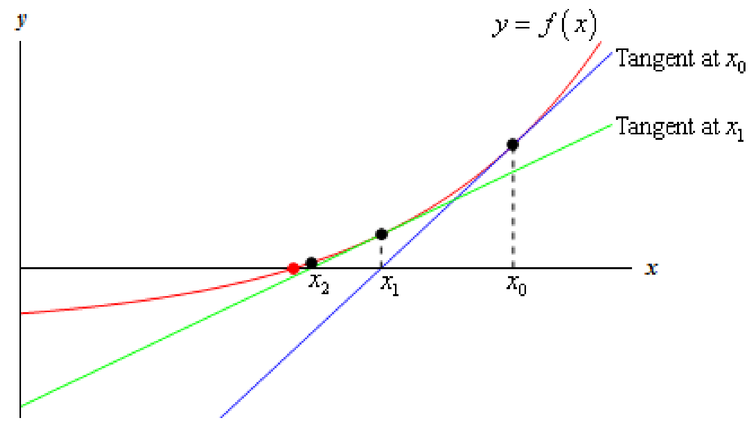

Figure 1: Graph of Newton's Method (Dawkins 2023)

The sample equation that was used by Newton to test this is $x^3 - 2x - 5 = 0$, also known as "Newton's Cubic." We will be exploring this function later on as well as other applications of this method.

2 Problem Statement

Our problem was divided into three different subsections of problems, referred to as "Part A", "Part B", "Part C". All use a form of using Newton's Method for finding their respective answer.

2.1 Part A

The task is to write a function "newton()" to implement Newton's method for finding the root of the given function F(x). The given parameters for Part A are to set TOL = 10^{-12} and find the real root of the "Newton's cubic" function $x^3 - 2x - 5 = 0$. Additionally, the program must output the initial guess (x_0), tolerance (ϵ), number of iterations (n), the root (x_n), and the residual ($|F(x_n)|$). The values of $F(x_n)$ and the derivative $F'(x_n)$ must be computed in a single function or two separate functions.

2.2 Part B

In a building, two intersecting halls with widths w1 = 9 feet and w2 = 7 feet meet at an angle = 125○, as shown below. Assuming a two-dimensional situation (i.e., ignoring the thickness of the board), what is the longest board that can negotiate the turn? [Hints: Express length of the board l = l1 + l2 in terms of the angle , then find the maximum of the function l() by solving the nonlinear equation l() = 0.]

2.3 Part C

President Barry Butler has been found dead in his office. At 8:00 pm, the county coroner determined the core temperature of the corpse to be 90○F. One hour later, the core temperature had dropped to 85○F. Maintenance reported that the building's air conditioning unit broke down at 4:00 pm. The temperature in President Butler's office was 68○F at that time. The computerized climate control system recorded that the office temperature rose at a rate of 1○F per hour after the air conditioning stopped working. Dr. Radosta believes that Dr. Sam killed President Butler. Dr. Sam, however, claims that he has an alibi. Dr. Sam was getting a spicy chicken sandwich at Chick-fil-A in the student union. The surveillance cameras show Dr. Sam walking into the SU at 6:35 pm, and Chick-fil-A staff confirm that Dr. Sam was eating a spicy chicken sandwich in the SU from 6:40 pm to 7:15 pm. Could Dr. Sam have killed President Butler?

3 Method/Analysis

3.1 Part A Subsection

For Part A we first hard-coded the initial guess and tolerance as 1 and 10^{-12} respectively. Additionally, we hard coded Newton's Cubic function

$$x^3 - 2x - 5 = 0 \tag{1}$$

as a function and the next roots to be tested (x_n1) were initialized as 0. It was then decided to make our 'Newtons_root' function able to be reused with different equations, so sympy was imported and made x_n the symbol 'x'. In the function 'fn' that was created with arguments of f, x_n, and x we returned f.subs(x_n,x) which allows for x to become a variable again. Next, the function fn_deriv was created with the arguments f, x_n, and x to take the limit definition of a derivative of any equation. It returned

$$(fn(f, x\_n, x + h) - fn(f, x\_n, x - h))/(2h) \tag{2}$$

with h being the limit of 1e-6.

'Newtons_root' function was then created with a for loop to run the iterations of guesses through to get where the root converged. This would happen when $|fx| < tol$. When this condition was met, the function printed how many iterations it took to converge, the root value, the error residual, and the amount of time the code took to run. If the condition was not met, the new guess was defined as $x - (fx/f_prime)$ and the iteration number, guess, and value of f(x_n) were printed out.

3.2 Part B Subsection

For Part B, we used the information of the hallway, such as the widths given (7 and 9), angles (alpha, beta, and gamma), and board lengths (l_1 and l_2), to create an equation that would be able to be used in Newton's Root Finding Theorem from Part A. First, we created sine functions, involving the opposite over hypotenuse values. These are the original equations found:

$$sin(\gamma) = \frac{7}{l_2} \tag{3}$$

$$sin(\beta) = \frac{9}{l_1} \tag{4}$$

Then, both equations needed to be in terms of γ. Since $\alpha = 125$ degrees, that means $\beta = 180 - 125 - \gamma$, which is $55 - gamma$. Then, we needed to solve for l_1 and l_2 because that would help solving for l. Here, we got these two equations:

$$l_2 = \frac{7}{sin(\gamma)} \tag{5}$$

$$l_1 = \frac{9}{sin(55 - \gamma)} \tag{6}$$

Since we needed to eventually find the maximum value of the board, we solved for l. Because $l = l_1 + l_2$:

$$l = \frac{9}{sin(55 - \gamma)} + \frac{7}{sin(\gamma)} \tag{7}$$

Next, since python works better with radians rather than degrees, we converted 55 degrees to 0.9599 radians and replaced that into the equation.

Now that we had an equation in terms of γ and solved for the length of the board, we needed to take the derivative of the function. The reason we needed to do this is because finding a maximum (or minimum) value of a function is finding wherever the derivative equals to zero. Newton's Root Finding can only solve for when a function itself equals zero, not the derivative. Because of this, we took the derivative by hand and hard-coded it into the function to avoid any other derivative issues that we encountered before. The code that we implemented directly into Part A, in the equation labeled "f" was:

$$f = \frac{-9cos(0.9599 - x)}{(sin(0.9599 - x))^2} - \frac{7cos(x)}{(sin(x))^2} + \frac{7}{sin(x)} \tag{8}$$

Unfortunately, when we plugged this function in, the limit definition of a derivative was not working properly, so we changed this part of the code to taking a derivative using the Sympy library and using the "sp.diff" function. The reason we did not originally want to use this method in the first place was because we would have to create symbols for the variables and then use "sp.subs" to substitute the symbols with a value again. We thought it would be easier to implement the limit definition instead, but had to change it for Part B.

After doing this, we ran into another issue with the convergence of the actual Root Finding. After graphing the function, we realized the problem was our initial guess. The function had infinite root values, including a vertical line close to x=1, which was our initial guess. This caused the code to create tangent lines closer to this value and keep going through iterations since it never truly converged. Even though the function had a root in this location, the code was unable to process convergence efficiently because the tangent lines were vertical and x was not getting any closer to the tolerance value. After noticing this issue, we changed the initial guess to 0.5, which fixed our problem.

After finally fixing all of the problems we came across, the code ran completely through, giving us a value of 0.451 radians and only took 4 iterations to run in 0.00839 seconds.

To check this answer, we graphed the derivative function and checked to see if this was the root of the equation. Looking at this image, we confirmed

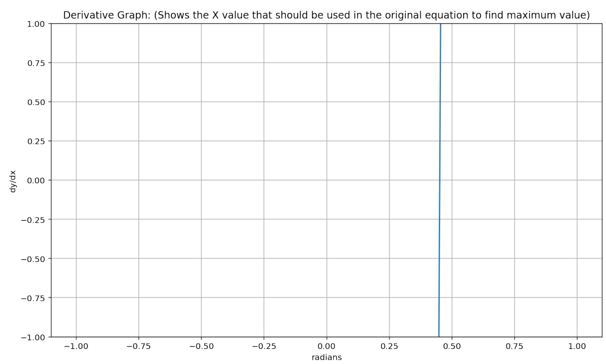

Figure 2: Graph of Derivative of Length Function

that the root value looks to be correct.

However, this value that came through was only the value of the root of the derivative and we needed to find the maximum length of the board. So, to do this, we plugged in that x value back into the original equation:

$$l = \frac{9}{sin(55-\gamma)} + \frac{7}{sin(\gamma)} \tag{9}$$

This would then give us our final answer.

3.3 Part C Subsection

The method used to solve this ravishing murder mystery consists of the application of Newtons Law of Cooling as shown in Equation 10 below.

$$T(t) = T_a + (T_0 - T_a)e^{-kt} \tag{10}$$

where:

$T(t)$= temperature at time t
T_0= initial temperature
T_1= temperature after 1 hour
T_a= ambient temperature
t_0= time at which A/C broke
t_1= time at which body was found
k = cooling constant
t = elapsed time

We are provided the following:

$T(t)$= 98.6°F
T_0= 90°F
T_1= 85°F
T_a= 68°F
t_0=4:00pm
t_1=8:00pm

In the initial problem statement, we are given a starting point in the form of an ordinary differential equation that relates temperature at time of death $T(t)$ to elapsed time t. This ODE is shown below in in Equation 11.

$$\frac{dT}{dt} = -k(T - 72 - t) \tag{11}$$

The following equations show the process of performing necessary integration of the ODE Equation 11 for further use.

$$\frac{dT}{T - 72 - t} = -k\,dt \tag{12}$$

$$\int \frac{dY}{T - 72 - t}\,dT = -k \int dt \tag{13}$$

U-sub $\rightarrow u = T - T_a - t, du = dt$

$$\int \frac{1}{u} = -k \int dt \tag{14}$$

$$\ln|u| = -kt + C_1 \tag{15}$$

$$e^{\ln|T-72-t|} = e^{-kt+C_1} \tag{16}$$

$$|T - 72 - t| = C_2 e^{-kt} \text{ where } C_2 = e^{C_1} \tag{17}$$

$$\boxed{T = t + 72 + C_2 e^{-kt}} \tag{18}$$

Using $T(0) = 90°\text{F}$, $t(0) = 0$:

$$T(0) = 0 + 72 + C_2 e^0 \rightarrow \boxed{C_2 = 18} \tag{19}$$

Using $T(1) = 85°\text{F}$, $t(1) = 1$:

$$T(1) = 1 + 72 + 18e^{-k(1)} \rightarrow \boxed{k = -\ln|2/3|} \tag{20}$$

Final equation:

$$\boxed{T(t) = t + 72 + 12e^{-t}} \tag{21}$$

In order to solve the mystery, we must find t when $T(t) = 98.6\,°F$

The code we created to find elapsed time starts by importing the necessary libraries of time, Numpy, and math and then implements their operations alongside Newton's function as previously defined in Part A. The code is built on the basis of convergence, with tolerance set to $1x10^{-12}$ and iterations set to 1000. Newton's cooling constant k is solved for. After hard coding an initial guess, the for loop starts implementing the guess into Newton's equation. This loop runs until the code reaches a convergence, at which time it will end and print the resulting value for t and the time it took to run the code. Note that variable t is depicted as x_i within the code.

4 Solutions/Results

4.1 Part A Subsection

When our program ran, it found the root of Newton's cubic to be 2.0945. It took 9 iterations for convergence with the tolerance of 10^{-12} in 0.00017 seconds. To check our answer, we graphed Newton's cubic equation and found where the root was on the x-axis.

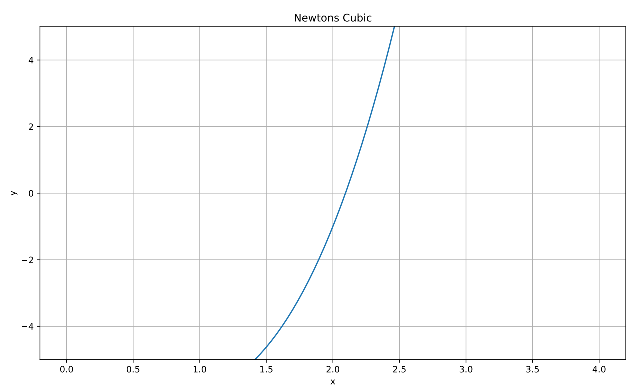

Figure 3: Graph of Newton's Cubic

4.2 Part B Subsection

When plugging in the x value of 0.451 radians into the original equation, we found the maximum length of the board to be 34.532 feet. We checked this calculation by graphing the function and seeing where the maximums and minimums are.

When examining this image, it is clear that the length value at 0.451 radians is around 34 to 35 feet, which confirms our maximum value of the length of the board is correct.

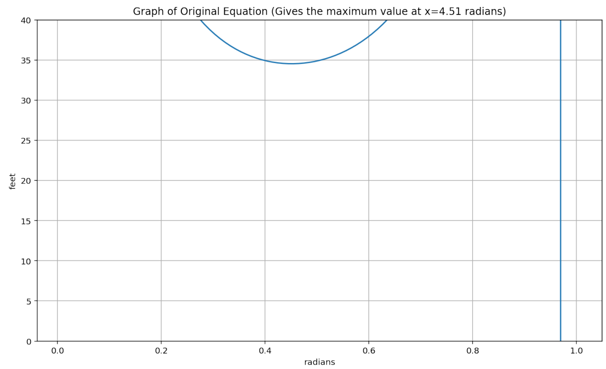

Figure 4: Graph of Length Function

4.3 Part C Subsection

By implementing Newton's Method alongside the pre-existing information that is available on the mystery, elapsed time after death was determined to be 1 hour and 25 minutes. This means that President Butler died at 6:35pm at which time Dr. Sam was seen walking into the Student Union to get a spicy chicken sandwich, therefore proving his innocence. The code took 2.50 x 10^{-4} seconds to run.

5 Discussion/Conclusions

We learned many things from creating this project. Not only can Newton's Root Finding Theorem be used to find roots of basic equations, but it can also be helpful to solve real world scenarios such as the board length and murder mystery problems. The Root Finding Theorem is efficient as well when considering how long it takes to run. If you use an initial guess that is close enough to the actual root value, the code should take plenty less than 1 second. However, that also brings it to its weakness. This code only works for one root at a time and really depends on your initial guess for which root it will converge towards. Sometimes, it may not even converge depending on what the function is which is definitely a flaw in this theorem.

We could attempt to create a way so that the code checks if there are multiple roots, but again this would just result in another flaw. If we were to create a for loop to go until there were no more roots, then some functions would never work, as they have infinite roots, like sine and cosine. Overall, our code was efficient with its time and found appropriate roots, even though there were some small issues. This theorem is still a very good way to find roots within a function.

6 References/Appendix

References

[1] Dawkins, Paul. "Section 4.13: Newton's Method." Calculus I - Newton's Method, 26 Oct. 2023, tutorial.math.lamar.edu/classes/calci/newtonsmethod.aspx.

[2] "Newton's Method." Calcworkshop, 22 Feb. 2021, calcworkshop.com/derivatives/newtons-method/: :text=Newton's

[3] Ypma, Tjalling, "Historical Development of the Newton-Raphson Method" (1995). Mathematics. 93. https://cedar.wwu.edu/math_facpubs/93

A Python Codes

Text introducing this appendix. Subsections and further divisions can also be used in appendices.

```python
#!/usr/bin/env python3
import math

"""
===============================================================================
MA305 - cw #: Kailey Turpening, Kelsey Kressler, Valerie Krys - 04/27/2024
Purpose: Complete Newton's Root Finding Theorem Project.
===============================================================================
"""
#a.write a function to implement Newton's method for finding a root of a given equation
import math
import numpy as np
import sympy as sp
import matplotlib.pylab as plt
import time


#Setting tolerance to given value of 10**-12

tolerance=10**-12

#Creates x_n as the symbol

x_n=sp.symbols('x')

#defining the numerical functions to be used
def fn(f,x_n, x):
    return f.subs(x_n,x) #allows for x to become vaiable

def fn_deriv(f,x_n,x):
    h=1e-6
    #limit definition of derivative to find derivative
    return (fn(f,x_n,x +h)-fn(f,x_n,x-h))/ (2*h)

#Newton's cubic function
f=x_n**3-2*x_n -5
#initial guess
x=1
x_n1=0

```

```
#defining Newton's Root function
def Newtons_root(f, x_n, x, tol, max_iter=1000):
    for n in range(max_iter):   #loops to get root
        start = time.time()     #starts timer
        fx = fn(f,x_n,x)
        #taking derivative
        f_prime = fn_deriv(f,x_n,x)

      #checking for convergence, making sure the value iterated is less than the tolerance. If not, goes back through for loop.
        if abs(fx) < tol:
            #How many iterations it took to converge/ have value be lower than the tolerance.
            print(f"Converged after {n} iterations.")
            #Prints the root value.
            print("Root = ", x)
            #Prints the error.
            print("Residual (|f(x_n)|):", abs(fx))
            #Finds the amount of time it took for the code to run through.
            end = time.time()
            print(f"Time it took to run the code:{end-start} seconds")
            return x

        x_n1=x-(fx/f_prime) #to get the new guess for the iterations.
        print(f"Iteration {n}: x_n = {x}, F(x_n)= {fx}") #Prints out the number of iterations and the values found.

        x = x_n1

Newtons_root(f, x_n, x, tolerance)


#graphing Newton's Cubic
fig=plt.figure()
x1=plt.linspace(0,4,100)
y1=x1**3-2*x1 -5
plt.plot(x1,y1)
plt.ylim(-5,5)
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.title('Newton''s Cubic')
plt.show()
fig.savefig('Newtons_Cubic.pdf')
```

```python
tolerance=10**-12

#Making the variable a symbol so the derivative can be taken using sp.diff
x_n=sp.symbols('x')

#defining the functions to be used, substituting x_n in for an actual value
def fn(f,x_n, x):
    return f.subs(x_n,x)
#Taking derivative, no longer using limit definition of a derivative, using sp.diff
def fn_deriv(f,x_n,x):
    return f.diff(x_n).subs(x_n,x)
#derivative of original function so that maximum length can be found
f=(9*sp.cos(0.9599-x_n))/((sp.sin(0.9599-x_n))**2) - (7*sp.cos(x_n))/(sp.sin(x_n))**2
x=0.5
#x_n1=0

#defining Newton's Root function, as used above in part a
def Newtons_root(f, x_n, x, tol, max_iter=1000):
    for n in range(max_iter):   #loops to get root
        start = time.time()
        fx = fn(f,x_n,x)
        f_prime = fn_deriv(f,x_n,x)

        #checking for convergence
        if abs(fx) < tol:
            print(f"Converged after {n} iterations.")
            print("Root = ", x)
            print("Residual (|f(x_n)|):", abs(fx))
            end = time.time()
            print(f"Time it took to run the code:{end-start} seconds")
            return x

        x_n1=x-(fx/f_prime) #x_n+1 (on top of page)
        print(f"Iteration {n}: x_n = {x}, F(x_n)= {fx}")

        x = x_n1

Newtons_root(f, x_n, x, tolerance)
#Finding the value of the max once the root value of the derivative is found
answer=(9/(math.sin(0.9599-0.451))+(7/(math.sin(0.451))))
print(answer)
```

```python
#Plotting the derivative to show where the root is, and the proper x value to be used
fig=plt.figure()
x1=plt.linspace(-1,1,100)
y1=(9*np.cos(0.9599-x1))/((np.sin(0.9599-x1))**2) - (7*np.cos(x1))/(np.sin(x1))**2
plt.plot(x1,y1)
plt.ylim(-1,1)
plt.xlabel('radians')
plt.ylabel('dy/dx')
plt.grid()
plt.title('Derivative Graph: (Shows the X value that should be used in the original equation to find maximum value)')
plt.show()
fig.savefig('Partb1.pdf')
#Plotting the original function to show the value of the maximum board length
fig=plt.figure()
x2=plt.linspace(0,1,100)
y2=(9/(np.sin(0.9599-x2))+(7/(np.sin(x2))))
plt.plot(x2,y2)
plt.ylim(0,40)
plt.xlabel('radians')
plt.ylabel('feet')
plt.grid()
plt.title('Graph of Original Equation (Gives the maximum value at x=4.51 radians)')
plt.show()
fig.savefig('Partb2.pdf')

T_0=90 #body temperature when found
T_1=85 #body temperature 1 hour after found
T_i=98.6 #body temperature at time of death

#solving for Newton's cooling constant k
def k_solve(k):
    T_hour=T_0+k
    return T_hour-T_1

#initial guess and degree that the temperature is dropping in F
guess=1

#solving using the initial guess
k_guess=k_solve(guess)

def Newtons(guess):
    #tolerance and max iterations, timer starts
    start = time.time()
    tolerance=10**-12
```

```
    iterations=1000

    x_i=guess
   #loop for finding the root (time before 8pm)
    for i in range(iterations):
        y=T_0-T_i+ (k_guess*x_i)
        y_d= x_i * k_guess
        x_i=x_i-(y/y_d)

    #finding convergence and timer ending
    if abs(y)<tolerance:
        end = time.time()
        print(f"Time took to run the code:{end-start}")
        return x_i


print(f"The death occured approximately {abs(round(Newtons(guess),2))} hours before the body was found at 8pm")
```