Proceedings of College of STEM Symposium

PROC(04ACSS2025)003
Clayton State University
05/2025

Implementing Newton's method in python for root-finding in differentiable functions

Christopher Raridan[1], *Spencer* Roberson[2], *and Alexa* Roberts[2]

[1] College of STEM, Department of Mathematics and Decision Science, Clayton State University, Morrow, GA 30260 USA
[2] College of STEM, School of Sciences, Clayton State University, Morrow, GA 30260 USA

Abstract. Newton's Method is an iterative technique that uses tangent lines to approximate roots of real-valued, differentiable functions. This paper explores the method's application as a strategy for finding real roots when exact solutions are difficult to obtain algebraically. The method was implemented in Python and applied to a variety of functions, including a low-degree polynomial, a high-degree polynomial, a trigonometric function, and an exponential function. The approximation process starts by evaluating each function at integer values between –10 and 10. If a function evaluated to zero at any of these values, a root is found immediately. Otherwise, the Intermediate Value Theorem is used to locate intervals where the function changed sign, indicating the presence of a root. One of these values is then selected as an initial approximation for Newton's Method. Using the iterative formula, each initial guess is refined to converge toward the actual root. The method produced accurate approximations across all function types when an initial guess was sufficiently close to the true root, demonstrating the efficiency of Newton's Method in iterative root-finding.

1 Introduction

Newton's Method is used in mathematics, physics, engineering, and computer science when an approximate solution to an equation or optimization problem is needed. We use an iterative form of Newton's Method that approximates the roots of real-valued, differentiable functions [1]. Newton's Method is a very efficient algorithm in this paper. Information about convergence of the method can be found in [2]. We will use the Intermediate Value Theorem to narrow the region in which we will search for a root before employing Newton's Method to search for the root more closely.

Given any function, we will evaluate that function at all integers between -10 and 10. We might find a root at an integer value, and in that case, some of our work will be done. Otherwise, we are looking for consecutive integers where the function's value changes from positive to negative or negative to positive because the Intermediate Value Theorem guarantees that there will be a real root between those integers. We will use one of these integers as the seed value for Newton's Method. This choice of seed value is already within one unit of the solution, which was sufficient for convergence in our examples. More information about the particulars of Newton's Method is provided in the Outline of the Method and the Discussion portions of this paper.

1.1 Formatting the title, authors and affiliations

There are three prominent figures in the development of what is now known as Newton's Method. The method, first published in 1685, bears the name of Sir Isaac Newton (Figure 1 left), but his formulation of the method required significant and tedious computations. The first step toward simplification came in 1690, when Joseph Raphson (Figure 1 center) determined that there was a way to use his original polynomials to aid in the process, and this allowed the method to be executed iteratively. The last piece fell into place in the mid-1700s when Thomas Simpson (Figure 1 right) updated the method to include what we now call differential calculus [3].

Fig. 1. Sir Isaac Newton (left) [4], Joseph Raphson (center) [5], and Thomas Simpson (right) [6] made significant contributions to the production and maintenance of Newton's Method.

1.2 Outline of the Method

To use Newton's Method, we need a function, $f(x)$, its derivative, $f'(x)$, and a starting value, x_0. In the Introduction, we described how we will use the Intermediate Value Theorem to help us pick x_0. Newton's Method uses the tangent line to the graph of $y = f(x)$ at $\big(x_0, f(x_0)\big)$. To find that line, we need to know its slope, which is where $f'(x_0)$ comes into play. The point where the tangent line to f at x_0 intersects the x-axis is the next iterate, x_1. This process is repeated by finding the "next" tangent line using the new x- value, $\big(x_1, f(x_1)\big)$ as well as its x-intercept, x_2 [7]. This process repeats until the difference between x_i and x_{i-1} is within some given tolerance, guaranteeing that consecutive iterates agree to a certain number of decimal places. The sequence of values x_0, x_1, x_2,etc., will almost always converge to the real root that is sought [2].

2 Equations and Mathematics

Four predetermined functions were used to serve as practical examples of how Newton's Method works:

$$f_1(x) = 2x^3 - 7x^2 + 10x - 6 \tag{1}$$
$$f_2(x) = x^8 - 4x^7 + 5x^6 - 17x^2 + 68x - 85 \tag{2}$$
$$f_3(x) = \cos(3x) + \sin(4x) \tag{3}$$
$$f_4(x) = e^x - 4x \tag{4}$$

Eq. 1 is a low degree polynomial function, eq. 2 is a higher degree polynomial function, eq. 3 is a trigonometric function, and eq. 4 is a combination of an exponential and a polynomial function. Equations 1 and 2 are similar, but we wanted to see how increasing the number of possible roots would affect Newton's Method.

The main equation of interest is the one used by Newton's Method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5}$$

where x_n is the first input of interest, $f(x_n)$ is the output of a function, and $f'(x_n)$ is the output of the derivative of a function.

Using equation 5 and applying it to equations 1-4, we can find a sample root for each function:

2.1 Equation 1:

Starting at $x_0 = 1$ since $f(1) < 0$ and $f(2) > 0$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 1 - \left(-\frac{1}{2}\right) = 1.5$$
$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 1.5 - \left(\frac{0}{2.5}\right) = 1.5$$
$$f(1.5) = 0$$

2.2 Equation 2:

Starting at $x_0 = 2$ since $f(1) < 0$ and $f(2) > 0$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 2 - \frac{47}{192} \approx 1.75520833333$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 1.75520833333 - \frac{12.9730762968}{12.9730762968} \approx 1.62541248722$$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} = 1.62541248722 - \frac{1.64305385911}{76.5442339972} \approx 1.60394707078$$

$$x_4 = x_3 - \frac{f(x_3)}{f'(x_3)} = 1.60394707078 - \frac{0.0313286176785}{73.663948116} \approx 1.60352177973$$

$$x_5 = x_4 - \frac{f(x_4)}{f'(x_4)} = 1.60352177973 - \frac{0.0000116462573487}{73.6091934926} \approx 1.60352162151$$

$$f(1.603521) \approx 0$$

2.3 Equation 3 (calculations are in radians):

Starting at $x_0 = 0$ as $f(0) > 0$ and $f(1) < 0$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 0 - \left(\frac{1}{4}\right) = -0.25$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = -0.25 - \left(\frac{-0.109782115934}{4.20612550354}\right) \approx -0.223899$$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} = -0.223899 - \left(\frac{0.002183}{4.367158}\right) \approx -0.224399$$

$$x_4 = x_3 - \frac{f(x_3)}{f'(x_3)} = -0.224399 - \left(\frac{6.81322 \times 10^{-7}}{4.364429}\right) \approx -0.224399$$

$$f(-0.224399) \approx 0$$

2.4 Equation 4:

Starting at $x_0 = 0$ as $f(0) > 0$ and $f(1) < 0$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 0 - \left(\frac{1}{-3}\right) = \frac{1}{3}$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = \frac{1}{3} - \left(\frac{0.06227909}{-2.60438757}\right) \approx 0.357246$$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} = 0.357246 - \left(\frac{0.000402232}{-2.57061186}\right) \approx 0.357403$$

$$x_4 = x_3 - \frac{f(x_3)}{f'(x_3)} = 0.357403 - \left(\frac{1.75 \times 10^{-8}}{-2.570388185}\right) \approx 0.357403$$

$$f(0.357403) \approx 0$$

3 Computer Code

This is a compressed version of the code. For defined functions and iterations, see Appendix 1. For complete code as a single block, see Appendix 2.

```
1 import sympy as sp
2 import pandas as pd
3 import numpy as np
4 from sympy import symbols, exp
5 x = sp.symbols('x')
6 def f1(x): # 2x^3-7x^2+10x-6
8 def f2(x): # x^8-4x^7+5x^6-17x^2+68x-85
```

```
    10 def f3(x): # cos(3x)+sin(4x)
    12 def f4(x): # e^x-4x
    14 funcList = [f1, f2, f3, f4]
    15 dfuncList = []
    16 inputList = []
    17 inputList2 = []
    18 outputList = []
    19 outputList2 = []
    20 finalRootList = []
    21 def table(func, i):# Produces lists to be displayed in table format of the
inputs and outputs.
    29 def values(func, i):# Produces lists to be used in finding the roots.
    38 def evaluate_func(i,first):# Evaluates a function with the given value.
    41 def evaluate_derv(i,first):# Evaluates the derivative of a function with
the given value.
    44 def find_root(a,b,i):# Determines a root of a given function.
    49 def IVT(inList,outList,i,count,k):# Calculates the roots of a function and
returns how many roots are found.
    81 for i in range (0,len(funcList)):# Creates a list of derivatives that go
respective to the four functions in funcList.
    85 bool = True
    86 while bool == True:# Makes sure a function is selected correctly.
    92 while bool == False:# Gives input of how many decimal places someone wants
to approximate to.
   101 ans = table(funcList,in1)
   102 solution = pd.DataFrame(ans, columns=[inputList])
   103 solution.index = ["x","f(x)"]
   104 print(solution)
   105 values(funcList,in1)
   106 count = 0
   107 k = 0
   108 newCount = IVT(inputList2,outputList2,0,count,k)
   109 print('\nThere are',newCount,'roots from',decimal,'decimals of
approximation:',finalRootList)
```

3.1 Code Discussion

```
PS C:\Users\spenc> & C:/Users/spenc/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/spenc/Documents/CSU Classes/Symposium Poster - Newton's Method/Alternate"
What function do you want to find the roots of? Enter 1, 2, 3, or 4.
1
How many decimal places do you want to round to? Give between 6 and 13 places.
8
        -10     -9      -8      -7     -6     -5     -4     -3     -2     -1     0    1    2     3     4     5     6     7     8     9      10
x     -10.0   -9.0    -8.0    -7.0   -6.0   -5.0   -4.0   -3.0   -2.0   -1.0   0.0  1.0  2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0   10.0
f(x) -2806.0 -2121.0 -1558.0 -1105.0 -750.0 -481.0 -286.0 -153.0 -70.0 -25.0 -6.0 -1.0  2.0  15.0  50.0  119.0  234.0  407.0  650.0  975.0  1394.0

There are 1 roots from 8 decimals of approximation: [1.5]
```

Fig. 2. Sample run displayed using equation 1 with eight decimals of approximation, giving a table of values and the roots of the function from inputs of -10 to 10.

At the start, necessary tools were imported, and the four functions were defined (Appendix 1.1) on lines 6-13 and put into an array, or list, in line 14. Several empty arrays were created in lines 15-20 as they would be used in later tasks. Next, the main methods were defined and written over lines 21-80 (Appendices 1.2-1.6) to carry out specific tasks later on when finding the roots of a function. A couple of input pieces were added to get some interaction with the user, and a table of inputs and outputs was printed on screen over a pre-defined interval (-10 to 10). Then, the full process of finding the root(s) of a function occurs in line 108 by calling the IVT function. First, outputs are compared to find when the function crosses the x-axis, or when one value is positive and the next is negative, or vice versa. Once that is found, the root computation process takes place, using the predefined arrays in lines 15-20 that cycle through various inputs until a value matches the previous input exactly with the specified number of decimals, thereby determining the best approximation of a root. This is then added to a separate list that contains all the roots of the function. Once all the roots have been identified, a final sentence is printed with how many roots are found within the interval and a list of all the roots in line 109, ending the process of Newton's Method as shown in Figure 2.

4 Conclusion

This paper demonstrated how Newton's Method can be applied to approximate real roots of differentiable functions, including polynomial, trigonometric, and exponential equations. Implementing the method in Python allowed for efficient testing and visualization of how the iterative formula performs across different types of equations. In all cases tested, the method successfully converged to a root when the initial approximation was sufficiently close to the actual value. However, the results also highlighted that poor initial guesses could lead to slow convergence or, in some cases, failure to converge.

Newton's Method proved effective for all equation types examined, particularly when the Intermediate Value Theorem was used to locate intervals containing roots, guiding the selection of starting points. Future investigations we will expand on this work by applying Newton's Method to a broader range of functions, including rational and piecewise-defined equations. In these cases, continuity must be ensured, since the Intermediate Value Theorem is only applicable when functions are continuous. This additional analysis will allow us to focus on how the choice of initial approximation within those intervals affects the rate of convergence, aiming to develop more efficient strategies for selecting starting points based on function behaviour.

References

1. J.R. Torregrosa, A. Cordero, F. Soleymani, *Iterative methods for solving nonlinear equations and systems*, MDPI, (**2019**) ISSN 2227-7390
2. Mathematics LibreTexts, *4.9: Newton's method*, (**2025**) https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/04%3A_Applications_of_Derivatives/4.09%3A_Newtons_Method
3. P. Deufhard, *A short history of Newton's method*, EMS, (**2010**), https://ems.press/content/book-chapter-files/27348
4. S. Kean, D.C. Engerman, J.E. Casto, P. Tonguette, H. Staff, S.S. Taylor, *Newton, the last magician*, NEH, (2011), https://www.neh.gov/humanities/2011/januaryfebruary/feature/newton-the-last-magician
5. C.M. Mata Rodriguez, *El método de Newton-Raphson para la obtención de raíces de ecuaciones mediante programación en Mathcad, Algoritmo de Cálculo*, (**2017**), Revista de Ingeniería Matemáticas y Ciencias de la Información **4(7):83-86** DOI:10.21017/rimci.2017.v4.n7.a25
6. J.J. O'Connor, E.F. Robertson, *Thomas Simpson – biography*, Maths History, (**2005**), https://mathshistory.st-andrews.ac.uk/Biographies/Simpson/
7. F.H. Bowers, *Newton's method*, M.S. thesis, Atlanta University, Atlanta, GA, (**1975**), http://hdl.handle.net/20.500.12322/cau.td:1975_bowers_fred_h

Appendix 1 – Functions and Loops used in Python code

1.1 – Equations 1-4 (Lines 6-13)

```
def f1(x): # 2x^3-7x^2+10x-6
    return 2*(x**3)-7*(x**2)+10*x-6
def f2(x): # x^8-4x^7+5x^6-17x^2+68x-85
    return x**8-4*(x**7)+5*(x**6)-17*(x**2)+68*x-85
def f3(x): # cos(3x)+sin(4x)
    return sp.cos(3*x) + sp.sin(4*x)
def f4(x): # e^x-4x
    return sp.exp(x) - 4*x
```

1.2 – Table (Line 21)

```
def table(func, i):# Produces lists to be displayed in table format of
the inputs and outputs.
    for j in range (-10,11):
        inputList.append(j)
        out = float(func[i-1](j))
        outputList.append(out)
    in_array = np.array(inputList)
    out_array = np.array(outputList)
    return in_array, out_array
```

1.3 – Values (Line 29)

```
def values(func, i):# Produces lists to be used in finding the roots.
    for j in range(-50,51):# Creates an alternate list that will be used
for computation.
        j = j/5
        inputList2.append(j)
        out2 = float(func[i-1](j))
        outputList2.append(out2)
    in_array2 = np.array(inputList2)
    out_array2 = np.array(outputList2)
    return in_array2, out_array2
```

1.4 – Evaluate Function (Line 38) and Evaluate Derivative (Line 41)

```
def evaluate_func(i,first):# Evaluates a function with the given value.
    result = funcList[i-1](first)
    return result
def evaluate_derv(i,first):# Evaluates the derivative of a function with the
given value.
    dresult = dfuncList[i-1](first)
    return dresult
```

1.5 – Find Root (Line (44)

```
def find_root(a,b,i):# Determines a root of a given function.
    if b == 0:
        raise ValueError("Derivative was zero, so a local extrema was
hit (cannot find a root any further).")
    finish = i - a/b
    return finish
```

1.6 – IVT (Line 49)

```
def IVT(inList,outList,i,count,k):# Calculates the roots of a function
and returns how many roots are found.
    for i in range(0,len(inList)-1):
        before = inList[i]
        after = inList[i+1]
        fx_before = outList[i]
        fx_after = outList[i+1]
        if (fx_before > 0 and fx_after < 0) or (fx_before < 0 and
fx_after > 0):
            count += 1
            one = abs(fx_before)
            two = abs(fx_after)
            rootList=[]
            if (one > two) or (one == two):
                rootList.append(after); rootList.append(after-1)
                while
(round(rootList[k],decimal)!=round(rootList[k+1],decimal)):
                    d = evaluate_derv(in1,float(after))
                    result = find_root(fx_after,d,after)
                    after = result
                    fx_after = evaluate_func(in1,float(after))
                    rootList.append(result)
                    k += 1
            else:
                rootList.append(after); rootList.append(after-1)
                while
(round(rootList[k],decimal)!=round(rootList[k+1],decimal)):
                    d = evaluate_derv(in1,float(before))
                    result = find_root(fx_before,d,before)
                    before = result
                    fx_before = evaluate_func(in1,float(before))
                    rootList.append(result)
                    k += 1
            finalRootList.append(round(result,decimal))
        k = 0
    return count
```

1.7 – For Loop (Line 81)

```
for i in range (0,len(funcList)):# Creates a list of derivatives that go
respective to the four functions in funcList.
    f = funcList[i](x)
    df = sp.diff(f, x)
    dfuncList.append(sp.lambdify(x, df, modules="numpy"))
```

1.8 – While Loop 1 (Line 86)

```
while bool == True:# Makes sure a function is selected correctly.
    in1 = int(input('What function do you want to find the roots of?
Enter 1, 2, 3, or 4.\n'))
    if in1 == 1 or in1 == 2 or in1 == 3 or in1 == 4:
        bool = False
    else:
        print('A function does not exist at',in1,'Try again.')
```

1.9 – While Loop 2 (Line 92)

```
while bool == False:# Gives input of how many decimal places someone
wants to approximate to.
    decimal = int(input('How many decimal places do you want to round
to? Give between 6 and 13 places.\n'))
    if 6 <= decimal <= 13:
        bool = True
    else:
        if decimal < 6:
            print('Your number of places is too small. Try again.\n')
        else:
            print('Your number of places is too large. Try again.\n')
```

The above code snippets show how the hidden functions and loops work for implementing Newton's Method. 1.1-1.6 show the defined functions that are called within the code itself, 1.7 is the loop for creating the outputs of a function and its derivative, and 1.8-1.9 are loops that use the user's input.

Appendix 2 – Entire Code in One Block

```
import sympy as sp
import pandas as pd #type: ignore
import numpy as np #type: ignore
from sympy import symbols, exp
x = sp.symbols('x')
def f1(x): # 2x^3-7x^2+10x-6
    return 2*(x**3)-7*(x**2)+10*x-6
def f2(x): # x^8-4x^7+5x^6-17x^2+68x-85
    return x**8-4*(x**7)+5*(x**6)-17*(x**2)+68*x-85
def f3(x): # cos(3x)+sin(4x)
    return sp.cos(3*x) + sp.sin(4*x)
def f4(x): # e^x-4x
    return sp.exp(x) - 4*x
funcList = [f1, f2, f3, f4]
dfuncList = []
inputList = []
inputList2 = []
outputList = []
outputList2 = []
finalRootList = []
def table(func, i):# Produces lists to be displayed in table format of the inputs
and outputs.
    for j in range (-10,11):
        inputList.append(j)
        out = float(func[i-1](j))
        outputList.append(out)
    in_array = np.array(inputList)
    out_array = np.array(outputList)
    return in_array, out_array
def values(func, i):# Produces lists to be used in finding the roots.
    for j in range(-50,51):# Creates an alternate list that will be used for
computation.
        j = j/5
        inputList2.append(j)
        out2 = float(func[i-1](j))
        outputList2.append(out2)
    in_array2 = np.array(inputList2)
    out_array2 = np.array(outputList2)
    return in_array2, out_array2
def evaluate_func(i,first):# Evaluates a function with the given value.
    result = funcList[i-1](first)
    return result
def evaluate_derv(i,first):# Evaluates the derivative of a function with the given
value.
    dresult = dfuncList[i-1](first)
    return dresult
def find_root(a,b,i):# Determines a root of a given function.
    if b == 0:
        raise ValueError("Derivative was zero, so a local extrema was hit (cannot
find a root any further).")
    finish = i - a/b
    return finish
def IVT(inList,outList,i,count,k):# Calculates the roots of a function and returns
how many roots are found.
    for i in range(0,len(inList)-1):
        before = inList[i]
        after = inList[i+1]
        fx_before = outList[i]
        fx_after = outList[i+1]
        if (fx_before > 0 and fx_after < 0) or (fx_before < 0 and fx_after > 0):
            count += 1
            one = abs(fx_before)
            two = abs(fx_after)
```

```
            rootList=[]
            if (one > two) or (one == two):
                rootList.append(after); rootList.append(after-1)
                while (round(rootList[k],decimal)!=round(rootList[k+1],decimal)):
                    d = evaluate_derv(in1,float(after))
                    result = find_root(fx_after,d,after)
                    after = result
                    fx_after = evaluate_func(in1,float(after))
                    rootList.append(result)
                    k += 1
            else:
                rootList.append(after); rootList.append(after-1)
                while (round(rootList[k],decimal)!= round(rootList[k+1], decimal)):
                    d = evaluate_derv(in1,float(before))
                    result = find_root(fx_before,d,before)
                    before = result
                    fx_before = evaluate_func(in1,float(before))
                    rootList.append(result)
                    k += 1
            finalRootList.append(round(result,decimal))
        k = 0
    return count
for i in range (0,len(funcList)):# Creates a list of derivatives that go respective
to the four functions in funcList.
    f = funcList[i](x)
    df = sp.diff(f, x)
    dfuncList.append(sp.lambdify(x, df, modules="numpy"))
bool = True
while bool == True:# Makes sure a function is selected correctly.
    in1 = int(input('What function do you want to find the roots of? Enter 1, 2, 3,
or 4.\n'))
    if in1 == 1 or in1 == 2 or in1 == 3 or in1 == 4:
        bool = False
    else:
        print('A function does not exist at',in1,'Try again.')
while bool == False: # Gives input of how many decimal places someone wants to
approximate to.
    decimal = int(input('How many decimal places do you want to round to? Give
between 6 and 13 places.\n'))
    if 6 <= decimal <= 13:
        bool = True
    else:
        if decimal < 6:
            print('Your number of places is too small. Try again.\n')
        else:
            print('Your number of places is too large. Try again.\n')
ans = table(funcList,in1)
solution = pd.DataFrame(ans, columns=[inputList])
solution.index = ["x","f(x)"]
print(solution)
values(funcList,in1)
count = 0
k = 0
newCount = IVT(inputList2,outputList2,0,count,k)
print('\nThere are',newCount,'roots from',decimal,'decimals of
approximation:',finalRootList)
```

The complete code is shown above, including all the functions and loops that were hidden from Computer Code. The main process is a user deciding which function to find the roots of, then deciding how many decimal places they would like to approximate to, while the code will perform the process behind the scenes. See Code Discussion for a brief walkthrough of how the code will operate.