Exploring Methods for Finding Roots of Functions and Their Applications in Physics

Luke Keely

Trinity College Dublin,
`keelyl@tcd.ie`

Abstract. In this lab exercise, the primary focus was to investigate the efficiency of two root-finding algorithms: the Bisection and Newton-Raphson methods. The algorithms were applied to find the roots of simple polynomials of various degrees. The two methods were studied under varying tolerances.
We look at the difference in the step growth on tighter tolerances and how each method might be applicable. If highly accurate answers are needed, the Newton-Raphson method is best suited as you can tighten tolerances. The Bisection method is faster and easier for simpler problems where larger tolerances are allowed.
To see the application of such root-finding algorithms, I worked on finding the minimum value of an ionic interaction potential function.

Table of Contents

1 Introduction

The primary focus of this lab is to evaluate the efficiency of two classic root-finding algorithms: the Bisection method and the Newton-Raphson method. Both algorithms were used to find roots of simple polynomial functions.

1.1 Bisection Method

Overview Bisection: we start with interval $[a, b]$, where the $f(x)$ changes sign, i.e., $f(a) \cdot f(b) < 0$. Then, we find the midpoint $c = \frac{a+b}{2}$ and use it to narrow down the interval where the root lies. The algorithm continues by selecting the sub-interval where $f(x)$ changes sign and repeating the process until the interval size is smaller than a given tolerance.

Formula

$$c = \frac{a+b}{2} \tag{1}$$

1.2 Newton-Raphson Method

Overview Newton-Raphson: this is a bit more complicated but is faster in general. Starting with the initial guess x_0, we use the function and its derivative to find a better approximation x_{n+1} using the formula:

Formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2}$$

2 Methodology

2.1 Implementation of the Bisection Method

The Bisection method was coded in Python as a function named `get_root_bisection`. This function was designed to accept polynomial coefficients as an input. This allowed testing on a variety of polynomial equations. To find suitable initial bounds automatically, another function was added to scan a range and select an interval where the function changes sign; if no suitable x_1 or x_3 was found, the scan steps were refined and ran again. This ensured that the algorithm began with a valid interval, making it easier for various functions with completely unknown roots.

```python
# Simplified snippet from get_root_bisection
def get_root_bisection(coefficients, tol, x_start, x_stop):
    x1, x3 = find_x1_x3(coefficients, x_start, x_stop)
    x2_attempts, y2_attempts = [], []
    while True:
        x2 = 0.5 * (x1 + x3)
        y2 = evaluate_polynomial(x2, coefficients)
        x2_attempts.append(x2)
        y2_attempts.append(y2)

        if abs(y2) <= tol:
            break
        if y2 > 0:
            x3 = x2
        else:
            x1 = x2
  return x2
```

2.2 Implementation of the Newton-Raphson Method

The Newton-Raphson algorithm was also coded in a Python function named `get_root_newton`. Like the Bisection method, this function was designed to accept polynomial coefficients, allowing for a wide range of functions. The function's derivative was also needed for this method, which was computed within the function, making it easier and quicker to run many tests.

```python
# Simplified snippet from get_root_newton
def get_root_newton(coefficients, tol, x_start, x_stop):
    x1 = x_start
    x2_attempts, y2_attempts = [], []
    while True:
        y1 = evaluate_polynomial(x1, coefficients)
        x_attempts.append(x1)
        y_attempts.append(y1)

        if abs(y1) <= tol:
            break
        x1 = x1 - y1 / evaluate_derivative(x1, coefficients)
    return x1
```

2.3 Comparative Analysis

The function `collect_data` was made to test direct comparisons of each method over a range of tolerances. For every polynomial I wanted to test, I ran each algorithm over a range of tolerances and recorded the number of steps and the root it returned. The data was stored in a Pandas Data Frame for easier manipulation and analysis. This data allowed me to plot the data found from each test to understand how the parameters affected the number of steps and accuracy.

```python
# Simplified snippet from collect_data function
for i, neg_log_tol in enumerate(np.arange(0, 4, 0.1)):
    tolerance = 10 ** -neg_log_tol
    data = []
    for i, neg_log_tol in enumerate(tqdm(np.arange(0, 4, 0.1), desc="
        Collecting Data")):
        tolerance = 10 ** -neg_log_tol
        root_bisection, nsteps_bisection = get_root_bisection(
            COEFFICIENTS, tolerance, X_START, X_STOP)
        root_newton, nsteps_newton = get_root_newton(COEFFICIENTS,
            tolerance, X_START, X_STOP)

        data.append({
            'Tolerance': tolerance,
            'log10(Tolerance)': -np.log10(tolerance),
            'Root Found (Bisection)': root_bisection,
            'Number of Steps (Bisection)': nsteps_bisection,
            'Root Found (Newton)': root_newton,
            'Number of Steps (Newton)': nsteps_newton
        })
    return data
```

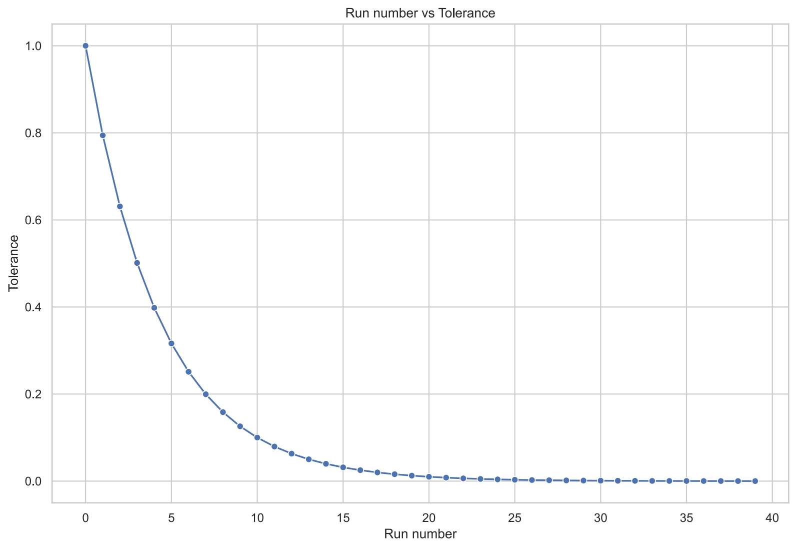

Fig. 2.1: Plot of tolerances tested against the iteration of test per function.

2.4 Application to Ionic Interaction Potential

The root-finding algorithms were not just limited to simple polynomials. They were also applied to find the minimum value of an ionic interaction potential function. This showed how these methods for finding roots were not only useful for arbitrary polynomials but instead practical for actual applications in computational physics. The ionic interaction potential function $V(x)$ was modeled using the equation:

$$V(x) = \frac{e^2}{4\pi\epsilon_0}\left(\frac{A}{x^p} - e^{-x}\right)$$

where $A = 1.0\, eV \cdot nm^p$ and $p = 6$.

The Newton-Raphson method was used to find the minimum of $V(x)$, starting with an initial guess of $x = 0.2\, nm$.

2.5 Software and Libraries

The project was made using Python 3.11.5, running in the Spyder IDE within the Anaconda environment. Several Python libraries were used in making this project. NumPy was essential for numerical operations, while Matplotlib and Seaborn were used for plotting the data. The Seaborn library, in particular, was great for data visualization, making it more aesthetic and functional compared to running Matplotlib alone.

2.6 Challenges and Modifications

The plotting part of the project had some challenges, mainly when representing the data accurately and visually appealingly. This was solved primarily using the Seaborn library, which offers cleaner and more customizable plots than Matplotlib. Additionally, the Pretty-Table library was used to display the data in a table format, making it visually appealing to interpret the data.

3 Results

Polynomial 1 $x^2 - \sqrt{2}$

The original choice for testing was $x^2 - 1$, but this was too simple for the Bisection method, which solved it in just one step. To provide a better test for each of the algorithms, I changed the function to $x^2 - \sqrt{2}$.

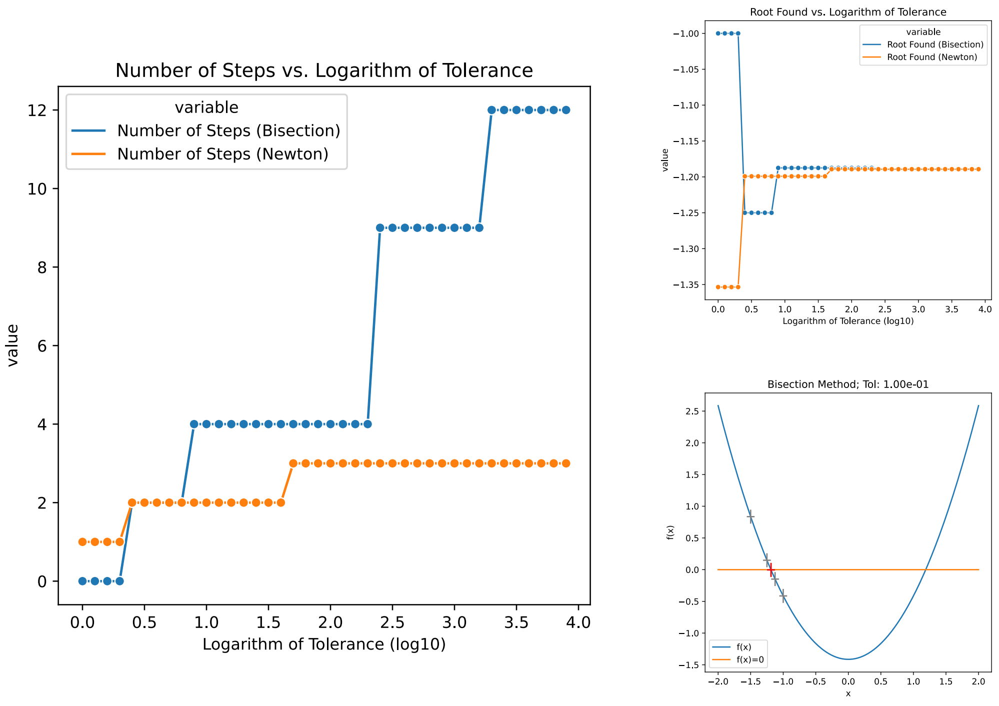

Fig. 3.1: Analysis of root finding for polynomial 1

Table 1: Comparison of Bisection and Newton-Raphson Methods for Polynomial 1

	Bisection	Newton-Raphson
Maximum steps	12	3
Root Found (at lowest tol)	-1.189208984375	-1.1892486211774982
Speed of Convergence	0.0311	0.0269
Sensitivity to Tolerance	1.0000	0.1667

Both methods had similar average convergence speeds, but the Bisection method was much more sensitive to changes in tolerance compared to the Newton-Raphson method, requiring more steps as the tolerance got smaller.

Polynomial 2 $x - \sqrt{2}$

The second function, $x - \sqrt{2}$, is a linear function. This is a more straightforward case compared to the previous quadratic. Still, using the irrational number to add complexity ensures finding the root is not trivially easy.

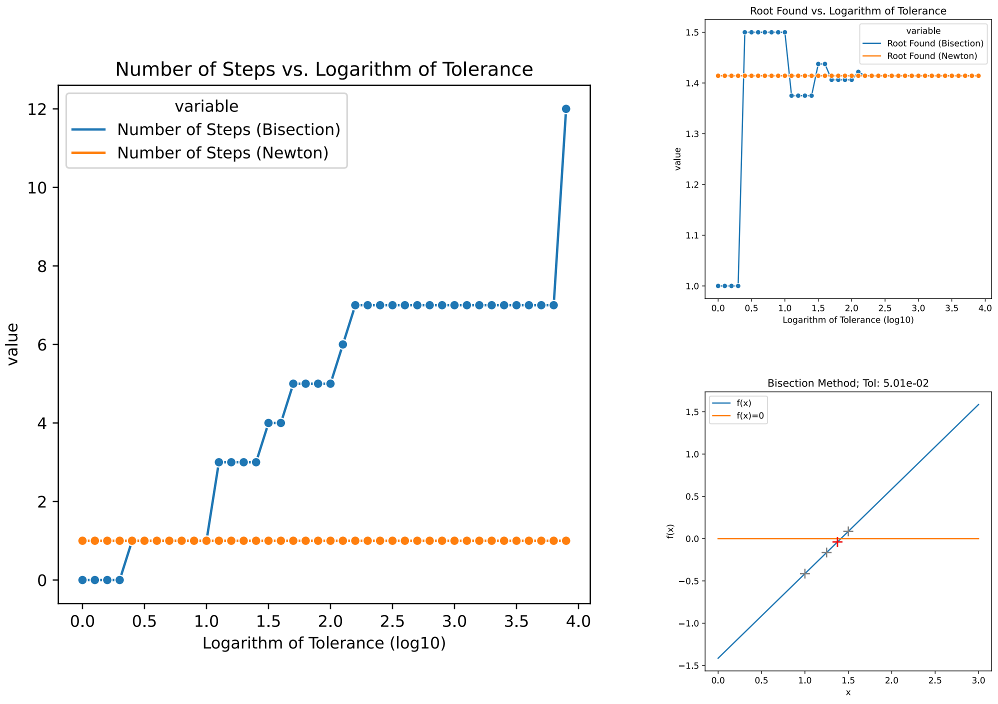

Fig. 3.2: Analysis of root finding for polynomial 2

Table 2: Comparison of Bisection and Newton-Raphson Methods for Polynomial 2

	Bisection	Newton-Raphson
Maximum steps	12	1
Root Found (at lowest tol)	1.414306640625	1.4142135623730951
Speed of Convergence	0.0259	1.0000
Sensitivity to Tolerance	1.0000	0.0000

The performance of the Bisection method was similar to the first function. However, the Newton-Raphson method quickly found the root, showing no sensitivity to tolerance changes; this seems to be because it was a linear function.

Polynomial 3 $-\frac{1}{2}x^3 + 8x - 2$

The third function, $-\frac{1}{2}x^3 + 8x - 2$, was chosen to explore what happens when a function has many roots. The cubic function was complicated enough and provided some interesting insights into how higher-degree polynomials affect the performance and accuracy of the Newton-Raphson method.

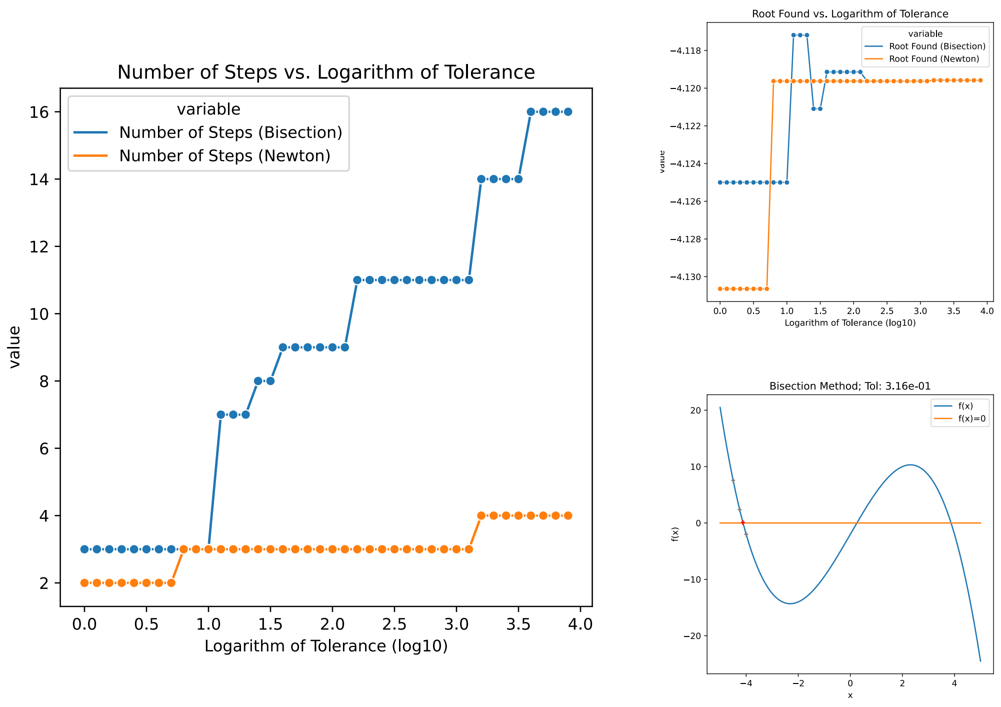

Fig. 3.3: Analysis of root finding for polynomial 3

Table 3: Comparison of Bisection and Newton-Raphson Methods for Polynomial 3

	Bisection	Newton-Raphson
Maximum steps	16	14
Root Found (at lowest tol)	-4.1195831298828125	-4.119583940858687
Speed of Convergence	0.0423	0.0257
Sensitivity to Tolerance	1.0000	0.1538

The Bisection method showed a higher average convergence speed than the Newton-Raphson method. However, the Bisection method was still much more sensitive to the tolerance change than the Newton-Raphson method.

Application to Ionic Interaction Potential

The Newton-Raphson method effectively found the minimum of $V(x)$, converging within 37 iterations. This real-world application showed the use of the Newton-Raphson method quickly and accurately, finding solutions to real-life physical problems.

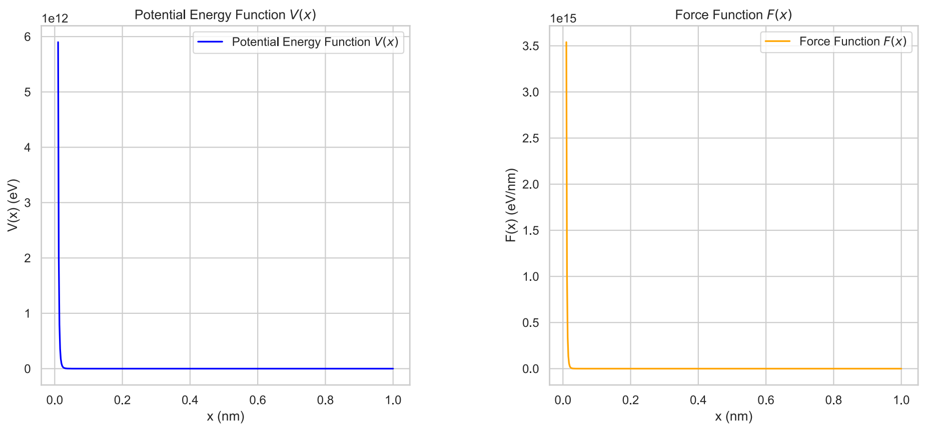

Fig. 3.4: Analysis of Potential energy function

Table 4: Newton-Raphson Method for Ionic Interaction Potential

Metric	Newton-Raphson
Minimum	1.6305
Iterations	37

Coding Exercise: Equilibrium Positions of Charges on a Ring

Similarly to finding the minimum of the ionic interaction potential, the Newton-Raphson method can also be used to find the equilibrium positions of three inequivalent charges on a ring. These charges are confined to move along the ring and will arrange themselves in positions that minimize the system's potential energy.

It has been left as an exercise to the reader to demonstrate how the Newton-Raphson method can be used in this more complicated application.

Hint: Use polar coordinates and trigonometric identities to express the distances between these charges.

4 Conclusion

In this lab session, we examined the efficiency and application of two popular root-finding algorithms: the Bisection and Newton-Raphson methods. Using Python and an assortment of libraries such as NumPy, Matplotlib, and Seaborn, I evaluated the performance of these algorithms on a range of polynomial functions, gauging their efficiency and response to various tolerance levels. The findings show that the Bisection method, while easy to implement and reliable for broader tolerances, loses its efficiency as the required precision increases. On the other hand, the more computationally intense Newton-Raphson method outperforms the former method when used in scenarios that require high precision.

In order to show the practical application of these sorts of algorithms, they were used in a real-world problem: finding the minimum of an ionic interaction potential function. This proved the usefulness of the Newton-Raphson method, which found the minimum value in 37 iterations.

To conclude, the best choice between these two algorithms for finding roots isn't one-size-fits-all; instead, it depends on the requirements of the specific task. The Bisection method is generally easier and quicker for simple problems, while the Newton-Raphson method is better suited for more complex, precision-sensitive problems.

A Supplemental Material

A.1 Detailed Analysis

Detailed analysis can be found in the attached Python files and data sets.

A.2 Code

The Python code used for the experiments is attached.

A.3 Data

The raw data is provided in a folder called 'run_data' in sub-folders per function.

A.4 Lab Manual

The lab manual used for this experiment is attached.

A.5 Images and Graphs

The images and graphs used for the analysis are found in the folder 'run_data'.

References

1. Department of Physics, Trinity College Dublin, *SF Computational Lab Handbook*.
2. Wikipedia, *Root-finding algorithms*, `https://en.wikipedia.org/wiki/Root-finding_algorithms`.
3. Documentation for NumPy, Matplotlib, and Seaborn.
4. Python Documentation, `https://docs.python.org/3/`.