Senior Research Project Report

Numerical Methods with Third-Order Convergence: Applications to Nonlinear Ordinary Differential Equations

Hebah Kamal
ID:100061305
Supervisor: Aymen Laadhari

Contents

1 Introduction

Nonlinear equations of the form $f(x) = 0$ frequently appear across various disciplines, including physics, engineering, finance, and data science[1]. Unlike linear equations, which can often be solved using straightforward algebraic techniques, nonlinear equations pose significant challenges due to their complexity and potential lack of explicit solutions. In many cases, these equations must be solved numerically rather than analytically, necessitating the development and application of efficient numerical methods[2,3].

The study of numerical methods for solving nonlinear equations is an essential branch of computational mathematics. These methods provide iterative approximations to the roots of functions and are widely used in scientific computing and real-world applications. They allow researchers and engineers to model complex systems, optimize processes, and make predictions based on mathematical models[2,3].

A fundamental approach to solving nonlinear equations is the fixed-point method. This method transforms the problem into the form $x = g(x)$ and iteratively applies $x_{n+1} = g(x_n)$ to approach the solution. The fixed-point method is simple to implement and does not require derivatives, making it useful for certain applications. However, it has significant drawbacks, such as slow convergence and strict conditions on the function $g(x)$ to ensure stability. If $g(x)$ is not chosen carefully, the method may fail to converge[4,5].

One of the most prominent numerical methods for solving nonlinear equations is Newton's method, originally introduced by Sir Isaac Newton and Joseph Raphson in the 17th century[6]. Newton's method is often preferred over the fixed-point method because of its faster convergence. It iteratively improves an initial guess to find a more accurate approximation of a function's root. Unlike the fixed-point method, which generally has linear convergence, Newton's method exhibits quadratic convergence, meaning it reaches an accurate solution much faster when the initial guess is close to the root. However, despite its effectiveness, Newton's method has limitations. It requires computing derivatives, which may be challenging for some functions, and it does not always converge if the initial guess is poorly chosen[3].

Researchers have introduced modifications to improve the reliability and efficiency of Newton's method. These refinements have enhanced its stability and convergence, making it widely applicable in various scientific and engineering fields [7].
Several works have explored high-order Newton variants to improve the resolution of nonlinear PDEs. Although limited in number, the main contributions include [9], [10], and [11], which solved 1D reaction-diffusion systems. In population dynamics, [12] focused on the 1D Fisher equation, while [13] studied a nonlinear heat conduction model. High-order methods were also employed in multiphase flows [14,15,16,17] and in fluid-membrane interaction models [18]. In this project, we study some specific Newton-type variants, focusing on their construction and use in solving root-finding problems and nonlinear ordinary differential equations.

Chapter 2

2 Numerical methods for solving nonlinear equations

Understanding numerical methods is crucial for solving nonlinear equations efficiently. This chapter introduces widely used numerical approaches for solving nonlinear equations, including Newton's method, the fixed-point method, and other iterative techniques. Key concepts such as iterative approaches, error analysis, Taylor series expansion, and method convergence are discussed to provide a strong foundation for understanding numerical solutions.

2.1 Key concepts

2.1.1 Iterative approaches

Most numerical methods rely on iterative approaches, where an initial guess x_0 is refined to produce a sequence of approximations $x_1, x_2, \ldots$ that ideally converge to the solution α. The general iterative form is:

$$x_{n+1} = g\left(x_n\right),$$

where $g(x)$ represents the iteration function specific to the numerical method being used, transforming an approximation x_n into a new estimate x_{n+1}.

Convergence of iterative methods depends on the properties of $g(x)$, including continuity and how approximations are updated. Some methods, like fixed-point iteration, require specific conditions for stability, while others, such as Newton's method, depend on properties of $f(x)$ and its derivatives.

2.1.2 Error analysis

To evaluate the accuracy of an iterative method, the error at the n-th iteration is defined as:

$$e_n = x_n - \alpha,$$

where α is the true root of $f(x) = 0$. The error equation relates the errors across iterations:

$$e_{n+1} = Ce_n^k,$$

where C is a constant and k is the order of the method, indicating the rate at which the method converges. Higher-order methods generally achieve faster convergence but may require additional computational effort.

2.1.3 Taylor series expansion and its role

The Taylor series expansion is essential for understanding many numerical methods, especially those that refine approximations through iteration. The expansion of a function $f(x)$ around a point x_n is given by:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2}(x - x_n)^2 + \ldots$$

This expansion simplifies the function, making it useful for approximations in methods like Newton's method. The Taylor series also helps explain how errors change between iterations, providing insight into how numerical methods improve accuracy step by step.

2.2 Numerical methods

2.2.1 Newton's method

Newton's method is an iterative technique for solving $f(x) = 0$ based on the Taylor series expansion. Starting with:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n),$$

Rearranging this equation results in:

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

Key features

-Error equation: The error e_{n+1} can be approximated by using the higher-order terms of the Taylor series expansion:

$$e_{n+1} \approx \frac{f''(\alpha)}{2f'(\alpha)} e_n^2.$$

The practical implementation of this error equation is discussed in Section 3.3 of the next chapter.

-Convergence: Quadratic ($k = 2$) under the assumption that $f'(\alpha) \neq 0$ and the initial guess is sufficiently close to α.

-Advantages: Fast convergence for smooth functions.

-Limitations: Requires computation of $f'(x)$ and may fail if $f'(x)$ is zero or poorly conditioned.

2.2.2 Fixed-point method

Fixed-point iteration solves $f(x) = 0$ by reformulating it as $x = g(x)$, and iteratively applying:

$$x_{n+1} = g(x_n).$$

Key features

-Error equation:
The Taylor series expansion of $g(x)$ about α helps analyze convergence:

$$g(x) \approx g(\alpha) + g'(\alpha)(x - \alpha),$$

where α is the fixed point. The error relation is:

$$e_{n+1} = g'(\alpha) e_n.$$

-Convergence: Linear ($k = 1$).

-Advantages: Simple to implement and does not require derivatives.

-Limitations: Convergence can be slow and depends on the choice of $g(x)$.

2.2.3 Secant method

The secant method improves upon Newton's method by avoiding the need to compute derivatives explicitly. Instead, it approximates the derivative using a finite difference. The method requires two initial guesses, x_n and x_{n-1}, and iteratively updates the solution using the formula:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

Key features

-Error equation: $\mathrm{e}_{n+1} \approx Ce_n^{1.618}$.

-Convergence: Superlinear convergence ($k \approx 1.618$).

-Advantages: Requires only function evaluations and does not need derivative computations.

-Limitations: Converges more slowly than Newton's Method.

2.2.4 Bisection method

The bisection method is a robust root-finding technique that requires the function $f(x)$ to change sign over an interval $[a, b]$. The interval is iteratively halved, and the subinterval containing the root is selected. The update formula is:

$$x_{n+1} = \frac{a_n + b_n}{2}.$$

Key features

-Error equation: $e_n = \frac{b_n - a_n}{2}$.

-Convergence: Linear ($k = 1$).

-Advantages: Guaranteed convergence, as the method always reduces the interval containing the root.

-Limitations: Slow convergence compared to other methods like Newton's method.

2.2.5 Method of false position (Regula falsi)

The method of false position combines the robustness of the bisection method with a secant-like update rule. It uses a weighted average:

$$x_{n+1} = x_n - \frac{f(x_n)\,(b_n - x_n)}{f(b_n) - f(x_n)}.$$

Key features

-Convergence: Linear ($k = 1$).

-Advantages: Faster than bisection in some cases.

-Limitations: May fail to converge for poorly chosen intervals.

Chapter 3

3 Newton's method for solving nonlinear equations

Newton's method is a fundamental numerical technique for finding the roots of nonlinear equations of the form $f(x) = 0$. By leveraging the derivative of the function, it refines approximations iteratively, often achieving rapid convergence near the root when the initial guess is sufficiently close. This chapter explores the derivation of the method, its practical implementation, convergence behavior, and limitations.

3.1 Derivation of Newton's method

Newton's method is derived using the Taylor series expansion of $f(x)$ around an initial guess x_n.The first-order Taylor series expansion is given by:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n).$$

Setting $f(x) = 0$ and solving for x in this linear approximation:

$$x \approx x_n - \frac{f(x_n)}{f'(x_n)}.$$

This leads to the iterative formula:

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

This is Newton's method.

3.2 Implementation

The implementation of Newton's method involves the following steps:
-Initial guess: Begin with an initial guess x_0.
-Iterative update: Apply the Newton iteration formula:

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

-Convergence check: Continue the iteration until one of the following stopping criteria is satisfied:

1. Absolute error criterion:
$$|x_{n+1} - x_n| < \text{tolerance}$$
2. Relative error criterion:
$$\left|\frac{x_{n+1} - x_n}{x_{n+1}}\right| < \text{tolerance}$$
3. Residual criterion:
$$|f(x_{n+1})| < \text{tolerance}$$

Throughout the report, the criterion used to examine the functions is as follows: if the exact solution is known, the error is calculated as the absolute difference between the exact and computed solutions. However, in cases where the exact solution is not known, the error is estimated based on the difference between consecutive approximated solutions.

3.3 Error analysis and convergence

Newton's method exhibits quadratic convergence, meaning the error decreases proportionally to the square of the previous error. Let the error at iteration n be $e_n = x_n - \alpha$, where α is the true root. Using the Taylor series expansion, the error at the next iteration can be approximated as:

$$e_{n+1} \approx \frac{f''(\alpha)}{2f'(\alpha)} e_n^2.$$

To analyze the error, we start by expanding $f(x)$ around the true root α using the Taylor series:

$$f(x_n) = f(\alpha) + f'(\alpha)(x_n - \alpha) + \frac{f''(\alpha)}{2}(x_n - \alpha)^2 + \dots$$

Since $f(\alpha) = 0$, this simplifies to:

$$f(x_n) = f'(\alpha)e_n + \frac{f''(\alpha)}{2} e_n^2 + \dots$$

The derivative $f'(x_n)$ can also be expanded around α:

$$f'(x_n) = f'(\alpha) + f''(\alpha)(x_n - \alpha) + \dots = f'(\alpha) + f''(\alpha)e_n + \dots$$

Substituting these expansions into Newton's iterative formula:

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

Simplify the denominator by assuming e_n is small, so $f'(x_n) \approx f'(\alpha)$. This yields:

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

The next error $e_{n+1} = x_{n+1} - \alpha$ becomes:

$$e_{n+1} \approx \frac{f''(\alpha)}{2f'(\alpha)} e_n^2.$$

3.4 Limitations of Newton's method

Despite its many advantages, Newton's method has some notable limitations:

-Dependency on initial guess: If the initial guess x_0 is far from the root, the method may fail to converge.

-Derivative requirement: Newton's method requires computing $f'(x)$, which can be challenging if the derivative is difficult to evaluate.

-Division by zero: The method fails if $f'(x_n) = 0$ at any iteration.

-Non-convergence: For functions with discontinuities, multiple roots, or inflection points, the method may fail to converge or oscillate indefinitely.

Chapter 4

4 Third-order modifications of Newton's method

4.1 General overview

Newton's method is widely used to solve nonlinear equations of the form $f(x) = 0$. Its iterative formula is:

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

and it typically achieves quadratic convergence near the root. However, recent advancements have focused on improving this convergence to cubic order by modifying the method.
These modifications rely on Newton's theorem, which expresses $f(x)$ as:

$$f(x) = f(x_n) + \int_{x_n}^{x} f'(t)\,dt.$$

Instead of assuming a constant value for $f'(t)$ in the interval $[x_n, x]$, these methods approximate the integral using quadrature rules such as the arithmetic mean, harmonic mean, or trapezoidal rule. By employing these integral approximation techniques, the iterative formulas achieve higher accuracy and cubic convergence, providing faster convergence than the classical Newton's method.

Each of these methods leverages a unique quadrature rule to compute the integral, leading to different modifications with specific characteristics. In this section, we introduce four such methods.

4.2 Midpoint-harmonic mean Newton (MHN) method

The following steps outline the derivation of a modified Newton method that achieves cubic convergence. This derivation uses Newton's theorem and approximates the integral through the harmonic mean rule.

1. Starting with Newton's theorem
Newton's theorem provides the integral representation of $f(x)$ as:

$$f(x) = f(x_n) + \int_{x_n}^{x} f'(t)\,dt.$$

2. Approximating the integral using the harmonic mean
Using the harmonic mean approximation for $f'(t)$ over the interval $[x_n, x]$, we write:

$$f'(t) \approx \frac{2}{\frac{1}{f'(x_n)} + \frac{1}{f'(x)}}$$

Substituting this into the integral:

$$\int_{x_n}^{x} f'(t)\,dt \approx (x - x_n) \cdot \frac{2}{\frac{1}{f'(x_n)} + \frac{1}{f'(x)}}$$

Substituting this into Newton's theorem gives:

$$f(x) = f(x_n) + (x - x_n) \cdot \frac{2}{\frac{1}{f'(x_n)} + \frac{1}{f'(x)}}.$$

3. Setting up the equation
We set $f(x) = 0$ to solve for the root:

$$0 = f(x_n) + (x - x_n) \cdot \frac{2}{\frac{1}{f'(x_n)} + \frac{1}{f'(x)}}$$

This simplifies to:

$$x = x_n - \frac{f(x_n)}{2} \cdot \left(\frac{1}{f'(x_n)} + \frac{1}{f'(x)} \right).$$

4. Approximating x
We approximate x by $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

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

5. Improving the representation
Introducing x^*_{n+1} as the classical Newton's approximation:

$$x^*_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

We can write:

$$x_{n+1} = x_n - \frac{f(x_n)}{2} \left(\frac{1}{f'(x_n)} + \frac{1}{f'\left(x^*_{n+1} \right)} \right).$$

6. Refining the expression
Rewriting the expression from step (5), we get:

$$x_{n+1} = x_n - \frac{f(x_n)}{2} \left(\frac{1}{f'(x_n)} + \frac{1}{2[\frac{f'(x^*_{n+1}) + f'(x_n)}{2}] - f'(x_n)} \right).$$

7. Using the midpoint value for the derivative
We replace the arithmetic mean $\frac{f'(x^*_{n+1}) + f'(x_n)}{2}$ with the midpoint value $f'\left(\frac{1}{2}(x_n + x^*_{n+1}) \right)$ to obtain:

$$x_{n+1} = x_n - \frac{f(x_n)}{2} \left(\frac{1}{f'(x_n)} + \frac{1}{2f'\left(\frac{1}{2}(x_n + x^*_{n+1}) \right) - f'(x_n)} \right)$$

The derived method is called the **midpoint-harmonic mean Newton (MHN) method**.

Proof of cubic convergence of the midpoint-harmonic mean Newton (MHN) method

We aim to demonstrate that the MHN method is of order 3 by showing its error equation.

1.Midpoint-harmonic mean Newton (MHN) method:

$$x_{n+1} = x_n - \frac{1}{2}\left(\frac{f(x_n)}{f'(x_n)} + \frac{f(x_n)}{2f'\left(\frac{x_n + x^*_{n+1}}{2}\right) - f'(x_n)}\right).$$

Using Taylor expansion and taking into account $f(\alpha) = 0$, we have:

2.

$$f(x_n) = f'(\alpha)\left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right].$$

3.

$$f'(x_n) = f'(\alpha)\left[1 + 2c_2 e_n + 3c_3 e_n^2 + O(e_n^3)\right].$$

4.Dividing (2) by (3), we get:

$$\frac{f(x_n)}{f'(x_n)} = \frac{f'(\alpha)\left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]}{f'(\alpha)\left[1 + 2c_2 e_n + 3c_3 e_n^2 + O(e_n^3)\right]}$$
$$= \frac{e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)}{1 + 2c_2 e_n + 3c_3 e_n^2 + O(e_n^3)}.$$

5.Using the Maclaurin series expansion to expand and simplify:

$$\frac{1}{1+x} = \frac{1}{1-(-x)} \approx 1 - x + x^2 + O(x^3),$$

We approximate the denominator as:

$$\frac{1}{1 + 2c_2 e_n + 3c_3 e_n^2 + O(e_n^3)} \approx 1 - 2c_2 e_n + (-3c_3 + 4c_2^2)e_n^2 + O(e_n^3).$$

6. Multiplying the numerator by the expanded denominator, we get:

$$\frac{f(x_n)}{f'(x_n)} = \left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]\left[1 - 2c_2 e_n + (-3c_3 + 4c_2^2)e_n^2 + O(e_n^3)\right]$$
$$= e_n - 2c_2 e_n^2 + (-3c_3 + 4c_2^2)e_n^3 + c_2 e_n^2 - 2c_2^2 e_n^3 + c_3 e_n^3 + O(e_n^4)$$
$$= e_n - c_2 e_n^2 + (-2c_3 + 2c_2^2)e_n^3 + O(e_n^4).$$

7. We then introduce:

$$x^*_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, e_n = x_n - \alpha$$

Simplifying, we get:

$$x^*_{n+1} = \alpha + e_n - \left(e_n - c_2 e_n^2 + (-2c_3 + 2c_2^2)e_n^3 + O(e_n^4)\right)$$
$$= \alpha + c_2 e_n^2 - (-2c_3 + 2c_2^2)e_n^3 + O(e_n^4).$$

8.Now, we find the midpoint between x_n and x^*_{n+1}:

$$\frac{x_n + x^*_{n+1}}{2} = \frac{e_n + \alpha + \alpha + c_2 e_n^2 - (-2c_3 + 2c_2^2)e_n^3 + O(e_n^4)}{2}$$

$$= \frac{2\alpha}{2} + \frac{e_n}{2} + \frac{c_2 e_n^2}{2} + \frac{2(c_3 - c_2^2)e_n^3}{2} + O(e_n^4)$$

$$= \alpha + \frac{e_n}{2} + \frac{c_2 e_n^2}{2} + (c_3 - c_2^2)e_n^3 + O(e_n^4).$$

9.Now we substitute $\frac{x_n + x_{n+1}^*}{2}$ in $f'(x_n)$ to find $f'\left(\frac{x_n + x_{n+1}^*}{2}\right)$, given $e_n = x_n - \alpha$:

$$f'\left(\frac{x_n + x_{n+1}^*}{2}\right) = f'(\alpha)\left(1 + 2c_2\left(\frac{x_n + x_{n+1}^*}{2} - \alpha\right) + 3c_3\left(\frac{x_n + x_{n+1}^*}{2} - \alpha\right)^2 + O(e_n^3)\right).$$

We know:

$$\frac{x_n + x_{n+1}^*}{2} - \alpha = \frac{1}{2}e_n + \frac{c_2}{2}e_n^2 + (c_3 - c_2^2)e_n^3 + O(e_n^4).$$

Therefore,

$$\begin{aligned} f'\left(\frac{x_n + x_{n+1}^*}{2}\right) &= f'(\alpha)\left(1 + 2c_2\left(\frac{1}{2}e_n + \frac{c_2}{2}e_n^2\right) + 3c_3\left(\frac{1}{2}e_n\right)^2 + O(e_n^3)\right) \\ &= f'(\alpha)\left(1 + c_2 e_n + c_2^2 e_n^2 + \frac{3}{4}c_3 e_n^2 + O(e_n^3)\right) \\ &= f'(\alpha)\left(1 + c_2 e_n + \left(c_2^2 + \frac{3}{4}c_3\right)e_n^2 + O(e_n^3)\right). \end{aligned}$$

10.

$$2f'\left(\frac{x_n + x_{n+1}^*}{2}\right) = f'(\alpha)\left(2 + 2c_2 e_n + \left(2c_2^2 + \frac{3}{2}c_3\right)e_n^2 + O(e_n^3)\right).$$

11.

$$2f'\left(\frac{x_{n+1}^* + x_n}{2}\right) - f'(x_n) = f'(\alpha)\left(1 + \left(2c_2^2 - \frac{3}{2}c_3\right)e_n^2 + O(e_n^3)\right).$$

12.

$$\frac{f(x_n)}{2f'\left(\frac{x_{n+1}^* + x_n}{2}\right) - f'(x_n)} = \frac{f'(\alpha)\left(e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right)}{f'(\alpha)\left(1 + \left(2c_2^2 - \frac{3}{2}c_3\right)e_n^2 + O(e_n^3)\right)}.$$

Using the same procedure as in (5) to expand the denominator:

$$\frac{1}{1 + \left(2c_2^2 - \frac{3}{2}c_3\right)e_n^2 + O(e_n^3)} \approx 1 - \left(2c_2^2 - \frac{3}{2}c_3\right)e_n^2 + O(e_n^3).$$

We then multiply the numerator by the expanded denominator:

$$\begin{aligned} \frac{f(x_n)}{2f'\left(\frac{x_{n+1}^* + x_n}{2}\right) - f'(x_n)} &= \left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]\left[1 - \left(2c_2^2 - \frac{3}{2}c_3\right)e_n^2 + O(e_n^3)\right] \\ &= e_n - \left(2c_2^2 - \frac{3}{2}c_3\right)e_n^3 + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4) \\ &= e_n + c_2 e_n^2 - \left(2c_2^2 - \frac{5}{2}c_3\right)e_n^3 + O(e_n^4). \end{aligned}$$

13.Subtracting α from both sides to express the MHN method in terms of the error e_n:

$$x_{n+1} - \alpha = x_n - \alpha - \frac{1}{2}\left(\frac{f(x_n)}{f'(x_n)} + \frac{f(x_n)}{2f'\left(\frac{x_n + x_{n+1}^*}{2}\right) - f'(x_n)}\right)$$

We then have:

$$e_{n+1} = e_n - \frac{1}{2}\left(\frac{f(x_n)}{f'(x_n)} + \frac{f(x_n)}{2f'\left(\frac{x_n + x^*_{n+1}}{2}\right) - f'(x_n)}\right)$$

Expanding:

$$\begin{aligned} e_{n+1} &= e_n - \frac{1}{2}\left[\left(e_n - c_2 e_n^2 + \left(2c_2^2 - 2c_3\right) e_n^3 + O(e_n^4)\right)\right. \\ &\quad \left. + \left(e_n + c_2 e_n^2 - \left(2c_2^2 - \frac{5}{2}c_3\right) e_n^3 + O(e_n^4)\right)\right] \\ &= e_n - \frac{1}{2}\left(2e_n + \left(2c_2^2 - 2c_3\right) e_n^3 - \left(2c_2^2 - \frac{5}{2}c_3\right) e_n^3 + O(e_n^4)\right) \\ &= e_n - \frac{1}{2}\left(2e_n + \frac{1}{2}c_3 e_n^3 + O(e_n^4)\right) \\ &= e_n - e_n - \frac{1}{4}c_3 e_n^3 + O(e_n^4) \\ &= -\frac{1}{4}c_3 e_n^3 + O(e_n^4) \end{aligned}$$

$$e_{n+1} = -\frac{1}{4}c_3 e_n^3 + O(e_n^4).$$

The exponent of e_n is 3, so the MHN method has cubic convergence.

4.3 Arithmetic mean Newton (AN) method

This section presents the derivation of a second modified Newton method with cubic convergence. The approach is based on Newton's theorem and applies an integral approximation using the arithmetic mean.

1. Start with Newton's theorem

$$f(x) = f(x_n) + \int_{x_n}^{x} f'(t)\,dt.$$

2.Approximate the integral

To approximate the integral $\int_{x_n}^{x} f'(t)\,dt$, the derivative $f'(t)$ is replaced with its arithmetic mean over the interval $[x_n, x]$:

$$f'(t) \approx \frac{f'(x_n) + f'(x)}{2}$$

Substituting this approximation into the integral:

$$\int_{x_n}^{x} f'(t)\,dt \approx \int_{x_n}^{x} \frac{f'(x_n) + f'(x)}{2}\,dt$$

Evaluating the integral gives:

$$\int_{x_n}^{x} f'(t)\,dt \approx \frac{f'(x_n) + f'(x)}{2}(x - x_n).$$

3. Substitute the integral approximation into Newton's theorem

$$f(x) = f(x_n) + \frac{f'(x_n) + f'(x)}{2}(x - x_n).$$

4. Solve for x

To find the root, set $f(x) = 0$:

$$0 = f(x_n) + \frac{f'(x_n) + f'(x)}{2}(x - x_n)$$

Rearranging terms:

$$x - x_n = -\frac{2f(x_n)}{f'(x_n) + f'(x)}$$

$$x = x_n - \frac{2f(x_n)}{f'(x_n) + f'(x)}.$$

5. Approximate x

To simplify the formula further, we approximate x by defining the next iteration x_{n+1} as:

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

Using this approximation, the iterative formula is refined as:

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

6. Simplify further

To simplify the iterative formula further, rewrite x_{n+1} as:

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) + f'(x^*_{n+1})},$$

where x^*_{n+1} is approximated using the classical Newton's method:

$$x^*_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

The derivation above results in the **arithmetic mean Newton (AN) method**.

Proof of cubic convergence of the arithmetic mean Newton (AN) method

We intend to demonstrate that the AN method exhibits cubic convergence by verifying its error equation, which shows that the exponent of e_n is 3.

1.Arithmetic mean Newton (AN) method:

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'\left(x^*_{n+1}\right) + f'(x_n)}, \quad x^*_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

2.Previously, we know $x^*_{n+1} = \alpha + c_2 e_n^2 - (-2c_3 + 2c_2^2)e_n^3 + O(e_n^4)$ and we have:

$$f'(x_n) = f'(\alpha)\left[1 + 2c_2 e_n + 3c_3 e_n^2 + O(e_n^3)\right]$$

Using these, we calculate $f'\left(x^*_{n+1}\right)$:

$$f'\left(x^*_{n+1}\right) = f'(\alpha)\left[1 + 2c_2\left(x^*_{n+1} - \alpha\right) + 3c_3\left(x^*_{n+1} - \alpha\right)^2 + O(e_n^3)\right]$$

$$= f'(\alpha)\left[1 + 2c_2\left(c_2 e_n^2\right) + O(e_n^3)\right]$$

$$= f'(\alpha)\left[1 + 2c_2^2 e_n^2 + O(e_n^3)\right].$$

3.

$$f'\left(x_{n+1}^*\right) + f'(x_n) = f'(\alpha)\left[2 + 2c_2 e_n + (2c_2^2 + 3c_3)e_n^2 + O(e_n^3)\right].$$

4.

$$\frac{2f(x_n)}{f'\left(x_{n+1}^*\right) + f'(x_n)} = \frac{2f'(\alpha)\left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]}{f'(\alpha)\left[2 + 2c_2 e_n + (2c_2^2 + 3c_3)e_n^2 + O(e_n^3)\right]}$$

We cancel out $f'(\alpha)$ and take 2 as a common factor:

$$= \frac{2\left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]}{2\left[1 + c_2 e_n + \frac{(2c_2^2+3c_3)}{2} e_n^2 + O(e_n^3)\right]}$$

$$= \frac{e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)}{1 + c_2 e_n + \frac{\left(2c_2^2+3c_3\right)}{2} e_n^2 + O(e_n^3)}.$$

5. We then simplify the fraction by using the Maclaurin series expansion:

$$\frac{1}{1+x} \approx 1 - x + x^2 + O(x^3)$$

$$\frac{1}{1 + c_2 e_n + \left(\frac{3c_3+2c_2^2}{2}\right) e_n^2 + O(e_n^3)} \approx$$

$$1 - \left(c_2 e_n + \left(\frac{3c_3 + 2c_2^2}{2}\right) e_n^2\right) + (c_2 e_n)^2 + O(e_n^3)$$

$$\approx 1 - c_2 e_n - \left(\frac{3c_3 + 2c_2^2}{2}\right) e_n^2 + c_2^2 e_n^2 + O(e_n^3)$$

$$\approx 1 - c_2 e_n - \left(\frac{3c_3}{2} + c_2^2 - c_2^2\right) e_n^2 + O(e_n^3)$$

$$\approx 1 - c_2 e_n - \frac{3}{2} c_3 e_n^2 + O(e_n^3).$$

6.We now multiply the numerator by the expanded denominator:

$$\frac{2f(x_n)}{f'\left(x_{n+1}^*\right) + f'(x_n)} = \left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]\left[1 - c_2 e_n - \frac{3}{2} c_3 e_n^2 + O(e_n^3)\right]$$

$$= e_n - c_2 e_n^2 - \frac{3}{2} c_3 e_n^3 + c_2 e_n^2 - c_2^2 e_n^3 + c_3 e_n^3 + O(e_n^4)$$

$$= e_n - \frac{1}{2} c_3 e_n^3 - c_2^2 e_n^3 + O(e_n^4).$$

7. Now after substituting $\frac{2f(x_n)}{f'\left(x_{n+1}^*\right)+f'(x_n)}$ in the method, we will then end up with the error equation:

$$e_{n+1} = e_n - \left(e_n - \frac{1}{2} c_3 e_n^3 - c_2^2 e_n^3 + O(e_n^4)\right)$$

$$e_{n+1} = \frac{1}{2} c_3 e_n^3 + c_2^2 e_n^3 + O(e_n^4)$$

$$e_{n+1} = \left(\frac{1}{2} c_3 + c_2^2\right) e_n^3 + O(e_n^4)$$

This concludes the proof.

4.4 Midpoint Newton (MN) Method

This section presents the derivation of a third modified Newton method that achieves cubic convergence. The approach is based on Newton's theorem and uses the trapezoidal rule to approximate the integral.

1. Start with Newton's theorem

$$f(x) = f(x_n) + \int_{x_n}^{x} f'(t)\,dt.$$

2. Approximate the integral

$$\int_{x_n}^{x} f'(t)\,dt \approx (x - x_n) f'\left(\frac{x_n + x}{2}\right)$$

Here, the derivative is evaluated at the midpoint $\frac{x_n+x}{2}$ to improve the accuracy of the approximation.

3. Substitute the approximation into Newton's theorem

$$f(x) = f(x_n) + (x - x_n) f'\left(\frac{x_n + x}{2}\right).$$

4. Solve for x

To find the root, set $f(x) = 0$:

$$0 = f(x_n) + (x - x_n) f'\left(\frac{x_n + x}{2}\right)$$

Rearranging terms to solve for x:

$$x = x_n - \frac{f(x_n)}{f'\left(\frac{x_n+x}{2}\right)}.$$

5. Approximate x

To simplify the computation, we approximate x by introducing the next iteration x_{n+1}:

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

Using this approximation, the iterative formula becomes:

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

Simplify the denominator further:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'\left(\frac{x_n + x^*_{n+1}}{2}\right)}, \quad \text{where } x^*_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The derived formula is known as the **midpoint Newton (MN) method**.

Proof of cubic convergence of midpoint Newton (MN) method

We now aim to show that the MN method converges cubically by showing its error equation.

1.Midpoint Newton method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'\left(\frac{1}{2}(x_n + x^*_{n+1})\right)}.$$

2.We previously know that

$$f(x_n) = f'(\alpha)\left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]$$

and

$$f'\left(\frac{1}{2}(x_n + x^*_{n+1})\right) = f'(\alpha)\left[1 + c_2 e_n + \left(c_2^2 + \frac{3}{4}c_3\right)e_n^2 + O(e_n^3)\right]$$

We then have:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'\left(\frac{1}{2}(x_n + x^*_{n+1})\right)}$$

$$= x_n - \frac{f'(\alpha)\left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]}{f'(\alpha)\left[1 + c_2 e_n + \left(c_2^2 + \frac{3}{4}c_3\right)e_n^2 + O(e_n^3)\right]}.$$

3. Using the Maclaurin series expansion:

$$\frac{1}{1 + c_2 e_n + \left(c_2^2 + \frac{3}{4}c_3\right)e_n^2 + O(e_n^3)} \approx 1 - c_2 e_n - \left(c_2^2 + \frac{3}{4}c_3\right)e_n^2 + c_2^2 e_n^2 + O(e_n^3)$$

$$\approx 1 - c_2 e_n - \frac{3}{4}c_3 e_n^2 + O(e_n^3).$$

4. Multiplying the numerator by the expanded denominator:

$$\frac{f(x_n)}{f'\left(\frac{1}{2}(x_n + x^*_{n+1})\right)} = \left[e_n + c_2 e_n^2 + c_3 e_n^3 + O(e_n^4)\right]\left[1 - c_2 e_n - \frac{3}{4}c_3 e_n^2 + O(e_n^3)\right]$$

$$= e_n - c_2 e_n^2 - \frac{3}{4}c_3 e_n^3 + c_2 e_n^2 - c_2^2 e_n^3 + c_3 e_n^3 + O(e_n^4)$$

$$= e_n + \left(\frac{1}{4}c_3 - c_2^2\right)e_n^3 + O(e_n^4).$$

5. We then conclude:

$$e_{n+1} = e_n - \left[e_n + \left(\frac{1}{4}c_3 - c_2^2\right)e_n^3 + O(e_n^4)\right]$$

$$= \left(c_2^2 - \frac{1}{4}c_3\right)e_n^3 + O(e_n^4)$$

This completes the proof that the MN converges cubically since the exponent of e_n is 3.

4.5 Harmonic mean Newton (HN) method

The derivation of the harmonic mean Newton (HN) method follows the exact steps as the MHN method. However, unlike the MHN method, it does not involve replacing the arithmetic mean with the midpoint value. This method achieves cubic convergence too.

The derived formula for the HN method is:

$$x_{n+1} = x_n - \frac{f(x_n)}{2}\left(\frac{1}{f'(x_n)} + \frac{1}{f'\left(x^*_{n+1}\right)}\right), \quad \text{where } x^*_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Proof of cubic convergence of harmonic mean Newton (HN) method

The HN method is also a method of order 3, the derivation below proves this.

1.Harmonic mean Newton (HN) Method:

$$x_{n+1} = x_n - \frac{1}{2}\left(\frac{f(x_n)}{f'(x_n)} + \frac{f(x_n)}{f'(x^*_{n+1})}\right).$$

2. We now calculate $\frac{f(x_n)}{f'(x^*_{n+1})}$ using $f(x_n)$ and $f'(x^*_{n+1})$ that we calculated previously:

$$\frac{f(x_n)}{f'(x^*_{n+1})} = \frac{f'(\alpha)\left[e_n + c_2e_n^2 + c_3e_n^3 + O(e_n^4)\right]}{f'(\alpha)\left[1 + 2c_2^2e_n^2 + O(e_n^3)\right]}.$$

3. Applying Maclaurin series expansion to approximate the denominator:

$$\frac{1}{1 + 2c_2^2e_n^2 + O(e_n^3)} \approx 1 - 2c_2^2e_n^2 + O(e_n^3).$$

4. Multiplying the expanded denominator by the Numerator:

$$\begin{aligned}\frac{f(x_n)}{f'(x^*_{n+1})} &= \left[e_n + c_2e_n^2 + c_3e_n^3 + O(e_n^4)\right]\left[1 - 2c_2^2e_n^2 + O(e_n^3)\right] \\ &= e_n - 2c_2^2e_n^3 + c_2e_n^2 + c_3e_n^3 + O(e_n^4) \\ &= e_n + c_2e_n^2 + \left(-2c_2^2 + c_3\right)e_n^3 + O(e_n^4).\end{aligned}$$

5. Previously we calculated $\frac{f(x_n)}{f'(x_n)} = e_n - c_2e_n^2 + (-2c_3 + 2c_2^2)e_n^3 + O(e_n^4)$ we then find:

$$\frac{f(x_n)}{f'(x_n)} + \frac{f(x_n)}{f'(x^*_{n+1})}$$

$$\begin{aligned}&= \mathrm{e}_n - c_2e_n^2 + (-2c_3 + 2c_2^2)e_n^3 + e_n + c_2e_n^2 + (-2c_2^2 + c_3)e_n^3 + O(e_n^4) \\ &= 2\ \mathrm{e}_n + (2c_2^2 - 2c_3 - 2c_2^2 + c_3)e_n^3 + O(e_n^4) \\ &= 2\ \mathrm{e}_n - c_3e_n^3 + O(e_n^4).\end{aligned}$$

6. Finally, we conclude:

$$\begin{aligned}e_{n+1} &= e_n - \frac{1}{2}\left[2e_n - c_3e_n^3 + O(e_n^4)\right] \\ &= e_n - e_n + \frac{c_3}{2}e_n^3 + O(e_n^4) \\ &= \frac{c_3}{2}e_n^3 + O(e_n^4)\end{aligned}$$

Thus, we have shown that the HN method has cubic convergence.

Chapter 5

5 Numerical exploration of Newton methods for solving nonlinear equations in one variable

In this chapter, we aim to study the performance of the CN method and its modifications- AN, MN, HN, and MHN for solving nonlinear equations. These methods are applied to a variety of functions to evaluate their effectiveness in finding simple and multiple roots.

The nonlinear functions under consideration are:

- **Functions with simple roots:**

$$f_1(x) = x^3 + 4x^2 - 10, \quad \alpha = 1.3652300134140969$$
$$f_2(x) = x^2 - e^x - 3x + 2, \quad \alpha = 0.25753028543986084$$
$$f_3(x) = xe^{x^2} - \sin^2(x) + 3\cos(x) + 5, \quad \alpha = -1.207647827130919$$
$$f_4(x) = \sin(x)e^{-x} + \ln(x^2 + 1), \quad \alpha = 0$$
$$f_5(x) = (x-1)^2 - 1, \quad \alpha_1 = 0, \quad \alpha_2 = 2$$
$$f_6(x) = (x-1)^3 - 2, \quad \alpha = 2.2599210498948734$$
$$f_7(x) = (x-1)^6 - 1, \quad \alpha_1 = 0, \quad \alpha_2 = 2$$
$$f_8(x) = \cos(x) - x, \quad \alpha = 0.73908513321516067$$
$$f_9(x) = \sin^2(x) - x^2 + 1, \quad \alpha_1 = 1.4044916482153411, \quad \alpha_2 = -1.4044916482153411$$
$$f_{10}(x) = e^{x^2+7x-30} - 1, \quad \alpha_1 = 3, \quad \alpha_2 = -10$$
$$f_{11}(x) = x + e^x + \frac{10}{1+x^2} - 5,$$

- **Functions with multiple roots:**

$$f_{12}(x) = (x-1)\ln(x), \quad \alpha_1 = \alpha_2 = 1$$
$$f_{13}(x) = x^5 + 2x^3, \quad \alpha_1 = \alpha_2 = \alpha_3 = 0$$

The iterative methods examined in this study are defined by the following formulas:

- **Classical Newton method (CN):**

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

- **Arithmetic mean Newton (AN) method:**

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

- **Midpoint Newton (MN) method:**

$$x_{n+1} = x_n - \frac{f(x_n)}{f'\left(\frac{x_n + x_{n+1}^*}{2}\right)}$$

- **Harmonic mean Newton (HN) method:**

$$x_{n+1} = x_n - \frac{f(x_n)}{2}\left(\frac{1}{f'(x_n)} + \frac{1}{f'\left(x_{n+1}^*\right)}\right)$$

- **Midpoint-harmonic mean Newton (MHN) method:**

$$x_{n+1} = x_n - \frac{f(x_n)}{2}\left(\frac{1}{f'(x_n)} + \frac{1}{2f'\left(\frac{1}{2}(x_n + x^*_{n+1})\right) - f'(x_n)}\right)$$

To evaluate the performance of these methods, we analyze the following metrics:

1. **Number of function evaluations (NFE):** This represents the total number of evaluations of $f(x)$ and its derivative $f'(x)$ required to achieve convergence. It is a key measure of the computational cost associated with each method.

2. **Accuracy of approximation:** Measured by the absolute error $|x_{n+1} - \alpha|$, which indicates how close the computed root is to the true root α. When the true root is unknown, $|x_{n+1} - x_n|$ is commonly used instead to assess the convergence and precision of the approximation.

3. **Residual:** The value of $|f(x_{n+1})|$ at the computed root, which quantifies how closely the computed root satisfies the equation.

4. **Order of convergence (COC):** The computational order of convergence provides insight into the rate at which the sequence of iterates converges to the root. The COC is calculated using the following formula:

$$COC \simeq \frac{\ln\left|\frac{x_{n+1}-\alpha}{x_n-\alpha}\right|}{\ln\left|\frac{x_n-\alpha}{x_{n-1}-\alpha}\right|}$$

   where α is the true root. This measure is crucial for assessing the convergence behavior of each method.

By analyzing these metrics, this chapter provides a detailed evaluation of the iterative methods applied to the selected nonlinear equations, focusing on both simple and multiple root cases. The study aims to understand the iterative behavior, convergence properties, and computational requirements of each method.

5.1 Functions with simple roots

Table 1:Number of function evaluations required by Newton methods across test functions

This table summarizes the number of function evaluations (NFE) required by various Newton methods applied to different test functions. Each method was executed with a maximum of 1000 iterations to avoid excessive computation in cases of slow or non-convergence. A stopping criterion of 1×10^{-8} was used, ensuring that the iteration process stopped once the approximate root reached the desired level of precision. The initial guess for all methods is denoted by x_0.

Function	x_0	**CN**	**AN**	**MN**	**HN**	**MHN**
f_1	-0.5	260	15	30	129	30
	-0.3	106	15	51	147	12
	1	8	9	9	6	6
	2	8	9	9	9	6
f_2	2	8	9	9	9	9
	3	10	12	12	12	12
f_3	-3	26	27	24	21	15
	-2	14	15	15	12	9
f_4	3	10	15	9	12	9
f_5	3.5	10	9	9	9	9
f_6	1.85	10	9	9	9	9
	3	10	9	9	9	9
f_7	1.5	30	1401	174	21	18
	3	16	15	15	12	12
f_8	-0.3	10	9	9	9	9
	1	6	6	6	6	6
	1.7	8	9	9	9	9
f_9	1	10	9	9	9	9
	2	10	9	9	9	9
f_{10}	3.25	14	15	15	12	9
	3.5	22	21	21	18	12
f_{11}	-0.8	8	9	9	9	9

The results presented in Table 1 of [7] use a stopping tolerance of 1×10^{-14}, whereas the results in this table are based on a tolerance of 1×10^{-8}. Despite this difference, both tables lead to the same conclusion: cubically convergent methods, especially MHN, perform better than CN. In many cases, MHN needs fewer function evaluations, which means it is more efficient than CN.

Table 2: Convergence behavior and error analysis for Newton methods and fixed point method applied on test functions

Each method was executed with a maximum of 1000 iterations and a stopping tolerance of 1×10^{-12}. The table reports the number of iterations required for convergence (N), the computational order of convergence (COC), the absolute error $e_N = |x_N - \alpha|$, and the residual $|f(x_N)|$ at the final iteration.

		N	COC	e_N	$\lvert f(x_N)\rvert$
$f_{1(x_0=-0.3)}$	CN	53	2.0204	2.2204e−16	3.5527e−15
	AN	6	3.0543	0	0
	MN	17	2.9935	1.1990e−14	1.9895e−13
	HN	50	3.0856	0	0
	MHN	5	2.9781	0	0
	FP	41	1.0001	5.7510e−13	9.4964e−12
$f_{2(x_0=2)}$	CN	4	2.0003	9.8754e−14	3.7281e−13
	AN	4	3.0085	0	0
	MN	3	2.9333	5.5511e−17	0
	HN	3	3.0152	3.1808e−14	1.2079e−13
	MHN	4	2.9696	5.5511e−17	0
	FP	22	1.0001	4.7989e−13	1.8137e−12
$f_{3(x_0=-3)}$	CN	13	2.0000	1.7542e−14	3.5794e−13
	AN	9	2.9469	0	2.6645e−15
	MN	8	3.0187	8.8818e−16	1.9540e−14
	HN	8	3.0123	0	2.6645e−15
	MHN	6	2.9972	0	2.6645e−15
	FP	1000	-0.0000	2.6582	1.7764e−15
$f_{4(x_0=3)}$	CN	6	1.7390	5.7627e−17	5.7627e−17
	AN	5	3.7265	6.5872e−17	6.5872e−17
	MN	4	2.9160	1.4290e−17	1.4290e−17
	HN	4	2.6615	1.0925e−16	1.0925e−16
	MHN	3	2.0546	7.7144e−17	7.7144e−17
	FP	4	2.6517	8.8596e−17	8.8596e−17
$f_{5(x_0=3.5)}$	CN	6	1.9997	0	0
	AN	4	2.9528	0	0
	MN	4	2.9528	0	0
	HN	3	3.3261	0	0
	MHN	3	3.3261	0	0
	FP	41	1	5.0893e−13	1.0179e−12
$f_{6(x_0=1.85)}$	CN	5	2.0000	2.7534e−13	1.3123e−12
	AN	4	3.0335	4.4409e−16	8.8818e−16
	MN	4	3.0193	4.4409e−16	8.8818e−16
	HN	3	3.1838	4.4409e−16	8.8818e−16
	MHN	3	3.7437	4.4409e−16	8.8818e−16
	FP	48	0.9980	7.5939e−13	3.6151e−12

$f_{7(x_0=1.5)}$	CN	15	1.9969	8.8818e−16	5.3291e−15
	AN	467	3.0708	0	0
	MN	58	2.9964	1.3323e−15	7.9936e−15
	HN	7	3.0151	0	0
	MHN	6	2.8617	0	0
	FP	269	0.9982	9.0217e−13	5.4130e−12
$f_{8(x_0=-0.3)}$	CN	5	2.0047	3.3307e−16	6.6613e−16
	AN	3	3.0560	4.4409e−16	6.6613e−16
	MN	4	3.0522	0	0
	HN	3	2.9932	1.0669e−13	1.7863e−13
	MHN	3	3.0330	3.3307e−16	6.6613e−16
	FP	68	0.9999	7.1143e−13	1.1906e−12
$f_{9(x_0=1)}$	CN	5	1.9998	3.0598e−13	7.5939e−13
	AN	4	3.0410	2.2204e−16	4.4409e−16
	MN	4	3.0274	0	3.3307e−16
	HN	3	3.6724	0	3.3307e−16
	MHN	3	4.4841	0	3.3307e−16
	FP	13	1.0001	7.9758e−13	1.9804e−12
$f_{10(x_0=3.25)}$	CN	8	1.9988	0	0
	AN	5	2.9915	1.6920e−13	2.1991e−12
	MN	5	2.9331	4.4409e−16	7.1054e−15
	HN	5	3.0204	0	0
	MHN	4	2.7648	0	0
	FP	171	0.9970	9.0550e−13	1.1770e−11

Table 3: Convergence behavior and error analysis for Newton methods and fixed point method applied to $\mathbf{f_{11}}$

Since the exact solution for $\mathbf{f_{11}}$ is not known, the type of error used was changed to $e_N = |x_N - x_{N-1}|$, which measures the difference between successive approximations. The initial guess used for all methods was $x_0 = -0.8$.

Method	N	COC	e_N	$\lvert f(x_N)\rvert$
CN	5	1.9998	1.1102e−16	8.8818e−16
AN	3	2.9028	2.2204e−14	8.8818e−16
MN	3	2.9858	8.2667e−13	8.8818e−16
HN	3	3.0622	1.3323e−14	8.8818e−16
MHN	3	3.1016	2.5535e−15	8.8818e−16
FP	20	1.0000	8.0780e−13	1.1351e−12

Interpretation of convergence behavior of the methods across all functions

Function 1 ($f_1(x_0 = -0.3)$)

- **Classical Newton (CN):** Converges quadratically (COC ≈ 2), requiring 53 iterations, which is relatively high. This indicates a slow convergence compared to modified methods.
- **Modified Newton methods (AN, MN, HN, MHN):** AN, MN, HN, and MHN achieve cubic convergence (COC ≈ 3), requiring fewer iterations (AN: 6, MN: 17, HN: 50, MHN: 5) compared to CN.
- **Fixed point method (FP):** Linear convergence (COC ≈ 1) is as expected, requiring 41 iterations. It performs better than CN in terms of iteration count but is much slower compared to modified methods, except HN.

- **Conclusion:** MHN is the best, requiring only 5 iterations and achieving cubic convergence (COC ≈ 3).

Function 2 ($f_2(x_0 = 2)$)

- **Classical Newton (CN):** Achieves quadratic convergence (COC ≈ 2) with just 4 iterations.
- **Modified Newton methods (AN, MN, HN, MHN):** All modified methods converge cubically, AN, and MHN converge in 4 iterations, while HN and MN converge in 3 iterations.
- **Fixed point method (FP):** Demonstrates linear convergence, converging in 22 iterations
- **Conclusion:** The modified Newton methods HN and MN are the most efficient, achieving cubic convergence in just 3 iterations.

Function 3 ($f_3(x_0 = -3)$)

- **Classical Newton (CN):** Achieves expected quadratic convergence (COC $= 2$) in 13 iterations.
- **Modified Newton methods (AN, MN, HN, MHN):** All converge cubically (COC ≈ 3) with iteration counts: AN (9), MN (8), HN (8), and MHN (6).
- **Fixed point method (FP):** Negative order of convergence (COC $= -0.0000$) with 1000 iterations.
- **Conclusion:** Among the modified Newton methods, MHN is the most efficient, converging cubically in just 6 iterations.

Function 4 ($f_4(x_0 = 3)$)

- **Classical Newton (CN):** Demonstrates quadratic convergence with convergence achieved in 6 iterations.
- **Modified Newton methods (AN, MN, HN, MHN):**
  - AN: This method has a high convergence order (COC ≈ 3.7265), indicating a faster than expected convergence with 5 iterations.
  - MN: Converges cubically in 4 iterations.
  - HN: Achieves convergence in 4 iterations, with COC ≈ 2.6615.
  - MHN: Converges in 3 iterations but with a reduced convergence order (COC ≈ 2), performing slightly better in iteration count.
- **Fixed point method (FP):** converges in 4 iterations and achieves (COC ≈ 2.6517), which is higher than its theoretical linear convergence (COC $= 1$).
- **Conclusion:** MHN is the best method, since it only needs 3 iterations to converge.

Function 5 ($f_5(x_0 = 3.5)$)

- **Classical Newton (CN):** Quadratic convergence is observed with only 6 iterations.
- **Modified Newton methods(AN, MN, HN, MHN):** All modified methods converge cubically, AN, and MN converge in 4 iterations, while HN and MHN converge in 3 iterations.
- **Fixed point method (FP):** Linear convergence in 41 iterations highlights its inefficiency compared to Newton methods.
- **Conclusion:** HN and MHN excel here with minimal iterations.

Function 6 ($f_6(x_0 = 1.85)$)

- **Classical Newton (CN):** Achieves quadratic convergence in 5 iterations, indicating good performance.
- **Modified Newton methods (AN, MN, HN, MHN):**
  - AN and MN: They converge cubically in 4 iterations.
  - HN: Converges cubically in 3 iterations.
  - MHN: This method has a high convergence order (COC ≈ 3.7437), indicating a faster than expected convergence with 3 iterations.
- **Fixed point method (FP):** Converges linearly in 48 iterations, much slower than Newton methods.
- **Conclusion:** MHN is the most efficient, with just 3 iterations and the highest convergence order.

Function 7 ($f_7(x_0 = 1.5)$)

- **Classical Newton (CN):** Quadratic convergence is achieved in 15 iterations.
- **Modified Newton methods (AN, MN, HN, MHN):**
  - AN: Converges cubically but takes a surprising 467 iterations, which is far higher than expected.
  - MN: Achieves cubic convergence in 58 iterations, still relatively slow.
  - HN: Converges cubically in 7 iterations, making it far more efficient than AN and MN.
  - MHN: Exhibits cubic convergence, reaching the desired accuracy within 6 iterations.
- **Fixed point method (FP):** Converges linearly in 269 iterations.
- **Conclusion:** MHN is the most efficient, converging in just 6 iterations with cubic convergence.

Function 8 ($f_8(x_0 = -0.3)$)

- **Classical Newton (CN):** Achieves quadratic convergence in 5 iterations, reflecting strong performance.
- **Modified Newton methods (AN, MN, HN, MHN):** All of them are converging cubically in just 3 iterations, except MN converges in 4 iterations.
- **Fixed point method (FP):** Linear convergence in 68 iterations is far slower than Newton methods, as expected.
- **Conclusion:** Modified Newton methods specifically (AN,HN,MHN) excel here with minimal iterations.

Function 9 ($f_9(x_0 = 1)$)

- **Classical Newton (CN):** Quadratic convergence is achieved in 5 iterations, indicating good performance.
- **Modified Newton methods (AN, MN, HN, MHN):**
  - AN and MN: They converge cubically in 4 iterations.
  - HN: Achieves convergence in 3 iterations, but with COC > 3, showing better performance than expected.
  - MHN: Similar to HN, it converges in 3 iterations with a better than expected convergence order (COC > 4).
- **Fixed point method (FP):** Linear convergence in 13 iterations is slower but relatively competitive for a fixed-point iteration.
- **Conclusion:** MHN is the most efficient converging in 3 iterations with (COC ≈ 4.4841).

Function 10 ($f_{10}(x_0 = 3.25)$)

- **Classical Newton (CN):** Quadratic convergence is observed in 8 iterations.
- **Modified Newton methods (AN, MN, HN, MHN):** All methods converge cubically, AN, MN, and HN converge in 5 iterations, while MHN converges in 4 iterations.
- **Fixed point method (FP):** Linear convergence with 171 iterations.
- **Conclusion:** MHN is the best, requiring only 4 iterations.

Function 11 ($f_{11}(x_0 = -0.8)$)

- **Classical Newton (CN):** Converges quadratically in 5 iterations.
- **Modified Newton methods (AN, MN, HN, MHN):** All exhibit cubic convergence, requiring only 3 iterations.
- **Fixed point method (FP):** Converges linearly in 20 iterations.
- **Conclusion:** The modified Newton methods are notably efficient, converging in just 3 iterations.

Table 4:Convergence order (COC) analysis for f_9 using various Newton methods and the fixed point method

n	CN	AN	MN	HN	MHN	FP
1	-	-	-	-	-	-
2	3.8949133938	3.3759734189	3.3392315721	3.6724291342	4.4840558791	1.3386767825
3	1.8671371089	3.0410364278	3.0273902123	-	-	1.1085048955
4	1.9877745769	0.9007499064	-	-	-	1.0173186105
5	1.9997591033	-	-	-	-	1.0021209587
6	-	-	-	-	-	1.0002481278
7	-	-	-	-	-	1.0000288637
8	-	-	-	-	-	1.0000033549
9	-	-	-	-	-	1.0000003969
10	-	-	-	-	-	1.0000001216
11	-	-	-	-	-	0.9999999448
12	-	-	-	-	-	1.0000048302
13	-	-	-	-	-	1.0001436004

Table 5: Convergence order (COC) analysis for f_3 using various Newton methods and the fixed point method

n	CN	AN	MN	HN	MHN	FP
1	-	-	-	-	-	-
2	1.1587386937	1.2563553487	1.3074420164	1.3630210580	1.8811214426	-2.8099873678
3	1.1837615890	1.3277232850	1.4152938219	1.5212041471	3.7133277232	-0.1204662990
4	1.2174703889	1.4468286185	1.6170328804	1.8512212551	2.6197729027	0.3029923923
5	1.2645871812	1.6627475939	2.0209562379	2.4969498865	2.9972129027	-1.2924466441
6	1.3328106842	2.0559173515	2.6431079604	3.0210252098	-	-0.7187331984
7	1.4335492457	2.6040777467	2.9750472473	3.0123382499	-	-0.5100536402
8	1.5780415265	2.9468760981	3.0186817631	-	-	-0.3694254984
9	1.7571047889	-	-	-	-	-0.2099456950
10	1.9113341776	-	-	-	-	0.2739484620
11	1.9832780722	-	-	-	-	-1.6889701642
12	1.9988392834	-	-	-	-	-0.6959366085
13	2.000022729	-	-	-	-	-0.5000316291

Unexpected convergence order in Newton methods and fixed point method

This section examines unexpected behaviors in the observed order of convergence for Newton methods and the fixed-point method. Refer to **Table 2** for detailed numerical results of each method applied to the corresponding functions.

Function 3: Fixed point method fails to converge with a lower than expected order

In the case of function 3, the fixed-point method exhibited poor convergence behavior. The iteration function used was

$$g(x) = \sqrt{\ln\left(\frac{\sin^2(x) - 3\cos(x) - 5}{x}\right)},$$

which does not satisfy the necessary fixed-point condition $f(x) = 0 \iff g(x) = x$. However, the derivative $|g'(x)|$ evaluated at the true root is less than 1, but the root itself is not a fixed point of the chosen $g(x)$. As a result, the method failed to converge to the actual solution and instead diverged.

Function 4: Fixed point method converging faster than expected

For function 4, defined as $f(x) = \sin(x)e^{-x} + \ln(x^2 + 1)$, the fixed-point method was applied using the iteration function $g(x) = x - f(x)$. While this method is generally expected to converge linearly, the observed order was higher than expected, as reported in Table 2. This is because, although the initial guess $x_0 = 3$ is far from the root, the iterations eventually approach the true root at $x = 0$. Near this root, the behavior of the method acts similarly to Newton's method, given by $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Since $f'(x) \approx 1$ around $x = 0$, the Newton formula simplifies to $x - f(x)$, which matches the fixed-point formula used. To support this, we compute the derivative:

$$f'(x) = \cos(x)e^{-x} - \sin(x)e^{-x} + \frac{2x}{x^2 + 1},$$

and by substituting a value near zero, such as $x = 0.01$, we obtain $f'(0.01) \approx 1.000097999$, confirming that the derivative is close to 1 in that region. This similarity leads to faster convergence than typically expected.

Function 9, 6, and 4: Unusually high than expected observed order

Functions 9, 6, and 4 all converged in very few iterations—3 iterations for function 9 using the MHN and HN methods, 3 iterations for function 6 using MHN, and 5 iterations for function 4 using AN. Because of this rapid convergence, the computed order of convergence appeared unusually high in all cases(see **Table 2**).

Function 4: Unusually low than expected observed order for MHN

For function $f(x) = \sin(x)e^{-x} + \ln(x^2 + 1)$, the MHN method gave a lower order of convergence than expected(see **Table 2**). This happened because the method found the root in just three iterations. When a method finishes too quickly, there aren't enough points to properly measure how fast it was getting closer to the root, so the result may not show the true speed of the method.

Figure 1. The error vs. iterations plot for each method applied to function 6 highlights how the modifications of Newton's method require fewer iterations than the classical Newton's method while maintaining good accuracy.

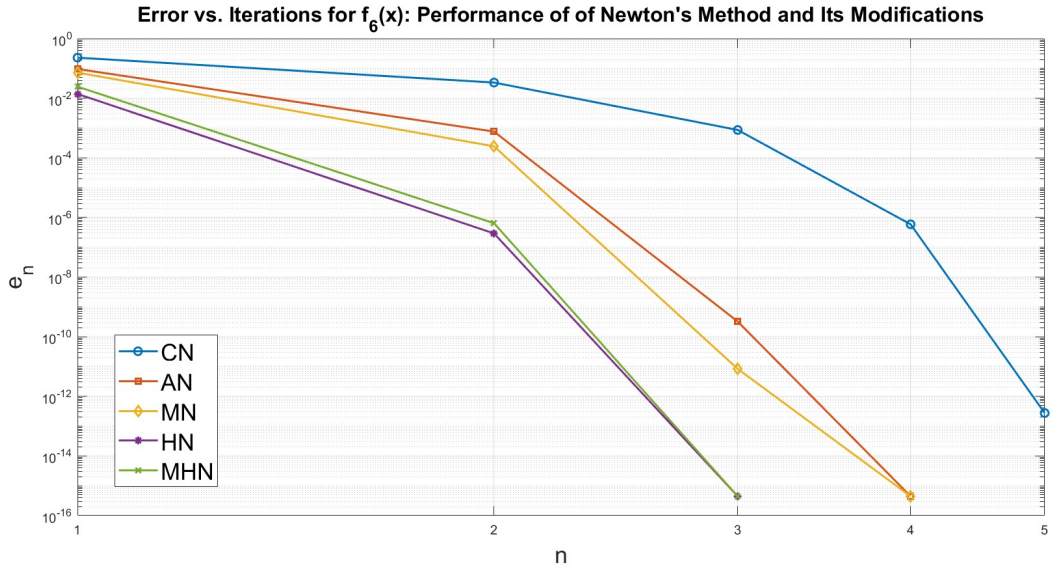
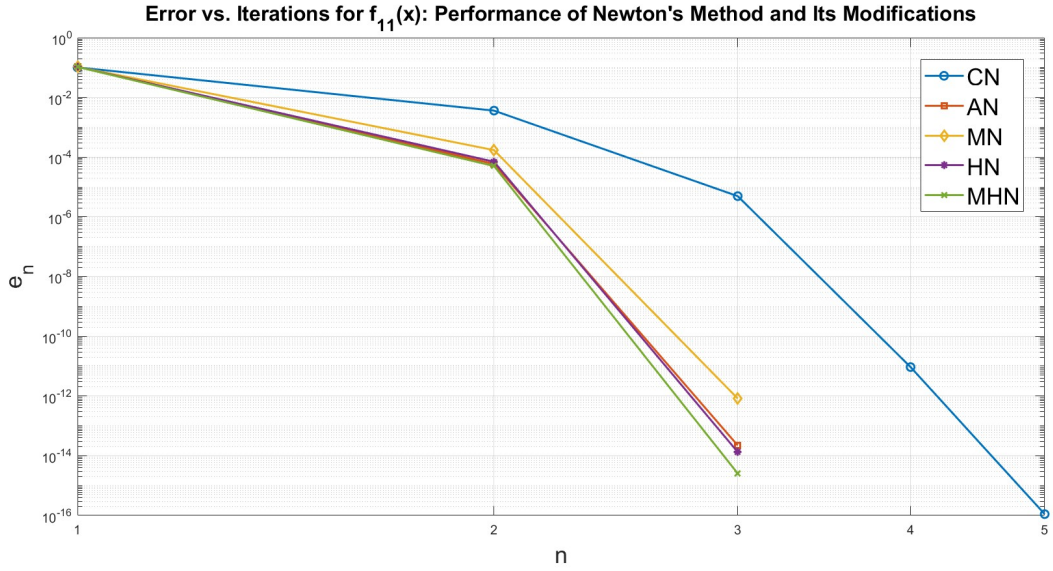


Figure 2. The error vs. iterations plot for each method applied to function 11 highlights how the modifications of Newton's method require fewer iterations



5.2 Functions with multiple roots

We now study the behavior of functions with multiple roots by analyzing function 12 and function 13.

1. $f_{12}(x) = (x-1)\ln(x)$

$f_{12}(x) = (x-1)\ln(x)$ has a double root at $x = 1$. This is verified analytically:

- $f(1) = (1-1)\ln(1) = 0$
- $f'(x) = \frac{x-1}{x} + \ln(x) \Rightarrow f'(1) = 0$
- $f''(x) = \frac{1}{x} + \frac{1}{x^2} \Rightarrow f''(1) = 2 \neq 0$

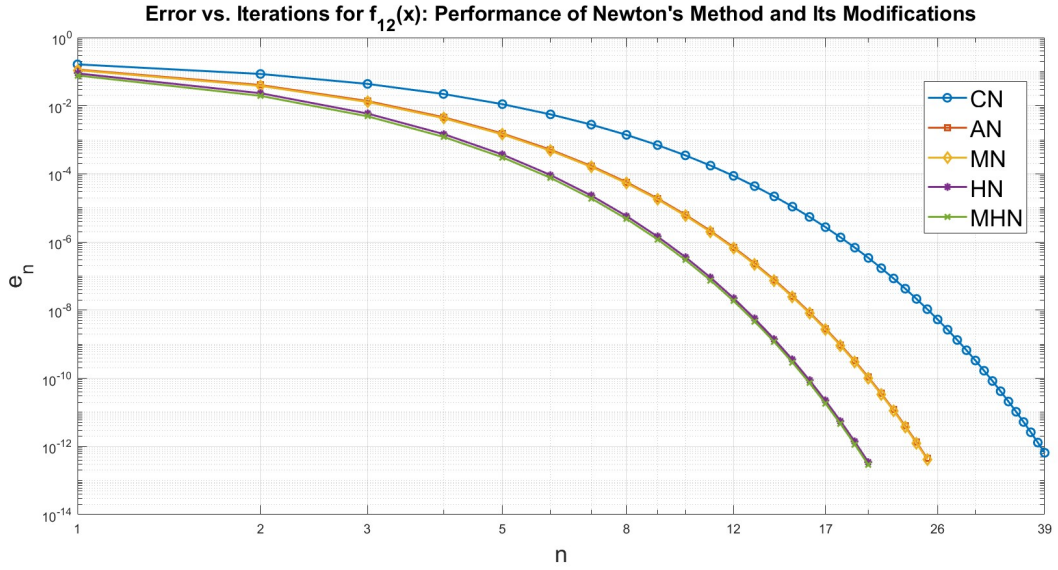
confirming a multiplicity of 2.

Due to this, Newton's method and its modifications experience a drop in convergence order from quadratic or cubic to linear, as shown in **Table 6**.The methods were applied using an initial guess of $x_0 = 0.7$.

Table 6: Performance of Newton methods on function 12 with a double root (without double root adjustment)

Method	N	NFE	COC	e_N	$\lvert f(x_N)\rvert$
CN	39	78	0.9999	6.5115e−13	4.2399e−25
AN	25	75	1.0000	4.4142e−13	1.9486e−25
MN	25	75	0.9999	4.1311e−13	1.7066e−25
HN	20	60	1.0000	3.4461e−13	1.1876e−25
MHN	20	60	1.0003	2.8444e−13	8.0906e−26

Figure 3



To restore faster convergence, the update term $\frac{f(x)}{f'(x)}$ was multiplied by 2 to match the root's multiplicity. This adjustment successfully recovered quadratic convergence for CN. However, the remaining methods (AN, MN, HN, MHN), which typically converge cubically, did not benefit similarly. This suggests that additional theoretical investigation is needed to adapt these methods for functions with multiple roots, which is outside the scope of this project. The performance of CN is shown in **Table 7**, where its convergence improves.

Table 7: Performance of CN with adjustment for the double root

Method	N	NFE	COC	e_N	$\lvert f(x_N)\rvert$
CN	4	8	2.0032	0	0

2. $f_{13}(x) = x^5 + 2x^3$

$f_{13}(x) = x^5 + 2x^3$ has a triple root at $x = 0$. This is confirmed through analytical differentiation:

- $f(0) = 0$
- $f'(x) = 5x^4 + 6x^2 \Rightarrow f'(0) = 0$
- $f''(x) = 20x^3 + 12x \Rightarrow f''(0) = 0$
- $f^{(3)}(x) = 60x^2 + 12 \Rightarrow f^{(3)}(0) = 12 \neq 0$

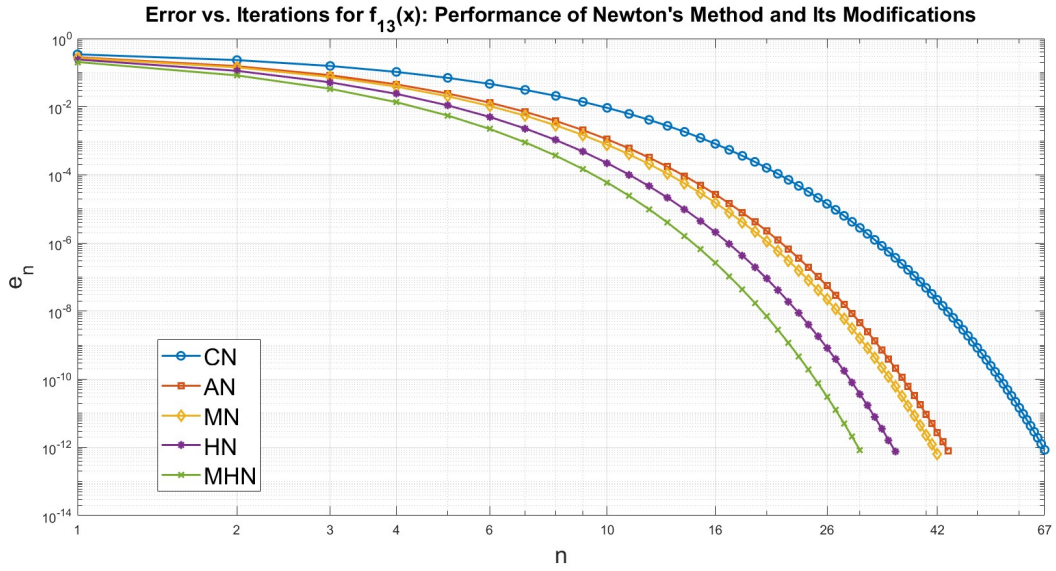
This confirms a multiplicity of 3.

Due to this, all Newton methods experience a drop in convergence order, displaying only linear convergence, as shown in **Table 8**.The methods were applied using an initial guess of $x_0 = 0.5$.

Table 8: Performance of Newton methods on function 13 with a triple root (without adjustment for the triple root)

Method	N	NFE	COC	e_N	$\lvert f(x_N)\rvert$
CN	67	134	1.00	8.5174e−13	1.2358e−36
AN	44	132	1.00	8.0140e−13	1.0294e−36
MN	42	126	1.00	6.2799e−13	4.9533e−37
HN	35	105	1.00	7.4978e−13	8.4301e−37
MHN	30	90	1.00	8.3466e−13	1.1630e−36

Figure 4



To address this, the update term $\frac{f(x)}{f'(x)}$ was multiplied by 3, in line with the root's multiplicity. This adjustment significantly improved the performance of the CN method, raising its COC from 1 to approximately 3, as shown in **Table 9**. However, the modifications (AN, MN, HN, MHN) did not benefit from the same adjustment. Extending similar improvements to these methods requires further theoretical investigation, which is considered outside the scope of this project.

Table 9: Performance of CN with adjustment for the triple root

Method	N	NFE	COC	e_N	$\lvert f(x_N)\rvert$
CN	3	6	2.9999	8.4848e−16	1.2217e−45

Interpretive conclusion:

Tables 2, 3, 6, and 8, along with Figs. 1–4, clearly show that the modified Newton methods usually converge faster and give more accurate results than both the CN and FP methods in most cases. Among all the methods, MHN often needs the fewest number of iterations, which shows that it works more efficiently. Also, MHN needs fewer function evaluations than CN in many cases, as seen in Tables 1, 6, and 8.

Some cases do not follow the expected theoretical behavior. For example, in function 7, the CN method needs fewer iterations than AN and MN. Also, for function 1, the FP method performs better than both CN and HN in terms of number of iterations.

This is mainly because the initial guess is far from the actual root, which affects how quickly each method converges. In function 7, the MN and AN methods require more iterations because their derivative approximations are less accurate at the start, which delays cubic convergence. In function 1, CN progresses slowly before reaching its usual quadratic convergence, while HN struggles because it relies on derivatives at multiple points, which also delays cubic convergence and increases the number of iterations.

Chapter 6

6 Solving nonlinear systems using Newton's method and its modifications

In this study, we aim to approximate the solution of nonlinear systems of equations using Newton's method and its modifications. A general nonlinear system is expressed as:

$$F(\mathbf{X}) = \mathbf{0}$$

where $\mathbf{X}$ is a vector of unknowns, and $F(\mathbf{X})$ is a system of nonlinear functions.

6.1 System of two nonlinear equations

Let $X = (x, y)^T = (x_1, x_2)^T$. Consider the nonlinear system $F(X) = 0$, where

$$F(X) = \begin{pmatrix} f_1(X) \\ f_2(X) \end{pmatrix}.$$

The Jacobian matrix is denoted by $J(X) \in R^{2\times 2}$, where

$$(J(X))_{ij} = \frac{\partial f_i(X)}{\partial x_j}, \quad i, j = 1, 2.$$

The solution is obtained iteratively by solving:

$$J(\mathbf{X})\Delta X = -F(\mathbf{X})$$

The update is then performed as:

$$\mathbf{X}_{n+1} = \mathbf{X}_n + \Delta X$$

We find ΔX by computing the inverse of $J(\mathbf{X})$:

$$\Delta X = -J(\mathbf{X})^{-1} F(\mathbf{X})$$

In practice, this can be calculated in MATLAB by solving the system using the backslash operator:

$$\Delta X = -J(\mathbf{X}) \setminus F(\mathbf{X}).$$

This procedure outlines the classical Newton method. The modified Newton methods follow the same general structure but include additional steps designed to improve convergence.

To measure the accuracy of iterative approximations, the error is computed as follows:

- If the exact solution $\boldsymbol{\alpha}$ is available, the error is the Euclidean norm of the difference between the numerical approximation $\mathbf{X}_{n+1}$ and $\boldsymbol{\alpha}$:

$$e_{n+1} = \|\mathbf{X}_{n+1} - \boldsymbol{\alpha}\|_2.$$

- If the exact solution is not available, the error is approximated by the difference between successive approximations:

$$e_{n+1} = \|\mathbf{X}_{n+1} - \mathbf{X}_n\|_2.$$

Example 1:
We consider the nonlinear system:

$$\begin{cases} x^2 + y^2 - 4 = 0 \\ e^x + y - 3 = 0 \end{cases}$$

Where the exact solution is $\begin{bmatrix} 0 & 2 \end{bmatrix}^T$.

Table 10:Performance of Newton methods for solving example 1

The initial guess used is $\begin{bmatrix} 1 & 1 \end{bmatrix}^T$, and the error is denoted by $e_N = \|\mathbf{X}_N - \boldsymbol{\alpha}\|_2$.

Method	N	NFE	COC	e_N
CN	6	12	2.0627	1.8141e−16
AN	4	12	3.0584	1.2033e−17
MN	4	12	3.0640	3.5823e−17
HN	3	9	3.0309	2.7769e−17
MHN	4	12	2.9125	2.6259e−16

Table 11: Convergence order (COC) analysis for various Newton methods

n	CN	AN	MN	HN	MHN
1	-	-	-	-	-
2	4.1699	3.2048	3.0111	4.2399	4.8030
3	2.0160	3.0584	3.0640	3.0309	2.9125
4	1.8554	1.7113	1.2962	-	0.8056
5	2.0627	-	-	-	-
6	1.3748	-	-	-	-

Example 2:
We consider the nonlinear system:

$$\begin{cases} (x-1)\ln(x)\ln(y) = 0 \\ (x-1)^2(y + y^2 - 1) = 0 \end{cases}$$

This system has infinitely many distinct solutions, all of the form $\begin{bmatrix} 1 & y \end{bmatrix}^T$ for $y \in R,\ y > 0$. Each of these is a multiple root, due to the repeated factor in the variable x, which makes the Jacobian singular at $x = 1$ and leads to a multiplicity greater than 1 for all solutions along this set.

Table 12: Performance of Newton methods for solving example 2

The initial guess used is $\begin{bmatrix}1.1 & 0.3\end{bmatrix}^T$. The error is denoted by $e_N = \|\mathbf{X}_N - \mathbf{X}_{N-1}\|_2$.The term $\|F(\mathbf{X}_N)\|_2$ represents the residual norm.

Method	N	NFE	COC	e_N	$\|\|F(X_N)\|\|_2$
CN	37	74	1.0000	9.3395e−13	7.7213e−25
AN	24	72	1.0002	9.3424e−13	1.9110e−25
MN	24	72	1.0006	8.6595e−13	9.8007e−26
HN	20	60	1.0001	3.9293e−13	1.5163e−26
MHN	20	60	1.0005	2.9919e−13	8.3283e−27

Since this system has roots with multiplicity greater than one, all Newton methods drop to linear convergence. To address this, the update term $J^{-1}F$ was multiplied by 2, which raised the COC of CN from 1 to approximately 2, as shown in **Table 13**. However, this adjustment had no effect on AN, MN, HN, and MHN. Adapting these methods requires further theoretical investigation.

Table 13: Performance of CN with adjustment for multiple roots

Method	N	NFE	COC	e_N	$\|\|F(X_N)\|\|_2$
CN	5	10	2.0095	3.3307e−16	0

6.2 System of three nonlinear equations

Let $X = (x, y, z)^T = (x_1, x_2, x_3)^T$. Consider the nonlinear system $F(X) = 0$, where

$$F(X) = \begin{pmatrix} f_1(X) \\ f_2(X) \\ f_3(X) \end{pmatrix}.$$

The Jacobian matrix is given by $J(X) \in R^{3\times 3}$, where

$$(J(X))_{ij} = \frac{\partial f_i(X)}{\partial x_j}, \quad i, j = 1, 2, 3.$$

To find the solution iteratively, we follow the same process:

$$J(\mathbf{X})\Delta X = -F(\mathbf{X})$$

$$\mathbf{X}_{n+1} = \mathbf{X}_n + \Delta X.$$

Example:

We consider the nonlinear system:

$$\begin{cases} 3x - \cos(yz) - \frac{1}{2} = 0, \\ x^2 - 81(y + 0.1)^2 + \sin(z) + 1.06 = 0, \\ e^{-xy} + 20z + \frac{10\pi - 3}{3} = 0. \end{cases}$$

where the exact solution is $\begin{bmatrix}0.5 & 0 & -\frac{\pi}{6}\end{bmatrix}^T$.

Table 14:Performance of Newton methods

The initial guess used is $\begin{bmatrix}0.1 & 0.1 & -0.1\end{bmatrix}^T$.

Methods	N	NFE	COC	e_N	$\|\|F(X_N)\|\|_2$
CN	5	10	1.9968	1.1110e−16	1.7764e−15
AN	3	9	2.9860	4.8708e−15	7.8625e−14
MN	3	9	2.9869	2.9859e−15	4.8406e−14
HN	3	9	1.1746	1.1106e−16	1.7764e−15
MHN	3	9	1.0344	1.1113e−16	1.7764e−15

Table 15: Convergence order (COC) analysis for various Newton methods

n	CN	AN	MN	HN	MHN
1	-	-	-	-	-
2	0.7369	1.5426	1.5471	2.4397	2.4643
3	1.9311	2.9860	2.9869	1.1746	1.0344
4	1.9968	-	-	-	-
5	1.6275	-	-	-	-

As shown in **Table 14** and **Table 15**, both the HN and MHN methods exhibit a lower-than-expected order of convergence. This is due to the few number of iterations, which provides too few data points to accurately estimate the convergence rate.

Interpretive Conclusion:

After analyzing the behavior of various systems—whether composed of two or three nonlinear equations—it was consistently observed that the modified Newton methods typically perform better than the CN method, particularly in terms of number of iterations and accuracy. Among these, the MHN method stands out, consistently achieving the lowest number of iterations in most cases.

6.3 Computational cost analysis of Newton's method and its modifications

After applying the CN method and its various modifications, it is important to evaluate the efficiency of each approach by analyzing the computational cost. In this context, the computational cost is measured using CPU time, which reflects how long each method takes to reach a desired level of accuracy. The timing was done using MATLAB's `tic` and `toc` functions. Methods that require more CPU time are considered more computationally expensive, as they consume more processing resources. By comparing the CPU time of each method, we can identify which ones are more efficient and better suited for practical use.

To identify which method takes the longest to run, we measured and compared the CPU time for each method using function 3. Each method was executed five times under identical conditions to ensure fairness and consistency in the comparison. The average CPU time from these runs was then calculated.The results are presented in **Table 16**.

Table 16:CPU Time comparison across different Newton methods applied to function 3

Method	CPU Time (seconds)
CN	0.036589
AN	2.7844e−3
MN	1.6266e−3
HN	2.9142e−3
MHN	2.1926e−3

The computational cost analysis for function 3 shows that the CN method takes the most CPU time to converge. In comparison, all the modified Newton methods achieve faster results. This indicates that these modifications not only enhance the convergence process but also offer better computational efficiency, making them more suitable for real-world applications.

Chapter 7

7 Application: Lotka–Volterra predator–prey model

The Lotka–Volterra system is a classical mathematical model used to describe the dynamics of predator–prey interactions in ecological systems. It consists of two nonlinear differential equations:

$$\begin{cases} \frac{dx}{dt} = ax - bxy \\ \frac{dy}{dt} = -cy + dxy \end{cases}$$

where $x(t)$ represents the prey population and $y(t)$ represents the predator population at time t. The parameters have the following biological interpretations:

- a is the natural growth rate of prey in the absence of predators.
- b is the rate at which predators consume prey.
- c is the natural death rate of predators in the absence of prey.
- d is the growth rate of predators due to the availability of prey.

To numerically approximate the solution of this nonlinear system, we apply the implicit (backward) Euler method for time discretization, which leads to a nonlinear system of equations at each time step. Specifically, letting x_n, y_n denote the prey and predator populations at time t_n, and x_{n+1}, y_{n+1} at $t_{n+1} = t_n + \Delta t$, the backward Euler method yields:

$$\begin{cases} \frac{x_{n+1}-x_n}{\Delta t} = ax_{n+1} - bx_{n+1}y_{n+1} \\ \frac{y_{n+1}-y_n}{\Delta t} = -cy_{n+1} + dx_{n+1}y_{n+1} \end{cases}$$

Rewriting this system gives:

$$\begin{cases} x_{n+1} - \Delta t\,(ax_{n+1} - bx_{n+1}y_{n+1}) - x_n = 0 \\ y_{n+1} - \Delta t\,(-cy_{n+1} + dx_{n+1}y_{n+1}) - y_n = 0 \end{cases}$$

This nonlinear system is then solved using Newton's method and several of its modifications to examine their performance and convergence behavior in the context of the Lotka–Volterra dynamics.

To solve the system using the CN method, we begin with an initial guess $X_0 = [x_0; y_0]$ and iteratively update the solution as follows:

1. Compute the Newton step:
$$\Delta X = -J^{-1}F$$
which in MATLAB is implemented efficiently using the backslash operator:
$$\Delta X = -J \setminus F.$$

2. Update the current approximation:
$$\mathbf{X}_{n+1} = \mathbf{X}_n + \Delta X.$$

Here, F represents the nonlinear system obtained from the backward Euler discretization, and J is the Jacobian matrix:

$$J = \begin{bmatrix} 1 - \Delta t(a - by_{n+1}) & \Delta t \cdot bx_{n+1} \\ -\Delta t \cdot dy_{n+1} & 1 - \Delta t(-c + dx_{n+1}) \end{bmatrix}$$

This procedure outlines the classical Newton method. The modified Newton methods follow the same overall structure but include additional steps designed to improve convergence performance.

Table 17:Performance of Newton methods

Method	N	NFE	COC	e_N	$\|\|F(X_N)\|\|$
CN	3	6	2	2.2204e−16	2.2204e−16
AN	2	6	3	4.8036e−15	1.1102e−16
MN	2	6	3	4.8036e−15	1.1102e−16
HN	2	6	3	2.2204e−16	2.2204e−16
MHN	2	6	3	2.2204e−16	2.2204e−16

The results in **Table 17** are based on the parameters $a = 2/3$, $b = 4/3$, $c = 2/3$, and $d = 1$, which govern the predator-prey dynamics. The initial condition is given by $X_0 = \begin{bmatrix} 2 & 1 \end{bmatrix}^T$, representing the initial prey and predator populations. The system is solved using a time step $\Delta t = 0.001$, a tolerance of 10^{-12}, and a maximum of 1000 iterations. All Newton methods return the same solution for the system, $\begin{bmatrix} 1.9987 & 1.0013 \end{bmatrix}^T$, which represents the approximated populations of prey and predator at the next time step.

This solution shows that the prey population decreases from 2 to 1.9987 as the predators consume them, while the predator population increases from 1 to 1.0013 due to the increased availability of prey. These dynamics demonstrate the typical oscillatory behavior seen in the predator-prey model, with both populations fluctuating over time.

CN Figures

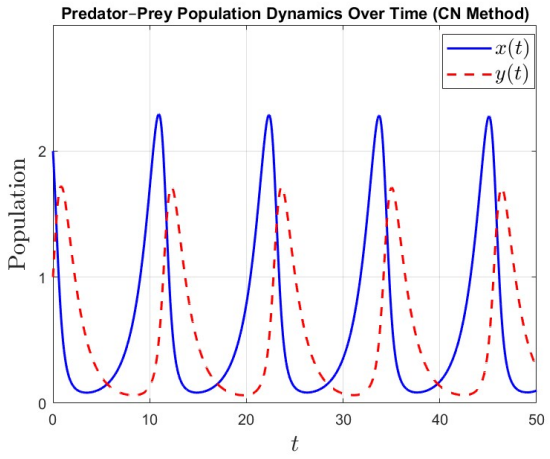


Figure 5

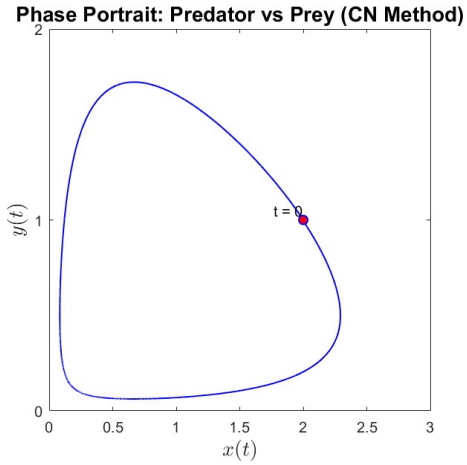


Figure 6

AN Figures

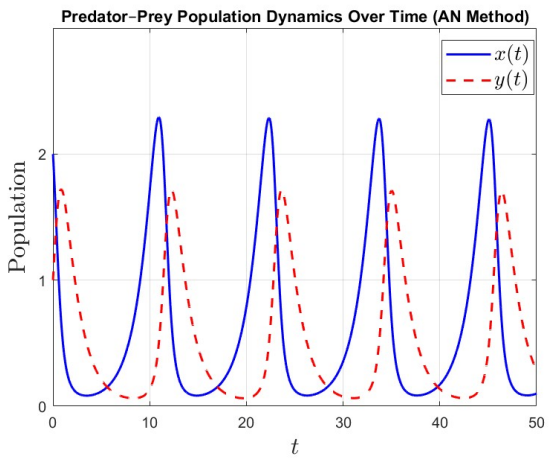


Figure 7

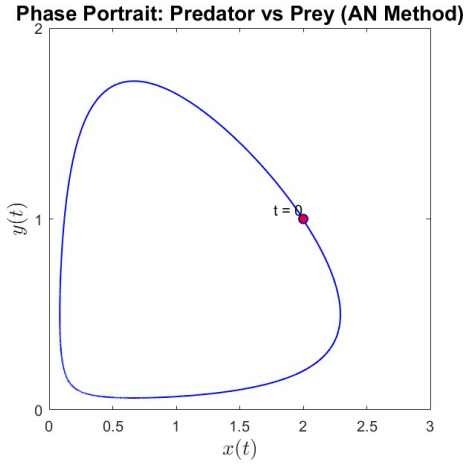


Figure 8

MN Figures

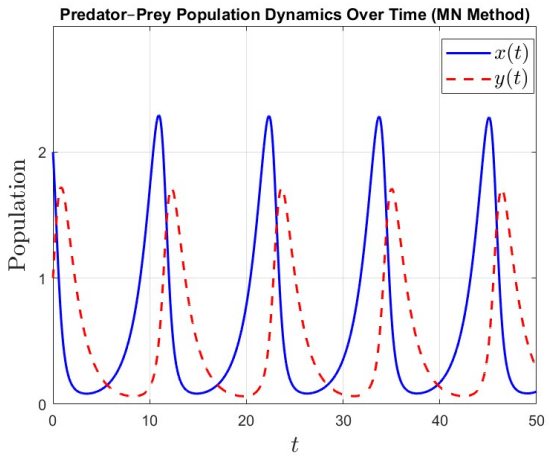


Figure 9

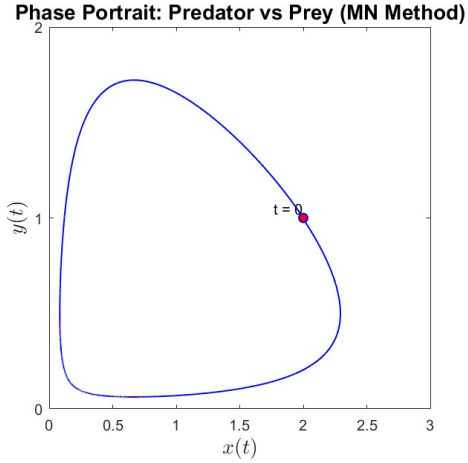


Figure 10

HN Figures

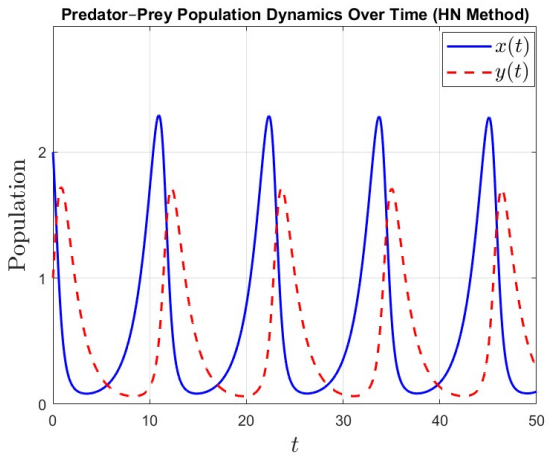


Figure 11

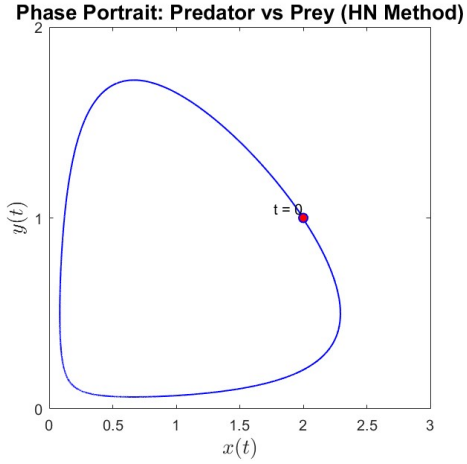


Figure 12

MHN Figures

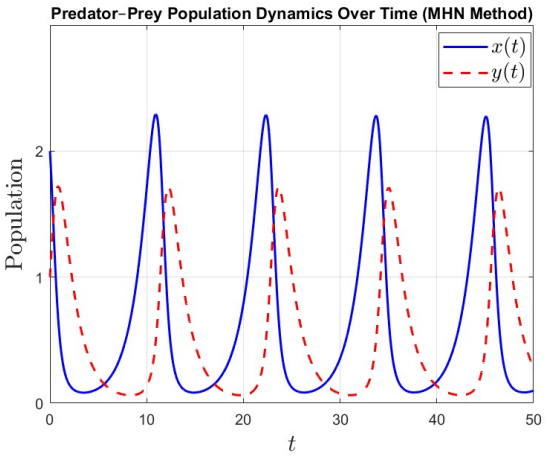


Figure 13

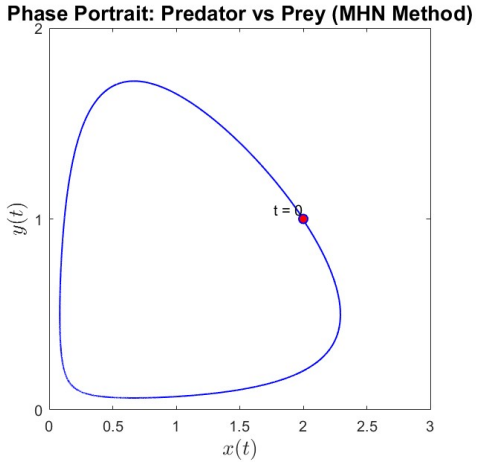


Figure 14

Figures 5–14 illustrate the behavior of the predator–prey system modeled by the Lotka–Volterra equations, solved using the backward Euler method. Each method—CN, AN, MN, HN, and MHN—is used to solve the resulting nonlinear system at each time step. For each method, we display:

- **Left Column (Figures 5, 7, 9, 11, and 13)**: The time evolution of the prey population $x(t)$ (blue solid line) and predator population $y(t)$ (red dashed line). All methods demonstrate the expected oscillatory behavior of predator–prey interactions, where the populations rise and fall in cycles due to their interdependence.
- **Right Column (Figures 6, 8, 10, 12, and 14)**: The phase portraits, which plot $y(t)$ against $x(t)$. These closed-loop trajectories show the cyclical nature of the system in the phase space, reflecting a repeating pattern in population interactions over time. The point marked $t = 0$ indicates the initial condition $x(0) = 2$, $y(0) = 1$.

While the solution curves are visually similar across all methods—demonstrating that they converge to the same numerical solution—the number of iterations and convergence behavior vary between them, as shown in **Table 17**. These results confirm that all Newton methods are capable of accurately solving the implicit discretization of the Lotka–Volterra system and capturing the characteristic predator–prey cycles—especially the modified methods, which not only require fewer iterations but also exhibit higher order of convergence compared to the classical Newton method.

The following figures present phase-space diagrams for the Lotka–Volterra predator–prey model, illustrating the relationship between prey and predator populations over time. Each plot corresponds to a different Newton method used to solve the system numerically. The trajectories show the system's response for varying initial predator populations, capturing the characteristic oscillatory behavior of predator–prey interactions.

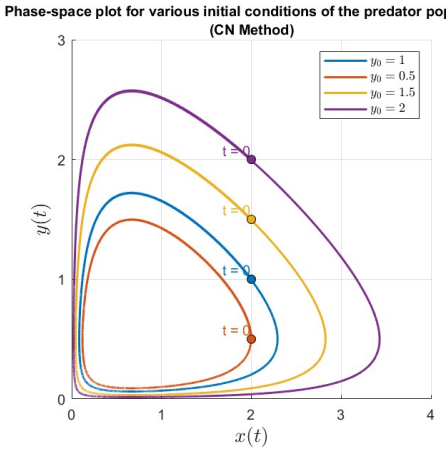


Figure 15

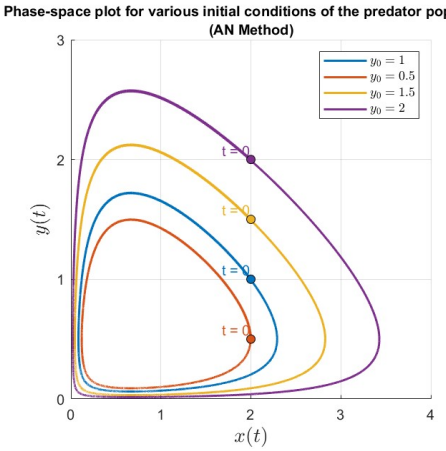


Figure 16

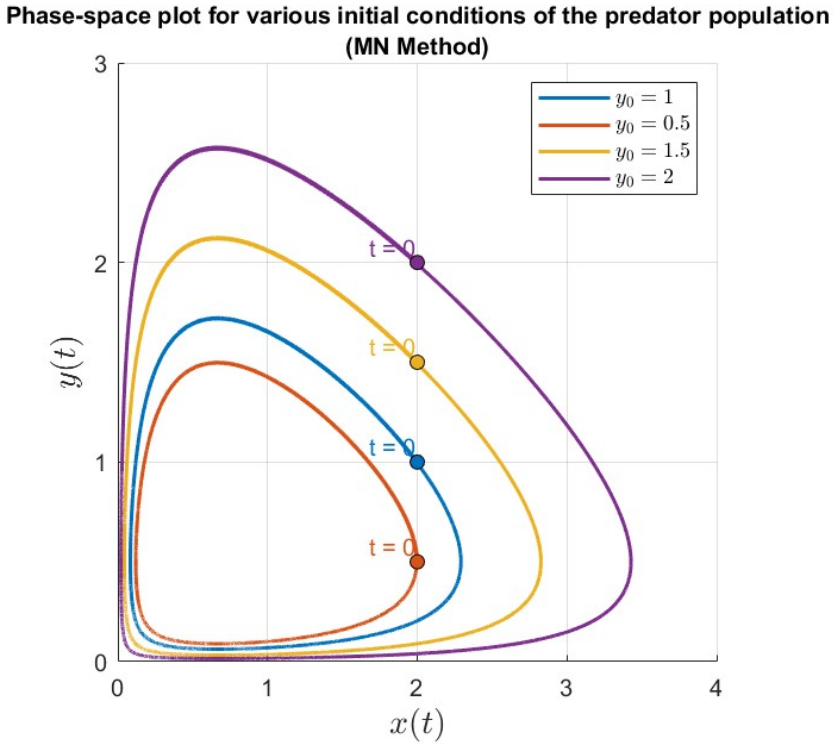


Figure 17

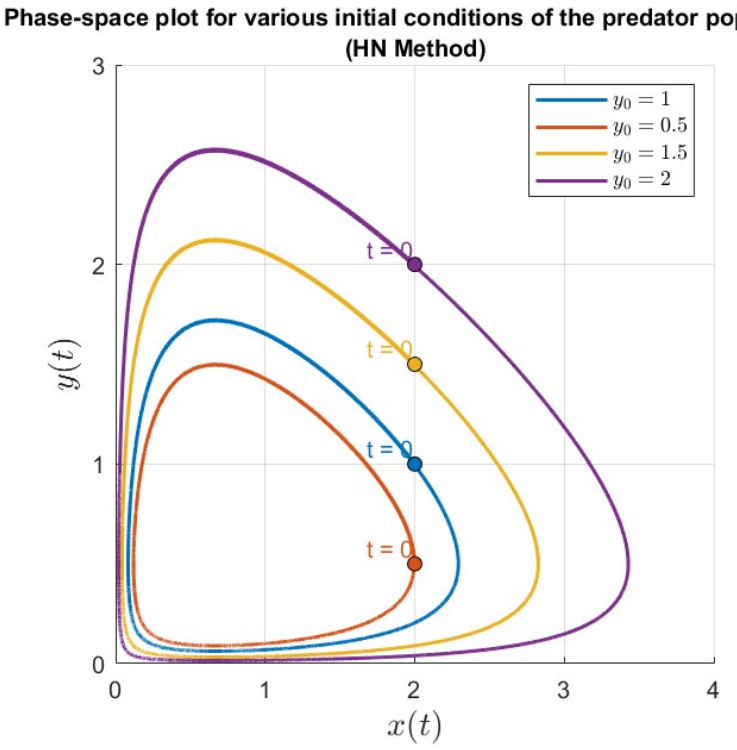


Figure 18

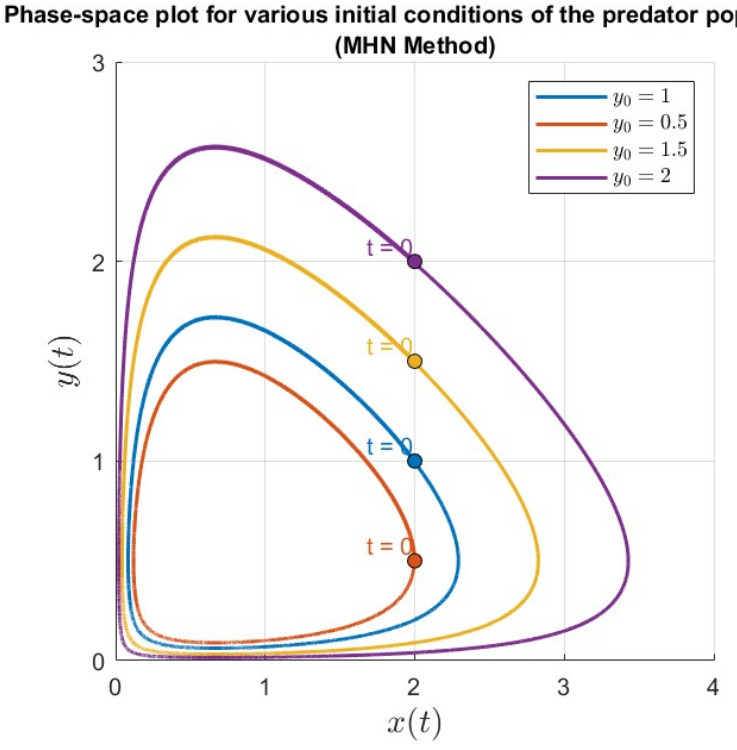


Figure 19

Table 18:CPU Time comparison across different Newton methods applied to Lotka–Volterra

Method	CPU Time (seconds)
CN	38.9725758
AN	32.7948936
MN	27.7836762
HN	30.7051416
MHN	36.4059108

To assess the computational efficiency of the various Newton methods applied to the Lotka–Volterra system, the CPU time was measured using MATLAB's `tic` and `toc` functions. Each method was executed five times under identical conditions to ensure fairness and consistency in comparison. The average CPU time from these five runs was then calculated to minimize the influence of any outliers or fluctuations in system performance.

The results are presented in **Table 18**. The table indicates that Newton's method appears to be less advantageous compared to the other methods in terms of computational cost. Based on these preliminary results regarding the use of high-order methods, we can assert that higher-order convergence may prove more effective for long-time simulations with relatively large time steps, making third-order methods particularly beneficial. However, further investigation is required for larger nonlinear systems to more accurately evaluate the computational efficiency of every method.

8 Bibliography

[1] M. Shams, N. Kausar, S. Araci, and G. I. Oros, "Artificial hybrid neural network-based simultaneous scheme for solving nonlinear equations: Applications in engineering," Alexandria Engineering Journal, vol. 108, pp. 292–305, Jul. 2024, doi: https://doi.org/10.1016/j.aej.2024.07.078.

[2] C. Remani, "Numerical Methods for Solving Systems of Nonlinear Equations." https://www.lakeheadu.ca/sites/default/files/uploads/77/docs/RemaniFinal.pdf.

[3] V. Kučera, "Numerical Methods for Nonlinear Equations." Accessed: Nov. 30, 2023. https://www.karlin.mff.cuni.cz/~kucera/Numerical_Methods_for_Nonlinear_Equations.pdf.

[4] M. Cameron, Fixed Point Iteration. College Park, MD: Department of Mathematics, University of Maryland. [Online]. https://www.math.umd.edu/~mariakc/AMSC466/LectureNotes/fixed_point.pdf.

[5] F. Dubeau and C. Gnang, "Fixed Point and Newton's Methods for Solving a Nonlinear Equation: From Linear to High-Order Convergence," SIAM Review, vol. 56, no. 4, pp. 691–708, Jan. 2014, doi: https://doi.org/10.1137/130934799.

[6] T. Ypma, "Historical Development of the Newton-Raphson Method," Western CEDAR, 2015. https://cedar.wwu.edu/math_facpubs/93/?utm_source.

[7] K. Jisheng, L. Yitian, and W. Xiuhua, "Third-order modification of Newton's method," Journal of Computational and Applied Mathematics, vol. 205, no. 1, pp. 1–5, Aug. 2007, doi: https://doi.org/10.1016/j.cam.2006.03.022.

[8] R. L. Burden, J. D. Faires, and A. M. Burden, Numerical Analysis, 10th ed. Boston, MA, USA: Cengage Learning, 2015, ISBN: 9781305253667.

[9] R. Suparatulatorn and S. Suantai, "Stability and convergence analysis of hybrid algorithms for Berinde contraction mappings and its applications," Results in Nonlinear Analysis, vol. 4, pp. 159–168, 2021.

[10] N. H. Tuan and H. S. Mohammadi, "A mathematical model for COVID-19 transmission by using the Caputo fractional derivative," Chaos, Solitons & Fractals, vol. 140, p. 110107, 2020.

[11] N. H. Tuan and Y. Zhou, "Well-posedness of an initial value problem for fractional diffusion equation with Caputo-Fabrizio derivative," Journal of Computational and Applied Mathematics, vol. 375, p. 112811, 2020.

[12] A. Cordero, C. Jordán, E. Sanabria-Codesal, and J. R. Torregrosa, "Highly efficient iterative algorithms for solving nonlinear systems with arbitrary order of convergence $p + 3$, $p \geq 5$," Journal of Computational and Applied Mathematics, vol. 330, pp. 748–758, 2018.

[13] A. Cordero, E. Gómez, and J. R. Torregrosa, "Efficient high-order iterative methods for solving nonlinear systems and their application on heat conduction problems," Complexity, vol. 2017, 2017.

[14] A. Laadhari and G. Szekely, "Fully implicit finite element method for the modeling of free surface flows with surface tension effect," Int. J. Numer. Methods Eng., vol. 111, no. 11, pp. 1047–1074, 2017, doi: https://doi.org/10.1002/nme.5493.

[15] A. Laadhari, "Exact Newton method with third-order convergence to model the dynamics of bubbles in incompressible flow," Appl. Math. Lett., vol. 69, pp. 138–145, 2017, doi: https://doi.org/10.1016/j.aml.2017.01.012.

[16] A. Laadhari, P. Saramito, C. Misbah, and G. Szekely, "Fully implicit methodology for the dynamics of biomembranes and capillary interfaces by combining the Level Set and Newton methods," J. Comput. Phys., vol. 343, pp. 271–299, 2017, doi: https://doi.org/10.1016/j.jcp.2017.04.019.

[17] A. Laadhari and H. Temimi, "Efficient finite element strategy using enhanced high-order and second-derivative-free variants of Newton's method," Applied Mathematics and Computation, vol. 486, p. 129058, 2025, doi: https://doi.org/10.1016/j.amc.2024.129058.

[18] A. Laadhari, "Implicit finite element methodology for the numerical modeling of incompressible two-fluid flows with moving hyperelastic interface," Applied Mathematics and Computation, vol. 333, pp. 376–400, 2018, doi: https://doi.org/10.1016/j.amc.2018.03.074.

9 Appendix

CN method code for 1 variable (Exact solution known)

```
1
2
3
4
5  function [sequence, errors_new, ratios, zero, niter, order, res, ek, NFE,cputime_cn] =
       newton_method2(fun, dfun, x0, tol, nmax, realroot)
6  % NEWTON_METHOD2 Applies Newton's method to find a root of a function.
7  %
8  % INPUTS:
9  % fun       - Function handle for f(x)
10 % dfun      - Function handle for f'(x)
11 % x0        - Initial guess
12 % tol       - Tolerance for stopping criterion
13 % nmax      - Maximum number of iterations
14 % realroot - The actual root (used to compute the error)
15 %
16 % OUTPUTS:
17 % sequence     - All approximated values of the root
18 % errors_new  - Absolute error at each iteration
19 % ratios       - Error ratios used to estimate order of convergence
20 % zero         - Final approximation of the root
21 % niter        - Total number of iterations performed
22 % order        - Estimated order of convergence
23 % res          - Final residual |f(x)|
24 % ek           - Final absolute error |x - alpha|
25 % NFE          - Number of function evaluations (f + f') = 2 per iteration
26
27 % Assign function and derivative
28 f = fun;
29 df = dfun;
30 alpha = realroot;
31
32 % Initialize parameters
33 max_iter = nmax;
34 x = x0;
35 iter = 0;
36 err = inf;
37
38 % Initialize containers for data
39 errors_new = [];
40 ratios = [];
41 sequence = [];
42
43 % Newton iteration loop
44 tic;
45 while err > tol && iter < max_iter
46
47     x_new = x - f(x) / df(x);
48
49     % Compute absolute error using the true root
50     err = abs(x_new - alpha);
51
52     % Estimate error ratio for convergence order
53     ratio = err / abs(x - alpha);
54
55     % Store current values
56     errors_new = [errors_new err];
57     ratios = [ratios ratio];
58     sequence = [sequence x];
59
60     % Update current approximation and iteration counter
61     x = x_new;
62     iter = iter + 1;
63 end
```

```
cputime_cn = toc;

fprintf('CPU time: %.6f seconds\n',cputime_cn);

% Output values
niter = iter;                   % Number of iterations
NFE = niter * 2;                % Function evaluations: f and f' per iteration
order = log(ratios(end)) / log(ratios(end - 1));  % Estimate order of convergence
fprintf('%.10f\n', order);      % Display order
zero = x;                       % Final approximation
res = abs(f(zero));             % Residual |f(x)|
ek = abs(zero - alpha);         % Final error

% Plot log-log error vs iteration
figure;
loglog(1:length(errors_new), errors_new, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Newton''s Method');

% Formatting for better plot readability
maxx = max(length(errors_new));
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

CN method code for 1 variable (Exact solution unknown)

```
function [sequence, errors_new, ratios, zero, niter, order, res, ek, NFE] =
    newton_method(fun, dfun, x0, tol, nmax)
% NEWTON_METHOD Applies Newton's method to solve f(x) = 0.
%
% INPUTS:
% fun       - Function handle for f(x)
% dfun      - Function handle for f'(x)
% x0        - Initial guess
% tol       - Tolerance for stopping criterion
% nmax      - Maximum number of iterations
%
% OUTPUTS:
% sequence  - All approximated values of the root
% errors_new - Absolute errors at each iteration
% ratios    - Error ratios used to estimate the order of convergence
% zero      - Final approximation of the root
% niter     - Total number of iterations performed
% order     - Estimated order of convergence
% res       - Final residual |f(x)|
% ek        - Final absolute error |x - alpha|
% NFE       - Number of function evaluations (1 for f(x) and 1 for f'(x) per iteration)

% Assign function and its derivative
f = fun;
df = dfun;

% Initialize variables
max_iter = nmax;
x = x0;
iter = 0;
err = inf;

% Initialize containers for storing iteration data
errors_new = [];
ratios = [];
sequence = [];

% Newton's method iteration loop
while err > tol && iter < max_iter
```

```
39
40      x_new = x - f(x) / df(x);
41
42      % Compute the error and store it
43      err = abs(x_new - x);
44      sequence = [sequence x];
45      errors_new = [errors_new err];
46
47      % Compute and store the error ratio (used for convergence order)
48      if iter > 0
49          ratios = [ratios, err / errors_new(end-1)];
50      end
51
52      % Update the current guess and iteration count
53      x = x_new;
54      iter = iter + 1;
55  end
56
57  % Output results
58  niter = iter;                       % Number of iterations performed
59  NFE = niter * 2;                    % Number of function evaluations (twice per
        iteration: f(x) and f'(x))
60
61  % Estimate the order of convergence if at least 2 ratios are available
62  if length(ratios) >= 2
63      order = log(ratios(end)) / log(ratios(end-1));
64  else
65      order = NaN;
66  end
67
68  fprintf('Estimated order of convergence: %.4f\n', order);  % Display the estimated
        order
69
70  % Store the final approximation of the root
71  zero = x;
72
73  % Calculate the residual (|f(x)|) and the final absolute error
74  res = abs(f(zero));
75  ek = err;  % Final error (|x - x_true|)
76
77  % Plot the error vs iteration in log-log scale
78  figure;
79  loglog(1:length(errors_new), errors_new, 'o-', 'LineWidth', 1.5);
80  xlabel('n');
81  ylabel('e_n');
82  title('Newton''s Method');
83
84  % Adjust axis ticks and range for better visualization
85  maxx = length(errors_new);
86  xticks(unique(round(logspace(0, log10(maxx), 10))));
87  xlim([1 maxx]);
88  grid on;
89  end
```

AN method code for 1 variable(Exact solution known)

```
1
2   function [sequence, errors_arth, ratios, zero, niter, order, res, ek, NFE,cputime_an] =
        arithmetic_mean_newton1(fun, dfun, x0, tol, nmax, realroot)
3   % ARITHMETIC_MEAN_NEWTON1 Applies the Arithmetic Mean Newton's method to find a root.
4   %
5   % INPUTS:
6   % fun      - Function handle for f(x)
7   % dfun     - Function handle for f'(x)
8   % x0       - Initial guess
9   % tol      - Tolerance for stopping criterion
10  % nmax     - Maximum number of iterations
11  % realroot - The actual root (used to compute the error)
```

```
%
% OUTPUTS:
% sequence      - All approximated values of the root
% errors_arth   - Absolute error at each iteration
% ratios        - Error ratios used to estimate order of convergence
% zero          - Final approximation of the root
% niter         - Total number of iterations performed
% order         - Estimated order of convergence
% res           - Final residual |f(x)|
% ek            - Final absolute error |x - alpha|
% NFE           - Number of function evaluations (3 per iteration: f, df, and df(x_star))

% Assign function and derivative
f = fun;
df = dfun;
alpha = realroot;

% Initialize parameters
max_iter = nmax;
x = x0;
iter = 0;
err = inf;

% Initialize containers for data
errors_arth = [];
ratios = [];
sequence = [];

% Arithmetic Mean Newton iteration loop
tic;
while err > tol && iter < max_iter

    x_star = x - f(x) / df(x);


    x_new = x - (2 * f(x)) / (df(x) + df(x_star));

    % Compute absolute error using the true root
    err = abs(x_new - alpha);

    % Estimate error ratio for convergence order
    ratio = err / abs(x - alpha);

    % Store current values
    errors_arth = [errors_arth err];
    ratios = [ratios ratio];
    sequence = [sequence x];

    % Update current approximation and iteration counter
    x = x_new;
    iter = iter + 1;
end
cputime_an = toc;

fprintf('CPU time: %.6f seconds\n',cputime_an);

% Output values
niter = iter;                       % Number of iterations
NFE = niter * 3;                    % Function evaluations: f, df, df(x_star) per iteration
order = log(ratios(end)) / log(ratios(end - 1));  % Estimate order of convergence
fprintf('%.10f\n', order);          % Display order
zero = x;                           % Final approximation
res = abs(f(zero));                 % Residual |f(x)|
ek = abs(zero - alpha);             % Final error

% Plot log-log error vs iteration
```

```
figure;
loglog(1:length(errors_arth), errors_arth, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Arithmetic Mean Newton''s Method');

% Formatting for better plot readability
maxx = max(length(errors_arth));
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

AN method code for 1 variable(Exact solution unknown)

```
function [sequence, errors_arth, ratios, zero, niter, order, res, ek, NFE] =
    arithmetic_mean_newton(fun, dfun, x0, tol, nmax)
% ARITHMETIC_MEAN_NEWTON Applies the Arithmetic Mean Newton's method to solve f(x) = 0.
%
% INPUTS:
% fun        - Function handle for f(x)
% dfun       - Function handle for f'(x)
% x0         - Initial guess
% tol        - Tolerance for stopping criterion
% nmax       - Maximum number of iterations
%
% OUTPUTS:
% sequence   - All approximated values of the root
% errors_arth - Absolute errors at each iteration
% ratios     - Error ratios used to estimate the order of convergence
% zero       - Final approximation of the root
% niter      - Total number of iterations performed
% order      - Estimated order of convergence
% res        - Final residual |f(x)|
% ek         - Final absolute error |x - alpha|
% NFE        - Number of function evaluations (3 per iteration: f(x), f'(x), and f(
    x_star))

% Assign function and its derivative
f = fun;
df = dfun;

% Initialize variables
x = x0;
iter = 0;
err = inf;

% Initialize containers for storing iteration data
errors_arth = [];
ratios = [];
sequence = [];
% Arithmetic Mean Newton's method iteration loop
while err > tol && iter < nmax

    x_star = x - f(x) / df(x);
    x_new = x - (2 * f(x)) / (df(x) + df(x_star));

    % Compute the error and store it
    err = abs(x_new - x);
    errors_arth = [errors_arth, err];

    % Compute and store the error ratio (used for convergence order)
    if iter > 0
        ratios = [ratios, err / errors_arth(end-1)];
    end

    % Store the current approximation and update for the next iteration
    sequence = [sequence, x];  % Store the sequence of approximations
```

```
    x = x_new;                  % Update current guess
    iter = iter + 1;            % Increment the iteration counter
end

% Output results
niter = iter;                    % Number of iterations performed
NFE = niter * 3;                 % Number of function evaluations (3 per iteration)

% Estimate the order of convergence if at least 2 ratios are available
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end

% Display the estimated order of convergence
fprintf('Estimated order of convergence: %.4f\n', order);

% Store the final approximation of the root
zero = x;

% Calculate the residual (|f(x)|) and the final absolute error
res = abs(f(zero));  % Residual: |f(x)|
ek = err;            % Final error (|x - x_true|)

% Plot the error vs iteration in log-log scale
figure;
loglog(1:length(errors_arth), errors_arth, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Arithmetic Mean Newton''s Method');

% Adjust axis ticks and range for better visualization
maxx = length(errors_arth);
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

MN method code for 1 variable(Exact solution known)

```

function [sequence, errors_mid, ratios, zero, niter, order, res, ek, NFE,cputime_mn ] =
     midpoint_newton1(fun, dfun, x0, tol, nmax, realroot)
% MIDPOINT_NEWTON1 Applies the Midpoint Newton's method to find a root.
%
% INPUTS:
% fun      - Function handle for f(x)
% dfun     - Function handle for f'(x)
% x0       - Initial guess
% tol      - Tolerance for stopping criterion
% nmax     - Maximum number of iterations
% realroot - The actual root (used to compute the error)
%
% OUTPUTS:
% sequence     - All approximated values of the root
% errors_mid   - Absolute error at each iteration
% ratios       - Error ratios used to estimate order of convergence
% zero         - Final approximation of the root
% niter        - Total number of iterations performed
% order        - Estimated order of convergence
% res          - Final residual |f(x)|
% ek           - Final absolute error |x - alpha|
% NFE          - Number of function evaluations (3 per iteration: f, df, and df(
    midpoint))

% Assign function and derivative
f = fun;
```

```
df = dfun;
alpha = realroot;

% Initialize parameters
max_iter = nmax;
x = x0;
iter = 0;
err = inf;

% Initialize containers for data
errors_mid = [];
ratios = [];
sequence = [];

% Midpoint Newton iteration loop
tic;
while err > tol && iter < max_iter

    x_star = x - f(x) / df(x);


    midpoint = (x + x_star) / 2;


    x_new = x - f(x) / df(midpoint);

    % Compute absolute error using the true root
    err = abs(x_new - alpha);

    % Estimate error ratio for convergence order
    ratio = err / abs(x - alpha);

    % Store current values
    errors_mid = [errors_mid err];
    ratios = [ratios ratio];
    sequence = [sequence x];

    % Update current approximation and iteration counter
    x = x_new;
    iter = iter + 1;
end
cputime_mn = toc;
fprintf('CPU time: %.6f seconds\n',cputime_mn);


% Output values
niter = iter;                       % Number of iterations
NFE = niter * 3;                    % Function evaluations: f, df, df(midpoint) per
    iteration
order = log(ratios(end)) / log(ratios(end - 1));  % Estimate order of convergence
fprintf('%.10f\n', order);          % Display order
zero = x;                           % Final approximation
res = abs(f(zero));                 % Residual |f(x)|
ek = abs(zero - alpha);             % Final error

% Plot log-log error vs iteration
figure;
loglog(1:length(errors_mid), errors_mid, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Midpoint Newton''s Method');

% Formatting for better plot readability
maxx = max(length(errors_mid));
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

MN method code for 1 variable(Exact solution unknown)

```
1
2  function [sequence, errors_mid, ratios, zero, niter, order, res, ek, NFE] =
       midpoint_newton(fun, dfun, x0, tol, nmax)
3  % MIDPOINT_NEWTON Applies the Midpoint Newton's method to solve f(x) = 0.
4  %
5  % INPUTS:
6  % fun        - Function handle for f(x)
7  % dfun       - Function handle for f'(x)
8  % x0         - Initial guess
9  % tol        - Tolerance for stopping criterion
10 % nmax       - Maximum number of iterations
11 %
12 % OUTPUTS:
13 % sequence   - All approximated values of the root
14 % errors_mid - Absolute errors at each iteration
15 % ratios     - Error ratios used to estimate the order of convergence
16 % zero       - Final approximation of the root
17 % niter      - Total number of iterations performed
18 % order      - Estimated order of convergence
19 % res        - Final residual |f(x)|
20 % ek         - Final absolute error |x - alpha|
21 % NFE        - Number of function evaluations (3 per iteration: f(x), f'(x), and f(
       midpoint))
22
23 % Assign function and its derivative
24 f = fun;
25 df = dfun;
26
27 % Initialize variables
28 x = x0;
29 iter = 0;
30 err = inf;
31
32 % Initialize containers for storing iteration data
33 errors_mid = [];
34 ratios = [];
35 sequence = [];
36
37 % Midpoint Newton's method iteration loop
38 while err > tol && iter < nmax
39
40     x_star = x - f(x) / df(x);
41     midpoint = (x + x_star) / 2;
42
43
44     x_new = x - f(x) / df(midpoint);
45
46     % Compute the error and store it
47     err = abs(x_new - x);
48     errors_mid = [errors_mid, err];
49
50     % Compute and store the error ratio (used for convergence order)
51     if iter > 0
52         ratios = [ratios, err / errors_mid(end-1)];
53     end
54
55     % Store the current approximation and update for the next iteration
56     sequence = [sequence, x];  % Store the sequence of approximations
57     x = x_new;                 % Update current guess
58     iter = iter + 1;           % Increment the iteration counter
59 end
60
61 % Output results
62 niter = iter;                  % Number of iterations performed
63 NFE = niter * 3;               % Number of function evaluations (3 per iteration)
64
```

```
% Estimate the order of convergence if at least 2 ratios are available
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end

% Display the estimated order of convergence
fprintf('Estimated order of convergence: %.4f\n', order);

% Store the final approximation of the root
zero = x;

% Calculate the residual (|f(x)|) and the final absolute error
res = abs(f(zero));  % Residual: |f(x)|
ek = err;            % Final error (|x - x_true|)

% Plot the error vs iteration in log-log scale
figure;
loglog(1:length(errors_mid), errors_mid, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Midpoint Newton''s Method');

% Adjust axis ticks and range for better visualization
maxx = length(errors_mid);
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

HN method code for 1 variable(Exact solution known)

```

function [sequence, errors_har, ratios, zero, niter, order, res, ek, NFE,cputime_hn] =
    harmonic_mean_newton1(fun, dfun, x0, tol, nmax, realroot)
% HARMONIC_MEAN_NEWTON1 Applies the Harmonic Mean Newton's method to find a root.
%
% INPUTS:
% fun      - Function handle for f(x)
% dfun     - Function handle for f'(x)
% x0       - Initial guess
% tol      - Tolerance for stopping criterion
% nmax     - Maximum number of iterations
% realroot - The actual root (used to compute the error)
%
% OUTPUTS:
% sequence     - All approximated values of the root
% errors_har   - Absolute error at each iteration
% ratios       - Error ratios used to estimate order of convergence
% zero         - Final approximation of the root
% niter        - Total number of iterations performed
% order        - Estimated order of convergence
% res          - Final residual |f(x)|
% ek           - Final absolute error |x - alpha|
% NFE          - Number of function evaluations (3 per iteration: f, df(x), df(x_star))

% Assign function and derivative
f = fun;
df = dfun;
alpha = realroot;

% Initialize parameters
max_iter = nmax;
x = x0;
iter = 0;
err = inf;

```

```
% Initialize containers for data
errors_har = [];
ratios = [];
sequence = [];

% Harmonic Mean Newton iteration loop
tic;
while err > tol && iter < max_iter

    x_star = x - f(x) / df(x);


    mult = 0.5 * (1 / df(x) + 1 / df(x_star));


    x_new = x - f(x) * mult;

    % Compute absolute error using the true root
    err = abs(x_new - alpha);

    % Estimate error ratio for convergence order
    ratio = err / abs(x - alpha);

    % Store current values
    errors_har = [errors_har err];
    ratios = [ratios ratio];
    sequence = [sequence x];

    % Update current approximation and iteration counter
    x = x_new;
    iter = iter + 1;
end

cputime_hn = toc;
fprintf('CPU time: %.6f seconds\n',cputime_hn);


% Output values
niter = iter;                           % Number of iterations
NFE = niter * 3;                        % Function evaluations: f, df(x), df(x_star)
    per iteration
order = log(ratios(end)) / log(ratios(end - 1));   % Estimate order of convergence
fprintf('%.10f\n', order);             % Display order
zero = x;                               % Final approximation
res = abs(f(zero));                     % Residual |f(x)|
ek = abs(zero - alpha);                 % Final error

% Plot log-log error vs iteration
figure;
loglog(1:length(errors_har), errors_har, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Harmonic Mean Newton''s Method');

% Formatting for better plot readability
maxx = max(length(errors_har));
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

HN method code for 1 variable(Exact solution unknown)

```

function [sequence, errors_har, ratios, zero, niter, order, res, ek, NFE] =
    harmonic_mean_newton(fun, dfun, x0, tol, nmax)
% HARMONIC_MEAN_NEWTON Applies the Harmonic Mean Newton's method to solve f(x) = 0.
%
```

```
% INPUTS:
% fun        - Function handle for f(x)
% dfun       - Function handle for f'(x)
% x0         - Initial guess
% tol        - Tolerance for stopping criterion
% nmax       - Maximum number of iterations
%
% OUTPUTS:
% sequence   - All approximated values of the root
% errors_har - Absolute errors at each iteration
% ratios     - Error ratios used to estimate the order of convergence
% zero       - Final approximation of the root
% niter      - Total number of iterations performed
% order      - Estimated order of convergence
% res        - Final residual |f(x)|
% ek         - Final absolute error |x - alpha|
% NFE        - Number of function evaluations (3 per iteration: f(x), f'(x), and f(
    x_star))

% Assign function and its derivative
f = fun;
df = dfun;

% Initialize variables
x = x0;
iter = 0;
err = inf;

% Initialize containers for storing iteration data
errors_har = [];
ratios = [];
sequence = [];

% Harmonic Mean Newton's method iteration loop
while err > tol && iter < nmax

    x_star = x - f(x) / df(x);


    mult = 0.5 * (1 / df(x) + 1 / df(x_star));
    x_new = x - f(x) * mult;

    % Compute the error and store it
    err = abs(x_new - x);
    errors_har = [errors_har, err];

    % Compute and store the error ratio (used for convergence order)
    if iter > 0
        ratios = [ratios, err / errors_har(end-1)];
    end

    % Store the current approximation and update for the next iteration
    sequence = [sequence, x];  % Store the sequence of approximations
    x = x_new;                 % Update current guess
    iter = iter + 1;           % Increment the iteration counter
end

% Output results
niter = iter;                   % Number of iterations performed
NFE = niter * 3;                % Number of function evaluations (3 per iteration)

% Estimate the order of convergence if at least 2 ratios are available
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end

```

```
% Display the estimated order of convergence
fprintf('Estimated order of convergence: %.4f\n', order);

% Store the final approximation of the root
zero = x;

% Calculate the residual (|f(x)|) and the final absolute error
res = abs(f(zero));
ek = err;

% Plot the error vs iteration in log-log scale
figure;
loglog(1:length(errors_har), errors_har, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Harmonic Mean Newton''s Method');

% Adjust axis ticks and range for better visualization
maxx = length(errors_har);
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

MHN method code for 1 variable(Exact solution known)

```
function [sequence, errors_midhar, ratios, zero, niter, order, res, ek, NFE,cputime_mhn
    ] = midpoint_harmonic_mean_newton1(fun, dfun, x0, tol, nmax, realroot)
% MIDPOINT_HARMONIC_MEAN_NEWTON1 Applies the Midpoint-Harmonic Mean Newton method to
    find a root.
%
% INPUTS:
% fun        - Function handle for f(x)
% dfun       - Function handle for f'(x)
% x0         - Initial guess
% tol        - Tolerance for stopping criterion
% nmax       - Maximum number of iterations
% realroot   - The actual root (used to compute the error)
%
% OUTPUTS:
% sequence        - All approximated values of the root
% errors_midhar   - Absolute error at each iteration
% ratios          - Error ratios used to estimate order of convergence
% zero            - Final approximation of the root
% niter           - Total number of iterations performed
% order           - Estimated order of convergence
% res             - Final residual |f(x)|
% ek              - Final absolute error |x - alpha|
% NFE             - Number of function evaluations (3 per iteration: f, df(x), df(
    midpoint))

clc;

% Assign function and derivative
f = fun;
df = dfun;
alpha = realroot;

% Initialize variables
max_iter = nmax;
x = x0;
iter = 0;
err = inf;

% Initialize containers for tracking convergence data
errors_midhar = [];
ratios = [];
sequence = [];
```

```

% Midpoint-Harmonic Mean Newton iteration loop
tic;
while err > tol && iter < max_iter

    x_star = x - f(x) / df(x);


    midpoint = 0.5 * (x + x_star);


    df_midpoint = df(midpoint);


    mult = 0.5 * (1 / df(x) + 1 / (2 * df_midpoint - df(x)));


    x_new = x - f(x) * mult;

    % Compute absolute error and ratio
    err = abs(x_new - alpha);
    ratio = err / abs(x - alpha);

    % Store current iteration data
    errors_midhar = [errors_midhar err];
    ratios = [ratios ratio];
    sequence = [sequence x];

    % Update x and iteration count
    x = x_new;
    iter = iter + 1;
end
cputime_mhn = toc;
fprintf('CPU time: %.6f seconds\n',cputime_mhn);

% Output values
niter = iter;                               % Number of iterations
NFE = niter * 3;                            % Function evaluations per iteration: f, df(
    x), df(midpoint)
order = log(ratios(end)) / log(ratios(end - 1));  % Estimate order of convergence
fprintf('%.10f\n', order);                 % Display order
zero = x;                                   % Final root approximation
res = abs(f(zero));                         % Residual |f(x)|
ek = abs(zero - alpha);                     % Final absolute error

% Plot error vs iteration on log-log scale
figure;
loglog(1:length(errors_midhar), errors_midhar, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Midpoint-Harmonic Mean Newton''s Method');

% Improve plot readability
maxx = max(length(errors_midhar));
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;

end
```

MHN method code for 1 variable(Exact solution unknown)

```
function [sequence, errors_midhar, ratios, zero, niter, order, res, ek, NFE] =
    midpoint_harmonic_mean_newton(fun, dfun, x0, tol, nmax)
% MIDPOINT_HARMONIC_MEAN_NEWTON Solves f(x) = 0 using the Midpoint-Harmonic Mean Newton
    's method.
%
% INPUTS:
```

```
% fun          - Function handle for f(x)
% dfun         - Function handle for f'(x)
% x0           - Initial guess
% tol          - Tolerance for stopping criterion
% nmax         - Maximum number of iterations
%
% OUTPUTS:
% sequence     - Sequence of approximated root values
% errors_midhar- Absolute errors at each iteration
% ratios       - Ratios of successive errors (used to estimate order)
% zero         - Final approximation of the root
% niter        - Total number of iterations performed
% order        - Estimated order of convergence
% res          - Final residual |f(x)|
% ek           - Final absolute error |x - x_true|
% NFE          - Number of function evaluations (3 per iteration)

% Assign function and its derivative
f = fun;
df = dfun;

% Initialize variables
x = x0;
iter = 0;
err = inf;
% Initialize storage vectors
errors_midhar = [];
ratios = [];
sequence = [];

% Iterative process of Midpoint-Harmonic Mean Newton's Method
while err > tol && iter < nmax


    x_star = x - f(x) / df(x);


    midpoint = 0.5 * (x + x_star);
    df_midpoint = df(midpoint);


    mult = 0.5 * (1 / df(x) + 1 / (2 * df_midpoint - df(x)));
    x_new = x - f(x) * mult;

    % Compute and store error
    err = abs(x_new - x);
    errors_midhar = [errors_midhar, err];

    % Compute and store ratio of successive errors
    if iter > 0
        ratios = [ratios, err / errors_midhar(end-1)];
    end

    % Store current approximation
    sequence = [sequence, x];

    % Update for next iteration
    x = x_new;
    iter = iter + 1;
end

% Final number of iterations
niter = iter;

% Estimate number of function evaluations (3 per iteration)
NFE = niter * 3;

% Estimate order of convergence if enough data
```

```
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end

% Display the estimated order
fprintf('Estimated order of convergence: %.4f\n', order);

% Final approximation of the root
zero = x;

% Final residual and error
res = abs(f(zero));
ek = err;

% Plot the error in log-log scale
figure;
loglog(1:length(errors_midhar), errors_midhar, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Midpoint-Harmonic Mean Newton''s Method');

% Adjust plot ticks and view range
maxx = length(errors_midhar);
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

FP method code for 1 variable(Exact solution known)

```

function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    fixed_point_method(fun, x0, tol, nmax, realroot)
% FIXED_POINT_METHOD Applies a fixed-point iteration to solve f(x) = 0.
%
% INPUTS:
% fun       - Function handle for f(x)
% x0        - Initial guess
% tol       - Tolerance for stopping criterion
% nmax      - Maximum number of iterations
% realroot  - The actual root (used to compute the error)
%
% OUTPUTS:
% sequence  - All approximated values of the root
% errors    - Absolute error at each iteration
% ratios    - Error ratios used to estimate order of convergence
% zero      - Final approximation of the root
% niter     - Total number of iterations performed
% order     - Estimated order of convergence
% res       - Final residual |f(x)|
% ek        - Final absolute error |x - alpha|
% NFE       - Number of function evaluations (1 per iteration)

% Assign function and parameters
f = fun;
alpha = realroot;

% Initialize variables
max_iter = nmax;
x = x0;
iter = 0;
err = inf;

% Initialize containers to track progress
errors = [];
ratios = [];
```

```
sequence = [];

% Fixed-point iteration loop
while err > tol && iter < max_iter
    % Perform fixed-point update
    x_new = g(x);

    % Compute error and ratio
    err = abs(x_new - alpha);
    ratio = err / abs(x - alpha);

    % Store current iteration data
    errors = [errors err];
    ratios = [ratios ratio];
    sequence = [sequence x];

    % Update current value and iteration count
    x = x_new;
    iter = iter + 1;
end

% Output results
niter = iter;                           % Number of iterations performed
NFE = niter;                            % One function evaluation per iteration
order = log(ratios(end)) / log(ratios(end - 1));  % Estimate order of convergence
fprintf('%.10f\n', order);             % Print estimated order
zero = x;                               % Final approximation of the root
res = abs(f(zero));                     % Final residual
ek = abs(zero - alpha);                 % Final error

% Plot error vs iteration in log-log scale
figure;
loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Fixed Point Method');

% Adjust axis ticks and range for readability
maxx = max(length(errors));
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;
end
```

FP method code for 1 variable(Exact solution unknown)

```

function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    fixed_point_method1(fun, x0, tol, nmax)
% FIXED_POINT_METHOD1 Solves f(x) = 0 using a fixed-point iteration scheme.
%
% INPUTS:
% fun      - Function handle for f(x)
% x0       - Initial guess
% tol      - Tolerance for stopping criterion
% nmax     - Maximum number of iterations
%
% OUTPUTS:
% sequence - All approximated values of the root
% errors   - Absolute errors at each iteration
% ratios   - Ratios of successive errors (used to estimate convergence order)
% zero     - Final approximation of the root
% niter    - Number of iterations performed
% order    - Estimated order of convergence
% res      - Final residual |f(x)|
% ek       - Final absolute error |x - x_true|
% NFE      - Number of function evaluations (1 per iteration)

```

```
% Assign function
f = fun;

% Initialize variables
x = x0;
iter = 0;
err = inf;

% Initialize containers to store iteration data
errors = [];
ratios = [];
sequence = [];

% Iterative process of the fixed-point method
while err > tol && iter < nmax

    x_new = g(x);

    % Compute and store absolute error
    err = abs(x_new - x);
    errors = [errors, err];

    % Store error ratio (used for estimating convergence order)
    if iter > 0
        ratios = [ratios, err / errors(end-1)];
    end

    % Store current value before update
    sequence = [sequence, x];

    % Update for next iteration
    x = x_new;
    iter = iter + 1;
end

% Final results
niter = iter;               % Total number of iterations
NFE = niter;                % Function evaluations (1 per iteration)
zero = x;                   % Final approximation of the root
res = abs(f(zero));         % Final residual
ek = err;                   % Final absolute error

% Estimate the order of convergence (if enough data)
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end
fprintf('Estimated order of convergence: %.4f\n', order);

% Plot error vs iteration on a log-log scale
figure;
loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
xlabel('n');
ylabel('e_n');
title('Fixed-Point Method');

% Adjust axis ticks for better readability
maxx = length(errors);
xticks(unique(round(logspace(0, log10(maxx), 10))));
xlim([1 maxx]);
grid on;

end
```

CN method code for solving a system of two nonlinear equations (Exact solution known)

```

function [sequence, errors, ratios, zero, niter, res, order, ek, NFE] = newton_system(
    fun, Jfun, X0, tol, nmax, realroot)
    % Initialize variables
    X = X0;                             % Initial guess
    alpha = realroot;                   % Actual root (used to compute error)
    iter = 0;
    err = inf;
    errors = [];
    ratios = [];
    sequence = X0;                      % Store iterates

    % Newton's method loop
    while err > tol && iter < nmax
        F = fun(X(1), X(2));            % Evaluate function at current guess
        J = Jfun(X(1), X(2));           % Evaluate Jacobian matrix

        delta = -J \ F;
        X_new = X + delta;

        err = norm(X_new - alpha, 2); % Compute error (Euclidean norm)
        ratio = err / norm(X - alpha);
        errors = [errors err];          % Store error
        ratios = [ratios ratio];        % Store error ratio
        sequence = [sequence, X_new]; % Store new iterate

        X = X_new;                      % Prepare for next iteration
        iter = iter + 1;
    end

    % Output final results
    zero = X;                           % Final approximation
    niter = iter;                       % Number of iterations
    NFE = niter * 2;                    % Function & Jacobian evaluations
    order = log(ratios(end)) / log(ratios(end-1)); % Estimated order
    fprintf('%.10f\n', order);
    res = norm(fun(zero(1), zero(2)));    % Residual
    ek = norm(zero - alpha);              % Final error

    % Plot error on log-log scale
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
    title('Newton''s Method for System of Two Functions');

    maxx = max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;
end
```

CN method code for solving a system of two nonlinear equations (Exact solution unknown)

```


function [sequence, errors, ratios, zero, niter, res, order, ek, NFE] = newton_systemm(
    fun, Jfun, X0, tol, nmax)

    X = X0;                         % Initial guess
    iter = 0;
    err = inf;

    errors = [];
    ratios = [];
    sequence = X;                   % Store all iterates
```

```
12
13      while err > tol && iter < nmax
14          F = fun(X(1), X(2));       % Evaluate function
15          J = Jfun(X(1), X(2));      % Evaluate Jacobian
16
17          delta = -J \ F;
18          X_new = X + delta;
19
20          err = norm(X_new - X, 2);       % Compute error (Euclidean norm)
21          errors = [errors, err];          % Store error
22
23          if iter > 0
24              ratios = [ratios, err / errors(end-1)];  % Error ratio
25          end
26
27          sequence = [sequence, X_new];  % Store iterate
28          X = X_new;
29          iter = iter + 1;
30      end
31
32      if length(ratios) >= 2
33          order = log(ratios(end)) / log(ratios(end-1));  % Convergence order
34      else
35          order = NaN;
36      end
37      fprintf('Estimated order of convergence: %.4f\n', order);
38
39      zero = X;                       % Final root
40      niter = iter;                   % Number of iterations
41      NFE = niter * 2;                % Function/Jacobian evaluations
42      res = norm(fun(zero(1), zero(2)));  % Final residual
43      ek = err;                       % Final error
44
45      figure;
46      loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);  % Error plot
47      xlabel('n');
48      ylabel('e_n');
49      title('Newton''s Method for System of Two Functions');
50
51      maxx = length(errors);
52      xticks(unique(round(logspace(0, log10(maxx), 10))));  % Log-spaced x-ticks
53      xlim([1 maxx]);                                        % X-axis range
54      grid on;
55
56  end
```

AN method code for solving a system of two nonlinear equations (Exact solution known)

```
1   function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
        arithmetic_system(fun, Jfun, X0, tol, nmax, realroot)
2
3       % Initialization
4       X = X0;                          % Initial guess
5       alpha = realroot;               % Actual root (for error measurement)
6       iter = 0;
7       err = inf;
8       errors = [];
9       ratios = [];
10      sequence = X0;
11
12      % Iterative loop for Arithmetic Mean Newton's Method
13      while err > tol && iter < nmax
14          F = fun(X(1), X(2));         % Evaluate function
15          J = Jfun(X(1), X(2));        % Evaluate Jacobian at current point
16
17          delta_star = -J \ F;
18          X_star = X + delta_star;
19
```

```
        J_star = Jfun(X_star(1), X_star(2));

        J_mean = 0.5 * (J + J_star);
        delta = -J_mean \ F;
        X_new = X + delta;

        err = norm(X_new - alpha, 2);               % Compute error (Euclidean norm)
        ratio = err / norm(X - alpha);              % Error ratio
        errors = [errors, err];                     % Store error
        ratios = [ratios ratio];                    % Store ratio
        sequence = [sequence, X_new];               % Store iterate

        X = X_new;                                   % Prepare for next iteration
        iter = iter + 1;
    end

    % Output results
    zero = X;                       % Final approximation
    niter = iter;                   % Iteration count
    NFE = niter * 3;                % Function/Jacobian evaluations
    order = log(ratios(end)) / log(ratios(end-1));  % Estimate order
    res = norm(fun(zero(1), zero(2)));              % Residual
    ek = norm(zero - alpha);                        % Final error

    % Plot log-log error graph
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
    title('Arithmetic Mean Newton''s Method for System of Two Functions');

    maxx = max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;

end
```

AN method code for solving a system of two nonlinear equations (Exact solution unknown)

```


function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    arithmetic_systemm(fun, Jfun, X0, tol, nmax)

    X = X0;                          % Initial guess
    iter = 0;
    err = inf;

    errors = [];
    ratios = [];
    sequence = X;                    % Store iterates

    while err > tol && iter < nmax
        F = fun(X(1), X(2));         % Evaluate function
        J = Jfun(X(1), X(2));        % Evaluate Jacobian

        delta_star = -J \ F;
        X_star = X + delta_star;

        J_star = Jfun(X_star(1), X_star(2));
        J_mean = 0.5 * (J + J_star);

        delta = -J_mean \ F;
        X_new = X + delta;

        err = norm(X_new - X, 2);    % Compute error (Euclidean norm)
        errors = [errors, err];
```

```
28
29          if iter > 0
30              ratios = [ratios, err / errors(end-1)];  % Error ratios
31          end
32
33          sequence = [sequence, X_new];  % Store iterate
34          X = X_new;
35          iter = iter + 1;
36      end
37
38      zero = X;                                   % Final approximation
39      niter = iter;                               % Iteration count
40      NFE = niter * 3;                            % Function evals (fun + 2 Jfun)
41      res = norm(fun(zero(1), zero(2)));          % Final residual
42      ek = err;                                   % Final error
43
44      if length(ratios) >= 2
45          order = log(ratios(end)) / log(ratios(end-1));  % Estimated order
46      else
47          order = NaN;
48      end
49      fprintf('Estimated order of convergence: %.4f\n', order);
50
51      figure;
52      loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);  % Plot error
53      xlabel('n');
54      ylabel('e_n');
55      title('Arithmetic Mean Newton''s Method for System of Two Functions');
56
57      maxx = length(errors);
58      xticks(unique(round(logspace(0, log10(maxx), 10))));  % Log-scaled x-ticks
59      xlim([1 maxx]);
60      grid on;
61
62  end
```

MN method code for solving a system of two nonlinear equations (Exact solution known)

```
1
2   function [sequence, errors,ratios, zero, niter,order, res, ek, NFE] = midpoint_system(
        fun, Jfun, X0, tol, nmax, realroot)
3
4       X = X0;                          % Initial guess
5       alpha = realroot;                % True root (for error analysis)
6       iter = 0;
7       err = inf;
8       errors = [];
9       ratios =[];
10      sequence = X0;                   % Track iterates
11
12      while err > tol && iter < nmax
13          F = fun(X(1), X(2));         % Evaluate function
14          J = Jfun(X(1), X(2));        % Evaluate Jacobian at X
15          delta_star = -J\ F;
16          X_star = X + delta_star;
17          X_mid = (X + X_star) / 2;
18          J_mid = Jfun(X_mid(1), X_mid(2));
19          delta = -J_mid \ F;
20          X_new = X + delta;
21          err = norm(X_new - alpha, 2);       % Compute error (Euclidean norm)
22          ratio = err / norm(X - alpha);      % Error ratio
23          errors = [errors, err];      % Store error
24          ratios = [ratios ratio];     % Store ratio
25          sequence = [sequence, X_new];       % Save iterate
26          X = X_new;                   % Prepare for next iteration
27          iter = iter + 1;
28      end
29
```

```
    zero = X;                       % Final approximation
    niter = iter;                   % Number of iterations
    NFE = niter * 3;                % Function/Jacobian evals
    order = log(ratios(end)) / log(ratios(end-1));  % Estimated order
    res = norm(fun(zero(1), zero(2)));              % Final residual
    ek = norm(zero - alpha);                        % Final error

    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5); % Plot error curve
    xlabel('n');
    ylabel('e_n');
    title(' Midpoint Newton''s Method for System of Two Functions');
    maxx= max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;

end
```

MN method code for solving a system of two nonlinear equations (Exact solution unknown)

```

function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    midpoint_systemm(fun, Jfun, X0, tol, nmax)

    X = X0;                         % Initial guess
    iter = 0;
    err = inf;

    errors = [];                    % List to store errors
    ratios = [];                    % List to store error ratios
    sequence = X;                   % Store iterates

    while err > tol && iter < nmax
        F = fun(X(1), X(2));        % Evaluate function at current point
        J = Jfun(X(1), X(2));       % Evaluate Jacobian at current point

        delta_star = -J \ F;
        X_star = X + delta_star;

        X_mid = (X + X_star) / 2;
        J_mid = Jfun(X_mid(1), X_mid(2));

        delta = -J_mid \ F;
        X_new = X + delta;

        err = norm(X_new - X, 2);   % Compute error (Euclidean norm)
        errors = [errors, err];     % Store error

        if iter > 0
            ratios = [ratios, err / errors(end-1)];  % Compute error ratio
        end

        sequence = [sequence, X_new];  % Append new point to sequence
        X = X_new;
        iter = iter + 1;
    end

    zero = X;                                % Final approximation
    niter = iter;                            % Total iterations
    NFE = niter * 3;                         % Function evaluations (1 fun + 2 Jfun per
        iter)
    res = norm(fun(zero(1), zero(2)));       % Residual at final approximation
    ek = err;                                % Final error

    if length(ratios) >= 2
        order = log(ratios(end)) / log(ratios(end-1));  % Estimated order of
            convergence
```

```
    else
        order = NaN;
    end
    fprintf('Estimated order of convergence: %.4f\n', order);

    % Plotting error vs iteration on log-log scale
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
    title('Midpoint Newton''s Method for System of Two Functions');

    maxx = length(errors);
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;

end
```

HN method code for a system of two nonlinear equations (Exact solution known)

```

function [sequence, errors,ratios, zero, niter,order, res, ek, NFE] = harmonic_system(
    fun, Jfun, X0, tol, nmax, realroot)

    X = X0;                          % Initial guess
    alpha = realroot;               % True solution (for error analysis)
    iter = 0;
    err = inf;
    errors = [];
    ratios = [];
    sequence = X0;                  % Store all iterates

    while err > tol && iter < nmax
        F = fun(X(1), X(2));        % Evaluate function at X
        J = Jfun(X(1), X(2));       % Evaluate Jacobian at X
        delta_star = -J \ F;
        X_star = X + delta_star;

        J_star = Jfun(X_star(1), X_star(2));

        J_mean_inv = 0.5 * (inv(J) + inv(J_star));
        delta = -J_mean_inv * F;
        X_new = X + delta;

        err = norm(X_new - alpha, 2);        % Compute error (Euclidean norm)
        ratio = err / norm(X - alpha);        % Error ratio
        errors = [errors, err];               % Track errors
        ratios = [ratios, ratio];             % Track ratios
        sequence = [sequence, X_new];         % Store iterates

        X = X_new;                  % Update current point
        iter = iter + 1;            % Increment iteration count
    end

    zero = X;                       % Final root approximation
    niter = iter;                   % Total iterations
    NFE = niter * 3;                % Function and Jacobian evaluations
    order = log(ratios(end)) / log(ratios(end-1));  % Convergence order
    res = norm(fun(zero(1), zero(2)));              % Final residual
    ek = norm(zero - alpha);                        % Final  error

    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5); % Error plot
    xlabel('n');
    ylabel('e_n');
    title(' Harmonic Mean Newton''s Method for System of Two Functions');

```

```
    maxx = max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));      % X-tick positions
    xlim([1 maxx]);                                             % X-axis limit
    grid on;

end
```

HN method code for a system of two nonlinear equations (Exact solution unknown)

```

function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    harmonic_systemm(fun, Jfun, X0, tol, nmax)

    X = X0;                          % Initial guess
    iter = 0;
    err = inf;

    errors = [];                     % Store error values
    ratios = [];                     % Store error ratios
    sequence = X;                    % Store iteration sequence

    while err > tol && iter < nmax
        F = fun(X(1), X(2));         % Evaluate function at current point
        J = Jfun(X(1), X(2));        % Evaluate Jacobian at current point

        delta_star = -J \ F;
        X_star = X + delta_star;

        J_star = Jfun(X_star(1), X_star(2));


        J_mean_inv = 0.5 * (inv(J) + inv(J_star));

        delta = -J_mean_inv * F;
        X_new = X + delta;

        err = norm(X_new - X, 2);  % Compute error (Euclidean norm)
        errors = [errors, err];     % Save error

        if iter > 0
            ratios = [ratios, err / errors(end-1)];  % Error ratio
        end

        sequence = [sequence, X_new];  % Store current iterate
        X = X_new;
        iter = iter + 1;
    end

    zero = X;                                % Final approximation
    niter = iter;                            % Total number of iterations
    NFE = niter * 3;                         % Function evaluations (1 fun + 2 Jfun per
        iter)
    res = norm(fun(zero(1), zero(2)));       % Residual at the solution
    ek = err;                                % Final error

    % Estimate order of convergence
    if length(ratios) >= 2
        order = log(ratios(end)) / log(ratios(end-1));
    else
        order = NaN;
    end
    fprintf('Estimated order of convergence: %.4f\n', order);

    % Plot error vs iteration
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
```

```
    title('Harmonic Mean Newton''s Method for System of Two Functions');

    maxx = length(errors);
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;

end
```

MHN Method code for solving a system of two nonlinear equations (Exact solution known)

```

function [sequence, errors,ratios, zero, niter,order, res, ek, NFE] =
    midpoint_harmonic_system(fun, Jfun, X0, tol, nmax, realroot)

    X = X0;                          % Initial guess
    alpha = realroot;               % True root (for error analysis)
    iter = 0;
    err = inf;
    errors = [];
    ratios = [];
    sequence = X0;                  % Store all iterates

    while err > tol && iter < nmax
        F = fun(X(1), X(2));        % Evaluate function at current point
        J = Jfun(X(1), X(2));       % Evaluate Jacobian at current point

        delta_star = -J \ F;
        X_star = X + delta_star;

        X_mid = 0.5 * (X + X_star);
        J_mid = Jfun(X_mid(1), X_mid(2));

        J_mod_inv = 0.5 * (inv(J) + inv(2 * J_mid - J));

        delta = -J_mod_inv * F;
        X_new = X + delta;

        err = norm(X_new - alpha, 2);               % Compute error (Euclidean norm)
        ratio = err / norm(X - alpha);              % Error ratio
        errors = [errors, err];                     % Store error
        ratios = [ratios, ratio];                   % Store ratio
        sequence = [sequence, X_new];               % Store iterates

        X = X_new;                  % Update current point
        iter = iter + 1;            % Next iteration
    end

    zero = X;                       % Final root
    niter = iter;                   % Total iterations
    NFE = niter * 3;                % Function and Jacobian evaluations
    res = norm(fun(zero(1), zero(2)));              % Final residual
    order = log(ratios(end)) / log(ratios(end-1));  % Convergence order
    ek = norm(zero - alpha);                        % Final error

    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5); % Error plot
    xlabel('n');
    ylabel('e_n');
    title(' Midpoint-Harmonic Mean Newton''s Method for System of Two Functions');

    maxx = max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));  % X-ticks for log scale
    xlim([1 maxx]);                                        % X-axis range
    grid on;

end
```

MHN method code for solving a system of two nonlinear equations (Exact solution uknown)

```

function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    midpoint_harmonic_systemm(fun, Jfun, X0, tol, nmax)

    X = X0;                          % Initial guess
    iter = 0;
    err = inf;

    errors = [];                     % Store error norms
    ratios = [];                     % Store error ratios
    sequence = X;                    % Store iteration sequence

    while err > tol && iter < nmax
        F = fun(X(1), X(2));         % Evaluate function at current point
        J = Jfun(X(1), X(2));        % Evaluate Jacobian at current point

        delta_star = -J \ F;
        X_star = X + delta_star;

        X_mid = 0.5 * (X + X_star);
        J_mid = Jfun(X_mid(1), X_mid(2));

        J_mod_inv = 0.5 * (inv(J) + inv(2 * J_mid - J));
        delta = -J_mod_inv * F;
        X_new = X + delta;

        err = norm(X_new - X, 2);  % Compute error (Euclidean norm)
        errors = [errors, err];     % Save error

        if iter > 0
            ratios = [ratios, err / errors(end-1)];  % Error ratio
        end

        sequence = [sequence, X_new];  % Store current iterate
        X = X_new;
        iter = iter + 1;
    end

    zero = X;                                % Final approximation
    niter = iter;                            % Total iterations
    NFE = niter * 3;                         % Function evaluations (1 fun + 2 Jfun per
        iter)
    res = norm(fun(zero(1), zero(2)));     % Residual at solution
    ek = err;                                % Final error

    % Estimate order of convergence
    if length(ratios) >= 2
        order = log(ratios(end)) / log(ratios(end-1));
    else
        order = NaN;
    end
    fprintf('Estimated order of convergence: %.4f\n', order);

    % Plot error vs. iteration (log scale)
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
    title('Midpoint-Harmonic Mean Newton''s Method for System of Two Functions');

    maxx = length(errors);
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;

end
```

CN method code for solving a system of three nonlinear equations

```
function [sequence, errors, ratios, zero, niter, res, order, ek, NFE] = newton_system2(
    fun, Jfun, X0, tol, nmax, realroot)

    X = X0;                         % Initial guess
    alpha = realroot;               % Known or expected true solution (used for error
        comparison)
    iter = 0;
    err = inf;

    errors = [];                    % Error norms at each step
    ratios = [];                    % Successive error ratios
    sequence = X0;                  % Store each iterate

    while err > tol && iter < nmax
        F = fun(X(1), X(2), X(3));       % Evaluate function at current guess
        J = Jfun(X(1), X(2), X(3));      % Evaluate Jacobian at current guess

        delta = -J \ F;
        X_new = X + delta;

        err = norm(X_new - alpha, 2);   % Compute error (Euclidean norm)
        ratio = err / norm(X - alpha);  % Error ratio between current and previous
            iteration

        errors = [errors, err];         % Store current error
        ratios = [ratios, ratio];       % Store error ratio
        sequence = [sequence, X_new];   % Append current iterate

        X = X_new;                      % Move to new guess
        iter = iter + 1;
    end

    zero = X;                           % Final approximation of the solution
    niter = iter;                       % Total number of iterations
    NFE = niter * 2;                    % Number of function evaluations (F and J per
        iteration)

    % Estimate order of convergence (last two ratios)
    order = log(ratios(end)) / log(ratios(end-1));
    fprintf('Estimated order of convergence: %.10f\n', order);

    res = norm(fun(zero(1), zero(2), zero(3)));  % Final residual norm
    ek = norm(zero - alpha);                    % Final error

    % Plot error on a log-log scale
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
    title('Newton''s Method for System of Three Functions');

    maxx = max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;

end
```

AN method code for solving a system of three nonlinear equations

```
function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    arithmetic_system2(fun, Jfun, X0, tol, nmax, realroot)
```

```
4
5       X = X0;                             % Initial guess
6       alpha = realroot;                   % Known or expected solution for reference error
7       iter = 0;
8       err = inf;
9
10      errors = [];                        % To store the errors at each iteration
11      ratios = [];                        % To store error ratios between steps
12      sequence = X0;                      % Store each iterate for analysis or plotting
13
14      while err > tol && iter < nmax
15          F = fun(X(1), X(2), X(3));              % Evaluate function vector at current
                guess
16          J = Jfun(X(1), X(2), X(3));             % Evaluate Jacobian matrix at current
                guess
17
18          delta_star = -J \ F;
19          X_star = X + delta_star;
20
21          J_star = Jfun(X_star(1), X_star(2), X_star(3));
22          J_mean = 0.5 * (J + J_star);
23
24          delta = -J_mean \ F;
25          X_new = X + delta;
26
27          err = norm(X_new - alpha, 2);           % Compute error (Euclidean norm)
28          ratio = err / norm(X - alpha);          % Successive error ratio
29          errors = [errors, err];                 % Store error
30          ratios = [ratios, ratio];               % Store ratio
31          sequence = [sequence, X_new];           % Store iterate
32
33          X = X_new;                              % Move to next step
34          iter = iter + 1;
35      end
36
37      zero = X;                                   % Final approximation
38      niter = iter;                               % Total iterations
39      NFE = niter * 3;                            % Number of function evaluations (F +
            2*J per iteration)
40
41      % Estimate order of convergence
42      order = log(ratios(end)) / log(ratios(end-1));
43      fprintf('Estimated order of convergence: %.10f\n', order);
44
45      res = norm(fun(zero(1), zero(2), zero(3)));   % Residual norm at the final
            approximation
46      ek = norm(zero - alpha);                    % Final error
47
48      % Log-log plot of error vs. iteration
49      figure;
50      loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
51      xlabel('n');
52      ylabel('e_n');
53      title('Arithmetic Mean Newton''s Method for System of Three Functions');
54
55      maxx = max(length(errors));
56      xticks(unique(round(logspace(0, log10(maxx), 10))));
57      xlim([1 maxx]);
58      grid on;
59
60  end
```

MN method code for solving a system of three nonlinear equations

```
1
2
3   function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
        midpoint_system2(fun, Jfun, X0, tol, nmax, realroot)
```

```
4
5       X = X0;                          % Initial approximation
6       alpha = realroot;
7       iter = 0;
8       err = inf;
9
10      errors = [];                     % Store error at each step
11      ratios = [];                     % Store error ratios between iterations
12      sequence = X0;                   % Track all iterates
13
14      while err > tol && iter < nmax
15          F = fun(X(1), X(2), X(3));               % Evaluate function vector
16          J = Jfun(X(1), X(2), X(3));              % Evaluate Jacobian at current point
17
18          delta_star = -J \ F;
19          X_star = X + delta_star;
20          X_mid = 0.5 * (X + X_star);
21          J_mid = Jfun(X_mid(1), X_mid(2), X_mid(3));
22
23          delta = -J_mid \ F;
24          X_new = X + delta;
25
26          err = norm(X_new - alpha, 2);        % Compute error (Euclidean norm)
27          ratio = err / norm(X - alpha);           % Ratio for convergence order
28          errors = [errors, err];
29          ratios = [ratios, ratio];
30          sequence = [sequence, X_new];            % Append current estimate to sequence
31
32          X = X_new;                               % Move to next iteration
33          iter = iter + 1;
34      end
35
36      zero = X;                                    % Final approximation of the root
37      niter = iter;                                % Total number of iterations performed
38      NFE = niter * 3;                             % Function evaluations (fun + 2*Jfun
            per iteration)
39
40      order = log(ratios(end)) / log(ratios(end-1));  % Estimate of convergence order
41      res = norm(fun(zero(1), zero(2), zero(3)));     % Norm of residual at final
            estimate
42      ek = norm(zero - alpha);                         % Final error
43
44      % Plotting error vs. iteration on a log-log scale
45      figure;
46      loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
47      xlabel('n');
48      ylabel('e_n');
49      title('Midpoint Newton''s Method for System of Three Functions');
50
51      maxx = max(length(errors));
52      xticks(unique(round(logspace(0, log10(maxx), 10))));
53      xlim([1 maxx]);
54      grid on;
55  end
```

HN method code for solving a system of three nonlinear equations

```
1
2
3   function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
        harmonic_system2(fun, Jfun, X0, tol, nmax, realroot)
4
5       X = X0;                           % Initial point
6       alpha = realroot;
7       iter = 0;
8       err = inf;
9
10      errors = [];                     % Track errors at each iteration
```

```
    ratios = [];                    % Track error ratios to estimate order
    sequence = X0;                  % Store all iterates for visualization

    while err > tol && iter < nmax
        F = fun(X(1), X(2), X(3));                  % Evaluate function vector at X
        J = Jfun(X(1), X(2), X(3));                 % Evaluate Jacobian at X

        delta_star = -J \ F;
        X_star = X + delta_star;

        J_star = Jfun(X_star(1), X_star(2), X_star(3));


        J_mean_inv = 0.5 * (inv(J) + inv(J_star));

        delta = -J_mean_inv * F;
        X_new = X + delta;

        err = norm(X_new - alpha, 2);               % Compute error (Euclidean norm)
        ratio = err / norm(X - alpha);              % Ratio for convergence analysis
        errors = [errors, err];
        ratios = [ratios, ratio];
        sequence = [sequence, X_new];               % Save current point

        X = X_new;                                   % Prepare for next iteration
        iter = iter + 1;
    end

    zero = X;                                       % Final approximate solution
    niter = iter;                                   % Total number of iterations
    NFE = niter * 3;                                % Number of function evaluations (1
        fun + 2 Jfun per iteration)

    order = log(ratios(end)) / log(ratios(end-1)); % Estimate convergence order
    res = norm(fun(zero(1), zero(2), zero(3)));    % Residual at the final solution
    ek = norm(zero - alpha);                        % Final error

    % Plot error vs. iteration (log scale)
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
    title('Harmonic Mean Newton''s Method for System of Three Functions');

    maxx = max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;
end
```

MHN method code for solving a system of three nonlinear equations

```


function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    midpoint_harmonic_system2(fun, Jfun, X0, tol, nmax, realroot)

    X = X0;                          % Starting point for iteration
    alpha = realroot;
    iter = 0;
    err = inf;

    errors = [];                    % List of error values at each iteration
    ratios = [];                    % List of error ratios (used to estimate
        convergence order)
    sequence = X0;                  % Stores the iterates

    while err > tol && iter < nmax
```

```
        F = fun(X(1), X(2), X(3));              % Evaluate function at current point
        J = Jfun(X(1), X(2), X(3));             % Evaluate Jacobian at current point

        delta_star = -J \ F;
        X_star = X + delta_star;

        X_mid = 0.5 * (X + X_star);
        J_mid = Jfun(X_mid(1), X_mid(2), X_mid(3));

        J_mod_inv = 0.5 * (inv(J) + inv(2 * J_mid - J));

        delta = -J_mod_inv * F;
        X_new = X + delta;

        err = norm(X_new - alpha, 2);              % Compute error (Euclidean norm)
        ratio = err / norm(X - alpha);          % Calculate error ratio for convergence
            check
        errors = [errors, err];
        ratios = [ratios, ratio];
        sequence = [sequence, X_new];           % Store current solution estimate

        X = X_new;                              % Prepare for next iteration
        iter = iter + 1;
    end

    zero = X;                                   % Final approximation of the root
    niter = iter;                               % Number of iterations completed
    NFE = niter * 3;                            % Function evaluations (1 F + 2 J per
        iteration)
    res = norm(fun(zero(1), zero(2), zero(3))); % Final residual
    order = log(ratios(end)) / log(ratios(end-1));  % Estimate convergence order
    ek = norm(zero - alpha);                    % Final error

    % Plotting log-log graph of errors
    figure;
    loglog(1:length(errors), errors, 'o-', 'LineWidth', 1.5);
    xlabel('n');
    ylabel('e_n');
    title('Midpoint-Harmonic Mean Newton''s Method for System of Three Functions');

    maxx = max(length(errors));
    xticks(unique(round(logspace(0, log10(maxx), 10))));
    xlim([1 maxx]);
    grid on;

end
```

CN method code for Lotka–Volterra system

```
function [sequence, errors, ratios, zero, niter, res, order, ek, NFE] =
    newton_systemm_lv(fun, Jfun, X0, x_old, y_old, dt, a, b, c, d, tol, nmax)
% Newton's method
%
% INPUTS:
% fun   - function handle for F(x, y, x_old, y_old, dt, a, b, c, d)
% Jfun  - Jacobian handle for J(x, y, x_old, y_old, dt, a, b, c, d)
% X0    - initial guess [x; y]
% x_old, y_old - values from previous time step
% dt    - time step size
% a, b, c, d -  L o t k a Volterra  parameters
% tol   - tolerance for Newton convergence
% nmax  - maximum number of iterations

X = X0;
iter = 0;
err = inf;

errors = [];
```

```
ratios = [];
sequence = X;

while err > tol && iter < nmax
    % Evaluate function and Jacobian with current guess
    F = fun(X(1), X(2), x_old, y_old, dt, a, b, c, d);
    J = Jfun(X(1), X(2), x_old, y_old, dt, a, b, c, d);


    delta = -J \ F;
    X_new = X + delta;

    err = norm(X_new - X, 2);
    errors = [errors, err];

    if iter > 0
        ratios = [ratios, err / errors(end-1)];
    end

    sequence = [sequence, X_new];
    X = X_new;
    iter = iter + 1;
end

% Estimate convergence order if possible
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end
fprintf('Estimated order of convergence: %.4f\n', order);

zero = X
niter = iter
NFE = niter * 2 % Each iteration uses 1 F and 1 J
res = norm(fun(zero(1), zero(2), x_old, y_old, dt, a, b, c, d))
ek = err
```

AN method code for Lotka–Volterra system

```
function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    arithmetic_systemm_lv(fun, Jfun, X0, x_old, y_old, dt, a, b, c, d, tol, nmax)
% Arithmetic Mean Newton's Method
%
% INPUTS and OUTPUTS: same as newton_systemm_lv

X = X0;
iter = 0;
err = inf;

errors = [];
ratios = [];
sequence = X;

while err > tol && iter < nmax
    % Evaluate function and Jacobian with current guess
    F = fun(X(1), X(2), x_old, y_old, dt, a, b, c, d);
    J = Jfun(X(1), X(2), x_old, y_old, dt, a, b, c, d);

    delta_star = -J \ F;
    X_star = X + delta_star;

    J_star = Jfun(X_star(1), X_star(2), x_old, y_old, dt, a, b, c, d);
    J_mean = 0.5 * (J + J_star);

    delta = -J_mean \ F;
    X_new = X + delta;

```

```
    % Compute error
    err = norm(X_new - X, 2);
    errors = [errors, err];

    if iter > 0
        ratios = [ratios, err / errors(end-1)];
    end

    sequence = [sequence, X_new];
    X = X_new;
    iter = iter + 1;
end

% Estimate convergence order
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end
fprintf('Estimated order of convergence: %.4f\n', order);

zero = X
niter = iter
NFE = niter * 3 % 1 F, 2 J evaluations per iteration
res = norm(fun(zero(1), zero(2), x_old, y_old, dt, a, b, c, d))
ek = err
```

MN method code for Lotka–Volterra system

```

function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    midpoint_systemm_lv(fun, Jfun, X0, x_old, y_old, dt, a, b, c, d, tol, nmax)
% Midpoint Newton's Method
%
% INPUTS and OUTPUTS: same as newton_systemm_lv

X = X0;
iter = 0;
err = inf;

errors = [];
ratios = [];
sequence = X;

while err > tol && iter < nmax
    % Evaluate function and Jacobian with current guess
    F = fun(X(1), X(2), x_old, y_old, dt, a, b, c, d);
    J = Jfun(X(1), X(2), x_old, y_old, dt, a, b, c, d);

    delta_star = -J \ F;
    X_star = X + delta_star;

    X_mid = (X + X_star) / 2;

    J_mid = Jfun(X_mid(1), X_mid(2), x_old, y_old, dt, a, b, c, d);

    delta = -J_mid \ F;
    X_new = X + delta;

    % Compute error
    err = norm(X_new - X, 2);
    errors = [errors, err];

    if iter > 0
        ratios = [ratios, err / errors(end-1)];
    end

    sequence = [sequence, X_new];
```

```
    X = X_new;
    iter = iter + 1;
end

% Final outputs
zero = X
niter = iter
NFE = niter * 3 % F, J, and J_mid per iteration
res = norm(fun(zero(1), zero(2), x_old, y_old, dt, a, b, c, d))
ek = err

% Order estimation
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end
fprintf('Estimated order of convergence: %.4f\n', order);
```

HN method code for Lotka–Volterra system

```


function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    harmonic_systemm_lv(fun, Jfun, X0, x_old, y_old, dt, a, b, c, d, tol, nmax)
% Harmonic Mean Newton's Method
%
% INPUTS:
% fun   - function handle for F(x, y, x_old, y_old, dt, a, b, c, d)
% Jfun  - Jacobian handle for J(x, y, x_old, y_old, dt, a, b, c, d)
% X0    - initial guess [x; y]
% x_old, y_old - values from previous time step
% dt    - time step size
% a, b, c, d - L o t k a Volterra parameters
% tol   - tolerance for Newton convergence
% nmax  - maximum number of iterations

X = X0;
iter = 0;
err = inf;

errors = [];
ratios = [];
sequence = X;

while err > tol && iter < nmax
   % Evaluate function and Jacobian with current guess
    F = fun(X(1), X(2), x_old, y_old, dt, a, b, c, d);
    J = Jfun(X(1), X(2), x_old, y_old, dt, a, b, c, d);

    delta_star = -J \ F;
    X_star = X + delta_star;

    J_star = Jfun(X_star(1), X_star(2), x_old, y_old, dt, a, b, c, d);

    J_mean_inv = 0.5 * (inv(J) + inv(J_star));
    delta = -J_mean_inv * F;
    X_new = X + delta;

    % Compute step-size error
    err = norm(X_new - X, 2);
    errors = [errors, err];

    if iter > 0
        ratios = [ratios, err / errors(end-1)];
    end

    sequence = [sequence, X_new];
```

```
    X = X_new;
    iter = iter + 1;
end

% Final results
zero = X
niter = iter
NFE = niter * 3  % F,J,J_star per iteration
res = norm(fun(zero(1), zero(2), x_old, y_old, dt, a, b, c, d))
ek = err

% Estimate convergence order
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end
fprintf('Estimated order of convergence: %.4f\n', order);
```

MHN method code for Lotka–Volterra system

```
function [sequence, errors, ratios, zero, niter, order, res, ek, NFE] =
    midpoint_harmonic_systemm_lv(fun, Jfun, X0, x_old, y_old, dt, a, b, c, d, tol, nmax
    )
% Midpoint-Harmonic Mean Newton's Method
%
% INPUTS:
% fun   - function handle for F(x, y, x_old, y_old, dt, a, b, c, d)
% Jfun  - Jacobian handle for J(x, y, x_old, y_old, dt, a, b, c, d)
% X0    - initial guess [x; y]
% x_old, y_old - values from previous time step
% dt    - time step size
% a, b, c, d -  L o t k a Volterra  parameters
% tol   - tolerance for Newton convergence
% nmax  - maximum number of iterations

X = X0;
iter = 0;
err = inf;

errors = [];
ratios = [];
sequence = X;

while err > tol && iter < nmax
    % Evaluate function and Jacobian at current guess
    F = fun(X(1), X(2), x_old, y_old, dt, a, b, c, d);
    J = Jfun(X(1), X(2), x_old, y_old, dt, a, b, c, d);


    delta_star = -J \ F;
    X_star = X + delta_star;


    X_mid = 0.5 * (X + X_star);
    J_mid = Jfun(X_mid(1), X_mid(2), x_old, y_old, dt, a, b, c, d);


    J_mod_inv = 0.5 * (inv(J) + inv(2 * J_mid - J));
    delta = -J_mod_inv * F;
    X_new = X + delta;

    % Step-size error
    err = norm(X_new - X, 2);
    errors = [errors, err];

    if iter > 0
        ratios = [ratios, err / errors(end-1)];
```

```
    end

    sequence = [sequence, X_new];
    X = X_new;
    iter = iter + 1;
end

% Final results
zero = X
niter = iter
NFE = niter * 3  % F,J,J_mid per iteration
res = norm(fun(zero(1), zero(2), x_old, y_old, dt, a, b, c, d))
ek = err

% Estimate convergence order
if length(ratios) >= 2
    order = log(ratios(end)) / log(ratios(end-1));
else
    order = NaN;
end
fprintf('Estimated order of convergence: %.4f\n', order);
```

Code for solving the Lotka–Volterra system using CN, AN, MN, HN, and MHN methods

```
% Define the  L o t k a Volterra  function system (backward Euler)
fun = @(x, y, x_old, y_old, dt, a, b, c, d) [
    x - dt * (a * x - b * x * y) - x_old;
    y - dt * (-c * y + d * x * y) - y_old
];

% Define the Jacobian of the system
Jfun = @(x, y, x_old, y_old, dt, a, b, c, d) [
    1 - dt * (a - b * y),  dt * b * x;
    -dt * d * y,           1 - dt * (-c + d * x)
];

% Parameters
a = 2/3; b = 4/3; c = 2/3; d = 1;
x_old =2; y_old =1;
X0 = [x_old; y_old];
dt = 0.001;
tol = 1e-12;
nmax = 1000;

% Call the Newton solver
[seq, errors, ratios, zero, niter, res, order, ek, NFE] =newton_systemm_lv(fun, Jfun,
    X0, x_old, y_old, dt, a, b, c, d, tol, nmax);
```

Code for time series and phase portrait of Lotka–Volterra system using CN, AN, MN, HN, and MHN methods

```

% Lotka-Volterra Parameters
a = 2/3; b = 4/3; c = 2/3; d = 1;

% Time setup
T = 55;                % Final time
dt = 0.001;             % Time step size
N = T / dt;            % Number of steps
t = linspace(0, T, N+1);

% Initial conditions
x0 = 2;
y0 = 1;
x = zeros(1, N+1);
y = zeros(1, N+1);
x(1) = x0;
```

```
y(1) = y0;

% Tolerance and max iterations
tol = 1e-12;
nmax = 1000;

% Define the  L o t k a Volterra  function system (backward Euler)
fun = @(x, y, x_old, y_old, dt, a, b, c, d) [
    x - dt * (a * x - b * x * y) - x_old;
    y - dt * (-c * y + d * x * y) - y_old
];

% Define the Jacobian of the system
Jfun = @(x, y, x_old, y_old, dt, a, b, c, d) [
    1 - dt * (a - b * y),  dt * b * x;
    -dt * d * y,           1 - dt * (-c + d * x)
];
tic;
% Time-stepping using CN (implicit Euler with Newton)
for n = 1:N
    X0 = [x(n); y(n)];
    [~, ~, ~, sol, ~, ~, ~, ~, ~] =newton_systemm_lv(fun, Jfun, X0, x(n), y(n), dt, a,
        b, c, d, tol, nmax);
    x(n+1) = sol(1);
    y(n+1) = sol(2);
end
cputime_cn = toc;
fprintf('Total CPU time (CN method) for interval [0, %.1f]: %.6f seconds\n', T,
    cputime_cn);

% Time series plot: x(t) and y(t)
figure;
plot(t, x, 'b', 'LineWidth', 1.5); hold on;
plot(t, y, 'r--', 'LineWidth', 1.5);
legend('$x(t)$', '$y(t)$', 'Interpreter', 'latex', 'Location', 'north', 'FontSize', 14)
xlabel('$t$', 'Interpreter', 'latex', 'FontSize', 16)
ylabel('Population', 'Interpreter', 'latex', 'FontSize', 16)
title('P r e d a t o r Prey  Population Dynamics Over Time (CN Method)', 'Interpreter', '
    latex', 'FontSize', 11)
xlim([0 50])             % Limit x-axis from 0 to 50
xticks(0:10:50)  % Set ticks at every 10 units
ylim([0 3])
yticks(0:1:2)
grid on


cutoff_index = find(t >=11.4, 1);  % stop at t = 11.4
x_plot = x(1:cutoff_index);
y_plot = y(1:cutoff_index);

% Phase portrait plot (x(t) vs. y(t))
figure;
plot(x_plot, y_plot, '-ob', 'LineWidth', 1, 'MarkerIndices', 1, ...
     'MarkerFaceColor', 'r', 'MarkerSize', 6);
text(x_plot(1), y_plot(1), 't = 0', 'FontSize', 10, ...
     'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right');

xlabel('$x(t)$', 'Interpreter', 'latex', 'FontSize', 14)
ylabel('$y(t)$', 'Interpreter', 'latex', 'FontSize', 14)
title('Phase Portrait: Predator vs Prey (CN Method)', 'FontSize', 14)
xlim([0 3])
ylim([0 2])
yticks(0:1:2)
axis square
```

Phase-Space plot for Lotka–Volterra system using CN, AN, MN, HN, and MHN methods with various initial conditions

```
1
2  % Lotka-Volterra Parameters
3  a = 2/3; b = 4/3; c = 2/3; d = 1;
4
5  % Time setup
6  T = 16;                % Final time
7  dt = 0.001;            % Time step size
8  N = T / dt;            % Number of steps
9  t = linspace(0, T, N+1);
10
11 % Tolerance and max iterations
12 tol = 1e-12;
13 nmax = 1000;
14
15 % Define the  L o t k a Volterra  function system (backward Euler)
16 fun = @(x, y, x_old, y_old, dt, a, b, c, d) [
17     x - dt * (a * x - b * x * y) - x_old;
18     y - dt * (-c * y + d * x * y) - y_old
19 ];
20
21 % Define the Jacobian of the system
22 Jfun = @(x, y, x_old, y_old, dt, a, b, c, d) [
23     1 - dt * (a - b * y),  dt * b * x;
24     -dt * d * y,           1 - dt * (-c + d * x)
25 ];
26
27 % Define different initial conditions (prey, predator)
28 initial_conditions = [2 1; 2 0.5; 2 1.5; 2 2];
29 colors = lines(size(initial_conditions, 1));
30
31 figure; hold on;
32
33 for i = 1:size(initial_conditions, 1)
34     % Initial conditions
35     x0 = initial_conditions(i, 1);
36     y0 = initial_conditions(i, 2);
37     x = zeros(1, N+1);
38     y = zeros(1, N+1);
39     x(1) = x0;
40     y(1) = y0;
41
42     % Time-stepping using CN method
43     for n = 1:N
44         X0 = [x(n); y(n)];
45         [~, ~, ~, sol, ~, ~, ~, ~, ~] = newton_systemm_lv(fun, Jfun, X0, x(n), y(n), dt
               , a, b, c, d, tol, nmax);
46         x(n+1) = sol(1);
47         y(n+1) = sol(2);
48     end
49
50
51     % Plot trajectory
52     plot(x, y, 'Color', colors(i,:), 'LineWidth', 1.5);
53
54     % Mark and label initial point (t = 0)
55     plot(x(1), y(1), 'o', 'MarkerSize', 6, ...
56          'MarkerFaceColor', colors(i,:), 'MarkerEdgeColor', 'k');
57     text(x(1), y(1), 't = 0', 'FontSize', 10, ...
58          'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right', ...
59          'Color', colors(i,:));
60 end
61
62 xlabel('$x(t)$', 'Interpreter', 'latex', 'FontSize', 14)
63 ylabel('$y(t)$', 'Interpreter', 'latex', 'FontSize', 14)
64 title({'Phase-space plot for various initial conditions of the predator population',
```

```
    ...
65        '(CN Method)'}, 'FontSize', 10)
66 xlim([0 4])
67 ylim([0 3])
68 yticks(0:1:3)
69 xticks(0:1:4)
70
71 axis square
72 grid on
73 legend({'$y_0 = 1$', '', '$y_0 = 0.5$', '', '$y_0 = 1.5$', '', '$y_0 = 2$', ''}, ...
74        'Interpreter', 'latex', 'Location', 'northeast');
```