Enhanced Newton's Methods for Efficiently Stimulating Biomedical Problems

Preprint · April 2025

DOI: 10.13140/RG.2.2.10599.97448

1 author:

Malik Mohammed
Khalifa University
1 PUBLICATION **0** CITATIONS

SEE PROFILE

جامعة خليفة
Khalifa University

MATH498 – Senior Research Project

Enhanced Newton's Methods for Efficiently Stimulating Biomedical Problems

Prepared by: Malik Mohammed

Student ID: 100059981

Supervised by: Dr. Aymen Laadhari

Date of Submission: April 24, 2025

Contents

Introduction

Root-finding problems constitute a fundamental aspect of numerical analysis and computational mathematics, serving as essential tools for solving equations of the form $f(x) = 0$. These problems arise in various fields, including engineering, physics, and economics, where determining the values that satisfy linear and nonlinear equations is critical for modeling and analysis. Effective techniques for finding roots are necessary for solving both theoretical and practical problems, as many scientific computations rely on the accurate determination of these values.

Among the various methods developed for root-finding, the fixed-point method is one of the simplest and most intuitive approaches. This technique involves rearranging the equation $f(x) = 0$ into the form $g(x) = x$, where $g(x)$ is a function that ideally converges to the root of the original equation. By iteratively substituting estimates into the function $g(x)$, the method generates a sequence of approximations that converge to the root, provided certain conditions are met. While the fixed-point method is conceptually straightforward and easy to implement, it has several limitations that can hinder its effectiveness. One significant drawback is its dependence on the choice of the function $g(x)$ and the initial guess, if these are not chosen wisely, the method can fail to converge or may converge very slowly. Additionally, the fixed-point method may struggle with functions that exhibit discontinuities or have roots that are not well-defined within the iteration process, limiting its applicability in more complex scenarios.

An alternative to the fixed-point method, and one of the most powerful techniques for root-finding, is Newton's method, also known as the Newton-Raphson method [1]. This iterative algorithm is renowned for its rapid convergence and efficiency, often yielding better accuracy than the fixed-point method, particularly when the initial guess is close to the actual root. Named after Sir Isaac Newton and Joseph Raphson, who contributed to its formulation in the 17th century, Newton's method remains a cornerstone of numerical analysis due to its effectiveness in solving root-finding problems across a wide range of applications. However, it is essential to consider its limitations, such as reliance on the function's derivative and sensitivity to the choice of the initial guess, which can impact its performance in certain scenarios.

In numerical analysis, Newton's method extends beyond basic root-finding problems and finds applications in various fields, including optimization, differential equations, and scientific computing. Its rapid convergence, combined with its ability to handle non-linear systems, makes it an indispensable tool across diverse applications. For example, its application to a specific biomedical problem shown in [2]. High-order Newton methods have been developed and applied to solve nonlinear partial differential equations in different contexts, although their use has mainly focused on systems of ordinary differential equations. In particular, the high-order methods developed in [3, 4, 5]. Some examples include 1D reaction-diffusion models [6, 7, 8], population dynamics via the Fisher equation [9], and nonlinear heat conduction [10]. Some of the high-order methods were also applied to solve fluid and multiphase flow problems [11, 12, 13, 14], as well as fluid-membrane interactions [15].

This project explores specific Newton variants for solving root-finding problems and nonlinear ordinary differentials equations used in some specific biomedical applications. This report is structured as follows, Chapter 1 lays the groundwork by introducing the general setting of root-finding problems, and the theoretical framework of the Newton's method and a couple of its modifications. Chapter 2 then presents a comparative numerical study of these schemes. Chapter 3 generalizes the methods to systems of nonlinear equations and formulates the generalized framework. Lastly, Chapter 4 applies the framework to initial-value problems in differential equations.

Chapter 1

Root-finding Problems: Setting and Methods

In this chapter, the general framework and setting of 1-dimensional root-finding problems are established, alongside an in-depth exploration of the Newton's method and three of its modifications, examining how they are derived thoroughly, as well as their theoretical convergence rates.

1.1 Generalities

In many cases, a function may have a root of order m, where the root of order m is defined as a point x^* such that

$$\begin{aligned} f(x^*) &= 0 \\ f'(x^*) &= 0 \\ &\vdots \\ f^{(m-1)}(x^*) &= 0 \\ f^{(m)}(x^*) &\neq 0 \end{aligned}$$

In this report, the focus will be on simple roots, which corresponds to the case when $m = 1$. To maintain consistency, the notation x^* will be used throughout to denote the simple root of a function $f(x)$.

Having established the concept of simple roots, some general conditions need to be established under which the Newton's method and its modifications effectively converge to such roots. First, the function $f(x)$ must belong to the class C^k, meaning it must be continuously differentiable up to some order k. This ensures the existence and continuity of the function and its derivatives, which are critical for Newton's iterative scheme. Additionally, convergence depends on the proximity of the initial guess x_0 to the actual root x^*. Specifically, there must exist some constant $\delta > 0$ such that $x_0 \in (x^* - \delta, x^* + \delta)$ ensuring that the method converges for sufficiently close initial guesses.

Furthermore, having established the conditions under which the Newton's method converges, it is crucial to define the rate of convergence of an iterative scheme. Newton's method is said to converge to the root x^* with order α if there exist positive constants α and λ such that

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^\alpha} = \lambda,$$

where e_n denotes the error at the n^{th} iteration, defined as

$$e_n = x_n - x^*.$$

Consequently, the error at the $(n+1)^{th}$ iteration can be expressed as

$$\begin{aligned} e_{n+1} &= x_{n+1} - x^* \\ &= F(x_n) - x^* \\ &= E(x_n), \end{aligned}$$

where $F(x_n)$ represents the iterative scheme and $E(x_n)$ is introduced as a convenient notation.

In the analysis of the order of convergence of an iterative scheme, the Taylor expansion is a critical tool. The n^{th} order Taylor expansion of a function $f(x)$ around a point x_0 can be expressed as

$$f(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i + O((x - x_0)^{n+1}),$$

where $f^{(i)}(x_0)$ denotes the i^{th} derivative of the function evaluated at the point x_0, and the big O term $O((x-x_0)^{n+1})$ accounts for the higher-order remainder, capturing the behavior of the function as x approaches x_0.

1.2 Newton's Method

The scheme is derived by performing a 1^{st} order Taylor expansion of $f(x_{n+1})$ around the point x_n, which results in the following

$$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n) + O((x_{n+1} - x_n)^2).$$

Assuming that $x_{n+1} \to x^*$ as $n \to \infty$, taking the limit as $n \to \infty$ results in the following

$$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0.$$

Performing some algebraic operations to isolate x_{n+1} results in the following

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{1.1}$$

which is the Newton's method scheme [1].

Theorem:
Let $f \in C^2[a, b]$, then the Newton's method scheme (1.1) converges quadratically.

Proof:
Consider a 2^{nd} order Taylor expansion of e_{n+1} around the point x^*

$$e_{n+1} = E(x^*) + E'(x^*)e_n + \frac{E''(x^*)}{2}e_n^2 + O(e_n^3).$$

Given that

$$E(x_n) = x_n - \frac{f(x_n)}{f'(x_n)} - x^*,$$

this implies that

$$E(x^*) = 0.$$

Consequently,

$$E'(x_n) = \frac{f(x_n)f''(x_n)}{f'(x_n)^2},$$

which implies that

$$E'(x^*) = 0.$$

Therefore,

$$e_{n+1} = \frac{E''(x^*)}{2}e_n^2 + O(e_n^3).$$

Dividing both sides by e_n^2 results in the following

$$\frac{e_{n+1}}{e_n^2} = \frac{E''(x^*)}{2} + O(e_n^3).$$

Lastly, as $n \to \infty$, the following is obtained

$$\begin{aligned}\lim_{n\to\infty} \frac{e_{n+1}}{e_n^2} &= \frac{E''(x^*)}{2} + \lim_{n\to\infty} O(e_n^3) \\ &= \frac{E''(x^*)}{2}.\end{aligned}$$

Therefore, the Newton's method scheme (1.1) converges quadratically.

1.3 Newton's Method: Kou Modification

A modification to the classical Newton's method scheme (1.1) has been proposed by J. Kou in [3], which supposedly convergences cubically. The modified scheme is derived by a different approach, which is the computation of the definite integral

$$f(x_{n+1}) = f(y_n) + \int_{y_n}^{x_{n+1}} f'(t)\ dt, \tag{1.2}$$

where

$$y_n = x_n + \frac{f(x_n)}{f'(x_n)}. \tag{1.3}$$

Using the midpoint formula to evaluate the integral in (1.2) results in the following

$$\begin{aligned} f(x_{n+1}) &= f(y_n) + (x_{n+1} - y_n) f'\left(\frac{x_{n+1} + y_n}{2}\right) \\ &= f(y_n) + x_{n+1} f'\left(\frac{x_{n+1} + y_n}{2}\right) - y_n f'\left(\frac{x_{n+1} + y_n}{2}\right). \end{aligned}$$

Assuming that $x_{n+1} \to x^*$ as $n \to \infty$, taking the limit as $n \to \infty$ results in the following

$$f(x_{n+1}) = f(y_n) + x_{n+1} f'\left(\frac{x_{n+1} + y_n}{2}\right) - y_n f'\left(\frac{x_{n+1} + y_n}{2}\right) = 0.$$

Performing some algebraic operations to isolate x_{n+1} results in the following

$$x_{n+1} = y_n - \frac{f(y_n)}{f'\left(\frac{x_{n+1} + y_n}{2}\right)}. \tag{1.4}$$

Lastly, substituting (1.1) into (1.4) results in the following

$$x_{n+1} = y_n - \frac{f(y_n)}{f'(x_n)}, \tag{1.5}$$

which is the modified Newton's method scheme proposed by J. Kou.

Theorem:
Let $f \in C^3[a,b]$, then the modified Newton's method scheme (1.5) converges cubically.

Proof:
Consider a 3^{rd} order Taylor expansion of $f(x)$ around the point x_n evaluated at the point x^*

$$\begin{aligned} f(x^*) &= f(x_n) - f'(x_n)e_n + \frac{f''(x_n)e_n^2}{2} - \frac{f^{(3)}(x_n)e_n^3}{3!} + O(e_n^4) \\ &= 0. \end{aligned}$$

This implies that

$$f(x_n) = f'(x_n)e_n - \frac{f''(x_n)e_n^2}{2} + \frac{f^{(3)}(x_n)e_n^3}{3!} + O(e_n^4). \tag{1.6}$$

Let

$$\begin{aligned} d_n &= y_n - x_n \\ &= \frac{f(x_n)}{f'(x_n)}, \end{aligned} \tag{1.7}$$

substituting (1.6) into (1.7) results in the following

$$d_n = e_n - \frac{f''(x_n)}{2f'(x_n)}e_n^2 + \frac{f^{(3)}(x_n)}{3!f'(x_n)}e_n^3 + O(e_n^4). \tag{1.8}$$

Consequently,

$$\begin{aligned} d_n^2 &= e_n^2 - \frac{f''(x_n)}{f'(x_n)}e_n^3 + O(e_n^4) \\ d_n^3 &= e_n^3 + O(e_n^4). \end{aligned}$$

Now consider a 3^{rd} order Taylor expansion around the same point x_n but of $f(y_n)$ instead

$$f(y_n) = f(x_n) + f'(x_n)d_n + \frac{f''(x_n)}{2}d_n^2 + \frac{f^{(3)}(x_n)}{3!}d_n^3 + O(d_n^4),$$

this implies that

$$f(y_n) - f(x_n) = f'(x_n)d_n + \frac{f''(x_n)}{2}d_n^2 + \frac{f^{(3)}(x_n)}{3!}d_n^3 + O(d_n^4). \tag{1.9}$$

Substituting (1.8) into (1.9) results in the following

$$f(y_n) - f(x_n) = f'(x_n)e_n - \frac{f''(x_n)^2}{2f'(x_n)}e_n^3 + \frac{f^{(3)}(x_n)}{3}e_n^3 + O(e_n^4). \tag{1.10}$$

Substituting (1.3) into (1.5) results in the following

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

Therefore, the error at the $(n+1)^{th}$ iteration can be expressed as

$$\begin{aligned} e_{n+1} &= e_n - \frac{f(y_n) - f(x_n)}{f'(x_n)}. \\ &= \frac{f'(x_n)e_n - (f(y_n) - f(x_n))}{f'(x_n)}. \end{aligned} \tag{1.11}$$

Substituting (1.10) into (1.11) results in the following

$$e_{n+1} = e_n^3 \left(\frac{f''(x_n)^2}{2f'(x_n)^2} + \frac{f^{(3)}(x_n)}{3f'(x_n)} \right) + O(e_n^4).$$

Dividing both sides by e_n^3 results in the following

$$\frac{e_{n+1}}{e_n^3} = \frac{f''(x_n)^2}{2f'(x_n)^2} + \frac{f^{(3)}(x_n)}{3f'(x_n)} + O(e_n^4).$$

Lastly, as $n \to \infty$, the following is obtained

$$\begin{aligned}\lim_{n\to\infty} \frac{e_{n+1}}{e_n^3} &= \lim_{n\to\infty} \left(\frac{f''(x_n)^2}{2f'(x_n)^2} + \frac{f^{(3)}(x_n)}{3f'(x_n)} + O(e_n^4) \right) \\ &= \frac{f''(x^*)^2}{2f'(x^*)^2} + \frac{f^{(3)}(x^*)}{3f'(x^*)}.\end{aligned}$$

Therefore, the modified Newton's method scheme (1.5) converges cubically.

1.4 Newton's Method: Homeier Modification

Another modification to the Newton's method scheme was proposed by H. Homeier in [4], which supposedly converges cubically as well. The iterative function is of the type

$$F(x) = x - \frac{f(x)}{f'(x + a(x)f(x))},$$

for some sufficiently smooth arbitrary function $a(x)$. Many choices of $a(x)$ could ensure cubic convergence, however, the choice of interest is

$$a(x) = -\frac{1}{2f'(x)}.$$

Therefore, the modified Newton's scheme proposed by H. Homeier is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(y_n)}, \tag{1.12}$$

where

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

Theorem:
Let $f \in C^3[a,b]$, if there exists some constant M such that $|F^{(3)}(x)| \leq M$, then the modified Newton's method scheme (1.12) converges cubically.

Proof:
Consider a 3^{rd} order Taylor expansion of e_{n+1} around the point x^*

$$e_{n+1} = E(x^*) + E'(x^*)e_n + \frac{E''(x^*)}{2}e_n^2 + \frac{E^{(3)}(x^*)}{3!}e_n^3 + O(e_n^4).$$

Given that

$$E(x_n) = x_n - \frac{f(x_n)}{f'(y_n)} - x^*,$$

this implies that

$$E(x^*) = 0.$$

Consequently,

$$E'(x_n) = 1 - \frac{f'(x_n)}{f'(y_n)} + \frac{f(x_n)f''(y_n)\left(1 + \frac{f(x_n)f''(x_n)}{f'(x_n)^2}\right)}{2f'(y_n)^2},$$

which implies that

$$E'(x^*) = 0,$$

and

$$\begin{aligned} E''(x_n) = &- \frac{f''(x_n)}{f'(y_n)} \\ &+ \frac{f'(x_n)f''(y_n)\left(1 + \frac{f(x_n)f''(x_n)}{f'(x_n)^2}\right)}{f'(y_n)^2} \\ &- \frac{f(x_n)f''(y_n)^2\left(1 + \frac{f(x_n)f''(x_n)}{f'(x_n)^2}\right)^2}{f'(y_n)^3} \\ &+ \frac{f(x_n)f^{(3)}(y_n)\left(1 + \frac{f(x_n)f''(x_n)}{f'(x_n)^2}\right)^2}{4f'(y_n)^2} \\ &+ \frac{f(x_n)^2f''(y_n)\left(\frac{f^{(3)}(x_n)f'(x_n)^2 - 2f'(x_n)f''(x_n)^2}{2f'(x_n)^4}\right)}{f'(y_n)^2} \\ &+ \frac{f(x_n)f''(y_n)\left(\frac{2f(x_n)f''(x_n)}{f'(x_n)^2} - \frac{f''(x_n)}{f'(x_n)}\right)}{2f'(y_n)^2}, \end{aligned}$$

which implies that

$$E''(x^*) = 0.$$

Therefore,

$$e_{n+1} = \frac{E^{(3)}(x^*)}{3!}e_n^3 + O(e_n^4).$$

Dividing both sides by e_n^3 results in the following

$$\frac{e_{n+1}}{e_n^3} = \frac{E^{(3)}(x^*)}{3!} + O(e_n^4).$$

Lastly, as $n \to \infty$, the following is obtained

$$\begin{aligned}\lim_{n\to\infty} \frac{e_{n+1}}{e_n^3} &= \lim_{n\to\infty} \left(\frac{E^{(3)}(x^*)}{3!} + O(e_n^4) \right) \\ &= \frac{E^{(3)}(x^*)}{3!}.\end{aligned}$$

The M condition ensures that the asymptotic error λ converges to some value rather than it diverging. Therefore, the modified Newton's method scheme (1.12) converges cubically.

1.5 Newton's Method: Weerakoon Modification

Another modification to the Newton's method scheme was proposed by S. Weerakoon in [5], which supposedly converges cubically as well. Similar to Kou's modification, the scheme is derived by the computation of the definite integral

$$f(x_{n+1}) = f(x_n) + \int_{x_n}^{x_{n+1}} f'(t)\,dt. \tag{1.13}$$

However, the trapezoidal rule is used to evaluate the integral in (1.13) instead of the midpoint formula, which results in the following

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

Assuming that $x_{n+1} \to x^*$ as $n \to \infty$, taking the limit as $n \to \infty$ results in the following

$$f(x_{n+1}) = f(x_n) + \frac{x_{n+1} - x_n}{2}(f'(x_n) + f'(x_{n+1})) = 0.$$

Performing some algebraic operations to isolate x_{n+1} results in the following

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) + f'(x_{n+1})}, \tag{1.14}$$

which is an implicit scheme as it requires $f'(x_{n+1})$ to compute x_{n+1}. Let

$$y_n = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{1.15}$$

which is defined as x_{n+1} in the classical Newton's scheme (1.1). Lastly, substituting (1.15) into (1.14) results in the following

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) + f'(y_n)}, \tag{1.16}$$

which is the modified Newton's method scheme proposed by S. Weerakoon.

Theorem:
Let $f \in C^3[a,b]$, then the modified Newton's method scheme (1.16) converges cubically.

Proof:
Let $C_j = \frac{f^{(j)}(x^*)}{j!f'(x^*)}$. Consider a 3^{rd} order Taylor expansion of $f(x_n)$ around the point x^*

$$\begin{aligned} f(x_n) &= f(x^*) + f'(x^*)e_n + \frac{f''(x^*)}{2}e_n^2 + \frac{f^{(3)}(x^*)}{3!}e_n^3 + O(e_n^4) \\ &= f'(x^*)(e_n + C_2e_n^2 + C_3e_n^3 + O(e_n^4)). \end{aligned} \tag{1.17}$$

Consider a 2^{nd} order Taylor expansion of $f'(x_n)$ around the point x^*

$$\begin{aligned} f'(x_n) &= f'(x^*) + f''(x^*)e_n + \frac{f^{(3)}(x^*)}{2}e_n^2 + O(e_n^3) \\ &= f'(x^*)(1 + 2C_2e_n + 3C_3e_n^2 + O(e_n^3)). \end{aligned} \tag{1.18}$$

Using (1.17) and (1.18),

$$\frac{f(x_n)}{f'(x_n)} = \frac{e_n + C_2e_n^2 + C_3e_n^3 + O(e_n^4)}{1 + 2C_2e_n + 3C_3e_n^2 + O(e_n^3)}.$$

It is clear that $\frac{1}{f'(x_n)}$ is of the form $\frac{1}{1+x}$ with $x = 2C_2e_n + 3C_3e_n^2 + O(e_n^3)$. Therefore, a 2^{nd} order Maclaurin expansion results in the following

$$\begin{aligned} \frac{f(x_n)}{f'(x_n)} &= (e_n + C_2e_n^2 + C_3e_n^3 + O(e_n^4))(1 - (2C_2e_n + 3C_3e_n^2 + O(e_n^3)) + (2C_2e_n + 3C_3e_n^2 + O(e_n^3))^2 + O(e_n^3)) \\ &= e_n - C_2e_n^2 + (2C_2^2 - 2C_3)e_n^3 + O(e_n^4). \end{aligned} \tag{1.19}$$

Substituting (1.19) into (1.15) results in the following

$$y_n = x^* + C_2e_n^2 + (2C_3 - 2C_2^2)e_n^3 + O(e_n^4). \tag{1.20}$$

Using (1.20), consider a 1^{st} order Taylor expansion of $f'(y_n)$ around the point x^*

$$\begin{aligned} f'(y_n) &= f'(x^*) + f''(x^*)(C_2e_n^2 + (2C_3 - 2C_2^2)e_n^3 + O(e_n^4)) + O(e_n^4) \\ &= f'(x^*)(1 + 2C_2^2e_n^2 + 4C_2(C_3 - C_2^2)e_n^3 + O(e_n^4)). \end{aligned} \tag{1.21}$$

Consequently, (1.18) and (1.21) result in the following

$$f'(x_n) + f'(y_n) = 2f'(x^*)(1 + C_2e_n + (C_2^2 + \frac{3}{2}C_3)e_n^2 + O(e_n^3)). \tag{1.22}$$

Substituting (1.22) into (1.16) results in the following

$$x_{n+1} = x_n - \frac{e_n + C_2e_n^2 + C_3e_n^3 + O(e_n^4)}{1 + C_2e_n + (C_2^2 + \frac{3}{2}C_3)e_n^2 + O(e_n^3)}. \tag{1.23}$$

Similarly, a 2^{nd} order Maclaurin expansion of the denominator in (1.23) results in the following

$$\begin{aligned} x_{n+1} &= x_n - (e_n + C_2e_n^2 + C_3e_n^3 + O(e_n^4))(1 - (C_2e_n + (C_2^2 + \frac{3}{2}C_3)e_n^2 + O(e_n^3)) + (C_2e_n + (C_2^2 + \frac{3}{2}C_3)e_n^2 + O(e_n^3))^2 + O(e_n^3)) \\ &= x_n - e_n + (C_2^2 + \frac{1}{2}C_3)e_n^3 + O(e_n^4). \end{aligned}$$

Therefore,

$$e_{n+1} = (C_2^2 + \frac{1}{2}C_3)e_n^3 + O(e_n^4).$$

Dividing both sides by e_n^3 results in the following

$$\frac{e_{n+1}}{e_n^3} = C_2^2 + \frac{1}{2}C_3 + O(e_n^4).$$

Lastly, as $n \to \infty$, the following is obtained

$$\begin{aligned}\lim_{n\to\infty} \frac{e_{n+1}}{e_n^3} &= \lim_{n\to\infty} \left(C_2^2 + \frac{1}{2} C_3 + O(e_n^4) \right) \\ &= C_2^2 + \frac{1}{2} C_3.\end{aligned}$$

Therefore, the modified Newton's method scheme (1.16) converges cubically.

Chapter 2

Numerical Testing and Convergence

In this chapter, the various schemes introduced are tested on a set of nonlinear functions in order to evaluate their convergence rates and computational efficiency. Table 2.1 presents the results, where N2 denote the classical quadratic Newton's scheme, and K3, H3, and W3 denote the cubic modifications proposed by Kou, Homeier, and Weerakoon, respectively. In the table, n represents the number of iterations required to converge within a tolerance of $\epsilon = 1\times10^{-12}$, NFE represent the number of fucntion evaluations, x_n is the computed root, and $|f(x_n)|$ represents the residual error at the root.

Table 2.1: Performance comparison of the Newton's method and its modifications across various test functions.

	n	NFE	x_n	$\lvert f(x_n)\rvert$
$f_1(x) = x^3 + 4x^2 - 15, \quad x_0 = -0.9$				
N2	25	50	1.63198080556606672786	0.00000000000006750156
K3	4	12	1.63198080556606339719	0.00000000000000355271
H3	9	27	1.63198080556606339719	0.00000000000000355271
W3	6	18	1.63198080556606339719	0.00000000000000355271
$f_2(x) = x^2\sin(x) - \cos(x), \quad x_0 = 6$				
N2	4	8	6.30830895523815726733	0.00000000000023636648
K3	3	9	6.30830895523815105008	0.00000000000001321165
H3	3	9	6.30830895523815105008	0.00000000000001321165
W3	3	9	6.30830895523815105008	0.00000000000001321165
$f_3(x) = e^{-x}\sin(x) + \ln(x^2+1), \quad x_0 = 3$				
N2	6	12	0.00000000000000005763	0.00000000000000005763
K3	4	12	0.00000000000000000001	0.00000000000000000001
H3	4	12	0.00000000000000001429	0.00000000000000001429
W3	5	15	-0.00000000000000006587	0.00000000000000006587

To complement the tabular comparison, the convergence behavior of the iterative schemes is illustrated for each of the three test functions. Figures 2.1–2.3 display a plot of $\log|f_i(x_n)|$ against the iteration index n for each all i.

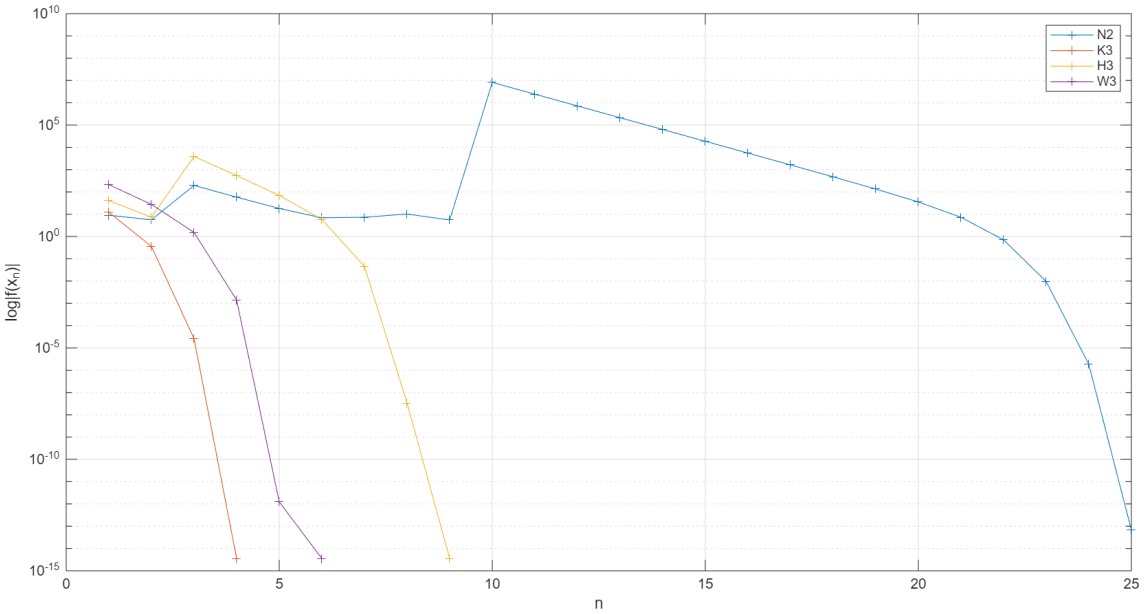

Figure 2.1: Error curves for the Newton's method and its modifications when applied to $f_1(x)$.

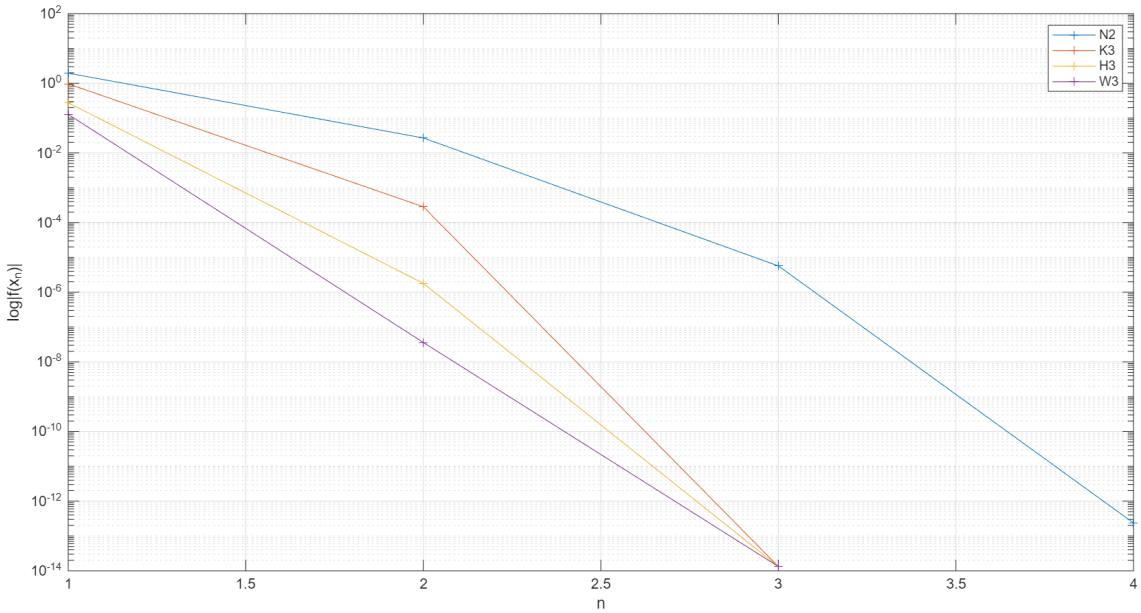

Figure 2.2: Error curves for the Newton's method and its modifications when applied to $f_2(x)$.

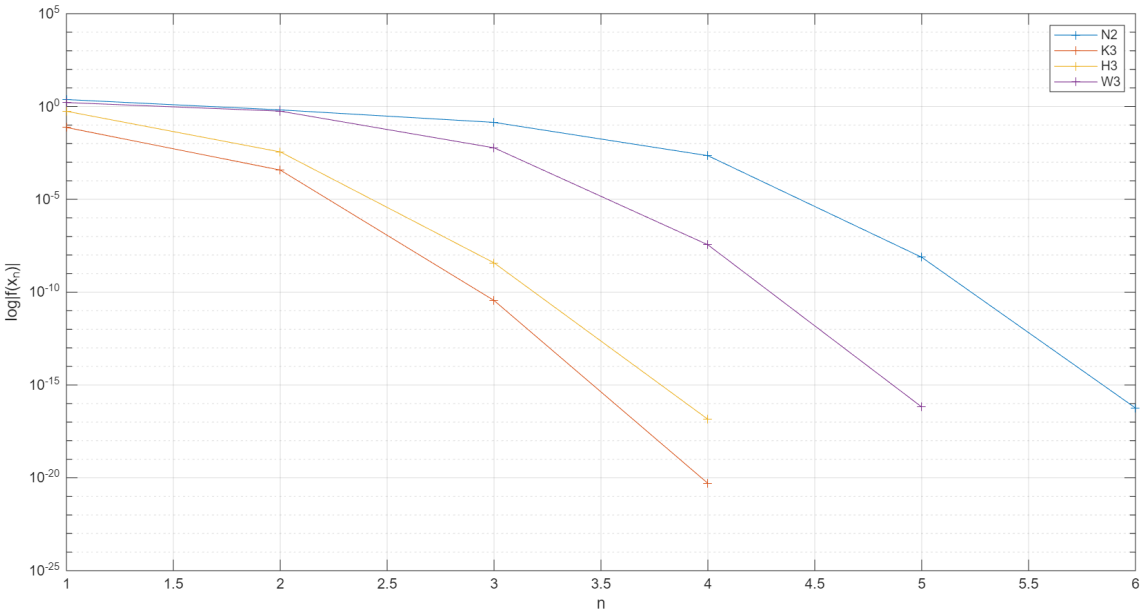

Figure 2.3: Error curves for the Newton's method and its modifications when applied to $f_3(x)$.

As observed in Table 2.1 and Figures 2.1–2.3, the Newton's method modifications converge more rapidly than the standard Newton method. The error curves for the modified schemes demonstrate a steeper decline, indicating that the desired tolerance is achieved in fewer iterations. Additionally, the residual errors remain consistently lower across most iterations, reflecting both faster convergence and improved accuracy.

As mentioned earlier, a numerical scheme is said to have a rate of convergence of order α if there exist positive constants α and λ such that

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^\alpha} = \lambda,$$

this definition can be equivalently expressed in terms of the previous error term as

$$\lim_{n\to\infty} \frac{|e_n|}{|e_{n-1}|^\alpha} = \lambda.$$

To determine the rate of convergence , the equations are manipulated as follows. Firstly, taking the limit of the ratio of these expressions for consecutive terms results in the following

$$\lim_{n\to\infty} \left| \frac{e_{n+1} e_{n-1}^\alpha}{e_n^{\alpha+1}} \right| = \frac{\lambda}{\lambda} = 1,$$

and by rearranging the terms the following is obtained

$$\lim_{n\to\infty} \left| \frac{e_{n+1}}{e_n} \right| = \lim_{n\to\infty} \left| \frac{e_n}{e_{n-1}} \right|^\alpha.$$

Taking the natural logarithm on both sides results in the following

$$\lim_{n\to\infty} \ln \left| \frac{e_{n+1}}{e_n} \right| = \lim_{n\to\infty} \alpha \cdot \ln \left| \frac{e_n}{e_{n-1}} \right|.$$

Lastly, some algebraic manipulation to isolate α results in the following

$$\alpha = \lim_{n\to\infty} \frac{\ln|e_{n+1}| - \ln|e_n|}{\ln|e_n| - \ln|e_{n-1}|}, \tag{2.1}$$

which is an expression that provides a practical means to approximate the rate of convergence of a scheme numerically.

To illustrate the practical use of (2.1), Tables 2.2–2.4 report the computed values of α for each iterative scheme applied to the test functions, as the iteration index n increases.

Consider the function $g_1(x) = \sin^2(x) - x^2 + 1$ and an initial guess $x_0 = 2$.

Table 2.2: Convergence rates of the Newton's method and its modifications when applied to $g_1(x)$

n	N2	K3	H3	W3
2	1.64537925317609468046	2.56079457487366468627	2.62546244421559160642	2.55613638981442958809
3	1.93264646500319581257	2.97016398131798853299	2.98765000675240477435	2.98700282430320251947
4	1.99639803533054149831	–	–	–
5	1.92192815536606520510	–	–	–

Consider the function $g_2(x) = \cos(x) - x$ and an initial guess $x_0 = 1.7$.

Table 2.3: Convergence rates the of Newton's method and its modifications when applied to $g_2(x)$

n	N2	K3	H3	W3
2	1.51224203202715923311	2.82318988190517750070	2.74168492283772469165	1.76500978241534234314
3	1.99052490015558380954	2.98122120661930267715	–	–
4	2.00577450638506826763	–	–	–

Consider the function $g_3(x) = xe^{x^2} - \sin^2(x) + 3\cos(x) + 5$ and an initial guess $x_0 = -2$.

Table 2.4: Convergence rates the of Newton's method and its modifications when applied to $g_3(x)$

n	N2	K3	H3	W3
2	1.51266462099767351468	2.52882975731273385023	2.24686992009604935561	1.97880229426542331161
3	1.68141431834645782573	3.27404666620059092708	2.83101320039386950000	2.51851007081365896312
4	1.85585568310294601879	3.05777635057063434942	2.99559208781607688721	2.91843698025694298082
5	1.96338828027112421992	–	–	3.00607222682107000367
6	1.99588022621549154856	–	–	–
7	1.99989185877951536341	–	–	–

As observed in Tables 2.2–2.4, for the Newton's method whose convergence rate is theoretically quadratic, α approaches 2 as the iterates converge to the root within the specified tolerance. Similarly, for the modified Newton's method schemes whose convergence rate is theoretically cubic, α approaches 3 as the root is approached. This demonstrates the effectiveness of (2.1) to numerically approximate the rate of convergence of an iterative scheme.

Chapter 3

Newton's Methods for Solving Nonlinear Systems

In this chapter, the discussion is extended to multivariate systems of nonlinear equations. It formulates the generalized framework for the n-dimensional Newton's method and its modifications, and validates each scheme through a set of examples.

3.1 Generalized Setting

In many practical applications, root-finding problems involve a system of nonlinear equations rather than a single equation. Such systems can be expressed in the general form

$$\boldsymbol{F}(\boldsymbol{X}) = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix},$$

where each function $f_i(\boldsymbol{X})$ is generally nonlinear, making the system significantly more challenging to solve compared to linear systems.

Unlike linear systems, which can be effectively solved using methods such as Gaussian elimination or LU decomposition, nonlinear systems require iterative approaches due to their complexity. A natural extension of Newton's method for a single equation provides an efficient numerical approach to solving such systems which generalizes the iterative scheme by incorporating the Jacobian matrix that is given by

$$J(\boldsymbol{X}) = \begin{pmatrix} \partial_{x_1} f_1(\boldsymbol{X}) & \cdots & \partial_{x_n} f_1(\boldsymbol{X}) \\ \vdots & \ddots & \vdots \\ \partial_{x_1} f_n(\boldsymbol{X}) & \cdots & \partial_{x_n} f_n(\boldsymbol{X}) \end{pmatrix}.$$

Similarly, there must exist some $\delta > 0$ such that $||\boldsymbol{X}_0 - \boldsymbol{X}^*|| < \delta$, meaning the initial guess $\boldsymbol{X}_0$ should be sufficiently close to the actual root $\boldsymbol{X}^*$ for the scheme to converge, and it is said to converge with order α if there exist positive constants α and λ such that

$$\lim_{n\to\infty} \frac{||\boldsymbol{e}_{n+1}||}{||\boldsymbol{e}_n||^\alpha} = \lambda.$$

Additionally, the rate of convergence could be approximated numerically using

$$\alpha = \lim_{n\to\infty} \frac{\ln||\boldsymbol{e}_{n+1}|| - \ln||\boldsymbol{e}_n||}{\ln||\boldsymbol{e}_n|| - \ln||\boldsymbol{e}_{n-1}||}.$$

3.2 Newton's Method Extension

In this section, the iterative schemes introduced for 1-dimensional root-finding problems will be extended to n-dimensional systems.

3.2.1 Newton's Method: Classical Scheme

The extension of the classical Newton's method (1.1) to multivariable systems is given by

$$\boldsymbol{X}_{n+1} = \boldsymbol{X}_n - J(\boldsymbol{X}_n)^{-1} \cdot \boldsymbol{F}(\boldsymbol{X}_n). \tag{3.1}$$

3.2.2 Newton's Method: Kou Modification

The extension of Kou's modification (1.5) to multivariable systems is given by

$$\boldsymbol{X}_{n+1} = \boldsymbol{Y}_n - J(\boldsymbol{X}_n)^{-1} \cdot \boldsymbol{F}(\boldsymbol{Y}_n), \tag{3.2}$$

where

$$\boldsymbol{Y}_n = \boldsymbol{X}_n + J(\boldsymbol{X}_n)^{-1} \cdot \boldsymbol{F}(\boldsymbol{X}_n).$$

3.2.3 Newton's Method: Homeier Modification

The extension of Homeier's modification (1.12) to multivariable systems is given by

$$\boldsymbol{X}_{n+1} = \boldsymbol{X}_n - J(\boldsymbol{Y}_n)^{-1} \cdot F(\boldsymbol{X}_n), \tag{3.3}$$

where

$$\boldsymbol{Y}_n = \boldsymbol{X}_n - \frac{1}{2} \cdot J(\boldsymbol{X}_n)^{-1} \cdot \boldsymbol{F}(\boldsymbol{X}_n).$$

3.2.4 Newton's Method: Weerakoon Modification

The extension of Weerakoon's modification (1.16) to multivariable systems is given by

$$\boldsymbol{X}_{n+1} = \boldsymbol{X}_n - 2 \cdot (J(\boldsymbol{X}_n) + J(\boldsymbol{Y}_n))^{-1} \cdot F(\boldsymbol{X}_n), \tag{3.4}$$

where

$$\boldsymbol{Y}_n = \boldsymbol{X}_n - J(\boldsymbol{X}_n)^{-1} \cdot \boldsymbol{F}(\boldsymbol{X}_n).$$

3.3 Tests and Validation

In this section, the behavior of the Newton's method and its modifications for nonlinear systems of increasing complexity will be analyzed based on the CPU computation runtime t_{CPU}, and number of iterations required to converge when the tolerance is set to $\epsilon = 1 \times 10^{-12}$.

3.3.1 Example 1: System of Two Equations

Consider the two-variable system

$$\boldsymbol{F}(\boldsymbol{X}) = \begin{pmatrix} \sin(x_1 x_2) + x_2^3 - 4 \\ e^{x_1} + x_1 \cos(x_2) - 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

with the Jacobian matrix given by

$$\boldsymbol{J}(\boldsymbol{X}) = \begin{pmatrix} x_2 \cos(x_1 x_2) & x_1 \cos(x_1 x_2) + 3x_2^2 \\ e^{x_1} + \cos(x_2) & -x_1 \sin(x_2) \end{pmatrix},$$

and an initial guess $\boldsymbol{X}_0 = (1 \quad 1)^t$.

Table 3.1: Performance comparison of the Newton's method and its modifications when applied to a system of two nonlinear equations.

	n	t_{CPU}	$\boldsymbol{X}_n$	$\|\|\boldsymbol{F}(\boldsymbol{X}_n)\|\|$
N2	5	0.01562500000000000000	$\begin{pmatrix}0.65936106092230228892\\1.46985549775510748738\end{pmatrix}$	0.00000000000000088818
K3	4	0.01562500000000000000	$\begin{pmatrix}0.65936106092230239994\\1.46985549775510770942\end{pmatrix}$	0.00000000000000000000
H3	3	0.01562500000000000000	$\begin{pmatrix}0.65936106092229718190\\1.46985549775506685322\end{pmatrix}$	0.00000000000028466638
W3	4	0.01562500000000000000	$\begin{pmatrix}0.65936106092230239994\\1.46985549775510770942\end{pmatrix}$	0.00000000000000000000

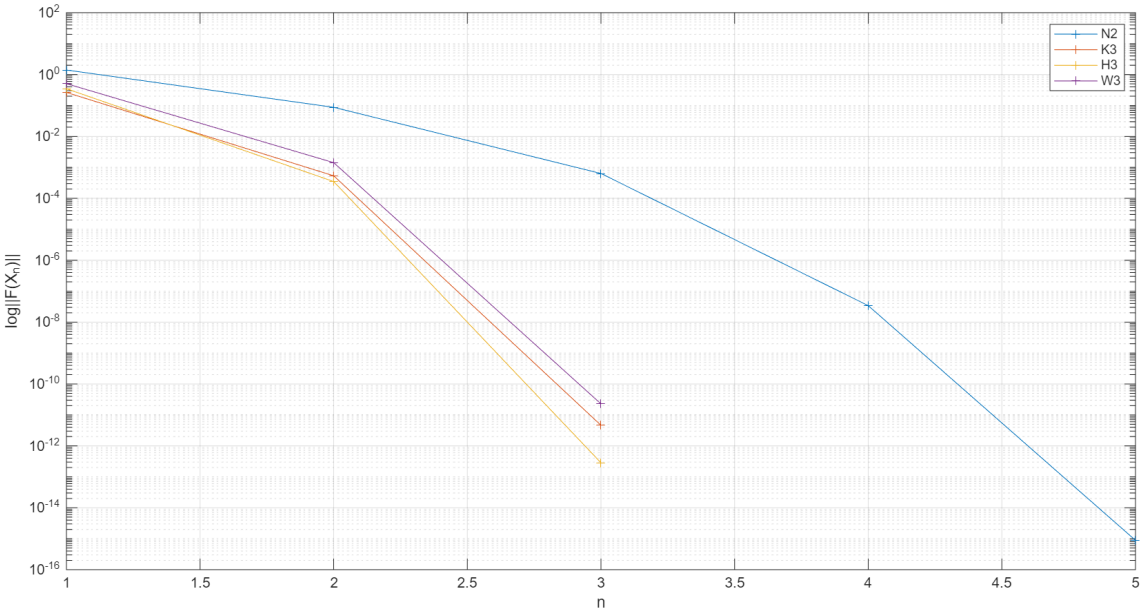

Figure 3.1: Error curves for the Newton's method and its modifications when applied to a system of two nonlinear equations.

Table 3.2: Convergence rates of the Newton's method and its modifications when applied to a system of two nonlinear equations.

n	N2	K3	H3	W3
2	2.79291846005032429190	2.98716605955174019371	2.90698739748779333425	2.92336856132228417593
3	1.74099503487297480042	2.98604855104152111522	–	3.03630699891795075018
4	2.00087687735370511888	–	–	–

As demonstrated in Table 3.1, the advantage of the Newton's method modifications over the classical Newton method extends beyond 1-dimensional root-finding problems to systems of nonlinear equations. Although all four schemes required similar computational time to converge due to the simplicity of the test problem, the modified schemed converged in at most four iterations, whereas the classical Newton's method required five. Furthermore, Table 3.2

confirms that the theoretical convergence rates established in earlier sections are preserved when the schemes are applied to multivariable systems. Additionally, Figure 3.1 illustrates that the error curves for the cubic modifications exhibit steeper curves, indicating faster convergence.

A common implementation error when applying Newton's method to systems of nonlinear equations is the use of an incorrect Jacobian matrix, consider the following Jacobian matrix instead

$$\boldsymbol{J}(\boldsymbol{X}) = \begin{pmatrix} x_2\cos(x_1x_2) & x_2\cos(x_1x_2) + 3x_2^2 \\ e^{x_1} + \cos(x_2) & -x_1\sin(x_2) \end{pmatrix},$$

which results in the following.

Table 3.3: Performance comparison of the Newton's method and its modifications when applied to a system of two nonlinear equations with an incorrect Jacobian matrix.

	n	t_{CPU}	$\boldsymbol{X}_n$	$\|\|\boldsymbol{F}(\boldsymbol{X}_n)\|\|$
N2	12	0.01562500000000000000	$\begin{pmatrix}0.65936106092230661879\\1.46985549775512036597\end{pmatrix}$	0.00000000000009059420
K3	14	0.01562500000000000000	$\begin{pmatrix}0.65936106092229385123\\1.46985549775508106407\end{pmatrix}$	0.00000000000018918200
H3	11	0.01562500000000000000	$\begin{pmatrix}0.65936106092228863318\\1.46985549775506485481\end{pmatrix}$	0.00000000000030508929
W3	11	0.01562500000000000000	$\begin{pmatrix}0.65936106092228363718\\1.46985549775504975578\end{pmatrix}$	0.00000000000041255894

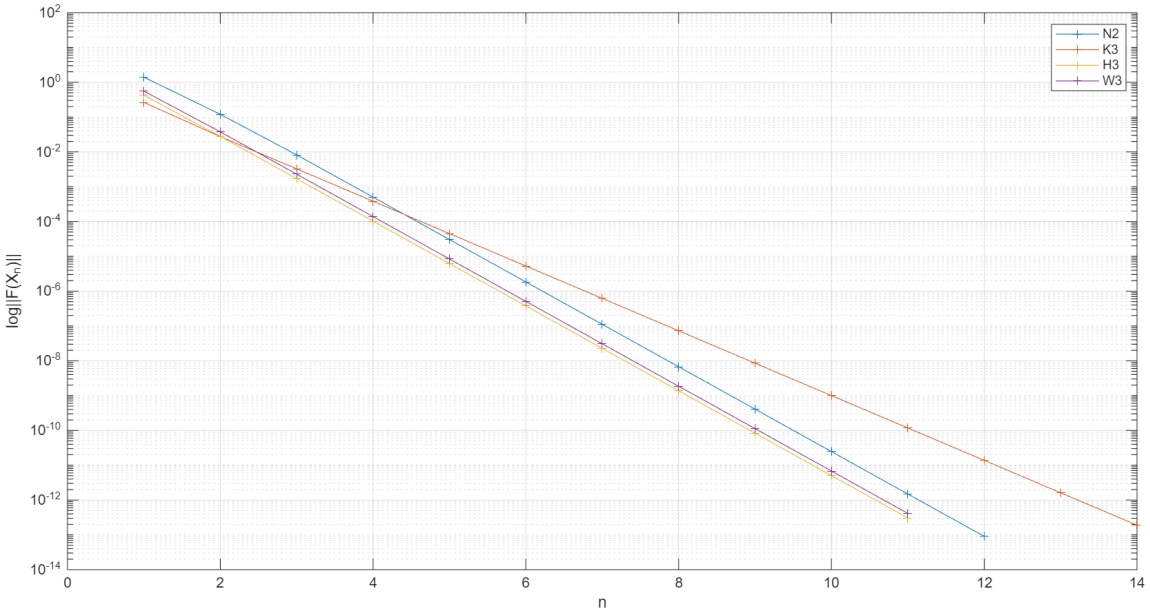

Figure 3.2: Error curves for the Newton's method and its modifications when applied to a system of two nonlinear equations with an incorrect Jacobian matrix.

Table 3.4: Convergence rates of the Newton's method and its modifications when applied to a system of two nonlinear equations with an incorrect Jacobian matrix.

n	N2	K3	H3	W3
2	2.47433154519346842903	1.39284943187978549339	1.25051214940531552067	1.41571707322612616586
3	1.06511631277430884879	0.73557040575698773299	1.01928983905414671796	1.02923014260926648511
⋮				
9	1.00007044705794023720	1.00000810118319605202	1.00115118320413154507	1.00115598751784040665
10	1.00115447296371229413	1.00006855497049707004	1.01971055914327979330	1.01970490895515797369
11	1.01969284606013710359	1.00058797106852392922	–	–
12	–	1.00501535817417719798	–	–
13	–	1.04565267254673233133	–	–

As shown in Table 3.3, implementing Newton's method with a slightly incorrect Jacobian still leads to convergence to the root, but requires a significantly greater number of iterations. For example, when the correct Jacobian matrix is used, all methods converge within five iterations. However, with an incorrect Jacobian, convergence requires at least twice as many iterations, exceeding ten for both the classical Newton's method and its modifications. Moreover, Figure 3.2 and Table 3.4 illustrate that, although convergence is eventually achieved, the rate of convergence becomes linear rather than quadratic or cubic.

3.3.2 Example 2: System of Three Equations

Consider the three-variable system

$$\boldsymbol{F}(\boldsymbol{X}) = \begin{pmatrix} x_1^2 + \sin(x_2 x_3) - 3 \\ \cos(x_1) + e^{x_2} - x_3^3 \\ x_1 + x_2 + x_3 - e^{x_1 x_2 x_3} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

with the Jacobian matrix given by

$$\boldsymbol{J}(\boldsymbol{X}) = \begin{pmatrix} 2x_1 & x_3\cos(x_2 x_3) & x_2\cos(x_2 x_3) \\ -\sin(x_1) & e^{x_2} & -3x_3^2 \\ 1 - x_2 x_3 e^{x_1 x_2 x_3} & 1 - x_1 x_3 e^{x_1 x_2 x_3} & 1 - x_1 x_2 e^{x_1 x_2 x_3} \end{pmatrix},$$

and an initial guess $\boldsymbol{X}_0 = (1 \quad 1 \quad 1)^t$.

Table 3.5: Performance comparison of the Newton's method and its modifications when applied to a system of three nonlinear equations.

	n	t_{CPU}	$\boldsymbol{X}_n$	$\|\|\boldsymbol{F}(\boldsymbol{x}_n)\|\|$
N2	7	0.01562500000000000000	$\begin{pmatrix} 1.50766583317275393306 \\ 0.64857678244495098330 \\ 1.25484098338041372145 \end{pmatrix}$	0.00000000000000022204
K3	4	0.01562500000000000000	$\begin{pmatrix} 1.50766583317275393306 \\ 0.64857678244495131636 \\ 1.25484098338041416554 \end{pmatrix}$	0.00000000000000349676
H3	4	0.01562500000000000000	$\begin{pmatrix} 1.50766583317275393306 \\ 0.64857678244495098330 \\ 1.25484098338041372145 \end{pmatrix}$	0.00000000000000022204
W3	4	0.01562500000000000000	$\begin{pmatrix} 1.50766583317275371101 \\ 0.64857678244495098330 \\ 1.25484098338041372145 \end{pmatrix}$	0.00000000000000108779

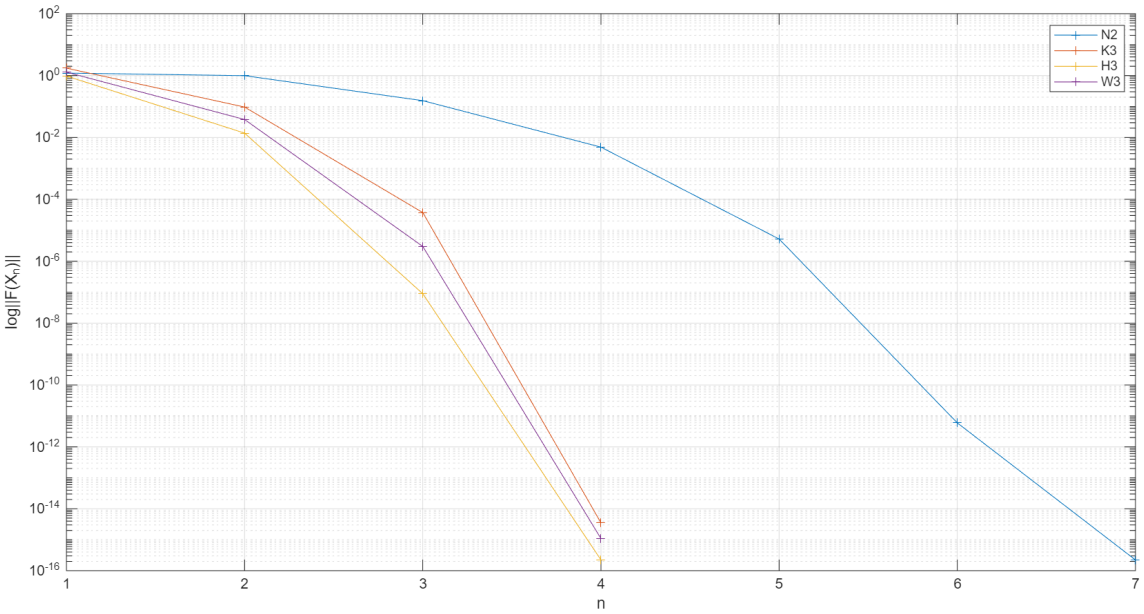

Figure 3.3: Error curves for the Newton's method and its modifications when applied to a system of two nonlinear equations.

Table 3.6: Convergence rates of the Newton's method and its modifications when applied to a system of three nonlinear equations.

n	N2	K3	H3	W3
2	2.02681433753216788674	5.64228532268375904124	3.52762641730369574944	3.19770726079867628755
3	1.49855229171175130531	2.34762091636102487868	2.78473857578697492343	2.83975097049805569327
4	2.23589651268094291581	–	–	–
5	1.99315648481491192179	–	–	–
6	2.00016937192150434655	–	–	–

Building on the previous example, Table 3.5 shows that the advantage of the modified Newton's methods over the classical scheme also extends to systems involving three equations. While all four methods require comparable computational time, again reflecting the relative simplicity of the problem, the modified schemes converge in at most four iterations, whereas the classical Newton's method requires seven. This performance difference is further supported by Figure 3.3, where the error curves for each cubic modification exhibit a steeper decline compared to the classical method. Additionally, Table 3.6 confirms that the theoretical orders of convergence are preserved.

3.3.3 Example 3: System of Four Equations

Consider the four-variable system

$$\boldsymbol{F}(\boldsymbol{X}) = \begin{pmatrix} x_1^3 - x_2 x_3 + \sin(x_4) - 1 \\ e^{x_2} + \cos(x_3) - x_1 x_4 \\ x_2 \sin(x_1) + x_3^2 - x_4^3 + 2 \\ x_1 + x_2 + x_3 + x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

with the Jacobian matrix given by

$$\boldsymbol{J}(\boldsymbol{X}) = \begin{pmatrix} 3x_1^2 & -x_3 & -x_2 & \cos(x_4) \\ -x_4 & e^{x_2} & -\sin(x_3) & -x_1 \\ x_2 \cos(x_1) & \sin(x_1) & 2x_3 & -3x_4^2 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

and an initial guess $\boldsymbol{X}_0 = (1 \quad 1 \quad 1 \quad 1)^t$.

Table 3.7: Performance comparison of the Newton's method and its modifications when applied to a system of four nonlinear equations.

	n	t_{CPU}	$\boldsymbol{X}_n$	$\|\|\boldsymbol{F}(\boldsymbol{X}_n)\|\|$
N2	8	0.01562500000000000000	$\begin{pmatrix} 1.00513291080073141615 \\ -1.44293561798560543430 \\ -0.61171044542410279998 \\ 1.04951315260897670711 \end{pmatrix}$	0.00000000000000049651
K3	9	0.01562500000000000000	$\begin{pmatrix} 1.00513291080073141615 \\ -1.44293561798560521225 \\ -0.61171044542410279998 \\ 1.04951315260897648507 \end{pmatrix}$	0.00000000000000074476
H3	6	0.01562500000000000000	$\begin{pmatrix} 1.00513291080073141615 \\ -1.44293561798560521225 \\ -0.61171044542410279998 \\ 1.04951315260897648507 \end{pmatrix}$	0.00000000000000074476
W3	7	0.01562500000000000000	$\begin{pmatrix} 1.00513291080073141615 \\ -1.44293561798560521225 \\ -0.61171044542410291101 \\ 1.04951315260897670711 \end{pmatrix}$	0.00000000000000022204

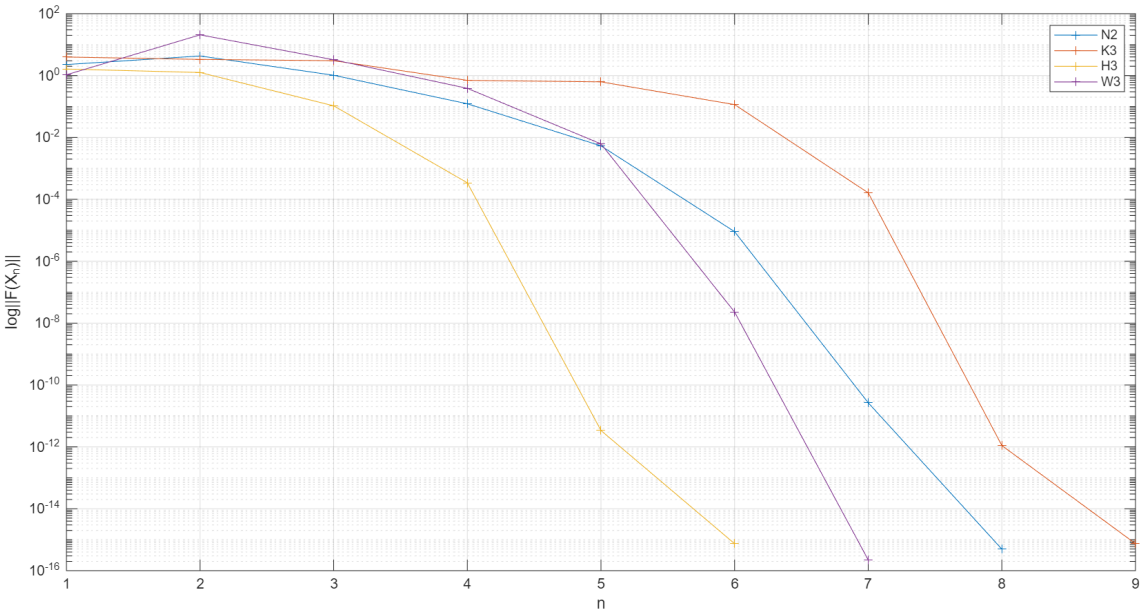

Figure 3.4: Error curves for the Newton's method and its modifications when applied to a system of two nonlinear equations.

Table 3.8: Convergence rates of the Newton's method and its modifications when applied to a system of four nonlinear equations.

n	N2	K3	H3	W3
2	0.52613239625398733335	1.52370184489422366703	1.62472257063293645807	-1.05202205106332402629
3	2.12342431258000274852	1.29649819228622686929	2.44327408769046883208	-0.90176172182205605043
4	1.65500062252773094684	2.48435384383253987650	3.06250544447473682652	1.10196169322217074615
5	2.65500062252773094684	0.56120325094096445984	2.79882911583742233219	3.45610879265566284246
6	1.67254414589940569869	4.12428475605311817276	–	3.05487768958442629241
7	2.14836007278489748984	3.13184767585827783520	–	–
8	–	3.14078662206505354604	–	–

Extending the analysis to a system of four equations, Table 3.7 demonstrates that the Newton's method modifications continue to outperform the classical scheme, converging in 4 to 5 iterations compared to 8 for the classical scheme, while maintaining similar CPU times. Furthermore, Figure 3.4 and Table 3.8 show that the faster error decay and higher convergence orders observed in lower dimensional cases are preserved as the system's dimension increases. These results reinforce the efficiency of the modified schemes, highlighting their consistent advantages across systems of nonlinear equation of increasing complexity.

Chapter 4

Initial Value Problems

In this chapter, the Newton's method and its modification are extended beyond algebraic equations to address initial value problems involving ordinary differential equations. By formulating fully implicit schemes, the schemes are employed to solve nonlinear systems arising from the discretization of the initial value problem. The chapter presents two applications to demonstrate the practical implementation and effectiveness of these iterative schemes in solving time-dependent problems.

4.1 Fully Implicit Discretization of Initial Value Problems

In many real-life applications, mathematical models are often expressed as a system of ordinary differential equations in an initial value problem, rather than a straightforward system of nonlinear equations. An initial value problem is generally expressed in the form

$$\frac{dy}{dt} = f(t, y) \qquad \text{with} \quad \begin{cases} a \leq t \leq b \\ y(a) = \alpha \end{cases},$$

where $y(t)$ is the unknown solution with the initial condition $y(a) = \alpha$. In many cases, an exact analytical solution to the initial value problem does not exist or is difficult to obtain, as a result, numerical approximations are often employed. A common approach to approximating the solution of an initial value problem is Euler's method. The method is derived by discretizing the time interval $[a, b]$ into $N+1$ mesh points given by $t_j = a+jh$, where $h = (b-a)/N$ is the step size, and N is a positive integer. By performing a 1^{st} order Taylor expansion of $y(t_j)$ around the point t_{j+1}, the following is obtained

$$y(t_j) = y(t_{j+1}) - hf(t_{j+1}, y(t_{j+1})) + O(h^2).$$

Since the step size h is typically chosen to be very small, the remainder term $O(h^2)$ is sufficiently small to be neglected. This results in an approximation w_j of the solution $y(j_i)$ at discrete mesh points given by

$$\begin{aligned} w_0 &= \alpha \\ w_{j+1} &= w_j + f(t_{j+1}, w_{j+1}) \end{aligned}$$

which is the backward Euler's method. It is important to note that in order to apply Euler's method, the initial value problem must be well-posed, meaning a unique solution exists and continuously depends on the initial condition. Let

$$D = \{(t, y) : a \leq t \leq b \text{ and } -\infty < y < \infty\},$$

if $f(t, y)$ is continuous in D and satisfies Lipschitz condition on y in D, that is

$$|f(t, y_i) - f(t, y_j)| \leq L|y_i - y_j|$$

for some constant L, then the initial value problem is well-posed. In contrast to the forward Euler's method, which is an explicit scheme that is only conditionally stable depending on the step size, the backward Euler method is an implicit scheme that requires solving an equation at each step. Despite the additional computational cost, it offers superior numerical stability as it is A-stable, meaning it is unconditionally stable.

In the case of a system of differential equation, the initial value problem is of the general form

$$\begin{aligned} \frac{dy_1}{dt} &= f_1(t, y_1, \dots y_n) \\ &\vdots \\ \frac{dy_n}{dt} &= f_n(t, y_1, \dots y_n) \end{aligned} \qquad \text{with} \quad \begin{cases} a \le t \le b \\ y_1(a) = \alpha_1 \\ \quad \vdots \\ y_n(a) = \alpha_n \end{cases}.$$

Let $w_{i,j}$ be the approximation of $y_i(t_j)$, the backward Euler's method is implemented as follows

$$\begin{aligned} w_{i,0} &= \alpha_i \\ w_{i,j+1} &= w_{i,j} + h f_i(t_{j+1}, w_{1,j+1}, \dots, w_{n,j+1}) \end{aligned}$$

Similarly, the initial value problem must be well-posed in order to implement the backward Euler's method. This is ensured when each function $f_i(t, y_1, \dots y_n)$ is continuous in

$$D = \{(t, y_1, \dots y_n) : a \le t \le b \text{ and } -\infty < y_i < \infty, \text{ for all } i\},$$

and satisfies Lipschitz condition in the variables $y_i, \dots, y_n$ on D, that is

$$|f_i(t, y_1, \dots y_n) - f_i(t, z_1, \dots z_n)| \le L \sum_{i=1}^{n} |y_i - z_i|$$

for some constant L, for all i. Due to the implicit nature of the backward Euler's method, it requires solving a system of equations at each mesh point. Let $\boldsymbol{W}_j = (w_{1,j}, \cdots, w_{n,j})^t$ be the vector of unknowns at discrete time t_j. For any t_j, $\boldsymbol{W}_j$ is known and the system to be solved is given by

$$\boldsymbol{F}(\boldsymbol{W}_{j+1}) = \begin{pmatrix} w_{1,j+1} - w_{1,j} - h f_1(\boldsymbol{W}_{j+1}) \\ \vdots \\ w_{n,j+1} - w_{n,j} - h f_n(\boldsymbol{W}_{j+1}) \end{pmatrix} \equiv \begin{pmatrix} g_1(\boldsymbol{W}_{j+1}) \\ \vdots \\ g_n(\boldsymbol{W}_{j+1}) \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}.$$

The corresponding Jacobian matrix writes

$$\boldsymbol{J}(\boldsymbol{W}_{j+1}) = \begin{pmatrix} \partial_{w_{1,j+1}} g_1(\boldsymbol{W}_{j+1}) & \cdots & \partial_{w_{n,j+1}} g_1(\boldsymbol{W}_{j+1}) \\ \vdots & \ddots & \vdots \\ \partial_{w_{1,j+1}} g_n(\boldsymbol{W}_{j+1}) & \cdots & \partial_{w_{n,j+1}} g_n(\boldsymbol{W}_{j+1}) \end{pmatrix}.$$

This formulation allows the use of Newton's method to iteratively solve for $\boldsymbol{W}_{j+1}$ at each step.

4.2 Applications

4.2.1 System of Ordinary Differential Equations with Exact Analytical Solution

Consider the initial value problem for a system of two ordinary differential equations

$$\begin{aligned} \frac{dy_1}{dt} &= -y_1 + y_1 y_2 \\ \frac{dy_2}{dt} &= -y_2 \end{aligned} \qquad \text{with} \quad \begin{cases} 0 \leq t \leq 5 \\ y_1(0) = 2 \\ y_2(0) = 2.5 \end{cases},$$

which has an exact solution given by

$$\begin{aligned} y_1(t) &= 2e^{-t}e^{2.5(1-e^{-t})} \\ y_2(t) &= 2.5e^{-t}. \end{aligned}$$

Let $w_{i,j}$ denote the approximation of $y_i(t_j)$. Using the backward Euler's method, the approximations are given by

$$\begin{aligned} w_{1,j+1} &= w_{1,j} + h(-w_{1,j+1} + w_{1,j+1}w_{2,j+1}), \\ w_{2,j+1} &= w_{2,j} - hw_{2,j+1}, \end{aligned}$$

with

$$\begin{aligned} w_{1,0} &= 2, \\ w_{2,0} &= 2.5. \end{aligned}$$

The time interval $[0, 5]$ is discretized into $N = 500$ subintervals, resulting in a uniform time step size of $h = 0.01$. The resulting mesh points are defined by $t_j = 0.01j$. At any t_j, $\boldsymbol{W}_j$ is known and the discretized nonlinear system is given by

$$\boldsymbol{F}(\boldsymbol{W}_{j+1}) = \begin{pmatrix} w_{1,j+1} - w_{1,j} + h(w_{1,j+1} - w_{1,j+1}w_{2,j+1}) \\ w_{2,j+1} - w_{2,j+1} + hw_{2,j+1} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The corresponding Jacobian matrix is given by

$$\boldsymbol{J}(\boldsymbol{W}_{j+1}) = \begin{pmatrix} 1 + h(1 - w_{2,j+1}) & -hw_{1,j+1} \\ 0 & 1 + h \end{pmatrix}.$$

Note that, at any t_j, the approximated $\boldsymbol{W}_j$ will be employed as an initial guess when computing solution $\boldsymbol{W}_{j+1}$. Table 4.1 presents the results, where n represents the total number of iterations required to converge to t within a tolerance of $\epsilon = 1 \times 10^{-12}$, summed over the entire time interval. The term $||\boldsymbol{F}(\boldsymbol{W}_n)||$ represents the residual error at the root of the system at each mesh point, summed over the entire time interval as well.

Table 4.1: Performance comparison of the Newton's method and its modifications in solving an initial value problem for a system of two ordinary differential equations using the backward Euler's method.

	n	t_{CPU}	$\|\|\boldsymbol{F}(\boldsymbol{W}_n)\|\|$
N2	1000	1.51562500000000000000	0.00000000000003656881
K3	813	0.57812500000000000000	0.00000000003568973799
H3	788	0.62500000000000000000	0.00000000003580280390
W3	788	0.62500000000000000000	0.00000000003580283165

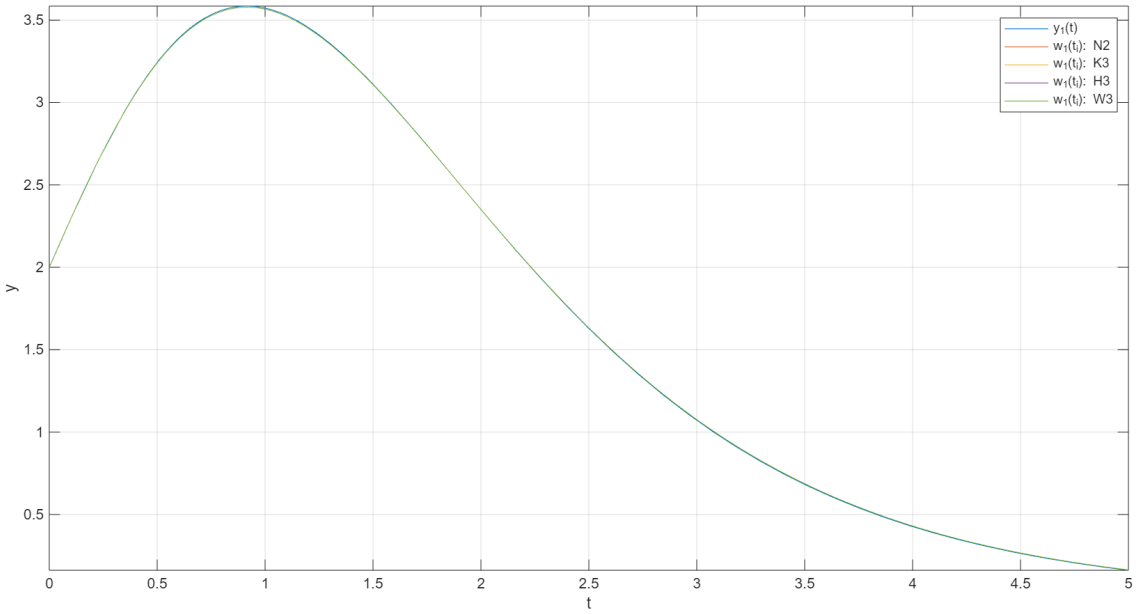

Figure 4.1: Comparison between the exact solution $y_1(t)$ and the numerical solutions $w_1(t_j)$ obtained using the Newton's method and its modifications.

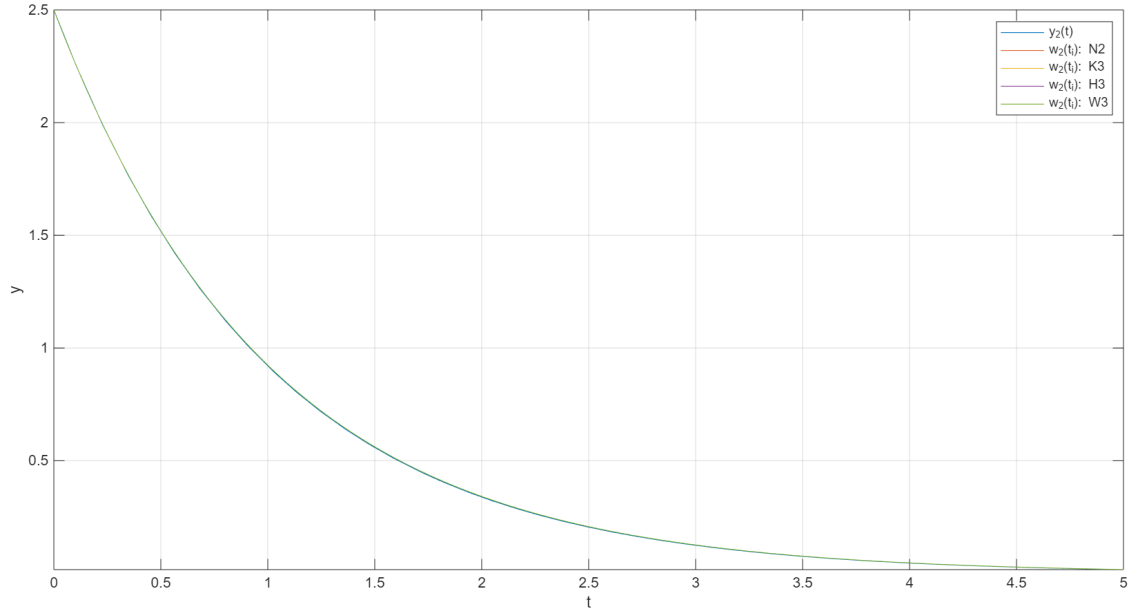

Figure 4.2: Comparison between the exact solution $y_2(t)$ and the numerical solutions $w_2(t_j)$ obtained using the Newton's method and its modifications.

In earlier sections, comparisons between the classical Newton's method and its modifications revealed slight differences in convergence iterations and computational time, which is due to the simplicity of the test problems. However, in the context of initial value problems, where the Newton's method must be applied iteratively at each mesh point, the difference in performance becomes significant. This difference grows exponentially when considering complex initial value problems involving even more ordinary differential equations. As demonstrated in Table 4.1, the Newton's method requires 1,000 iterations to approximate the solution, whereas its modifications achieve comparable results in approximately 800 iterations with less than half the computational time. While the total error for the Newton's method is marginally smaller than that of its modifications, both the Newton's method and its modifications result in approximations of comparable quality, as illustrated in Figures 4.1 and 4.2. This demonstrates that even though the modification might not result in improved accuracy, they allow faster convergence at a lower computational cost in many cases.

4.2.2 Mathematical Model for the Activation of Cardiac Muscle

Mathematical models play an important role in biomedical research by providing a structured way to understand complex biological processes. By formulating biological phenomena as mathematical problems, researchers can study system dynamics, make predictions, and evaluate hypotheses that may be difficult or even impossible to test experimentally. Among the various types of models, those based on initial value problems are particularly valuable, as they capture the time-dependent evolution of biological variables, such as those involved in disease progression, organ function, or cellular interactions.

An important area of application is cardiac biomechanics, where understanding the interplay between mechanical forces and biochemical signals within heart cells is critical. A study presented in [2] examined the influence of fluid–structure interactions and calcium dynamics on myocardial contraction at the cellular level. The aim was to simulate the propagation of calcium concentrations within the cell and their role in initiating contraction. Calcium serves as a key regulatory signal in heart muscle activation, and its dynamics can be effectively described using a reaction-diffusion framework. Using a reaction-diffusion model, the time-dependent changes in calcium concentration in both the cytosol and sarcoplasmic reticulum can be analyzed.

In the study, we consider a model formulated as a system of three ordinary differential equations that define an initial value problem. Let $y_1(t)$ denote the concentration of cytosolic calcium, which acts as a trigger signal and regulates the timing and intensity of muscle contraction. Let $y_2(t)$ denote the concentration of sarcoplasmic calcium, which serves as an internal reservoir responsible for the release of calcium that elevates the cytosolic concentration. Finally, let $\gamma(t)$ represent the activation signal. The initial value problem is then given by

$$\begin{aligned}
\frac{dy_1}{dt} &= v_1 - \frac{v_2 y_1^2}{k_1 + y_1^2} + \frac{v_3 y_1^4 y_2^2}{(k_2 + y_2^2)(k_3 + y_1^4)} - v_4 y_1 \\
\frac{dy_2}{dt} &= \frac{v_2 y_1^2}{k_1 + y_1^2} - \frac{v_3 y_1^4 y_2^2}{(k_2 + y_2^2)(k_3 + y_1^4)} - v_5 y_2 \qquad \text{with} \quad \begin{cases} 0 \le t \le 5 \\ y_1(0) = 0 \\ y_2(0) = 1.5 \\ \gamma(0) = 0 \end{cases} \\
\frac{d\gamma}{dt} &= -d_1 y_1 - d_2 \gamma
\end{aligned} .$$

Table 4.2: Model parameters.

v_1	v_2	v_3	v_4	v_5
$1.58\ \frac{\mu\text{M}}{s}$	$16\ \frac{\mu\text{M}}{s}$	$91\ \frac{\mu\text{M}}{s}$	$2\ \text{s}^{-1}$	$0.2\ \text{s}^{-1}$
k_1	k_2	k_3	d_1	d_2
$1\ \mu\text{M}$	$4\ \mu\text{M}^2$	$0.7841\ \mu\text{M}^4$	$0.5\ \frac{1}{\mu\text{Ms}}$	$2.5\ \frac{1}{\text{s}}$

Let $w_{1,j}$ and $w_{2,j}$ denote the approximations of $y_1(t_j)$ and $y_2(t_j)$, respectively, and let $w_{3,j}$ denote the approximation of $\gamma(t_j)$. Using the backward Euler method, the discretized problem is given by

$$\begin{aligned}
w_{1,j+1} &= w_{1,j} + h\left(v_1 - \frac{v_2 w_{1,j+1}^2}{k_1 + w_{1,j+1}^2} + \frac{v_3 w_{1,j+1}^4 w_{2,j+1}^2}{(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4)} - v_4 w_{1,j+1}\right), \\
w_{2,j+1} &= w_{2,j} + h\left(\frac{v_2 w_{1,j+1}^2}{k_1 + w_{1,j+1}^2} - \frac{v_3 w_{1,j+1}^4 w_{2,j+1}^2}{(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4)} - v_5 w_{2,j+1}\right), \\
w_{3,j+1} &= w_{3,j} - h\left(d_1 w_{1,j+1} + d_2 w_{3,j+1}\right),
\end{aligned}$$

with

$$
\begin{aligned}
w_{1,0} &= 0,\\
w_{2,0} &= 1.5,\\
w_{3,0} &= 0.
\end{aligned}
$$

The time interval $[0, 50]$ is uniformly discretized into $N = 5000$ subintervals, yielding a time step size of $h = 0.01$. The corresponding mesh points are given by $t_j = 0.01j$. At any t_j, the discretized system using the backward Euler method reduces to the nonlinear system

$$
\boldsymbol{F}(\boldsymbol{W}_{j+1}) = \begin{pmatrix}
w_{1,j+1} - w_{1,j} - h\left(v_1 - \dfrac{v_2 w_{1,j+1}^2}{k_1 + w_{1,j+1}^2} + \dfrac{v_3 w_{1,j+1}^4 w_{2,j+1}^2}{(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4)} - v_4 w_{1,j+1}\right)\\
w_{2,j+1} - w_{2,j} - h\left(\dfrac{v_2 w_{1,j+1}^2}{k_1 + w_{1,j+1}^2} - \dfrac{v_3 w_{1,j+1}^4 w_{2,j+1}^2}{(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4)} - v_5 w_{2,j+1}\right)\\
w_{3,j+1} - w_{3,j} + h\left(d_1 w_{1,j+1} + d_2 w_{3,j+1}\right)
\end{pmatrix} = \begin{pmatrix} 0\\ 0\\ 0 \end{pmatrix}.
$$

The corresponding Jacobian matrix given by

$$
\boldsymbol{J}(\boldsymbol{W}_{j+1}) = \begin{pmatrix}
\partial_{w_{1,j+1}} g_1(\boldsymbol{W}_{j+1}) & \partial_{w_{2,j+1}} g_1(\boldsymbol{W}_{j+1}) & 0\\
\partial_{w_{1,j+1}} g_2(\boldsymbol{W}_{j+1}) & \partial_{w_{2,j+1}} g_2(\boldsymbol{W}_{j+1}) & 0\\
h d_1 & 0 & 1 + h d_2
\end{pmatrix},
$$

where

$$
\begin{aligned}
\partial_{w_{1,j+1}} g_1(\boldsymbol{W}_{j+1}) &= 1 - h\left(\frac{(4v_3 w_{1,j+1}^3 w_{2,j+1}^2)(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4) - (4v_3 w_{1,j+1}^7 w_{2,j+1}^2)(k_2 + w_{2,j+1}^2)}{((k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4))^2}\right.\\
&\quad \left. - \frac{2v_2 k_1 w_{1,j+1}}{(k_1 + w_{1,j+1}^2)^2} + v_4\right),\\
\partial_{w_{2,j+1}} g_1(\boldsymbol{W}_{j+1}) &= -h\left(\frac{(2v_3 w_{1,j+1}^4 w_{2,j+1})(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4) - (2v_3 w_{1,j+1}^4 w_{2,j+1}^3)(k_3 + w_{1,j+1}^4)}{((k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4))^2}\right),\\
\partial_{w_{1,j+1}} g_2(\boldsymbol{W}_{j+1}) &= -h\left(\frac{2v_2 k_1 w_{1,j+1}}{(k_1 + w_{1,j+1}^2)^2} + \frac{(4v_3 w_{1,j+1}^7 w_{2,j+1}^2)(k_2 + w_{2,j+1}^2) - (4v_3 w_{1,j+1}^3 w_{2,j+1}^2)(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4)}{((k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4))^2}\right),\\
\partial_{w_{2,j+1}} g_2(\boldsymbol{W}_{j+1}) &= 1 - h\left(\frac{(2v_3 w_{1,j+1}^4 w_{2,j+1}^3)(k_3 + w_{1,j+1}^4) - (2v_3 w_{1,j+1}^4 w_{2,j+1})(k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4)}{((k_2 + w_{2,j+1}^2)(k_3 + w_{1,j+1}^4))^2}\right).
\end{aligned}
$$

Similarly, the approximation $\boldsymbol{W}_j$ will be employed as an initial guess when approximating solution $\boldsymbol{W}_{j+1}$. Using the physiological model parameters listed in Table 4.2, the problem is solved over the entire time interval by employing Newton method and its variants. Table 4.3 presents the total number of linear system solves, the CPU time, and the final residuals for each method.

Table 4.3: Performance comparison of the Newton's method and its modifications in solving the cardiac muscle activation model using the backward Euler's method.

	n	t_{CPU}	$\|\|\boldsymbol{F}(\boldsymbol{W}_n)\|\|$
N2	37159	14.996940000000000000	0.00000001053570266 69
K3	46840	31.875000000000000000	0.00000001597262649 71
H3	37165	15.087360000000000000	0.00000001051900786 28
W3	37152	14.937500000000000000	0.00000001061542877 09

The results demonstrate the performance of the Newton's method. The classical Newton's method required approximately 37000 iterations and 15 seconds of CPU time, while Homeier's and Weerakoon's modifications performed similarly, with approximately 37000 iterations, and comparable CPU times. In contrast, Kou's modification exhibited notably poorer efficiency, demanding 46,840 iterations and 31.875 seconds of CPU time. It is important to note that these results are preliminary. Due to the limited time frame, the impact of the time step size has not been explored. However, it is anticipated that increasing the time step may lead to improved performance of the Newton variants in terms of CPU time, consistent with the results present in Table 4.1.

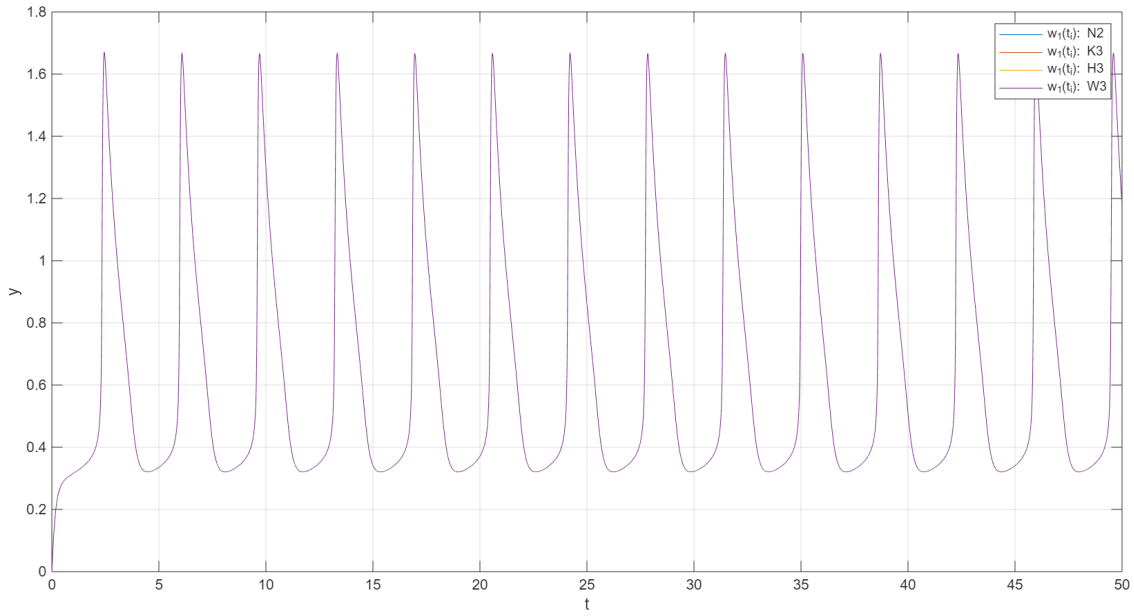

Figure 4.3: Time evolution of the approximated solutions $w_1(t_j)$ using the Newton's method and its modifications.

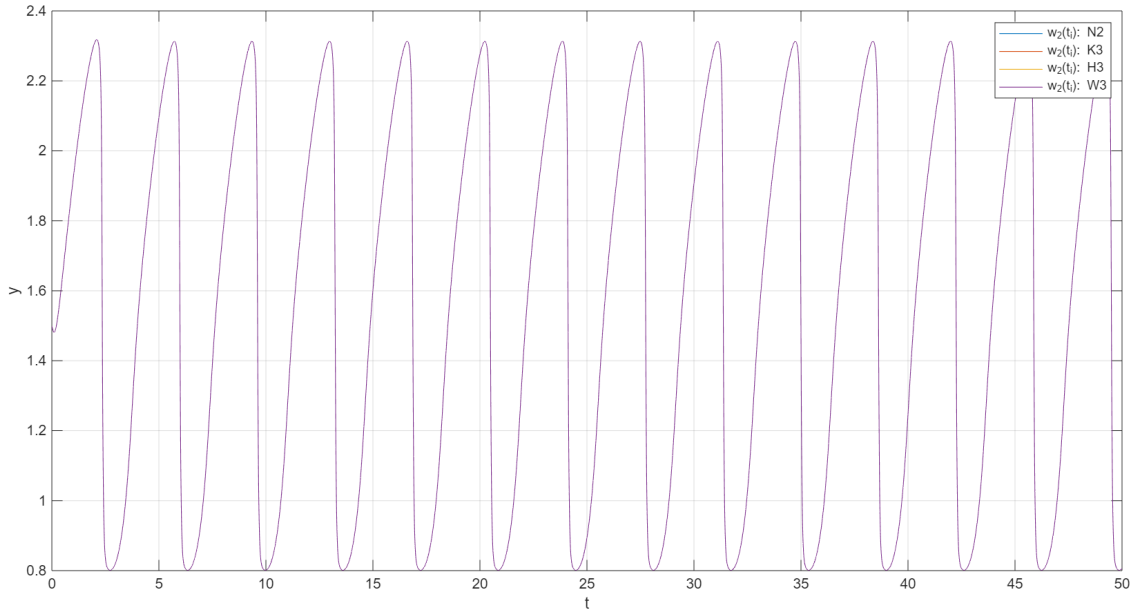

Figure 4.4: Time evolution of the approximated solutions $w_2(t_j)$ using the Newton's method and its modifications.

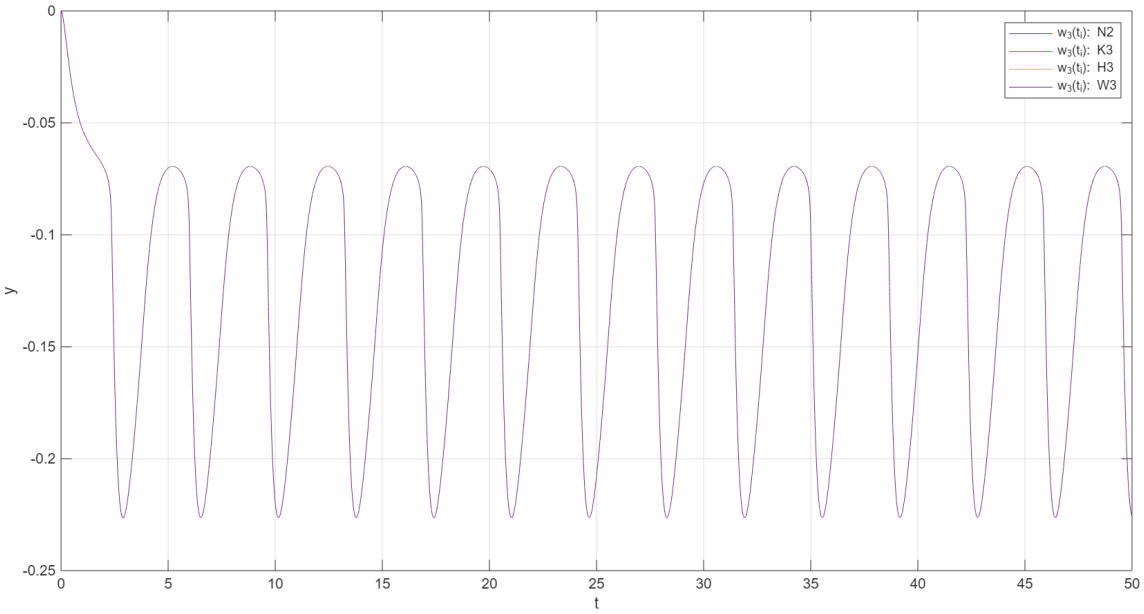

Figure 4.5: Time evolution of the approximated solutions $w_3(t_j)$ using the Newton's method and its modifications.

As illustrated in Figures 4.3–4.5, the functions $y_i(t)$ for all i exhibit perfectly periodic oscillations rather than converging to a steady state, indicating the model's intrinsic limit-cycle behavior. Physiologically, this mimics a heart cell's beat. In each cycle, $y_1(t)$ jumps up rapidly and peaks at approximately 1.67 when cytosolic calcium is released from the cell's storage compartment and then falls more slowly as the cell pumps the calcium out. At the same time, the storage level $y_2(t)$ plunges to approximately 0.80 during release and then refills during the resting phase where it peaks at approximately 2.32. The activation signal $\gamma(t)$ peaks shortly after the calcium spike—reflecting the finite time needed to switch the contractile machinery on—and then returns to zero as the cell relaxes.

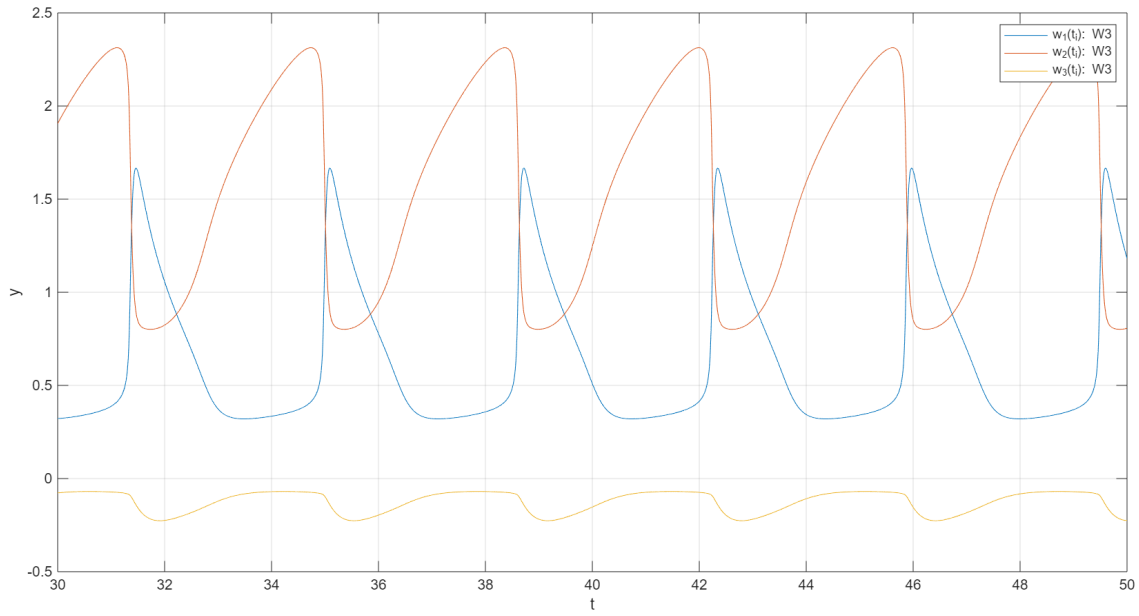

Figure 4.6: Time evolution of the approximated solutions $w_1(t_j)$, $w_2(t_j)$, and $w_3(t_j)$ using Weerakoon's modification on the interval $[30, 50]$.

Figure 4.6 highlits the rhythmic oscillations essential for simulating heart muscle contractions. This visualization confirms the synchronized interplay of these variables, as their periodic behavior, previously observed in individual time-series plots, sustains the system's stable, repetitive dynamics. In the overlaid time-series of $w_1(t)$, $w_2(t)$, and $\gamma(t)$, the tight coupling among cytosolic calcium release, sarcoplasmic calcium refilling, and contractile activation is even more apparent. As w_1 spikes, γ follows after a constant delay, and w_2 reaches its minimum exactly when w_1

peaks. The loop then closes as γ falls and w_2 begins to rise, showing a tightly locked sequence.

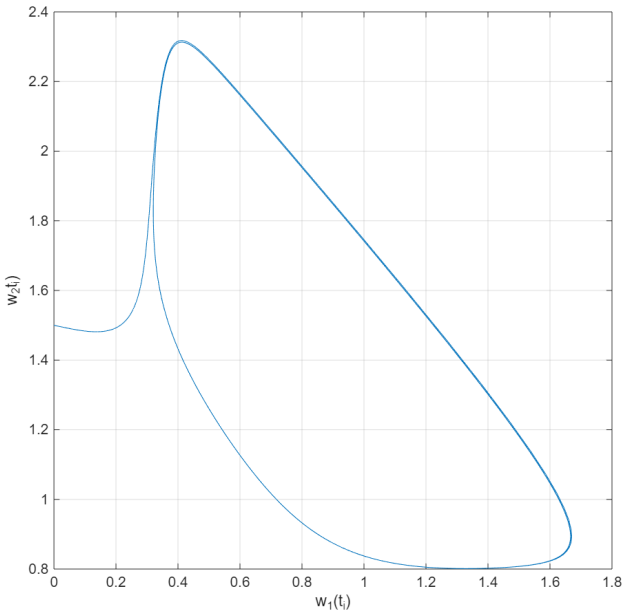

Figure 4.7: Phase plot of $w_2(t_j)$ versus $w_1(t_j)$ over the time interval $[0, 50]$.

Figure 4.7 shows the sarcoplasmic calcium load w_2 plotted against cytosolic calcium w_1 over one full cycle. The trajectory is a closed, clockwise loop, as w_1 rises, w_2 falls sharply during calcium release, once w_1 peaks and begins to decline, w_2 then refills more slowly.

Conclusion

This project has explored the classical Newton's method and its modifications, which were proposed by Kou, Homeier, and Weerakoon, for solving root-finding problems. The theoretical analysis established that the classical Newton's method achieves quadratic convergence, while the modified schemes attain cubic convergence, as demonstrated through rigorous proofs using Taylor expansions. Numerical testing on single-variable functions, systems of nonlinear equations, and initial value problems confirmed the superior efficiency of the modified methods, which consistently required fewer iterations and lower computational time in most cases when compared to the classical approach.

The application of these methods to systems of nonlinear equations highlighted the robustness of the cubic schemes which exhibited faster convergence and preserved third-order convergence even in higher-dimensional problems. The analysis of initial value problems, including a system of ordinary differential equations and a cardiac muscle activation model, further highlighted the practical advantages of the modified methods. In the context of the cardiac model, the Weerakoon modification proved particularly efficient, accurately capturing the periodic calcium dynamics critical to myocardial contraction with reduced computational cost.

Despite these advantages, the performance of the modified methods can vary depending on the problem's complexity and the choice of initial guesses, as observed in the cardiac model where Kou's modification underperformed. Additionally, the sensitivity of all methods to the accuracy of the Jacobian matrix was evident, with incorrect Jacobian matrices leading to slower, linear convergence. These findings suggest that while the modified Newton's methods offer significant improvements, careful implementation and problem-specific tuning are essential for optimal performance.

In conclusion, the enhanced Newton's methods schemes explored in this project provide powerful tools for efficiently solving complex root-finding problems in biomedical applications. Their ability to achieve higher-order convergence and handle intricate systems makes them invaluable for computational mathematics and scientific computing. Future work could explore applications to numerically solve partial differential equations, as well as adaptive initial guess strategies to further improve convergence and robustness, particularly for highly nonlinear biomedical models.

Bibliography

[1] R. Burden and J. Faires, *Numerical Analysis*, 10^{th} ed. Boston, MA: Cengage Learning, 2016.

[2] A. Laadhari, R.Ruiz-Baier, and A. Quarteroni. " Fully Eulerian finite element approximation of a fluid-structure interaction problem in cardiac cells," *International Journal for Numerical Methods in Engineering*. 96 (2013) 712–738.

[3] J. Kou, Y. Li, and X. Wang. "A modification of Newton method with third-order convergence," *Applied Mathematics and Computation*. 181 (2006) 1106-1111.

[4] H. Homeier. "A modified Newton method for rootfinding with cubic convergence," *Journal of Computational and Applied Mathematics*. 157 (2003) 227-230.

[5] S. Weerakoon and T. Fernando. "A Variant of Newton's with Accelerated Third-Order Convergence," *Applied Mathematics Letters*. 13 (2000) 87-93.

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

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

[8] 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*, 375 (2020) 112811.

[9] 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*, 330 (2018) 748–758.

[10] 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*, 2017 (2017) Article ID 3708461.

[11] A. Laadhari and G. Szekely. "Fully implicit finite element method for the modeling of free surface flows with surface tension effect," *International Journal for Numerical Methods in Engineering*, 111 (2017) 1047–1074. `https://doi.org/10.1002/nme.5493`

[12] A. Laadhari. "Exact Newton method with third-order convergence to model the dynamics of bubbles in incompressible flow," *Applied Mathematics Letters*, 69 (2017) 138–145. `https://doi.org/10.1016/j.aml.2017.01.012`

[13] 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," *Journal of Computational Physics*, 343 (2017) 271–299. `https://doi.org/10.1016/j.jcp.2017.04.019`.

[14] 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*, 486 (2025) 129058. `https://doi.org/10.1016/j.amc.2024.129058`.

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

Appendices

The appendix includes the main Matlab scripts for solving root-finding problems using Newton's method and its variants by Kou, Homeier, and Weerakoon. It also contains code for solving nonlinear systems with Newton's method and the model simulating the cardiac muscle activation.

Appendix A: Newton's Method MATLAB Code

```
function [xNewtons, x_nNewtons, e_nNewtons] = newtonsMethod(func, funcDerivative, x0, Tolerance, maxK)

    % func is the function and funcDerivative is the functions derivative
    % x0 is the initial guess for the Newtons method
    % Tolerance is the error tolernce for convergence
    % maxK is the maximum number of iterations
    % x_nNewtons is the sequence generated by the Newtons method
    % e_nNewtons is the residual error of the sequence
    % xNewtons is the root the sequence converges to within the error tolerance

    fprintf("\nNewtons Method\n")
    x = x0;
    x_nNewtons = [];
    e_nNewtons = [];
    xNewtons = 0;
    for K = 1:maxK
        x = x - func(x) / funcDerivative(x);
        x_nNewtons(K) = x;
        e_nNewtons(K) = abs(func(x));
        fprintf("x_%d: %.20f \t Error: %.20f\n", K, x_nNewtons(K), e_nNewtons(K))
        if abs(func(x)) <= Tolerance
            xNewtons = x;
            fprintf("The method converged to root %.20f in %d iterations within %.20f tolerance.\n",
                xNewtons, K, Tolerance)
            return;
        end
    end
    fprintf("The method didnt converge within the maxmimum number of iterations with the desired
        tolerance.\n")

end
```

Appendix B: Newton's Method: Kou Modification MATLAB Code

```
function [xKou, x_nKou, e_nKou] = kouNewtonsMethod(func, funcDerivative, x0, Tolerance, maxK)

    % func is the function and funcDerivative is the functions derivative
    % x0 is the initial guess for the Newtons method
    % Tolerance is the error tolernce for convergence
    % maxK is the maximum number of iterations
    % x_nKou is the sequence generated by Kous modification to the Newtons method
    % e_nKou is the residual error of the sequence
    % xKou is the root the sequence converges to within the error tolerance

    fprintf("\nNewtons Method: Kou Modification\n")
    x = x0;
    xKou = 0;
    x_nKou = [];
    e_nKou = [];
    for K = 1:maxK
        y = x + func(x) / funcDerivative(x);
        x = y - func(y) / funcDerivative(x);
        x_nKou(K) = x;
        e_nKou(K) = abs(func(x));
        fprintf("x_%d: %.20f \t Error: %.20f\n", K, x_nKou(K), e_nKou(K));
        if abs(func(x)) <= Tolerance
            xKou = x;
            fprintf("The method converged to root %.20f in %d iterations within %.20f tolerance.\n", xKou,
                K, Tolerance);
            return;
        end
    end
    fprintf("The method didnt converge within the maxmimum number of iterations with the desired
        tolerance.\n")

end
```

Appendix C: Newton's Method: Homeier Modification MATLAB Code

```
function [xHomeier, x_nHomeier, e_nHomeier] = homeierNewtonsMethod(func, funcDerivative, x0, Tolerance,
    maxK)

    % func is the function and funcDerivative is the functions derivative
    % x0 is the initial guess for the Newtons method
    % Tolerance is the error tolernce for convergence
    % maxK is the maximum number of iterations
    % x_nHomeier is the sequence generated by Homeiers modification to the Newtons method
    % e_nHomeier is the residual error of the sequence
    % xHomeier is the root the sequence converges to within the error tolerance

    fprintf("\nNewtons Method: Homeier Modification\n")
    x = x0;
    xHomeier = 0;
    x_nHomeier = [];
    e_nHomeier = [];
    for K = 1:maxK
        y = x - 1 / 2 * func(x) / funcDerivative(x);
        x = x - func(x) / funcDerivative(y);
        x_nHomeier(K) = x;
        e_nHomeier(K) = abs(func(x));
        fprintf("x_%d: %.20f \t Error: %.20f\n", K, x_nHomeier(K), e_nHomeier(K));
        if abs(func(x)) <= Tolerance
            xHomeier = x;
            fprintf("The method converged to root %.20f in %d iterations within %.20f tolerance.\n",
                xHomeier, K, Tolerance);
            return;
        end
    end
    fprintf("The method didnt converge within the maxmimum number of iterations with the desired
        tolerance.\n")

end
```

Appendix D: Newton's Method: Weerakoon Modification MATLAB Code

```
function [xWeerakoon, x_nWeerakoon, e_nWeerakoon] = weerakoonNewtonsMethod(func, funcDerivative, x0,
    Tolerance, maxK)

    % func is the function and funcDerivative is the function's derivative
    % x0 is the initial guess for the Newton's method
    % Tolerance is the error tolernce for convergence
    % maxK is the maximum number of iterations
    % x_nWeerakoon is the sequence generated by Weerakoon's modification to the Newton's method
    % e_nWeerakoon is the residual error of the sequence
    % xWeerakoon is the root the sequence converges to within the error tolerance

    fprintf("\nNewtons Method: Weerakoon Modification\n")
    x = x0;
    xWeerakoon = 0;
    x_nWeerakoon = [];
    e_nWeerakoon = [];
    for K = 1:maxK
        y = x - func(x) / funcDerivative(x);
        x = x - 2*func(x) / (funcDerivative(x) + funcDerivative(y));
        x_nWeerakoon(K) = x;
        e_nWeerakoon(K) = abs(func(x));
        fprintf("x_%d: %.20f \t Error: %.20f\n", K, x_nWeerakoon(K), e_nWeerakoon(K));
        if abs(func(x)) <= Tolerance
            xWeerakoon = x;
            fprintf("The method converged to root %.20f in %d iterations within %.20f tolerance.\n",
                xWeerakoon, K, Tolerance);
            return;
        end
    end
    fprintf("The method didn't converge within the maxmimum number of iterations with the desired
        tolerance.\n")

end
```

Appendix E: Error Plot MATLAB Code

```
function errorPlot(e_nNewtons, e_nKou, e_nHomeier, e_nWeerakoon)

    figure;
    semilogy(1:length(e_nNewtons), e_nNewtons, '+-', 'DisplayName', 'N2');
    hold on;
    semilogy(1:length(e_nKou), e_nKou, '+-', 'DisplayName', 'K3');
    semilogy(1:length(e_nHomeier), e_nHomeier, '+-', 'DisplayName', 'H3');
    semilogy(1:length(e_nWeerakoon), e_nWeerakoon, '+-', 'DisplayName', 'W3');
    hold off;
    xlabel('n');
    ylabel('log|f(x_n)|');
    legend('Location', 'best');
    grid on;

end
```

Appendix F: System of Equations Newton's Method MATLAB Code

```
function [xNewtons, x_nNewtons, e_nNewtons, K] = systemNewtonsMethod(func, funcDerivative, x0, Tolerance,
    maxK)

    % func is the array consisting of the system of functions and funcDerivative is its Jacobian
    % x0 is the initial guess for the Newtons method
    % Tolerance is the error tolernce for convergence
    % maxK is the maximum number of iterations
    % x_nNewtons is the sequence generated by the Newtons method
    % e_nNewtons is the residual error of the sequence
    % xNewtons is the root the sequence converges to within the error tolerance

    fprintf("\nNewtons Method: System of Equations\n")
    x = x0;
    x_cell = num2cell(x);
    x_nNewtons = [];
    e_nNewtons = [];
    xNewtons = zeros(length(x0),1);
    for K = 1:maxK
        if det(feval(funcDerivative, x_cell{:})) ~= 0
            x = x - inv(feval(funcDerivative, x_cell{:})) * feval(func, x_cell{:});
            x_cell = num2cell(x);
            x_nNewtons = [x_nNewtons, x];
            e_nNewtons = [e_nNewtons, norm(feval(func, x_cell{:}))];
            fprintf("x_%d:\n", K)
            fprintf("%.20f\n", x_nNewtons(:,K))
            fprintf("Error: %.20f\n", e_nNewtons(K))
            if norm(feval(func, x_cell{:})) <= Tolerance
                xNewtons = x;
                fprintf("The method converged to root\n")
                fprintf("%.20f\n", xNewtons)
                fprintf("in %d iterations within %.20f tolerance.\n", K, Tolerance)
                return;
            end
        else
            fprintf("The Jacobian can't be inverted hence the algorithm is terminated")
            xNewtons = NaN
            return;
        end
    end
    fprintf("The method didnt converge within the maxmimum number of iterations with the desired
        tolerance.\n")

end
```

Appendix G: System of Three Ordinary Differential Equations MATLAB Code

---

```
% System of Three Ordinary Differential Equations Initial Value Problem

a = 0;
b = 50;
N = 5000;
h = (b - a) / N;
t = linspace(a, b, N+1);
alpha1 = 0;
alpha2 = 1.5;
alpha3 = 0;
Tolerance = 1e-12;
maxK = 1000;


v1 = 1.58;
v2 = 16;
v3 = 91;
v4 = 2;
v5 = 0.2;
k1 = 1;
k2 = 4;
k3 = 0.7841;
d1 = 0.5;
d2 = 2.5;


% Newton's Method

wNewtons = [alpha1; alpha2; alpha3];
w1Newtons = [alpha1];
w2Newtons = [alpha2];
w3Newtons = [alpha3];
totalIterations = 0;
totalError = 0;
T = cputime;
for i = 1:N
    fprintf("\nIteration %d:\n", i)
    a1  = wNewtons(1);
    a2 = wNewtons(2);
    a3 = wNewtons(3);
    func = @(w1, w2, w3) [
    w1 - a1 - h*(v1 - (v2*w1^2)/(k1+w1^2) + (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v4*w1);
    w2 - a2 - h*((v2*w1^2)/(k1+w1^2) - (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v5*w2);
    w3 - a3 + h*(d1*w1 + d2*w3);
    ];
    funcDerivative = @(w1, w2, w3) [
    1 - h*(((4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 + w1^4) - (4*v3*w1^7*w2^2)*(k2 + w2^2))/((k2 + w2^2)*(k3 +
        w1^4))^2 - (2*v2*k1*w1)/(k1^2 + w1^2)^2 + v4), - h*(((2*v3*w1^4*w2)*(k2 + w2^2)*(k3 + w1^4) -
        (2*v3*w1^4*w2^3)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    - h*((2*v2*k1*w1)/(k1^2 + w1^2)^2 + ((4*v3*w1^7*w2^2)*(k2 + w2^2) - (4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 +
        w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 1 - h*(((2*v3*w1^4*w2^3)*(k3 + w1^4) - (2*v3*w1^4*w2)*(k2 +
        w2^2)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    h*d1, 0, 1 + h*d2
    ];
    w0 = wNewtons;
    [xNewtons, x_nNewtons, e_nNewtons, K] = systemNewtonsMethod(func, funcDerivative, w0, Tolerance, maxK);
    wNewtons = xNewtons;
    w1Newtons = [w1Newtons, wNewtons(1)];
```

```
    w2Newtons = [w2Newtons, wNewtons(2)];
    w3Newtons = [w3Newtons, wNewtons(3)];
    totalIterations = totalIterations + length(e_nNewtons);
    totalError = totalError + e_nNewtons(length(e_nNewtons));
end
fprintf("\nElapsed Time: %.20f\n", cputime - T);
fprintf("\nTotal Iterations: %d\n", totalIterations);
fprintf("\nTotal Error: %.20f\n", totalError);


% Newton's Method: Kou Modification

wKou = [alpha1; alpha2; alpha3];
w1Kou = [alpha1];
w2Kou = [alpha2];
w3Kou = [alpha3];
totalIterations = 0;
totalError = 0;
T = cputime;
for i = 1:N
    fprintf("\nIteration %d:\n", i)
    ti = a + i*h;
    a1  = wKou(1);
    a2 = wKou(2);
    a3 = wKou(3);
    func = @(w1, w2, w3) [
    w1 - a1 - h*(v1 - (v2*w1^2)/(k1+w1^2) + (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v4*w1);
    w2 - a2 - h*((v2*w1^2)/(k1+w1^2) - (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v5*w2);
    w3 - a3 + h*(d1*w1 + d2*w3);
    ];
    funcDerivative = @(w1, w2, w3) [
    1 - h*(((4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 + w1^4) - (4*v3*w1^7*w2^2)*(k2 + w2^2))/((k2 + w2^2)*(k3 +
        w1^4))^2 - (2*v2*k1*w1)/(k1^2 + w1^2)^2 + v4), - h*(((2*v3*w1^4*w2)*(k2 + w2^2)*(k3 + w1^4) -
        (2*v3*w1^4*w2^3)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    - h*((2*v2*k1*w1)/(k1^2 + w1^2)^2 + ((4*v3*w1^7*w2^2)*(k2 + w2^2) - (4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 +
        w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 1 - h*(((2*v3*w1^4*w2^3)*(k3 + w1^4) - (2*v3*w1^4*w2)*(k2 +
        w2^2)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    h*d1, 0, 1 + h*d2
    ];
    w0 = wKou;
    [xKou, x_nKou, e_nKou] = systemKouNewtonsMethod(func, funcDerivative, w0, Tolerance, maxK);
    wKou = xKou;
    w1Kou = [w1Kou, wKou(1)];
    w2Kou = [w2Kou, wKou(2)];
    w3Kou = [w3Kou, wKou(3)];
    totalIterations = totalIterations + length(e_nKou);
    totalError = totalError + e_nKou(length(e_nKou));
end
fprintf("\nElapsed Time: %.20f\n", cputime - T);
fprintf("\nTotal Iterations: %d\n", totalIterations);
fprintf("\nTotal Error: %.20f\n", totalError);


% Newton's Method: Homeier Modification

wHomeier = [alpha1; alpha2; alpha3];
w1Homeier = [alpha1];
w2Homeier = [alpha2];
w3Homeier = [alpha3];
totalIterations = 0;
totalError = 0;
T = cputime;
for i = 1:N
```

```
    fprintf("\nIteration %d:\n", i)
    ti = a + i*h;
    a1  = wHomeier(1);
    a2 = wHomeier(2);
    a3 = wHomeier(3);
    func = @(w1, w2, w3) [
    w1 - a1 - h*(v1 - (v2*w1^2)/(k1+w1^2) + (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v4*w1);
    w2 - a2 - h*((v2*w1^2)/(k1+w1^2) - (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v5*w2);
    w3 - a3 + h*(d1*w1 + d2*w3);
    ];
    funcDerivative = @(w1, w2, w3) [
    1 - h*(((4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 + w1^4) - (4*v3*w1^7*w2^2)*(k2 + w2^2))/((k2 + w2^2)*(k3 +
        w1^4))^2 - (2*v2*k1*w1)/(k1^2 + w1^2)^2 + v4), - h*(((2*v3*w1^4*w2)*(k2 + w2^2)*(k3 + w1^4) -
        (2*v3*w1^4*w2^3)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    - h*((2*v2*k1*w1)/(k1^2 + w1^2)^2 + ((4*v3*w1^7*w2^2)*(k2 + w2^2) - (4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 +
        w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 1 - h*(((2*v3*w1^4*w2^3)*(k3 + w1^4) - (2*v3*w1^4*w2)*(k2 +
        w2^2)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    h*d1, 0, 1 + h*d2
    ];
    w0 = wHomeier;
    [xHomeier, x_nHomeier, e_nHomeier] = systemHomeierNewtonsMethod(func, funcDerivative, w0, Tolerance,
        maxK);
    wHomeier = xHomeier;
    w1Homeier = [w1Homeier, wHomeier(1)];
    w2Homeier = [w2Homeier, wHomeier(2)];
    w3Homeier = [w3Homeier, wHomeier(3)];
    totalIterations = totalIterations + length(e_nHomeier);
    totalError = totalError + e_nHomeier(length(e_nHomeier));
end
fprintf("\nElapsed Time: %.20f\n", cputime - T);
fprintf("\nTotal Iterations: %d\n", totalIterations);
fprintf("\nTotal Error: %.20f\n", totalError);


% Newton's Method: Weerakoon Modification

wWeerakoon = [alpha1; alpha2; alpha3];
w1Weerakoon = [alpha1];
w2Weerakoon = [alpha2];
w3Weerakoon = [alpha3];
totalIterations = 0;
totalError = 0;
T = cputime;
for i = 1:N
    fprintf("\nIteration %d:\n", i)
    ti = a + i*h;
    a1  = wWeerakoon(1);
    a2 = wWeerakoon(2);
    a3 = wWeerakoon(3);
    func = @(w1, w2, w3) [
    w1 - a1 - h*(v1 - (v2*w1^2)/(k1+w1^2) + (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v4*w1);
    w2 - a2 - h*((v2*w1^2)/(k1+w1^2) - (v3*w1^4*w2^2)/((k2 + w2^2)*(k3 + w1^4)) - v5*w2);
    w3 - a3 + h*(d1*w1 + d2*w3);
    ];
    funcDerivative = @(w1, w2, w3) [
    1 - h*(((4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 + w1^4) - (4*v3*w1^7*w2^2)*(k2 + w2^2))/((k2 + w2^2)*(k3 +
        w1^4))^2 - (2*v2*k1*w1)/(k1^2 + w1^2)^2 + v4), - h*(((2*v3*w1^4*w2)*(k2 + w2^2)*(k3 + w1^4) -
        (2*v3*w1^4*w2^3)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    - h*((2*v2*k1*w1)/(k1^2 + w1^2)^2 + ((4*v3*w1^7*w2^2)*(k2 + w2^2) - (4*v3*w1^3*w2^2)*(k2 + w2^2)*(k3 +
        w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 1 - h*(((2*v3*w1^4*w2^3)*(k3 + w1^4) - (2*v3*w1^4*w2)*(k2 +
        w2^2)*(k3 + w1^4))/((k2 + w2^2)*(k3 + w1^4))^2), 0;
    h*d1, 0, 1 + h*d2
    ];
```

```
    w0 = wWeerakoon;
    [xWeerakoon, x_nWeerakoon, e_nWeerakoon] = systemWeerakoonNewtonsMethod(func, funcDerivative, w0,
        Tolerance, maxK);
    wWeerakoon = xWeerakoon;
    w1Weerakoon = [w1Weerakoon, wWeerakoon(1)];
    w2Weerakoon = [w2Weerakoon, wWeerakoon(2)];
    w3Weerakoon = [w3Weerakoon, wWeerakoon(3)];
    totalIterations = totalIterations + length(e_nWeerakoon);
    totalError = totalError + e_nWeerakoon(length(e_nWeerakoon));
end
fprintf("\nElapsed Time: %.20f\n", cputime - T);
fprintf("\nTotal Iterations: %d\n", totalIterations);
fprintf("\nTotal Error: %.20f\n", totalError);


figure;
plot(t, w1Newtons, 'displayname', 'w_1(t_i): N2');
hold on
plot(t, w1Kou, 'displayname', 'w_1(t_i): K3');
plot(t, w1Homeier, 'displayname', 'w_1(t_i): H3');
plot(t, w1Weerakoon, 'displayname', 'w_1(t_i): W3');
hold off
xlabel('t');
ylabel('y');
legend('location', 'best');
grid on;


figure;
plot(t, w2Newtons, 'displayname', 'w_2(t_i): N2');
hold on
plot(t, w2Kou, 'displayname', 'w_2(t_i): K3')
plot(t, w2Homeier, 'displayname', 'w_2(t_i): H3')
plot(t, w2Weerakoon, 'displayname', 'w_2(t_i): W3')
hold off
xlabel('t');
ylabel('y');
legend('location', 'best');
grid on;


figure;
plot(t, w3Newtons, 'displayname', 'w_3(t_i): N2');
hold on
plot(t, w3Kou, 'displayname', 'w_3(t_i): K3')
plot(t, w3Homeier, 'displayname', 'w_3(t_i): H3')
plot(t, w3Weerakoon, 'displayname', 'w_3(t_i): W3')
hold off
xlabel('t');
ylabel('y');
legend('location', 'best');
grid on;


figure;
plot(t(3001:5001), w1Weerakoon(3001:5001), 'displayname', 'w_1(t_i): W3');
hold on
plot(t(3001:5001), w2Weerakoon(3001:5001), 'displayname', 'w_2(t_i): W3');
plot(t(3001:5001), w3Weerakoon(3001:5001), 'displayname', 'w_3(t_i): W3');
hold off
xlabel('t');
ylabel('y');
legend('location', 'best');
```

```
grid on;

figure;
plot(w1Weerakoon, w2Weerakoon);
xlabel('w_1(t_i)');
ylabel('w_2t_i)');
grid on;
```