

FACULTY OF ENGINEERING AND TECHNOLOGY

THE REPORT OF MATLAB ASSIGNMENT DONE BY GROUP TWO MEMBERS

COURES: COMPUTER PROGRAMMING

GROUP 2

LINK: https://github.com/matlab-group-2/group-2-matlab

SUBMISSION DATE: ............/............../............

## ACKNOWLEDGEMENT

First and foremost, we would like to take this opportunity to thank the Almighty Father for the grace of life He has given to us and also giving us knowledge and ideas during the process of doing the assignment. We also want to thank each and every group two members for their effort and sacrifices they made during the due course of doing the assignment.

## DEDICATION

We dedicate this report to Almighty God who gave us the knowledge, good health and protection during the time of doing the assignment. We are really grateful to Him for what He has done in our lives.

Also, we want to dedicate this report to our dear beloved lecturer **ENG. BENEDICTO MASERUKA** for the knowledge he is giving us as far as the course unit is concerned.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STUDENT NAME** | **COURSE** | **REGISTRATION NUMBER** | **GITHUB LINK** | **SIGNATURE** |
| Chelimo Sandra | WAR | BU/UP/2024/5402 | https://github.com/chelimosandra6-png/Chelimo-Sandra-.git |  |
| Nakazibwe Ethel | AMI | BU/UP/2024/0835 | https://github.com/ethelnakazibwe9/Nakazibwe-Ethel-.git |  |
| Poffia Adongo | AMI | BU/UP/2024/0823 | https://github.com/padongo240823-ops/Poffia2004#:~:text=https%3A//github.com/padongo240823%2Dops/Poffia%2D2004.git |  |
| Ogutu Daniel Wafula | WAR | BU/UP/2024/1060 | https://github.com/ogutudanielwafula-collab/ogutu-daniel-wafula |  |
| Owor Hamidu | PTI | BU/UP/2024/2600 | https://github.com/hamiduowor/hamiduowor.git |  |
| Muhangi Mouris Mathew | MEB | BU/UP/2024/5345 | https://github.com/mourismuhangi-prog/Muhangi-Mouris-Mathew-#:~:text=https%3A//github.-,com,-  /mourismuhangi%2Dprog/Muhangi |  |
| Odongo Joseph Aryong | AMI | BU/UP/2024/0840 | https://github.com/odongjoseph12/joseph.git |  |
| Obur Charles | APE | BU/UP/2024/3743 | https://github.com/charlesobur86/Obur-Charles-.git |  |
| Onanyang Francis | WAR | BU/UP/2024/4757 | https://github.com/onanyangfrancis8  00-ux/ONANYANG-FRANCIS-.git |  |
| Econi Ronald | AMI | BU/UG/2024/5092 | https://github.com/econironald543-  design/ECONI-RONALD-.git |  |

## APPROVAL

We are presenting this report which has been written and produced under our own effort to be approved by our lecturer of computer programming.

Lecturer;

Name: .......................................................................................................................

Signature: .................................................................................................................

Date: .........................................................................................................................

## ABSTRACT

This report explores the numerical methods for solving mathematical functions and differential equations using MATLAB, focusing on algorithm development, control structures, and modular programming as outlined in Modules 1-4.

The study examines various numerical approximation techniques, including Newton-Raphson for root-finding, and several methods for solving differential equations, such as Euler’s method and the Runge-Kutta methods.

Practical real-world problems are employed to illustrate the application of these methods, allowing for a comparative analysis of their accuracy and computational efficiency. Each numerical method is tested on the same set of problems to facilitate a clear comparison with analytical solutions.

Graphical representations are provided to visualize the discrepancies between numerical and analytical solutions, alongside a detailed examination of computation times for each method. This work not only demonstrates the relevance of numerical methods in solving complex problems but also highlights the importance of algorithmic efficiency in computational mathematics.

TABLE OF CONTENTS

Table of Contents

[ACKNOWLEDGEMENT ii](#_Toc210111359)

[DEDICATION iii](#_Toc210111360)

[APPROVAL iv](#_Toc210111361)

[ABSTRACT v](#_Toc210111362)

[CHAPTER ONE 1](#_Toc210111363)

[1.1 INTRODUCTION 1](#_Toc210111364)

[1.2 PART A 1](#_Toc210111365)

[1.2.1 DEFINING THE FUNCTIONS AND DERIVATIVES 1](#_Toc210111366)

[1.2.2 BISECTION METHOD 2](#_Toc210111367)

[1.2.3 NEWTON RAPHSON METHOD 4](#_Toc210111368)

[1.2.4 SECANT METHOD 7](#_Toc210111369)

[1.2.5 FIXED POINT ITERATION 9](#_Toc210111370)

[1.3 PART B 13](#_Toc210111371)

[1.4 INTRODUCTION 13](#_Toc210111372)

[1.4.1 PARAMETERS USED 13](#_Toc210111373)

[1.4.2 EULER METHOD 15](#_Toc210111374)

[1.4.3 RUNGE-KUTTA METHOD 16](#_Toc210111375)

[1.4.4 HEUN’S METHOD 20](#_Toc210111376)

[1.4.5 MIDPOINT METHOD 21](#_Toc210111377)

[1.5 REFERENCES 23](#_Toc210111378)

[1.6 CONCLUSION 24](#_Toc210111379)

# CHAPTER ONE

## INTRODUCTION

In this question, we explore the application of numerical methods and algorithm development techniques to solve real-world engineering and scientific problems. The assignment focuses on two major areas of computational mathematics:

1. Numerical approximation methods for finding the solutions to nonlinear equations, such as Newton–Raphson, Bisection, and Secant methods.
2. Numerical methods for solving differential equations, such as Euler’s Method and Runge–Kutta Methods.

The approach taken in this work emphasizes the use of MATLAB as a powerful tool for implementing algorithms, applying control structures, and managing modular programming. By developing and executing various algorithms, we aim to demonstrate how numerical techniques can approximate analytical solutions where closed-form solutions are difficult or impossible to obtain.

Additionally, the study applies the above methods to practical real-world problems, enabling us to compare their accuracy, convergence rates, and computational efficiency. Graphical analysis will also be used to visualize and compare the analytical solutions with numerical results, alongside an evaluation of the computation time for each method.

## PART A

### DEFINING THE FUNCTIONS AND DERIVATIVES

f=@(x) x.^5-2\*x.^3+3\*x-1;

df=@(x) 5\*x.^4-6\*x^2+3;

#### Explanation:

f = @(x): This command defines an anonymous function named f. The @ symbol indicates that it is a function handle. The variable x is the input to this function.

x.^5 - 2\*x.^3 + 3\*x - 1: This is the mathematical expression for the polynomial function. The. ^ operator is used for element-wise exponentiation, meaning it can handle arrays as input.

df = @(x): Similar to f, this defines the derivative of the function f.

5\*x.^4 - 6\*x.^2 + 3: This expression calculates the derivative of the polynomial, which will be used in methods like Newton-Raphson.

### BISECTION METHOD

This is a numerical method used to find the root of an equation f(x) = 0. It is based on repeatedly having an interval where the function changes sign.

a=0; b=1;

tol=1e-6;

max1=50;

tic

for k=1:max1

c=(a+b)/2;

if f(c)==0 || (b-a)/2 <tol

break;

end

if f(a)\*f(c)<0

b=c;

else

a=c;

end

end

root4bisection =c;

time4bisection =toc;

#### Initialization

a = 0; b = 1;

tol = 1e-6;

max1 = 50;

* a = 0; b = 1; Initializes the endpoints of the interval \ ([a, b] \) where the function will be evaluated. The root is expected to lie within this interval.
* tol = 1e-6; Sets the tolerance level for convergence. This value determines how close the approximation of the root must be to zero to consider the method successful.
* max1 = 50; Specifies the maximum number of iterations the algorithm will perform. This prevents infinite loops in case the method does not converge.

#### Iterative Process

tic

for k = 1:max1

c = (a + b) / 2;

* tic: Starts a timer to measure the execution time of the algorithm, which will be stopped by toc.
* for k = 1:max1: Begins a for-loop that will iterate from 1 to max1. Each iteration represents one step in the bisection process.
* c = (a + b) / 2; Calculates the midpoint of the interval \ ([a, b] \). This point c is where the function will be evaluated to check for a root.

#### Checking for Roots

if f(c) == 0 || (b - a) / 2 < tol

break;

end

* if f(c) == 0: Checks if the function value at c is zero, which means c is a root.
* (b - a) / 2 < tol: Checks if the width of the interval is smaller than the tolerance. If true, the method is considered to have converged, and the loop will break.

#### Updating the Interval

if f(a) \* f(c) < 0

b = c;

else

a = c;

end

* if f(a) \* f(c) < 0: Determines if the root lies between a and c by checking if the product of f(a) and f(c) is negative. If it is, the root is in the interval \ ([a, c] \).
* b = c; or a = c; Updates the interval based on where the root is believed to be.

#### Storing Results

root4bisection = c;

time4bisection = toc;

* root4bisection = c; Stores the approximate root found by the bisection method.
* time4bisection = toc; Stops the timer and records the elapsed time since tic was called.

### NEWTON RAPHSON METHOD

The **Newton–Raphson method** (also called the **Newton method**) is an **iterative numerical technique** used to find **roots** (solutions) of real-valued equations. In other words, it helps solve equations of the form f(x)=0 by successively improving an initial guess.

x0=0.5;

tic

for k=1:max1

x1=x0-f(x0)/df(x0);

if abs(x1-x0)<tol

break;

end

x0=x1;

end

root4newton=x1;

time4newton=toc;

#### Initialization

x0 = 0.5;

tic

* x0 = 0.5; Sets the initial guess for the root. This value should ideally be close to the actual root for the method to converge effectively.
* tic: Starts a timer to measure the execution time for the Newton-Raphson method.

#### Iterative Process

for k = 1:max1

x1 = x0 - f(x0) / df(x0);

* for k = 1:max1: Initiates a loop that will run up to max1 iterations.
* x1 = x0 - f(x0) / df(x0); Calculates the next approximation of the root using the Newton-Raphson formula. This formula uses the current guess x0, the function value f(x0), and its derivative df(x0).

#### Convergence

if abs (x1 - x0) < tol

break;

end

* if abs (x1 - x0) < tol: Checks if the change between successive approximations is less than the tolerance. If true, the method has converged, and the loop will be exited.

**Storing Results**

x0 = x1;

* x0 = x1; Updates the guess for the next iteration.

#### Final Results

root4newton = x1;

time4newton = toc;

* root4newton = x1; Stores the root found by the Newton-Raphson method.
* time4newton = toc; Stops the timer and records the elapsed time.

Root finding methods: Bisection vs Newton Raphson Method

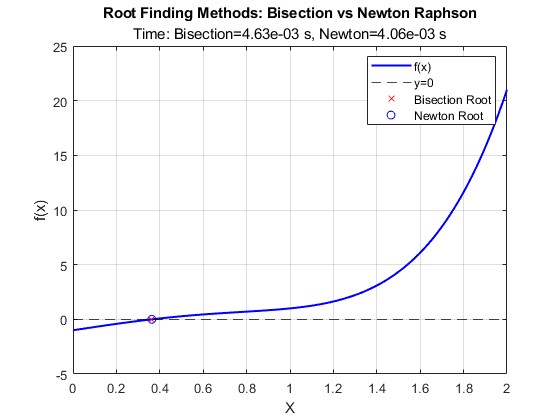


Figure Root finding method

plot(x, f(x), 'b-', 'LineWidth', 1.5);

hold on;

yline(0, 'k--');

* plot(x, f(x), 'b-', 'LineWidth', 1.5); Plots the function f(x) against x with a blue line ('b-') and a line width of 1.5.
* hold on; Retains the current plot so that additional plots can be added without erasing the existing one.
* yline(0, 'k--');Draws a horizontal dashed line at \( y = 0 \) (the x-axis) in black('k--')

#### Highlighting Roots

plot(root4bisection, f(root4bisection), 'rx', 'DisplayName', 'Bisection Root');

* plot(root4bisection, f(root4bisection), 'rx', 'DisplayName', 'Bisection Root'); Plots the root found by the bisection method as a red 'x' marker on the graph.

#### Adding Legends and Labels

legend('f(x)', 'y=0', 'Bisection Root', 'Newton Root');

xlabel('X'); ylabel('f(x)');

title('Root Finding Methods: Bisection vs Newton Raphson');

* legend(...): Creates a legend that labels the various elements in the plot.
* xlabel('X'); ylabel('f(x)'); Sets the labels for the x-axis and y-axis respectively.
* title(...): Adds a title to the plot.

#### Displaying Execution Time

subtitle(sprintf('Time: Bisection=%.2e s, Newton=%.2e s', time4bisection, time4newton));

grid on;

* subtitle(...): Displays a subtitle that shows the execution time for both the bisection and Newton methods using formatted output.
* grid on; Adds a grid to the plot for better readability.

### SECANT METHOD

The secant method is an iterative numerical technique used to find the root of a function by repeatedly drawing a straight line (secant) through two successive approximations of the root and using the x-intercept of that line as the next approximation. It is essentially a derivative-free version of the Newton–Raphson method.

f2 = @(x) sin(x)-x.^2+1;

x0 = 1.0; x1 = 2.0;

tic

for k = 1:max1

x2 = x1 - f2(x1)\*(x1 - x0)/(f2(x1) - f2(x0));

if abs(x2 - x1) < tol

break;

end

x0 = x1;

x1 = x2;

end

root4secant = x2;

time4secant = toc;

#### Function Definition and Initialization

f2 = @(x) sin(x) - x.^2 + 1;

x0 = 1.0; x1 = 2.0;

tic

* f2 = @(x): Defines another anonymous function f2 representing a different equation.
* x0 = 1.0; x1 = 2.0; Initializes two initial guesses for the roots, which are required for the secant method.
* tic: Starts a timer for measuring execution time.

#### Iterative Process

for k = 1:max1

x2 = x1 - f2(x1) \* (x1 - x0) / (f2(x1) - f2(x0));

* for k = 1:max1: Begins a loop that will run up to max1 iterations.
* x2 = x1 - f2(x1) \* (x1 - x0) / (f2(x1) - f2(x0)); Applies the secant formula to find the next approximation x2. The formula approximates the derivative using the values of the function at x1 and x0.

#### Convergence Check

if abs(x2 - x1) < tol

break;

end

* if abs(x2 - x1) < tol: Checks if the change between iterations is smaller than the tolerance. If so, the method has converged.

#### Updating Values

x0 = x1;

x1 = x2

* x0 = x1; Updates x0 for the next iteration.
* x1 = x2; Updates x1 for the next iteration.

#### Final Results

root4secant = x2;

time4secant = toc;

* root4secant = x2; Records the root found by the secant method.
* time4secant = toc; Stops the timer and records the elapsed time.

### FIXED POINT ITERATION

Fixed Point Iteration is an iterative numerical method used to solve equations of the form x=g(x)x = g(x)x=g(x) by repeatedly substituting the current estimate into g(x)g(x)g(x) to produce the next estimate, until the values converge.

It’s called “fixed point” because you’re looking for a point x∗x^\*x∗ where g(x∗)=x∗g(x^\*) = x^\*g(x∗)=x∗.

g = @(x) sqrt(sin(x)+1);

x0 = 1.0;

tic

for k = 1:max1

x1 = g(x0);

if abs(x1 - x0) < tol

break;

end

x0 = x1;

end

root4fixed = x1;

time4fixed = toc;

#### Function Definition and Initialization

g = @(x) sqrt(sin(x) + 1);

x0 = 1.0;

tic

* g = @(x): Defines the function g for fixed-point iteration.
* x0 = 1.0; Sets the initial guess for the root.
* tic: Starts a timer for measuring execution time.

#### Iterative Process

for k = 1:max1

x1 = g(x0);

* for k = 1:max1: Initiates a loop that will run up to max1 iterations.
* x1 = g(x0); Computes the next value using the fixed-point iteration formula.

#### Convergence Check

if abs(x1 - x0) < tol

break;

end

* if abs(x1 - x0) < tol: Checks if the difference between successive approximations is within the tolerance.

#### Updating Values

x0 = x1;

* x0 = x1; Updates the current guess for the next iteration.

#### Final Results

root4fixed = x1;

time4fixed = toc;

* root4fixed = x1; Stores the root found by the fixed-point iteration.
* time4fixed = toc; Stops the timer and records the elapsed time.

**ROOT FINDING METHODS: SECANT VS FIXED-POINT ITERATION**

x = linspace(0,2,100);

figure;

plot(x, f2(x), 'b-', 'LineWidth', 1.5); hold on;

yline(0, 'k--');

plot(root4secant, f2(root4secant), 'r\*', 'MarkerSize', 8, 'DisplayName','Secant Root');

plot(root4fixed, f2(root4fixed), 'gs', 'MarkerSize', 8, 'DisplayName','Fixed-Point Root');

legend('f2(x)', 'y=0','Secant Root','Fixed-Point Root');

xlabel('x'); ylabel('f2(x)');

title('Root Finding Methods: Secant vs Fixed-Point Iteration');

subtitle(sprintf('Time: Secant=%.2e s, Fixed=%.2e s',time4secant,time4fixed));

grid on;

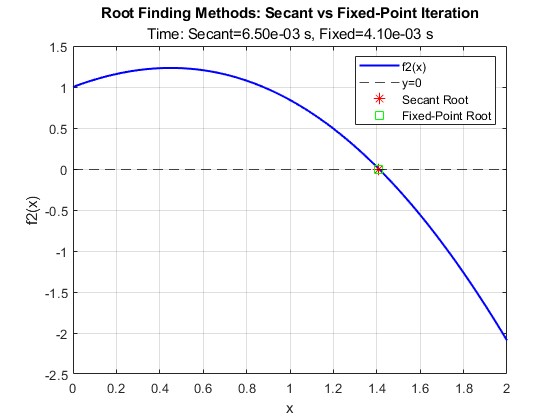


Figure Secant vs fix-point iteration

#### Visualization Setup

x = linspace(0, 2, 100);

figure;

* x = linspace(0, 2, 100); Generates a vector x with 100 points evenly spaced between 0 and 2 for plotting.
* figure; Opens a new figure window for the plot.

**Plotting the Function**

plot(x, f2(x), 'b-', 'LineWidth', 1.5);

hold on;

yline(0, 'k--');

* plot(x, f2(x), 'b-', 'LineWidth', 1.5); Plots the function f2(x) against x with a blue line.
* hold on; Retains the current plot to add more elements.
* yline(0, 'k--'); Draws a dashed line at \( y = 0 \) (the x-axis) in black.

#### Highlighting Roots

plot(root4secant, f2(root4secant), 'r\*', 'MarkerSize', 8, 'DisplayName', 'Secant Root');

plot(root4fixed, f2(root4fixed), 'gs', 'MarkerSize', 8, 'DisplayName', 'Fixed-Point Root');

* plot(root4secant, f2(root4secant), 'r\*', ...\*\*: Marks the root found by the secant method with a red star.
* plot(root4fixed, f2(root4fixed), 'gs', ...\*\*: Marks the root found by the fixed-point iteration with a green square.

#### Adding Legends and Labels

legend('f2(x)', 'y=0', 'Secant Root', 'Fixed-Point Root');

xlabel('x'); ylabel('f2(x)');

title('Root Finding Methods: Secant vs Fixed-Point Iteration');

* legend(...): Creates a legend to label the elements of the plot.
* xlabel('x'); ylabel('f2(x)'); Sets labels for the x-axis and y-axis.
* title(...): Adds a title to the plot.

#### Displaying Execution Time

subtitle(sprintf('Time: Secant=%.2e s, Fixed=%.2e s', time4secant, time4fixed));

grid on;

* subtitle(...): Displays a subtitle showing the execution times for the secant and fixed-point methods.
* grid on; Adds a grid to the plot for better readabilit

## PART B

## INTRODUCTION

Here we looked at different methods of solving differential equations numerically. Which include Euler, Runge-Kutta and many others.

They have been explained below

### PARAMETERS USED

T1 = 90;

Environmental Temperature

T\_env = 25;

v=T\_env;

Cooling rate

k = 0.5;

Interval

tspan = 0:1:60;

Differential Equation

f = @(T) -k\*(T-v);

#### Parameter Definitions

T1 = 90; Initial temperature

T\_env = 25; Environmental temperature

v = T\_env; Set v to the environmental temperature

k = 0.5; Cooling rate constant

tspan = 0:1:60; Time span from 0 to 60 minutes in 1-minute intervals

#### Explanation:

* T1 = 90: This variable represents the initial temperature of the object being cooled, set to 90 degrees Celsius.
* T\_env = 25: This is the constant environmental temperature surrounding the object, set to 25 degrees Celsius.
* v = T\_env: The variable v is assigned the value of T\_env, indicating that v represents the equilibrium temperature the object will approach as it cools.
* k = 0.5: This variable defines the cooling rate constant, which determines how quickly the object cools. A higher k means faster cooling.
* tspan = 0:1:60: This creates an array of time points from 0 to 60 minutes, with a step size of 1 minute. The numerical methods will use these time points to calculate the temperature at each moment.

#### Differential Equation

f = @(T) -k\*(T-v);

f = @(T) -k\*(T-v); % Define the differential equation

#### Explanation:

* f = @(T) -k\*(T-v): This line defines an anonymous function f that represents the rate of change of temperature. According to Newton's Law of Cooling:
* The rate of cooling of an object is proportional to the difference between its temperature \( T \) and the surrounding temperature \( v \).
* The negative sign indicates that as the temperature of the object \( T \) approaches the environmental temperature \( v \), the rate of change becomes smaller.

### EULER METHOD

The Euler method is a numerical technique used to approximate the solution of ordinary differential equations (ODEs). It's a first-order method, meaning it uses the current estimate to calculate the next estimate.

T\_euler = zeros(size(tspan));

T\_euler(1) = T1;

tic

for i = 1:length(tspan)-1

T\_euler(i+1) = T\_euler(i) + f(T\_euler(i))\*(tspan(i+1)-tspan(i));

end

time4euler = toc;

T\_euler = zeros(size(tspan)); Initialize array for Euler method results

T\_euler(1) = T1; Set the initial temperature

tic; Start timer for performance measurement

for i = 1:length(tspan)-1

T\_euler(i+1) = T\_euler(i) + f(T\_euler(i)) \* (tspan(i+1) - tspan(i));

end

time4euler = toc; Stop timer and record the time taken

#### Explanation:

* T\_euler = zeros(size(tspan)): Initializes an array T\_euler to store the temperature values calculated using the Euler method. It is the same size as tspan.
* T\_euler(1) = T1: Sets the first element of T\_euler to the initial temperature \( T1 \).
* tic: Starts a timer to measure the execution time of the method.
* for i = 1:length(tspan)-1: Loops through each time step, excluding the last one because it will calculate the next temperature.
* T\_euler(i+1) = T\_euler(i) + f(T\_euler(i)) \* (tspan(i+1) - tspan(i)):
* This line updates the temperature at the next time step based on the current temperature.
* The term f(T\_euler(i)) computes the rate of change at the current temperature.
* The product f(T\_euler(i)) \* (tspan(i+1) - tspan(i)) gives the change in temperature over the interval, which is added to the current temperature.
* time4euler = toc: Stops the timer and stores the execution time in time4euler.

### RUNGE-KUTTA METHOD

T\_rk4 = zeros(size(tspan));

T\_rk4(1) = T1;

tic

for i = 1:length(tspan)-1

h = tspan(i+1)-tspan(i);

k1 = f(T\_rk4(i));

k2 = f(T\_rk4(i) + 0.5\*h\*k1);

k3 = f(T\_rk4(i) + 0.5\*h\*k2);

k4 = f(T\_rk4(i) + h\*k3);

T\_rk4(i+1) = T\_rk4(i) + (h/6)\*(k1 + 2\*k2 + 2\*k3 + k4);

end

time4rk4 = toc;

T\_exact = v + (T1-v)\*exp(-k\*tspan);

T\_rk4 = zeros(size(tspan)); Initialize array for RK4 results

T\_rk4(1) = T1; Set the initial temperature

tic Start timer

for i = 1:length(tspan)-1

h = tspan(i+1) - tspan(i); Calculate time step

k1 = f(T\_rk4(i)); Estimate slope at initial point

k2 = f(T\_rk4(i) + 0.5\*h\*k1); Estimate slope at midpoint

k3 = f(T\_rk4(i) + 0.5\*h\*k2); Estimate slope at midpoint again

k4 = f(T\_rk4(i) + h\*k3); Estimate slope at end point

T\_rk4(i+1) = T\_rk4(i) + (h/6)\*(k1 + 2\*k2 + 2\*k3 + k4); Update temperature

end

time4rk4 = toc; Stop timer and record time taken

#### Explanation:

* T\_rk4 = zeros(size(tspan)): Initializes an array T\_rk4 to store the temperature values calculated using the RK4 method.
* T\_rk4(1) = T1: Sets the first element of T\_rk4 to the initial temperature \( T1 \).
* tic: Starts the timer.
* for i = 1:length(tspan)-1: Loops through each time step.
* h = tspan(i+1) - tspan(i): Calculates the time step size.
* k1 = f(T\_rk4(i)): Computes the first slope at the current temperature.
* k2 = f(T\_rk4(i) + 0.5\*h\*k1): Computes the slope at the midpoint using the first slope.
* k3 = f(T\_rk4(i) + 0.5\*h\*k2): Computes the slope at the midpoint again, but using the second slope.
* k4 = f(T\_rk4(i) + h\*k3): Computes the slope at the end of the interval using the third slope.
* T\_rk4(i+1) = T\_rk4(i) + (h/6)\*(k1 + 2\*k2 + 2\*k3 + k4):
* Updates the temperature for the next time step using a weighted average of the four slopes:
* \( k1 \) contributes 1/6,
* \( k2 \) contributes 2/6,
* \( k3 \) contributes 2/6,
* \( k4 \) contributes 1/6.
* time4rk4 = toc: Stops the timer for the RK4 method.

**A plot showing Euler, Runge\_Kutta and Exact**

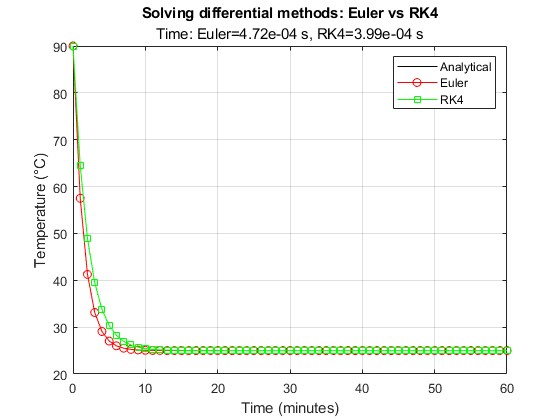


Figure A plot showing Euler, Runge-Kutta and Analytical

figure;

plot(tspan, T\_exact, 'k-');

hold on;

plot(tspan, T\_euler, 'ro-');

plot(tspan, T\_rk4, 'gs-');

xlabel('Time (minutes)');

ylabel('Temperature (°C)');

title('Solving differential methods: Euler vs RK4');

subtitle(sprintf('Time: Euler=%.2e s, RK4=%.2e s',time4euler,time4rk4));

legend('Analytical','Euler','RK4');

grid on;

figure; Create a new figure

plot(tspan, T\_exact, 'k-'); Plot the analytical solution in black

hold on; Hold the plot for overlaying additional data

plot(tspan, T\_euler, 'ro-');Plot Euler method results in red circles

plot(tspan, T\_rk4, 'gs-'); Plot RK4 results in green squares

xlabel('Time (minutes)'); Label x-axis

ylabel('Temperature (°C)');Label y-axis

title('Solving differential methods: Euler vs RK4'); Title for the plot

subtitle(sprintf('Time: Euler=%.2e s, RK4=%.2e s', time4euler, time4rk4)); Subtitle showing execution time

legend('Analytical', 'Euler', 'RK4'); Legend for the plot

grid on; Add grid to the plot.

#### Explanation:

* figure: Initializes a new figure window for plotting the results.
* plot(tspan, T\_exact, 'k-'): Plots the analytical solution (exact solution) in black.
* hold on: Keeps the current plot so that additional data can be overlaid.
* plot(tspan, T\_euler, 'ro-'): Plots the results from the Euler method using red circles connected by lines.
* plot(tspan, T\_rk4, 'gs-'): Plots the results from the RK4 method using green squares connected by lines.
* xlabel('Time (minutes)'): Labels the x-axis to indicate that it represents time in minutes.
* ylabel('Temperature (°C)'): Labels the y-axis to indicate that it represents temperature in degrees Celsius.
* title('Solving differential methods: Euler vs RK4'): Sets the title of the plot to describe what is being shown.
* subtitle(sprintf('Time: Euler=%.2e s, RK4=%.2e s', time4euler, time4rk4)): Adds a subtitle that displays the execution times for the Euler and RK4 methods.
* legend('Analytical', 'Euler', 'RK4'): Adds a legend to the plot to identify which line corresponds to which method.
* grid on: Enables a grid on the plot for better readability.

### HEUN’S METHOD

T\_heun = zeros(size(tspan));

T\_heun(1) = T1;

tic

for i = 1:length(tspan)-1

h = tspan(i+1)-tspan(i);

k1 = f(T\_heun(i));

k2 = f(T\_heun(i) + h\*k1);

T\_heun(i+1) = T\_heun(i) + (h/2)\*(k1 + k2);

end

time4heun = toc;

**Heun's Method**

T\_heun = zeros(size(tspan)); Initialize array for Heun's method

T\_heun(1) = T1; Set the initial temperature

tic Start timer

for i = 1:length(tspan)-1

h = tspan(i+1) - tspan(i); Calculate time step

k1 = f(T\_heun(i)); Estimate slope at the initial point

k2 = f(T\_heun(i) + h\*k1); Estimate slope at the end of the interval

T\_heun(i+1) = T\_heun(i) + (h/2)\*(k1 + k2); Update temperature

end

time4heun = toc; Stop timer and record time taken

#### Explanation:

* T\_heun = zeros(size(tspan)): Initializes an array for Heun's method.
* T\_heun(1) = T1: Sets the initial temperature.
* tic: Starts the timer.
* for i = 1:length(tspan)-1: Loops through each time step.
* h = tspan(i+1) - tspan(i): Calculates the time step size.
* k1 = f(T\_heun(i)): Computes the first slope at the current temperature.
* k2 = f(T\_heun(i) + h\*k1): Computes the slope at the end of the interval using the first slope.
* T\_heun(i+1) = T\_heun(i) + (h/2)\*(k1 + k2):
* Updates the temperature using the average of the slopes at the start and end of the

### MIDPOINT METHOD

The Midpoint Method is a numerical technique for solving ordinary differential equations (ODEs) that improves on Euler’s method by estimating the slope at the midpoint of the interval instead of at the beginning.

T\_mid = zeros(size(tspan));

T\_mid(1) = T1;

tic

for i = 1:length(tspan)-1

h = tspan(i+1)-tspan(i);

k1 = f(T\_mid(i));

k2 = f(T\_mid(i) + 0.5\*h\*k1);

T\_mid(i+1) = T\_mid(i) + h\*k2;

end

time4mid = toc;

T\_mid = zeros(size(tspan)); Initialize array for Midpoint method

T\_mid(1) = T1; Set the initial temperature

tic ; Start timer

for i = 1:length(tspan)-1

h = tspan(i+1) - tspan(i); Calculate time step

k1 = f(T\_mid(i)); Estimate slope at the initial point

k2 = f(T\_mid(i) + 0.5\*h\*k1); Estimate slope at the midpoint

T\_mid(i+1) = T\_mid(i) + h\*k2; Update temperature

end

time4mid = toc; Stop timer and record time taken

#### Explanation:

* T\_mid = zeros(size(tspan)): Initializes an array for the Midpoint method.
* T\_mid(1) = T1: Sets the initial temperature.
* tic: Starts the timer.
* for i = 1:length(tspan)-1: Loops through each time step.
* h = tspan(i+1) - tspan(i): Calculates the time step size.
* k1 = f(T\_mid(i)): Computes the slope at the current temperature.
* k2 = f(T\_mid(i) + 0.5\*h\*k1): Computes the slope at the midpoint of the interval using the first slope.
* T\_mid(i+1) = T\_mid(i) + h\*k2: Updates the temperature using the slope at the midpoint, which typically yields better results than the basic Euler method.
* time4mid = toc: Stops the timer.

Heun, Midpoint and Analytical

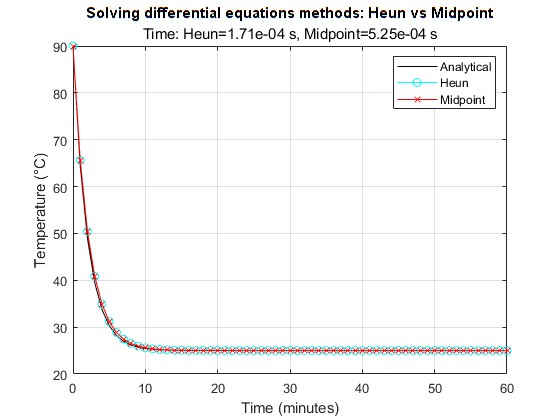


Figure A graph showing Heun, Midpoint and Analytical

figure;

plot(tspan, T\_exact, 'k-');

hold on;

plot(tspan, T\_heun, '-co');

plot(tspan, T\_mid, 'rx-');

xlabel('Time (minutes)');

ylabel('Temperature (°C)');

title('Solving differential equations methods: Heun vs Midpoint');

subtitle(sprintf('Time: Heun=%.2e s, Midpoint=%.2e s',time4heun,time4mid));

legend('Analytical','Heun','Midpoint');

grid on;

## REFERENCES

1. Chapra, S. C. (2018). Applied Numerical Methods with MATLAB for Engineers and Scientists. McGraw-Hill Education.
2. Palm, W. J. (2018). Introduction to MATLAB for Engineers. McGraw-Hill Education.
3. Gilat, A. (2017). MATLAB: An Introduction with Applications. John Wiley & Sons.
4. Newton, I. (1701). Scala graduum caloris et frigoris.
5. Incropera, F. P., & DeWitt, D. P. (2002). Fundamentals of Heat and Mass Transfer. John Wiley & Sons.

## CONCLUSION

This MATLAB code has provided a comprehensive exploration of numerical methods for solving differential equations, specifically Newton's Law of Cooling. Through the implementation of Euler, Runge-Kutta (RK4), Heun's, and Midpoint methods, we've gained valuable insights into the strengths and limitations of each approach. The code's visual representations and execution time comparisons have further enhanced our understanding of these methods' performance.

This knowledge can be applied to various real-world problems, such as modelling population growth, chemical reactions, or electrical circuits. By mastering these numerical methods, we can develop more accurate and efficient solutions to complex problems, ultimately driving innovation and progress in fields like engineering, physics, and beyond.