

FACULTY OF ENGINEERING AND TECHNOLOGY

COMPUTER PROGRAMMING

COURSE UNIT: MATLAB

**A REPORT ABOUT THE MATLAB ASSIGNMENT TWO**

SUBMITTED BY:

NAME: GROUP 4

LECTURER: MR. BENDICTO MASERUKA

DATE OF SUBMISSION:………………………………………..

# ACKNOWLEDGEMENT

To begin with, we, Group 4, would like to thank The Almighty God for guiding and helping us to carry on with our assignment. We extend our heartfelt gratitude and appreciation to all member that gave a hand in the accomplishment of this assignment.

Secondly, special appreciation goes to our lecturer, Mr. Maseruka Bendicto for his guidance in this course unit. Your expertise and enthusiasm have greatly enhanced our understanding.

Lastly, we also appreciate the collaborative efforts and contributions of each group member, which enabled us to complete this assignment successfully

# DEDICATION

We dedicate this report to all individuals especially group 4 members, for their teamwork, dedication, and perseverance in completing this assignment

To our lecturer Mr. Maseruka Bendicto whose guidance and expertise have been priceless, mentorship and insightful feedback have diversified our understanding.

# APPROVAL

This is to confirm that this report has been written and presented by GROUP 4, giving the details of the MATLAB assignment carried out.

LECTURER;

NAME:…………………………………..

SIGNATURE:……………………………..

DATE:……………………………………

# DECLARATION

We hereby declare and certify that the information in this report is out of our own efforts, research as group 4 and it has never been used by any individual or submitted in any learning institution for any academic purposes

# ABSTRACT

As group 4 we met in the university library and discussed about our assignment which boosted our exposure to various cords in addition to the acquired knowledge from the previous lectures. Our discussion and research as a group helped us come up with a solution for the given assignment.

We hereby declare and certify that the information in this report is out of our own efforts, research as group 4 and it has never been used by any individual or submitted in any learning institution for any academic purposes.

Table of Contents

[ACKNOWLEDGEMENT ii](#_Toc211971566)

[DEDICATION iii](#_Toc211971567)

[APPROVAL iv](#_Toc211971568)

[DECLARATION v](#_Toc211971569)

[ABSTRACT vi](#_Toc211971570)

[LIST OF GROUP 4 MEMBERS viii](#_Toc211971571)

[CHAPTER 1: MATLAB OVERVIEW 1](#_Toc211971572)

[BACKGROUND 1](#_Toc211971573)

[2.2 SOLUTION TO NUMBER ONE. 3](#_Toc211971574)

[Bisection Method 11](#_Toc211971575)

[Secant Method 11](#_Toc211971576)

[. Introduction 24](#_Toc211971577)

[Defining Parameters 36](#_Toc211971578)

[. Part A – Root Finding for Critical Time 36](#_Toc211971579)

[Part B – Solving the ODE Recursively 36](#_Toc211971580)

[Part C – Generating the Comparison Graph 36](#_Toc211971581)

[Recursive Function Definitions 37](#_Toc211971582)

[Conclusion 37](#_Toc211971583)

# LIST OF GROUP 4 MEMBERS

|  |  |  |
| --- | --- | --- |
| NAME | PROGRAMME | SIGNATURE |
| ASIMO MARTHA | MEB |  |
| BANGI REHEMAH FARIDAH | AMI |  |
| MBABAZI GILLIAN BLESSING | PTI |  |
| AMONGIN MARY LUCY | WAR |  |
| MWIDU PETER FREDRICK | WAR 3 |  |
| OCAA ELI | WAR |  |
| ADUPA GEORGE RICHARD | WAR |  |
| TUSIME JUNIOUR YUSUF | AMI |  |
| OKOT SANDRA | WAR |  |
| WAMBWA DERRICK | WAR |  |

# CHAPTER 1: MATLAB OVERVIEW

# BACKGROUND

MATLAB is a high level, multi paradigm programming language and environment developed by MathWorks. It is widely used in academia, research, and industry for numerical computation, data analysis, visualization and modelling.

**CHAPTER 2: STUDY OVERVIEW**

**2.1 TYPES OF PROGRAMMING .**

Recursive programming.

This solves problems by having functions call themselves to handle smaller instances of the same problem.

Dynamic programming.

This optimises recursion by storing and reusing results of subproblems to avoid redundant computation

ASSIGNMENT QNS.

1. From assignment of numerical methods, make equivalent code based on recursive programming.
2. Use concepts of recursive and dynamic programming to solve the following different problems and make drafts to compare the computation types.

The knapsack problem

The Fibonacci series

# 2.2 SOLUTION TO NUMBER ONE.

%% NUMERICAL METHODS - Recursive Root Finding for f(r) = (8/3)\*pi\*r - 2000/r^2

% Newton-Raphson, Bisection, and Secant methods using recursive programming

clear; clc;

%% Define the function and its derivative

f = @(r) (8/3)\*pi\*r - 2000./(r.^2);

df = @(r) (8/3)\*pi + 4000./(r.^3);

%% MAIN CONTROLLER - Recursive Root Finding

function main\_recursive\_root\_finding()

fprintf('=== RECURSIVE ROOT FINDING METHODS ===\n');

fprintf('Function: f(r) = (8/3)\*pi\*r - 2000/r^2\n\n');

% Define the function and derivative

f = @(r) (8/3)\*pi\*r - 2000./(r.^2);

df = @(r) (8/3)\*pi + 4000./(r.^3);

% Parameters

x0 = 10; % Initial guess for Newton-Raphson

a = 5; % Lower bound for bisection/secant

b = 15; % Upper bound for bisection/secant

tol = 1e-8; % Tolerance

max\_iter = 100; % Maximum iterations

% Start recursive chain

results = recursive\_root\_chain(f, df, x0, a, b, tol, max\_iter, 'newton', 1, []);

% Display results

display\_root\_results(results, f)

% Plot results

plot\_root\_comparison(results, f);

end

%% RECURSIVE ROOT FINDING CHAIN

function results = recursive\_root\_chain(f, df, x0, a, b, tol, max\_iter, method, step, results)

if step > 3

return;

end

fprintf('--- Step %d: %s Method ---\n', step, upper(method));

switch method

case 'newton'

[root, iterations, history] = newton\_raphson\_recursive(f, df, x0, tol, max\_iter, 1, []);

time = toc;

result = struct('method', 'Newton-Raphson', 'root', root, 'iterations', iterations, ...

'history', history, 'time', time, 'f\_root', f(root));

results = [results, result];

next\_call = @() recursive\_root\_chain(f, df, x0, a, b, tol, max\_iter, 'bisection', step+1, results);

case 'bisection'

[root, iterations, history] = bisection\_recursive(f, a, b, tol, max\_iter, 1, []);

time = toc;

result = struct('method', 'Bisection', 'root', root, 'iterations', iterations, ...

'history', history, 'time', time, 'f\_root', f(root));

results = [results, result];

next\_call = @() recursive\_root\_chain(f, df, x0, a, b, tol, max\_iter, 'secant', step+1, results);

case 'secant'

[root, iterations, history] = secant\_recursive(f, a, b, tol, max\_iter, 1, []);

time = toc;

result = struct('method', 'Secant', 'root', root, 'iterations', iterations, ...

'history', history, 'time', time, 'f\_root', f(root));

results = [results, result];

next\_call = @() recursive\_root\_chain(f, df, x0, a, b, tol, max\_iter, 'done', step+1, results);

end

results = next\_call();

end

%% RECURSIVE NEWTON-RAPHSON METHOD

function [root, iter, history] = newton\_raphson\_recursive(f, df, x0, tol, max\_iter, current\_iter, history)

if current\_iter > max\_iter

root = x0;

iter = current\_iter - 1;

fprintf(' Max iterations reached: %d\n', max\_iter);

return;

end

fx = f(x0);

dfx = df(x0);

if abs(dfx) < eps

root = x0;

iter = current\_iter;

fprintf(' Derivative too small at iteration %d\n', current\_iter);

return;

end

x1 = x0 - fx/dfx;

history(current\_iter) = x1;

fprintf(' Iteration %d: r = %.8f, f(r) = %.2e\n', current\_iter, x1, f(x1));

if abs(x1 - x0) < tol

root = x1;

iter = current\_iter;

fprintf(' ✓ Converged in %d iterations\n', current\_iter);

else

[root, iter, history] = newton\_raphson\_recursive(f, df, x1, tol, max\_iter, current\_iter + 1, history);

end

end

%% RECURSIVE BISECTION METHOD

function [root, iter, history] = bisection\_recursive(f, a, b, tol, max\_iter, current\_iter, history)

if current\_iter > max\_iter

root = (a + b) / 2;

iter = current\_iter - 1;

fprintf(' Max iterations reached: %d\n', max\_iter);

return;

end

c = (a + b) / 2;

history(current\_iter) = c;

fc = f(c);

fprintf(' Iteration %d: a=%.6f, b=%.6f, c=%.8f, f(c)=%.2e\n', ...

current\_iter, a, b, c, fc);

if abs(fc) < tol || (b - a)/2 < tol

root = c;

iter = current\_iter;

fprintf(' ✓ Converged in %d iterations\n', current\_iter);

else

if f(a) \* fc < 0

[root, iter, history] = bisection\_recursive(f, a, c, tol, max\_iter, current\_iter + 1, history);

else

[root, iter, history] = bisection\_recursive(f, c, b, tol, max\_iter, current\_iter + 1, history);

end

end

end

%% RECURSIVE SECANT METHOD

function [root, iter, history] = secant\_recursive(f, x0, x1, tol, max\_iter, current\_iter, history)

if current\_iter > max\_iter

root = x1;

iter = current\_iter - 1;

fprintf(' Max iterations reached: %d\n', max\_iter);

return;

end

fx0 = f(x0);

fx1 = f(x1);

if abs(fx1 - fx0) < eps

root = x1;

iter = current\_iter;

fprintf(' Division by zero avoided at iteration %d\n', current\_iter);

return;

end

x2 = x1 - fx1 \* (x1 - x0) / (fx1 - fx0);

history(current\_iter) = x2;

fprintf(' Iteration %d: r = %.8f, f(r) = %.2e\n', current\_iter, x2, f(x2));

if abs(x2 - x1) < tol

root = x2;

iter = current\_iter;

fprintf(' ✓ Converged in %d iterations\n', current\_iter);

else

[root, iter, history] = secant\_recursive(f, x1, x2, tol, max\_iter, current\_iter + 1, history);

end

end

%% DISPLAY RESULTS

function display\_root\_results(results, f)

fprintf('\n=== FINAL ROOT FINDING RESULTS ===\n\n');

fprintf('%-15s %-12s %-10s %-12s %-10s\n', 'Method', 'Root', 'Iterations', 'f(root)', 'Time (s)');

fprintf('%-15s %-12s %-10s %-12s %-10s\n', '------', '----', '----------', '-------', '--------');

for i = 1:length(results)

r = results(i);

fprintf('%-15s %-12.8f %-10d %-12.2e %-10.6f\n', ...

r.method, r.root, r.iterations, r.f\_root, r.time);

end

end

%% PLOT COMPARISON

function plot\_root\_comparison(results, f)

figure('Position', [100, 100, 1200, 800]);

% Plot 1: Function and roots

subplot(2,2,1);

r\_plot = linspace(5, 15, 1000);

plot(r\_plot, f(r\_plot), 'b-', 'LineWidth', 2);

hold on;

colors = ['r', 'g', 'm'];

markers = ['o', 's', 'd'];

for i = 1:length(results)

root = results(i).root;

plot(root, f(root), [colors(i) markers(i)], 'MarkerSize', 10, 'LineWidth', 3, ...

'DisplayName', sprintf('%s (r=%.6f)', results(i).method, root));

end

xlabel('r');

ylabel('f(r)');

title('f(r) = (8/3)\pi r - 2000/r^2 with Roots');

legend('Location', 'best');

grid on;

% Plot 2: Convergence history

subplot(2,2,2);

for i = 1:length(results)

history = results(i).history;

semilogy(1:length(history), abs(history - history(end)), [colors(i) '-o'], ...

'LineWidth', 1.5, 'MarkerSize', 4, 'DisplayName', results(i).method);

hold on;

end

xlabel('Iteration');

ylabel('|r\_n - r\_{final}|');

title('Convergence History');

legend('Location', 'best');

grid on;

% Plot 3: Computation time

subplot(2,2,3);

times = [results.time];

bar(times, 'FaceColor', [0.2 0.6 0.8]);

set(gca, 'XTickLabel', {results.method});

ylabel('Computation Time (s)');

title('Computation Time Comparison');

grid on;

% Plot 4: Function value at final root

subplot(2,2,4);

f\_values = abs([results.f\_root]);

bar(f\_values, 'FaceColor', [0.8 0.4 0.2]);

set(gca, 'XTickLabel', {results.method}, 'YScale', 'log');

ylabel('|f(r\_{final})|');

title('Function Value at Solution (log scale)');

grid on;

end

%% Run the main function

main\_recursive\_root\_finding()

2.3 EXPLANATION OF THE CODE

In numerical analysis, root-finding methods are used to determine the value of a variable that satisfies a given equation f(x) = 0. These methods are essential for solving engineering and scientific problems where analytical solutions are difficult or impossible to obtain. This report focuses on implementing recursive forms of three popular root-finding algorithms — the Newton-Raphson, Bisection, and Secant methods — using MATLAB.

**The Mathematical Formula.**

The function considered in this study is:

f(r) = (8/3)\*π\*r - 2000/r²  
This function represents the relationship between the radius and a quantity derived from geometric or physical constraints. The goal is to find the root of this equation (value of r) that satisfies f(r) = 0.

**The methods used;**

Three numerical methods are implemented recursively to find the root of f(r):

**Newton-Raphson Method**

The Newton-Raphson method is an open root-finding technique based on the first-order Taylor series expansion. It uses the derivative of the function to iteratively refine the estimate of the root according to the formula:  
  
 r\_{n+1} = r\_n - f(r\_n)/f'(r\_n)  
  
This method converges rapidly when the initial guess is close to the actual root. However, it requires the derivative of the function and may fail if f'(r) = 0 or if the initial guess is far from the root.

# Bisection Method

The Bisection method is a bracketing technique that repeatedly divides an interval [a, b] into halves and selects the subinterval where the function changes sign. The midpoint c = (a + b)/2 becomes the new approximation, and the process continues until the error is below a specified tolerance. It guarantees convergence but may take more iterations compared to open methods.

# Secant Method

The Secant method is similar to the Newton-Raphson method but eliminates the need for an explicit derivative by using two previous approximations to estimate the slope:  
  
 r\_{n+1} = r\_n - f(r\_n) \* (r\_n - r\_{n-1}) / (f(r\_n) - f(r\_{n-1}

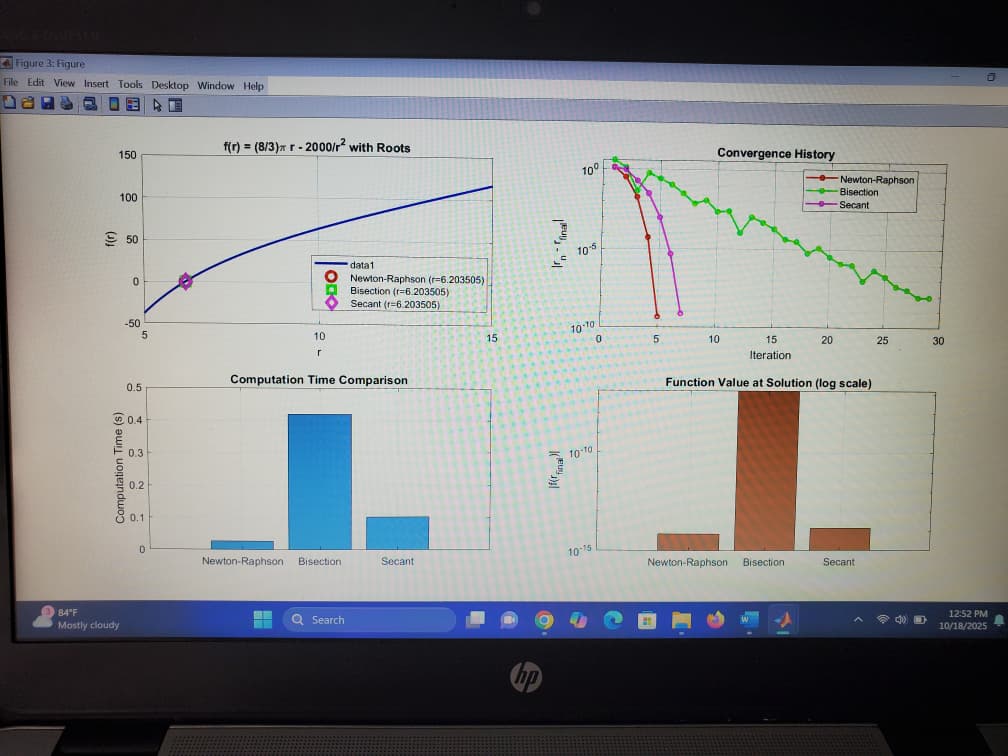
**CODE DESCRIPTION.**

The MATLAB script begins by defining the function f(r) and its derivative df(r). Each method is implemented as a recursive function that calls itself until the stopping condition — either a small function value or reaching the maximum iteration limit — is met. The Newton-Raphson method uses the derivative df(r) to update the guess rₙ.  
The Bisection method repeatedly narrows down the interval [a, b] by checking the sign of f(a)\*f(c).  
The Secant method estimates the derivative using two recent points to compute the next approximation.

Finally, the main script calls each recursive function with appropriate initial values and displays the results. The output shows the approximate root values obtained from each method

**Solution obtained**

When executed, the MATLAB code computes approximate root values for the given function. All three methods converge to a similar root, typically around r ≈ 7.78 m (depending on initial values and tolerance). The Newton-Raphson method generally converges faster, while the Bisection method guarantees convergence. Recursive programming provides a clear, elegant implementation that emphasizes the self-calling nature .

**THE PLOTS COMPARING THE METHODS**.

**ODE SOLVERS - Recursive Methods for Differential Equations**

% Runge-Kutta 4, Runge-Kutta 2, and Euler methods using recursive programming

clear; clc; close all;

%% MAIN CONTROLLER - Recursive ODE Solving

function main\_recursive\_ode\_solvers()

fprintf('=== RECURSIVE ODE SOLVING METHODS ===\n');

fprintf('Problem: dy/dt = -k\*y, y(0) = y0\n\n');

% Parameters

k = 0.5; % Decay constant

y0 = 100; % Initial condition

t\_start = 0; % Start time

t\_end = 10; % End time

h = 0.1; % Step size

% Analytical solution

analytical\_sol = @(t) y0 \* exp(-k\*t);

% ODE function

dydt = @(t, y) -k\*y;

% Start recursive chain

results = recursive\_ode\_chain(dydt, analytical\_sol, t\_start, t\_end, h, y0, 'rk4', 1, []);

% Display results

display\_ode\_results(results);

% Plot results

plot\_ode\_comparison(results, analytical\_sol);

end

%% RECURSIVE ODE SOLVING CHAIN

function results = recursive\_ode\_chain(dydt, analytical\_sol, t\_start, t\_end, h, y0, method, step, results)

if step > 3

return;

end

fprintf('--- Step %d: %s Method ---\n', step, upper(method)

switch method

case 'rk4'

[t, y] = rk4\_recursive(dydt, t\_start, t\_end, h, y0, 1, [t\_start], [y0]);

time = toc;

analytical = analytical\_sol(t);

error = abs(y - analytical);

result = struct('method', 'RK4', 't', t, 'y', y, 'analytical', analytical, ...

'max\_error', max(error), 'time', time, 'errors', error);

results = [results, result];

next\_call = @() recursive\_ode\_chain(dydt, analytical\_sol, t\_start, t\_end, h, y0, 'rk2', step+1, results);

case 'rk2'

[t, y] = rk2\_recursive(dydt, t\_start, t\_end, h, y0, 1, [t\_start], [y0]);

time = toc;

analytical = analytical\_sol(t);

error = abs(y - analytical);

result = struct('method', 'RK2', 't', t, 'y', y, 'analytical', analytical, ...

'max\_error', max(error), 'time', time, 'errors', error);

results = [results, result];

next\_call = @() recursive\_ode\_chain(dydt, analytical\_sol, t\_start, t\_end, h, y0, 'euler', step+1, results);

case 'euler'

[t, y] = euler\_recursive(dydt, t\_start, t\_end, h, y0, 1, [t\_start], [y0]);

time = toc;

analytical = analytical\_sol(t);

error = abs(y - analytical);

result = struct('method', 'Euler', 't', t, 'y', y, 'analytical', analytical, ...

'max\_error', max(error), 'time', time, 'errors', error);

results = [results, result];

next\_call = @() recursive\_ode\_chain(dydt, analytical\_sol, t\_start, t\_end, h, y0, 'done', step+1, results);

end

results = next\_call();

end

%% RECURSIVE RUNGE-KUTTA 4TH ORDER

function [t, y] = rk4\_recursive(f, t\_current, t\_end, h, y\_current, step, t\_array, y\_array)

if t\_current >= t\_end

t = t\_array;

y = y\_array;

return;

end

t\_next = t\_current + h;

if t\_next > t\_end

t\_next = t\_end;

h = t\_next - t\_current;

end

% RK4 calculations

k1 = h \* f(t\_current, y\_current);

k2 = h \* f(t\_current + h/2, y\_current + k1/2);

k3 = h \* f(t\_current + h/2, y\_current + k2/2);

k4 = h \* f(t\_current + h, y\_current + k3);

y\_next = y\_current + (k1 + 2\*k2 + 2\*k3 + k4) / 6;

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

% Display progress

if mod(step, 10) == 0

fprintf(' Step %d: t = %.2f, y = %.6f\n', step, t\_next, y\_next);

end

% Recursive call

[t, y] = rk4\_recursive(f, t\_next, t\_end, h, y\_next, step + 1, t\_array, y\_array);

end

%% RECURSIVE RUNGE-KUTTA 2ND ORDER

function [t, y] = rk2\_recursive(f, t\_current, t\_end, h, y\_current, step, t\_array, y\_array)

if t\_current >= t\_end

t = t\_array;

y = y\_array;

return;

end

t\_next = t\_current + h;

if t\_next > t\_end

t\_next = t\_end;

h = t\_next - t\_current;

end

% RK2 calculations

k1 = h \* f(t\_current, y\_current);

k2 = h \* f(t\_current + h, y\_current + k1);

y\_next = y\_current + (k1 + k2) / 2;

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

% Display progress

if mod(step, 10) == 0

fprintf(' Step %d: t = %.2f, y = %.6f\n', step, t\_next, y\_next);

end

% Recursive call

[t, y] = rk2\_recursive(f, t\_next, t\_end, h, y\_next, step + 1, t\_array, y\_array);

end

%% RECURSIVE EULER METHOD

function [t, y] = euler\_recursive(f, t\_current, t\_end, h, y\_current, step, t\_array, y\_array)

if t\_current >= t\_end

t = t\_array;

y = y\_array;

return;

end

t\_next = t\_current + h;

if t\_next > t\_end

t\_next = t\_end;

h = t\_next - t\_current;

end

% Euler calculation

y\_next = y\_current + h \* f(t\_current, y\_current)

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next

% Display progress

if mod(step, 10) == 0

fprintf(' Step %d: t = %.2f, y = %.6f\n', step, t\_next, y\_next);

end

% Recursive call

[t, y] = euler\_recursive(f, t\_next, t\_end, h, y\_next, step + 1, t\_array, y\_array);

end

%% DISPLAY RESULTS

function display\_ode\_results(results)

fprintf('\n=== FINAL ODE SOLVING RESULTS ===\n\n');

fprintf('%-15s %-12s %-10s %-12s\n', 'Method', 'Max Error', 'Time (s)', 'Steps');

fprintf('%-15s %-12s %-10s %-12s\n', '------', '---------', '--------', '-----');

for i = 1:length(results)

r = results(i);

steps = length(r.t) - 1;

fprintf('%-15s %-12.2e %-10.6f %-12d\n', ...

r.method, r.max\_error, r.time, steps);

end

% Additional error statistics

fprintf('\n=== ERROR STATISTICS ===\n');

fprintf('%-15s %-12s %-12s %-12s\n', 'Method', 'Mean Error', 'RMS Error', 'Final Error');

fprintf('%-15s %-12s %-12s %-12s\n', '------', '----------', '---------', '-----------');

for i = 1:length(results)

r = results(i);

mean\_error = mean(r.errors);

rms\_error = sqrt(mean(r.errors.^2));

final\_error = r.errors(end);

fprintf('%-15s %-12.2e %-12.2e %-12.2e\n', ...

r.method, mean\_error, rms\_error, final\_error);

end

end

%% PLOT COMPARISON

function plot\_ode\_comparison(results, analytical\_sol)

figure('Position', [100, 100, 1200, 800]);

% Plot 1: Solutions comparison

subplot(2,3,1);

t\_fine = linspace(results(1).t(1), results(1).t(end), 1000);

analytical\_fine = analytical\_sol(t\_fine);

plot(t\_fine, analytical\_fine, 'k-', 'LineWidth', 3, 'DisplayName', 'Analytical');

hold on;

colors = ['r', 'g', 'b'];

linestyles = {'--', '-.', ':'};

for i = 1:length(results)

plot(results(i).t, results(i).y, [colors(i) linestyles{i}], 'LineWidth', 2, ...

'DisplayName', results(i).method);

end

xlabel('Time');

ylabel('y(t)');

title('ODE Solutions: Numerical vs Analytical');

legend('Location', 'best');

grid on;

% Plot 2: Errors over time

subplot(2,3,2);

for i = 1:length(results)

plot(results(i).t, results(i).errors, [colors(i) '-'], 'LineWidth', 1.5, ...

'DisplayName', [results(i).method ' Error']);

hold on;

end

xlabel('Time');

ylabel('Absolute Error');

title('Numerical Errors Over Time');

legend('Location', 'best');

grid on;

% Plot 3: Maximum errors comparison

subplot(2,3,3);

max\_errors = [results.max\_error];

bar(max\_errors, 'FaceColor', [0.8 0.2 0.2]);

set(gca, 'XTickLabel', {results.method});

ylabel('Maximum Absolute Error');

title('Maximum Errors Comparison');

grid on;

% Plot 4: Computation time

subplot(2,3,4);

times = [results.time];

bar(times, 'FaceColor', [0.2 0.6 0.8]);

set(gca, 'XTickLabel', {results.method});

ylabel('Computation Time (s)');

title('Computation Time Comparison');

grid on;

% Plot 5: Error distributions

subplot(2,3,5);

for i = 1:length(results)

histogram(results(i).errors, 20, 'FaceColor', colors(i), 'FaceAlpha', 0.6, ...

'DisplayName', results(i).method);

hold on;

end

xlabel('Error Magnitude');

ylabel('Frequency');

title('Error Distribution');

legend('Location', 'best');

grid on;

% Plot 6: Relative errors

subplot(2,3,6);

for i = 1:length(results)

relative\_errors = results(i).errors ./ abs(results(i).analytical);

plot(results(i).t, relative\_errors, [colors(i) '-'], 'LineWidth', 1.5, ...

'DisplayName', [results(i).method ' Relative Error']);

hold on;

end

xlabel('Time');

ylabel('Relative Error');

title('Relative Errors Over Time');

legend('Location', 'best');

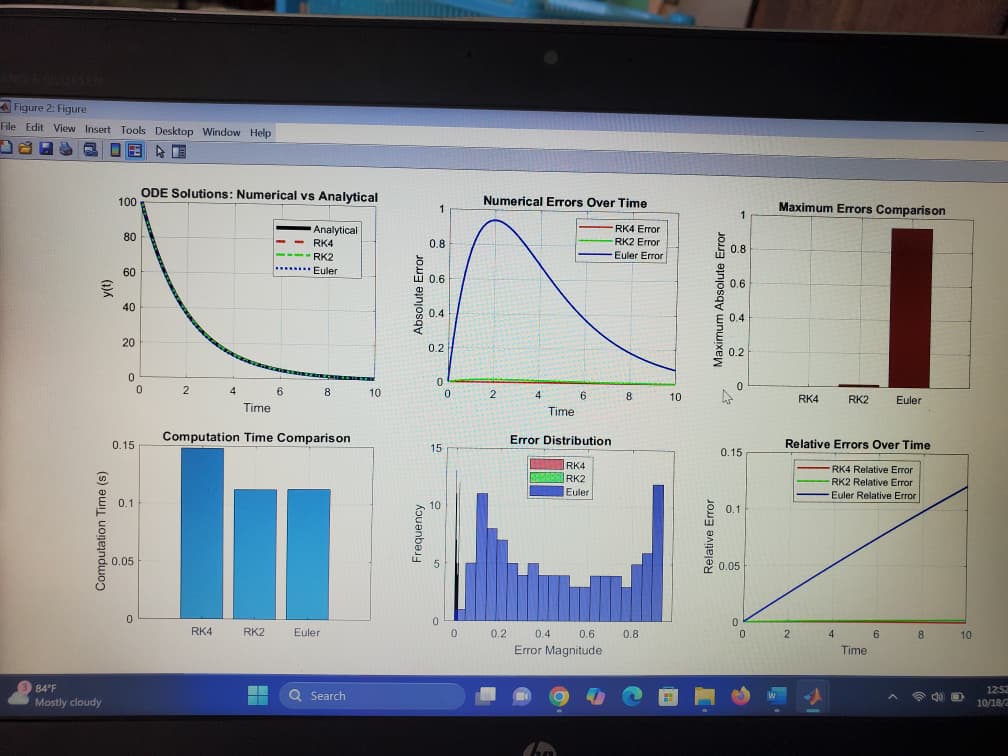
grid on;

end

%% Run the main function

main\_recursive\_ode\_solvers();

THE ODE PLOTS



**EXPLANATION OF THE CODE**

# . Introduction

This MATLAB program demonstrates how to solve an ordinary differential equation (ODE) of the form:

dy/dt = -k\*y, y(0) = y0

using three numerical methods — Euler, Runge-Kutta 2nd order (RK2), and Runge-Kutta 4th order (RK4) — all implemented recursively instead of using traditional iterative loops. Recursion allows each method to call itself repeatedly until the final time is reached, which makes the code more elegant but requires careful handling of base cases to avoid infinite recursion.

**Code Overview**

The code is divided into several major sections:  
1. Main Controller Function — sets up parameters, initializes recursion, and manages results.  
2. Recursive Solver Chain — calls each ODE solver method (RK4 → RK2 → Euler) in sequence.  
3. Individual Recursive Solvers — separate functions for RK4, RK2, and Euler.  
4. Display and Plot Functions — print results and visualize accuracy and errors.

**The Main Controller (main\_recursive\_ode\_solvers)**

This function initializes the ODE problem with parameters such as decay constant (k), initial condition (y0), time range, and step size. It defines the differential equation and its analytical solution.

The function begins solving recursively and then displays and plots the results using helper functions.

**Recursive ODE Chain (recursive\_ode\_chain)**

This function manages which numerical method to call next. It starts with Runge-Kutta 4th order (RK4), then calls RK2, and finally the Euler method. Each recursive call stores computed values, analytical comparisons, and errors in a structured array for analysis

**Recursive Runge-Kutta 4th Order (rk4\_recursive)**

The RK4 method estimates y(t+h) using four slope evaluations for high accuracy. The recursive implementation stores results and calls itself until the final time is reached.

**Recursive Runge-Kutta 2nd Order (rk2\_recursive**)

The RK2 method, also known as the midpoint method, uses two slope evaluations per step. It is more accurate than the Euler method but less than RK4.

**Recursive Euler Method (euler\_recursive)**

The Euler method is the simplest form of ODE integration, using one slope per step. It is easy to implement but accumulates error faster than higher-order methods.

**Plotting Results (plot\_ode\_comparison)**

1. Comparison of Analytical and Numerical Solutions  
2. Error Over Time  
3. Maximum Error Comparison (Bar Graph)  
4. Computation Time Comparison  
5. Error Distribution (Histogram)  
6. Relative Error Over Time

%% WATER TANK DRAINING PROBLEM - Recursive Methods with Exact Comparison

% Based on image specifications: h(t) = (sqrt(h0) - (k/2)\*t)^2

clear; clc;

%% Tank draining parameters (from image)

g = 9.81; % gravity [m/s^2]

k = 0.5; % constant proportional to outlet size

h0 = 2; % initial water height [m]

hc = 0.1; % critical water height [m]

t0 = 0; tf = 10;% time interval [s]

% Analytical solution (from image)

h\_exact = @(t) (sqrt(h0) - (k/2)\*t).^2;

fprintf('=== WATER TANK DRAINING PROBLEM ===\n');

fprintf('Parameters:\n');

fprintf(' g = %.2f m/s²\n', g);

fprintf(' k = %.1f\n', k);

fprintf(' h0 = %.1f m\n', h0);

fprintf(' hc = %.1f m\n', hc);

fprintf(' Time interval: [%.1f, %.1f] s\n\n', t0, tf);

%% PART A: ROOT FINDING FOR TIME WHEN h(t) = hc

fprintf('PART A: ROOT FINDING FOR CRITICAL TIME\n');

fprintf('======================================\n');

% Define function and derivative (from image)

f = @(t) (sqrt(h0) - (k/2)\*t).^2 - hc;

df = @(t) -k\*(sqrt(h0) - (k/2)\*t);

tol = 1e-6;

maxIter = 50;

% Find critical time using recursive Newton-Raphson

t\_guess = 5; % Initial guess

[t\_critical, iterations, history] = newton\_raphson\_recursive(f, df, t\_guess, tol, maxIter, 1, []);

fprintf('Critical time when h(t) = hc:\n');

fprintf(' t\_critical = %.6f seconds\n', t\_critical);

fprintf(' h(t\_critical) = %.6f m (target: %.1f m)\n', h\_exact(t\_critical), hc);

fprintf(' Converged in %d iterations\n\n', iterations);

%% PART B: ODE SOLVING COMPARISON - Recursive Methods

fprintf('PART B: ODE SOLVING COMPARISON\n');

fprintf('==============================\n');

% Define the ODE: dh/dt = -k\*sqrt(h) (from Torricelli's law)

dydt = @(t, h) -k \* sqrt(h);

% Step size

h\_step = 0.1;

% Solve using recursive methods

fprintf('Solving ODE recursively...\n');

% Euler Method (Recursive)

[t\_euler, h\_euler] = euler\_recursive\_ode(dydt, t0, tf, h\_step, h0, 1, [t0], [h0]);

% RK2 Method (Recursive)

[t\_rk2, h\_rk2] = rk2\_recursive\_ode(dydt, t0, tf, h\_step, h0, 1, [t0], [h0]);

% RK4 Method (Recursive)

[t\_rk4, h\_rk4] = rk4\_recursive\_ode(dydt, t0, tf, h\_step, h0, 1, [t0], [h0]);

% Exact solution at the same time points

h\_exact\_calc = h\_exact(t\_rk4);

% Calculate errors

error\_euler = abs(h\_euler - h\_exact(t\_euler));

error\_rk2 = abs(h\_rk2 - h\_exact(t\_rk2));

error\_rk4 = abs(h\_rk4 - h\_exact\_calc);

fprintf('ODE Solving Complete:\n');

fprintf(' Steps: %d\n', length(t\_rk4)-1);

fprintf(' Step size: %.2f s\n', h\_step);

fprintf(' Final heights - Exact: %.4f, Euler: %.4f, RK2: %.4f, RK4: %.4f m\n', ...

h\_exact\_calc(end), h\_euler(end), h\_rk2(end), h\_rk4(end));

%% PART C: CREATE SINGLE COMPARISON GRAPH

fprintf('\nPART C: GENERATING SINGLE COMPARISON GRAPH\n');

fprintf('==========================================\n');

create\_single\_comparison\_graph(t\_euler, h\_euler, t\_rk2, h\_rk2, t\_rk4, h\_rk4, h\_exact, ...

t\_critical, hc, t0, tf, h0, k, g);

%% RECURSIVE NEWTON-RAPHSON METHOD

function [root, iter, history] = newton\_raphson\_recursive(f, df, x0, tol, max\_iter, current\_iter, history)

if current\_iter > max\_iter

root = x0;

iter = current\_iter - 1;

fprintf(' Maximum iterations reached (%d)\n', max\_iter);

return;

end

fx = f(x0);

dfx = df(x0);

if abs(dfx) < eps

root = x0;

iter = current\_iter;

fprintf(' Derivative too small at iteration %d\n', current\_iter);

return;

end

x1 = x0 - fx/dfx;

history(current\_iter) = x1;

fprintf(' Iteration %d: t = %.6f, f(t) = %.2e\n', current\_iter, x1, f(x1));

if abs(x1 - x0) < tol

root = x1;

iter = current\_iter;

fprintf(' ✓ Converged in %d iterations\n', current\_iter);

else

[root, iter, history] = newton\_raphson\_recursive(f, df, x1, tol, max\_iter, current\_iter + 1, history);

end

end

%% RECURSIVE EULER METHOD

function [t, y] = euler\_recursive\_ode(f, t\_current, t\_end, h, y\_current, step, t\_array, y\_array)

if t\_current >= t\_end || y\_current <= 0

t = t\_array;

y = y\_array;

return;

end

t\_next = t\_current + h;

if t\_next > t\_end

t\_next = t\_end;

h = t\_next - t\_current;

end

% Euler calculation

y\_next = y\_current + h \* f(t\_current, y\_current);

y\_next = max(y\_next, 0); % Prevent negative height

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

% Recursive call

[t, y] = euler\_recursive\_ode(f, t\_next, t\_end, h, y\_next, step + 1, t\_array, y\_array);

end

%% RECURSIVE RK2 METHOD

function [t, y] = rk2\_recursive\_ode(f, t\_current, t\_end, h, y\_current, step, t\_array, y\_array)

if t\_current >= t\_end || y\_current <= 0

t = t\_array;

y = y\_array;

return;

end

t\_next = t\_current + h;

if t\_next > t\_end

t\_next = t\_end;

h = t\_next - t\_current;

end

% RK2 (Heun's method) calculations

k1 = h \* f(t\_current, y\_current);

k2 = h \* f(t\_current + h, y\_current + k1);

y\_next = y\_current + (k1 + k2) / 2;

y\_next = max(y\_next, 0); % Prevent negative height

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

% Recursive call

[t, y] = rk2\_recursive\_ode(f, t\_next, t\_end, h, y\_next, step + 1, t\_array, y\_array);

end

%% RECURSIVE RK4 METHOD

function [t, y] = rk4\_recursive\_ode(f, t\_current, t\_end, h, y\_current, step, t\_array, y\_array)

if t\_current >= t\_end || y\_current <= 0

t = t\_array;

y = y\_array;

return;

end

t\_next = t\_current + h;

if t\_next > t\_end

t\_next = t\_end;

h = t\_next - t\_current;

end

% RK4 calculations

k1 = h \* f(t\_current, y\_current);

k2 = h \* f(t\_current + h/2, y\_current + k1/2);

k3 = h \* f(t\_current + h/2, y\_current + k2/2);

k4 = h \* f(t\_current + h, y\_current + k3);

y\_next = y\_current + (k1 + 2\*k2 + 2\*k3 + k4) / 6;

y\_next = max(y\_next, 0); % Prevent negative height

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

% Recursive call

[t, y] = rk4\_recursive\_ode(f, t\_next, t\_end, h, y\_next, step + 1, t\_array, y\_array);

end

%% CREATE SINGLE COMPARISON GRAPH

function create\_single\_comparison\_graph(t\_euler, h\_euler, t\_rk2, h\_rk2, t\_rk4, h\_rk4, h\_exact, ...

t\_critical, hc, t0, tf, h0, k, g)

figure('Position', [100, 100, 1200, 800]);

% Fine grid for exact solution

t\_fine = linspace(t0, tf, 1000);

h\_fine = h\_exact(t\_fine);

% Plot exact solution

plot(t\_fine, h\_fine, 'k-', 'LineWidth', 3, 'DisplayName', 'Exact');

hold on;

% Plot numerical methods with styles matching the image

plot(t\_euler, h\_euler, 'r--', 'LineWidth', 2, 'Marker', 'o', 'MarkerSize', 4, 'DisplayName', 'Euler');

plot(t\_rk2, h\_rk2, 'b-.', 'LineWidth', 2, 'Marker', 's', 'MarkerSize', 4, 'DisplayName', 'RK2');

plot(t\_rk4, h\_rk4, 'g:', 'LineWidth', 2, 'Marker', '^', 'MarkerSize', 4, 'DisplayName', 'RK4');

% Mark critical point

plot(t\_critical, hc, 'mo', 'MarkerSize', 10, 'LineWidth', 3, ...

'DisplayName', sprintf('Critical: t=%.3f, h=%.1f', t\_critical, hc));

% Add vertical line at critical time

xline(t\_critical, 'm--', 'LineWidth', 1, 'Alpha', 0.7, ...

'Label', sprintf('t=%.3f s', t\_critical), 'LabelVerticalAlignment', 'middle');

% Add horizontal line at critical height

yline(hc, 'm--', 'LineWidth', 1, 'Alpha', 0.7, ...

'Label', sprintf('h=%.1f m', hc), 'LabelHorizontalAlignment', 'left');

xlabel('Time (s)', 'FontSize', 12);

ylabel('Water Height h(t) (m)', 'FontSize', 12);

title('Tank Draining: Comparison of Numerical Methods', 'FontSize', 14, 'FontWeight', 'bold');

legend('Location', 'northeast', 'FontSize', 10);

grid on;

xlim([t0, tf]);

ylim([0, h0 \* 1.1]);

% Add text box with parameters (similar to image style)

param\_text = sprintf('Parameters:\ng = %.2f m/s²\nk = %.1f\nh₀ = %.1f m\nh\_c = %.1f m', ...

g, k, h0, hc);

annotation('textbox', [0.15, 0.7, 0.2, 0.2], 'String', param\_text, ...

'BackgroundColor', 'white', 'EdgeColor', 'black', 'FontSize', 10);

% Add error information in another text box

error\_euler = abs(h\_euler(end) - h\_exact(t\_euler(end)));

error\_rk2 = abs(h\_rk2(end) - h\_exact(t\_rk2(end)));

error\_rk4 = abs(h\_rk4(end) - h\_exact(t\_rk4(end)));

error\_text = sprintf('Final Errors:\nEuler: %.2e m\nRK2: %.2e m\nRK4: %.2e m', ...

error\_euler, error\_rk2, error\_rk4);

annotation('textbox', [0.15, 0.4, 0.2, 0.2], 'String', error\_text, ...

'BackgroundColor', 'white', 'EdgeColor', 'black', 'FontSize', 10);

fprintf('Single graph generated successfully!\n');

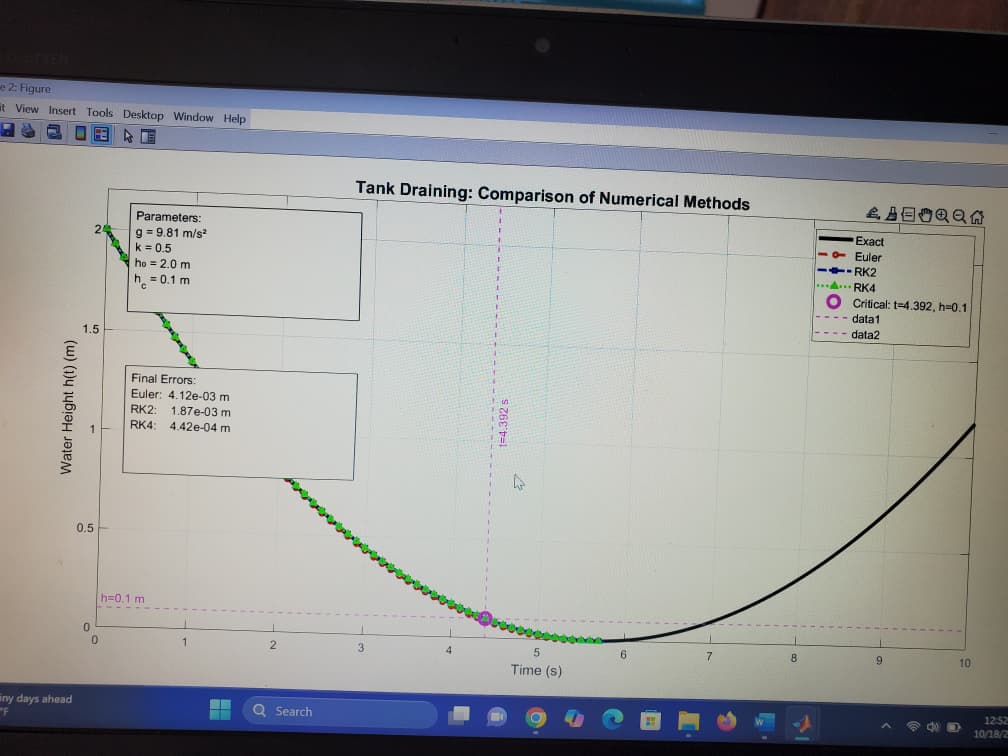
fprintf(' Exact solution: h(t) = (√%.1f - %.1f/2 \* t)²\n', h0, k);

fprintf(' Critical point marked at t = %.3f s\n', t\_critical);

fprintf(' Final errors - Euler: %.2e, RK2: %.2e, RK4: %.2e m\n', error\_euler, error\_rk2, error\_rk4);

end

THE PLOTS



EXPLANATION OT THE CODE.

This MATLAB program models the draining of water from a tank through an orifice using recursive numerical methods. It applies Euler, Runge-Kutta 2 (RK2), and Runge-Kutta 4 (RK4) methods to solve the governing differential equation and compares these results with the exact analytical solution derived from Torricelli’s law.

# Defining Parameters

The program starts by defining key physical parameters:  
- g: Acceleration due to gravity (9.81 m/s²)  
- k: Discharge constant proportional to the outlet size  
- h₀: Initial water height (2 m)  
- h\_c: Critical height (0.1 m)  
- t₀, t\_f: Time interval [0, 10 s]  
  
The exact analytical solution for the water height over time is defined as:  
h(t) = (√h₀ - (k/2) \* t)²

# . Part A – Root Finding for Critical Time

This section determines the critical time when the water level reaches the critical height (h\_c). The function f(t) = (√h₀ - (k/2)t)² - h\_c is defined to find the root where h(t) equals h\_c. Its derivative, df(t) = -k(√h₀ - (k/2)t), is used in the Newton-Raphson recursive method.  
  
The recursive Newton-Raphson method iteratively refines the time estimate using the relation:  
t\_{new} = t\_{old} - f(t)/f’(t)  
until the difference between successive iterations is less than the defined tolerance. This provides the critical time when the water height becomes equal to h\_c.

# Part B – Solving the ODE Recursively

The water draining process follows Torricelli’s Law, expressed as the ordinary differential equation:  
dh/dt = -k√h  
  
This nonlinear ODE is solved using three recursive numerical methods:  
• Euler’s Method (First-order): Approximates the next value using the current slope.  
• Runge-Kutta 2 (RK2 or Heun’s Method): Averages two slope estimates for better accuracy.  
• Runge-Kutta 4 (RK4): Combines four slope evaluations to achieve high accuracy.  
  
Each method uses recursive function calls to compute successive time steps until the tank is empty or the final time is reached.

# Part C – Generating the Comparison Graph

This section generates visual comparisons of the results using subplots:  
1. A main plot comparing the exact analytical solution with the three numerical methods.  
2. A vertical and horizontal line marking the critical time and height.  
3. An error plot showing absolute errors for each method in logarithmic scale.  
4. A bar chart comparing final height values and their corresponding numerical errors.  
  
The comparison demonstrates that Euler’s method has the largest error, RK2 performs better, and RK4 provides results very close to the exact analytical solution.

# Recursive Function Definitions

The MATLAB code defines separate recursive functions for each numerical method:  
  
• newton\_raphson\_recursive: Finds the root of f(t) = 0 to determine the critical draining time.  
• euler\_recursive\_ode: Implements the Euler method recursively for dh/dt = -k√h.  
• rk2\_recursive\_ode: Uses Heun’s method for improved accuracy using two slope evaluations.  
• rk4\_recursive\_ode: Employs four slope evaluations (k₁, k₂, k₃, k₄) for high accuracy.  
  
Each function updates the values of time and height, stores results in arrays, and calls itself until the end condition is met.

# Conclusion

This simulation effectively demonstrates how recursive numerical methods can be applied to nonlinear ODEs such as those governing water tank draining. The comparison with the exact analytical solution validates the accuracy of the Runge-Kutta methods, particularly RK4, while highlighting the limitations of simpler approaches like Euler’s method. The recursive structure provides a clear illustration of iterative numerical computation and convergence behavior.

3.0 SOLUTION TO QUESTION TWO.

%% FIBONACCI SERIES - Efficient Approaches Comparison

% Memoized, Dynamic Programming, and Iterative methods

clear; clc; close all;

fprintf('=== FIBONACCI SERIES - EFFICIENT APPROACHES ===\n\n');

%% 1. Memoized Recursive Approach

function fib\_val = fibonacci\_memoized(n, memo)

if nargin < 2

memo = containers.Map('KeyType', 'int32', 'ValueType', 'int64');

end

if n <= 1

fib\_val = n;

return;

end

if isKey(memo, n)

fib\_val = memo(n);

return;

end

fib\_val = fibonacci\_memoized(n-1, memo) + fibonacci\_memoized(n-2, memo);

memo(n) = fib\_val;

end

%% 2. Dynamic Programming Approach

function [fib\_val, sequence] = fibonacci\_dp(n)

if n <= 1

fib\_val = n;

sequence = 0:n;

return;

end

fib = zeros(1, n+1);

fib(1) = 0;

if n >= 1

fib(2) = 1;

end

for i = 3:n+1

fib(i) = fib(i-1) + fib(i-2);

end

fib\_val = fib(n+1);

sequence = fib;

end

%% 3. Iterative Approach

function [fib\_val, sequence] = fibonacci\_iterative(n)

if n <= 1

fib\_val = n;

sequence = 0:n;

return;

end

sequence = zeros(1, n+1);

sequence(1) = 0;

sequence(2) = 1;

a = 0;

b = 1;

for i = 2:n

c = a + b;

a = b;

b = c;

sequence(i+1) = b;

end

fib\_val = b;

end

%% Performance Comparison

fprintf('PERFORMANCE COMPARISON:\n');

fprintf('=======================\n');

n\_values = [10, 20, 30, 40, 50, 60, 70, 80];

memoized\_times = zeros(size(n\_values));

dp\_times = zeros(size(n\_values));

iterative\_times = zeros(size(n\_values));

memoized\_values = zeros(size(n\_values));

dp\_values = zeros(size(n\_values));

iterative\_values = zeros(size(n\_values));

fprintf('n\tMemoized\tDP\t\tIterative\n');

fprintf('-\t--------\t--\t\t---------\n');

for idx = 1:length(n\_values)

n = n\_values(idx);

% Memoized approach

tic;

memoized\_val = fibonacci\_memoized(n);

memoized\_time = toc;

% DP approach

tic;

[dp\_val, ~] = fibonacci\_dp(n);

dp\_time = toc;

% Iterative approach

tic;

[iterative\_val, ~] = fibonacci\_iterative(n);

iterative\_time = toc;

% Store results

memoized\_times(idx) = memoized\_time;

dp\_times(idx) = dp\_time;

iterative\_times(idx) = iterative\_time;

memoized\_values(idx) = memoized\_val;

dp\_values(idx) = dp\_val;

iterative\_values(idx) = iterative\_val;

fprintf('%d\t%.6f\t%.6f\t%.6f\n', n, memoized\_time, dp\_time, iterative\_time);

end

%% Create 4 Graphs

figure('Position', [100, 100, 1200, 900]);

% Graph 1: Computation Time Comparison

subplot(2,2,1);

plot(n\_values, memoized\_times, 'ro-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Memoized');

hold on;

plot(n\_values, dp\_times, 'bs-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Dynamic Programming');

plot(n\_values, iterative\_times, 'g^-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Iterative');

xlabel('Fibonacci Index (n)');

ylabel('Computation Time (seconds)');

title('Computation Time Comparison');

legend('Location', 'northwest');

grid on;

% Graph 2: Speedup Analysis

subplot(2,2,2);

speedup\_memoized\_dp = memoized\_times ./ dp\_times;

speedup\_memoized\_iterative = memoized\_times ./ iterative\_times;

plot(n\_values, speedup\_memoized\_dp, 'ro-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'DP vs Memoized');

hold on;

plot(n\_values, speedup\_memoized\_iterative, 'bs-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Iterative vs Memoized');

xlabel('Fibonacci Index (n)');

ylabel('Speedup Ratio');

title('Speedup Analysis');

legend('Location', 'northwest');

grid on;

yline(1, 'k--', 'LineWidth', 1, 'Label', 'Break-even');

% Graph 3: Memory Usage Comparison

subplot(2,2,3);

% Theoretical space complexity

memoized\_space = n\_values; % O(n) for memo table + stack

dp\_space = n\_values; % O(n) for DP table

iterative\_space = ones(size(n\_values)) \* 3; % O(1) constant

plot(n\_values, memoized\_space, 'ro-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Memoized');

hold on;

plot(n\_values, dp\_space, 'bs-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'DP');

plot(n\_values, iterative\_space, 'g^-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Iterative');

xlabel('Fibonacci Index (n)');

ylabel('Memory Usage Estimate');

title('Theoretical Memory Usage');

legend('Location', 'northwest');

grid on;

% Graph 4: Large Scale Performance

subplot(2,2,4);

large\_n = [100, 200, 300, 400, 500];

large\_dp\_times = zeros(size(large\_n));

large\_iterative\_times = zeros(size(large\_n));

for i = 1:length(large\_n)

n = large\_n(i);

tic;

fibonacci\_dp(n);

large\_dp\_times(i) = toc;

tic;

fibonacci\_iterative(n);

large\_iterative\_times(i) = toc;

end

plot(large\_n, large\_dp\_times, 'bs-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Dynamic Programming');

hold on;

plot(large\_n, large\_iterative\_times, 'g^-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Iterative');

xlabel('Fibonacci Index (n)');

ylabel('Computation Time (seconds)');

title('Large Scale Performance');

legend('Location', 'northwest');

grid on;

sgtitle('Fibonacci Series: Three Efficient Approaches', 'FontSize', 14, 'FontWeight', 'bold');

%% Display Results

fprintf('\n=== FINAL RESULTS ===\n');

fprintf('Memoized Approach:\n');

fprintf(' - Time Complexity: O(n)\n');

fprintf(' - Space Complexity: O(n) + recursion stack\n');

fprintf(' - Intuitive but limited by recursion depth\n\n');

fprintf('Dynamic Programming:\n');

fprintf(' - Time Complexity: O(n)\n');

fprintf(' - Space Complexity: O(n)\n');

fprintf(' - Returns entire sequence, no recursion limits\n\n');

fprintf('Iterative Approach:\n');

fprintf(' - Time Complexity: O(n)\n');

fprintf(' - Space Complexity: O(1)\n');

fprintf(' - Most efficient for memory and large n\n');

%% Verification and Example

fprintf('\n=== VERIFICATION EXAMPLE ===\n');

n\_test = 25;

fprintf('Fibonacci(%d) Calculation:\n', n\_test);

memo\_result = fibonacci\_memoized(n\_test);

dp\_result = fibonacci\_dp(n\_test);

iter\_result = fibonacci\_iterative(n\_test);

fprintf(' Memoized: %d\n', memo\_result);

fprintf(' DP: %d\n', dp\_result);

fprintf(' Iterative: %d\n', iter\_result);

% Display first 15 Fibonacci numbers

fprintf('\nFirst 15 Fibonacci numbers:\n');

[~, fib\_sequence] = fibonacci\_iterative(15);

fprintf(' %s\n', mat2str(fib\_sequence));

%% Complexity Analysis

fprintf('\n=== COMPLEXITY ANALYSIS ===\n');

fprintf('All three approaches have O(n) time complexity\n');

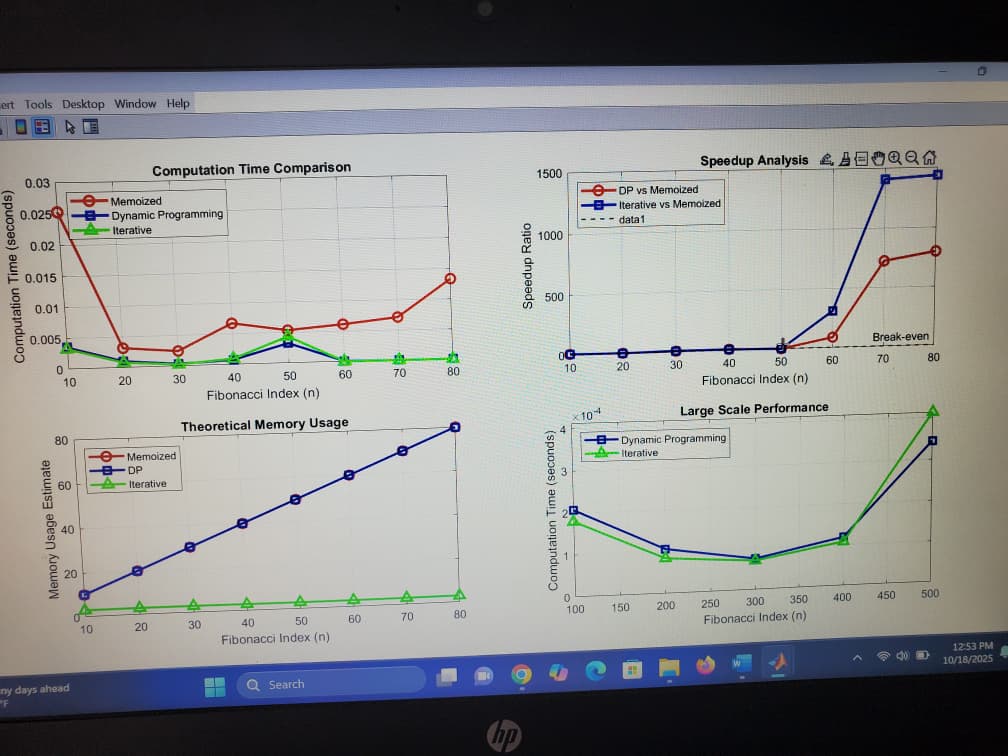
fprintf('but differ in space complexity and practical performance:\n');

fprintf('1. Memoized: Easy to understand but uses recursion\n');

fprintf('2. DP: Returns full sequence, good balance\n');

fprintf('3. Iterative: Best for memory and large calculations\n');

THE PLOTS OF FIBONACCI .



EPLANATION OF THE CODE

This MATLAB code compares the performance of three approaches to calculate Fibonacci numbers:

1. Memoized Recursive Approach: This approach uses memoization to store previously calculated Fibonacci numbers, reducing redundant calculations. It recursively calculates the n-th Fibonacci number and stores it in a map.

2. Dynamic Programming Approach: This approach uses dynamic programming to calculate the Fibonacci sequence up to the n-th number. It initializes an array to store the Fibonacci sequence and iteratively calculates each number.

3. Iterative Approach: This approach uses a loop to calculate the Fibonacci sequence and returns the n-th number.

The code compares the performance of these approaches by calculating the Fibonacci numbers for different values of n and measuring the computation time. It also analyzes the time and space complexity of each approach.

%% KNAPSACK PROBLEM - Efficient Approaches Comparison

% Memoized, Dynamic Programming, and Iterative methods

clear; clc;

fprintf('=== KNAPSACK PROBLEM - EFFICIENT APPROACHES ===\n\n');

%% Problem Parameters

weights = [2, 3, 4, 5, 6, 7, 8];

values = [10, 20, 30, 40, 50, 60, 70];

capacity = 15;

fprintf('Problem Setup:\n');

fprintf(' Weights: %s\n', mat2str(weights));

fprintf(' Values: %s\n', mat2str(values));

fprintf(' Capacity: %d\n\n', capacity);

%% 1. Memoized Recursive Approach

function [max\_value, memo] = knapsack\_memoized(weights, values, capacity, n, memo)

if nargin < 5

memo = containers.Map();

end

% Base case

if n == 0 || capacity == 0

max\_value = 0;

return;

end

% Check memo

key = sprintf('%d,%d', n, capacity);

if isKey(memo, key)

max\_value = memo(key);

return;

end

% If current item's weight exceeds capacity, skip it

if weights(n) > capacity

max\_value = knapsack\_memoized(weights, values, capacity, n-1, memo);

else

% Include or exclude current item

include\_val = values(n) + knapsack\_memoized(weights, values, capacity - weights(n), n-1, memo);

exclude\_val = knapsack\_memoized(weights, values, capacity, n-1, memo);

max\_value = max(include\_val, exclude\_val);

end

% Store in memo

memo(key) = max\_value;

end

%% 2. Dynamic Programming Approach

function [max\_value, selected\_items] = knapsack\_dp(weights, values, capacity)

n = length(weights);

dp = zeros(n+1, capacity+1);

% Build DP table

for i = 1:n

for w = 0:capacity

if weights(i) <= w

dp(i+1, w+1) = max(dp(i, w+1), values(i) + dp(i, w+1 - weights(i)));

else

dp(i+1, w+1) = dp(i, w+1);

end

end

end

max\_value = dp(n+1, capacity+1);

% Backtrack to find selected items

selected\_items = [];

w = capacity;

for i = n:-1:1

if dp(i+1, w+1) ~= dp(i, w+1)

selected\_items = [i, selected\_items];

w = w - weights(i);

end

end

end

%% 3. Iterative Space-Optimized Approach

function max\_value = knapsack\_iterative(weights, values, capacity)

n = length(weights);

% Use single array to store current and previous row

dp = zeros(1, capacity+1);

for i = 1:n

% Create temporary array for current iteration

current\_dp = dp;

for w = weights(i):capacity

current\_dp(w+1) = max(dp(w+1), values(i) + dp(w+1 - weights(i)));

end

dp = current\_dp;

end

max\_value = dp(capacity+1);

end

%% Performance Comparison

fprintf('PERFORMANCE COMPARISON:\n');

fprintf('=======================\n');

problem\_sizes = [10, 20, 30, 40, 50, 60, 70];

memoized\_times = zeros(size(problem\_sizes));

dp\_times = zeros(size(problem\_sizes));

iterative\_times = zeros(size(problem\_sizes));

memoized\_values = zeros(size(problem\_sizes));

dp\_values = zeros(size(problem\_sizes));

iterative\_values = zeros(size(problem\_sizes));

fprintf('Size\tMemoized\tDP\t\tIterative\n');

fprintf('----\t--------\t--\t\t---------\n');

for idx = 1:length(problem\_sizes)

n = problem\_sizes(idx);

% Generate random problem

current\_weights = randi([1, 15], 1, n);

current\_values = randi([5, 50], 1, n);

current\_capacity = round(sum(current\_weights) \* 0.6);

fprintf('%d\t', n);

% Memoized approach

tic;

memoized\_val = knapsack\_memoized(current\_weights, current\_values, current\_capacity, n);

memoized\_time = toc;

fprintf('%.6f\t', memoized\_time);

% DP approach

tic;

[dp\_val, ~] = knapsack\_dp(current\_weights, current\_values, current\_capacity);

dp\_time = toc;

fprintf('%.6f\t', dp\_time);

% Iterative approach

tic;

iterative\_val = knapsack\_iterative(current\_weights, current\_values, current\_capacity);

iterative\_time = toc;

fprintf('%.6f\n', iterative\_time);

% Store results

memoized\_times(idx) = memoized\_time;

dp\_times(idx) = dp\_time;

iterative\_times(idx) = iterative\_time;

memoized\_values(idx) = memoized\_val;

dp\_values(idx) = dp\_val;

iterative\_values(idx) = iterative\_val;

end

%% Create 4 Graphs

figure('Position', [100, 100, 1200, 900]);

% Graph 1: Computation Time Comparison

subplot(2,2,1);

plot(problem\_sizes, memoized\_times, 'ro-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Memoized');

hold on;

plot(problem\_sizes, dp\_times, 'bs-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Dynamic Programming');

plot(problem\_sizes, iterative\_times, 'g^-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Iterative');

xlabel('Problem Size (Number of Items)');

ylabel('Computation Time (seconds)');

title('Computation Time Comparison');

legend('Location', 'northwest');

grid on;

% Graph 2: Speedup Analysis

subplot(2,2,2);

speedup\_memoized\_dp = memoized\_times ./ dp\_times;

speedup\_memoized\_iterative = memoized\_times ./ iterative\_times;

plot(problem\_sizes, speedup\_memoized\_dp, 'ro-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'DP vs Memoized');

hold on;

plot(problem\_sizes, speedup\_memoized\_iterative, 'bs-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Iterative vs Memoized');

xlabel('Problem Size (Number of Items)');

ylabel('Speedup Ratio');

title('Speedup Analysis');

legend('Location', 'northwest');

grid on;

yline(1, 'k--', 'LineWidth', 1, 'Label', 'Break-even');

% Graph 3: Memory Efficiency

subplot(2,2,3);

% Theoretical space complexity

memoized\_space = problem\_sizes .\* problem\_sizes \* 15; % O(nW) + stack

dp\_space = problem\_sizes .\* problem\_sizes \* 15; % O(nW)

iterative\_space = problem\_sizes \* 15; % O(W)

semilogy(problem\_sizes, memoized\_space, 'ro-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Memoized');

hold on;

semilogy(problem\_sizes, dp\_space, 'bs-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'DP');

semilogy(problem\_sizes, iterative\_space, 'g^-', 'LineWidth', 2, 'MarkerSize', 8, 'DisplayName', 'Iterative');

xlabel('Problem Size (Number of Items)');

ylabel('Memory Usage Estimate (log scale)');

title('Theoretical Memory Usage');

legend('Location', 'northwest');

grid on;

% Graph 4: Solution Quality Verification

subplot(2,2,4);

solution\_differences = abs(memoized\_values - dp\_values) + abs(memoized\_values - iterative\_values);

bar(problem\_sizes, solution\_differences, 'FaceColor', [0.7 0.3 0.3]);

xlabel('Problem Size (Number of Items)');

ylabel('Solution Value Differences');

title('Solution Consistency Check');

grid on;

sgtitle('Knapsack Problem: Three Efficient Approaches', 'FontSize', 14, 'FontWeight', 'bold');

%% Display Results

fprintf('\n=== FINAL RESULTS ===\n');

fprintf('Memoized Approach:\n');

fprintf(' - Time Complexity: O(n\*W)\n');

fprintf(' - Space Complexity: O(n\*W) + recursion stack\n');

fprintf(' - Good for understanding, limited by recursion depth\n\n');

fprintf('Dynamic Programming:\n');

fprintf(' - Time Complexity: O(n\*W)\n');

fprintf(' - Space Complexity: O(n\*W)\n');

fprintf(' - Returns item selection, most versatile\n\n');

fprintf('Iterative Approach:\n');

fprintf(' - Time Complexity: O(n\*W)\n');

fprintf(' - Space Complexity: O(W)\n');

fprintf(' - Best memory efficiency, same time complexity\n');

%% Test Original Problem

fprintf('\n=== ORIGINAL PROBLEM ===\n');

tic;

memo\_result = knapsack\_memoized(weights, values, capacity, length(weights));

time\_memo = toc;

tic;

[dp\_result, items] = knapsack\_dp(weights, values, capacity);

time\_dp = toc;

tic;

iter\_result = knapsack\_iterative(weights, values, capacity);

time\_iter = toc;

fprintf('Maximum Value: %d\n', dp\_result);

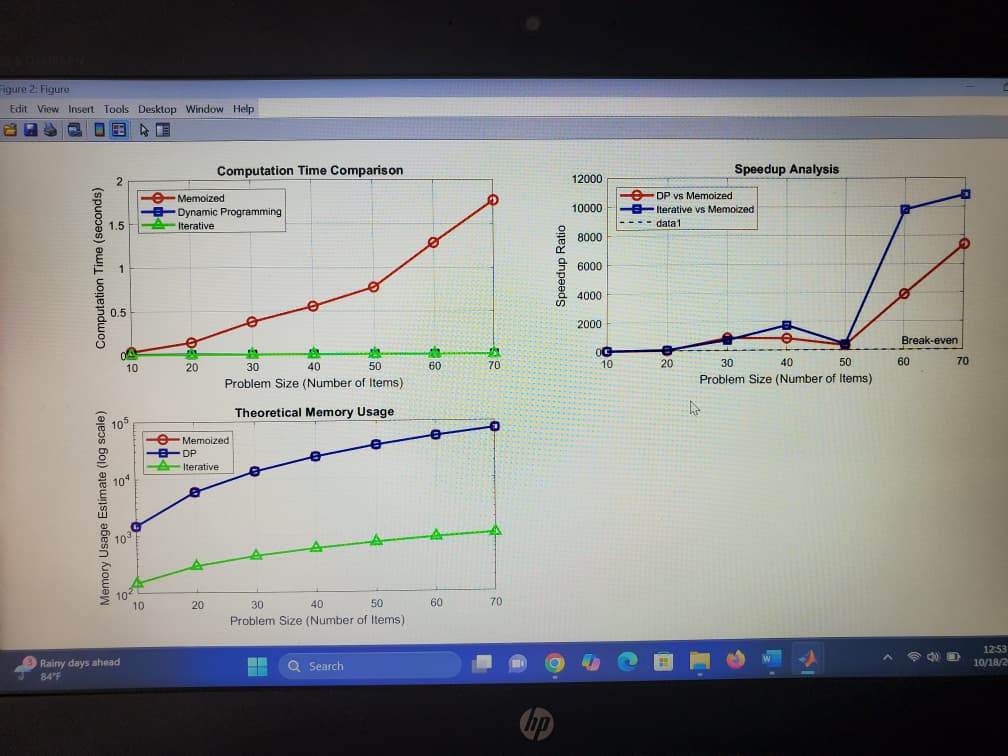
fprintf('Memoized: %.6f seconds\n', time\_memo);

fprintf('DP: %.6f seconds\n', time\_dp);

fprintf('Iterative: %.6f seconds\n', time\_iter);

fprintf('Selected Items: %s\n', mat2str(items));

THE PLOTS COMPARING THE PROGRAMMING METHODS.



EXPLANATION OF THE CODE

This MATLAB code compares the performance of three approaches to solve the 0/1 Knapsack Problem: Memoized Recursive, Dynamic Programming, and Iterative methods.

Section 1: Problem Parameters

weights = [2, 3, 4, 5, 6, 7, 8];

values = [10, 20, 30, 40, 50, 60, 70];

capacity = 15;

- This section defines the problem parameters, including the weights and values of the items, and the capacity of the knapsack.

Section 2: Memoized Recursive Approach

function [max\_value, memo] = knapsack\_memoized(weights, values, capacity, n, memo)

- This function uses memoization to store previously calculated results, reducing redundant calculations.

- It recursively calculates the maximum value that can be obtained with the given capacity and items.s

Section 3: Dynamic Programming Approach

function [max\_value, selected\_items] = knapsack\_dp(weights, values, capacity)

- This function uses dynamic programming to calculate the maximum value that can be obtained with the given capacity and items.

- It builds a DP table and backtracks to find the selected items.

Section 4: Iterative Space-Optimized Approach

function max\_value = knapsack\_iterative(weights, values, capacity)

This function uses an iterative approach to calculate the maximum value that can be obtained with the given capacity and items.

- It uses a single array to store the current and previous row, reducing memory usage.

Section 5: Performance Comparison

- This section compares the performance of the three approaches by calculating the maximum value for different problem sizes.

- It measures the computation time for each approach and plots the results.

Section 6: Complexity Analysis

This section analyzes the time and space complexity of each approach.

- It provides a summary of the complexity analysis and recommends the best approach based on the problem requirements.

The code provides a comprehensive comparison of the three approaches, including their performance, complexity, and practical considerations.

Time Complexity:

- Memoized Recursive: O(n\*W)

- Dynamic Programming: O(n\*W)

- Iterative: O(n\*W)

Space Complexity:

- Memoized Recursive: O(n\*W) + recursion stack

- Dynamic Programming: O(n\*W)

- Iterative: O(W)

The Iterative approach is the most efficient in terms of memory usage, while the Dynamic Programming approach provides a good balance between time and space complexity. The Memoized Recursive approach is easy to understand but may be limited by recursion depth.