FACULTY OF ENGINEERIG AND TECHONOLOGY

COMPUTER PROGRAMMING

ASSIGNMENT REPORT ON APPLICATION OF KNOWLEDGE OBTAINED FROM MODULES ONE TO FOUR USING MATLAB

BY GROUP 12

PRESENTED TO MR. MASERUKA BENEDICTO

DATE OF SUBMISSION; 21st October, 2025.

GROUP 12 MEMBERS

NAME	COURSE	REG NUMBER	SIGNATURE
LWASAMPIJJA MIKE CRISPUS	AMI	BU/UG/2024/2678	
AGABA JOSHUA	WAR	BU/UP/2024/1009	
APIO MIRIAM	WAR	BU/UP/2024/3827	
OBBO DANIEL BENJAMIN	WAR	BU/UP/2024/1054	
GINAH RUTH	PTI	BU/UP/2024/5468	
MIREMBE ROSE	WAR	BU/UP/2024/3835	
ISABIRIYE EDMOND	AMI	BU/UP/2024/3733	
TANGA RIGAN	WAR	BU/UP/2024/1071	
WANDABWA ELVIS	WAR	BU/UP/2024/1076	

DECLARATION

We declare that the information in this report is our own, to the best of our knowledge and shows the skills and knowledge obtained through the continuous assignment meetings.

APPROVAL

This is to confirm that this report has been written and presented by group 12, given the details of the assignment carried out.

Signature…….

Date…….

Course lecturer.

ACKNOWLEDGEMENT

First and foremost, we would like to thank the Almighty God for giving us the knowledge and guidance while doing our assignment as group 12. We extend our gratitude to all the persons with whose help we managed to make it this far. The love of every group member to invest time and provide all they could to see the assignment a success. Finally, we would like to express our gratitude to all the sources and references that have been cited in this report.

ABSTRACT

We started our first meeting for research on 17th October, 2025 in the university library and managed to achieve successful completion of this assignment through group work and division of tasks, and the achieved knowledge through the first assignment we did as a group as a tremendous means of consolidating the knowledge obtained to further the visualization of different parameters, patterns, trends and relationships of various variables represented as detailed plots as per the second assignment.

DEDICATION.

We dedicate this report to all Group 12 members, who have been the very cooperative towards the success of this report.

To our lecturer Mr. Maseruka Benedicto whose guidance and expertise have been so needful, your mentorship and lecturing has built our understanding.

TABLE OF CONTENTS

1. CHAPTER ONE

1.1 INTRODUCTION

1.1.2 HISTORICAL BACKGROUND

MATLAB, which stands for matrix laboratory, is a high performance programming language and environment designed primarily for technical computing. Its origins trace back to the late 1970s when Cleve Moler, a professor of computer science, developed it to provide his students with easy access to mathematical software libraries without requiring them to learn Fortran.

MATLAB is built around the concept of matrices, making it particularly effective for linear algebra and matrix manipulation. It provides a vast library of built-in functions for mathematical operations, statistics, optimization, and other specialized tasks.

MATLAB offers powerful tools for creating 2D and 3D plots, enabling users to visualize data effectively. Specialized toolboxes extend MATLAB's capabilities, providing functions tailored for specific applications like signal processing, image processing, control systems, and machine learning.

MATLAB can interface with other programming languages (like C, C++, and Python) and software tools, allowing for flexible integration into larger systems. Its interactive environment features a command window, workspace, and editor, making it accessible for both beginners and advanced users.

1.1.3 Historical Background

The first version of MATLAB was created in Fortran in the late 1970s as a simple interactive matrix calculator. This early iteration included basic matrix operations and was built on top of two significant mathematical libraries: LINPACK and EISPACK, which were developed for numerical linear algebra and eigenvalue problems, respectively.

Recent versions of MATLAB have introduced features like the *Live Editor*, which allows users to create interactive documents that combine code, output, and formatted text. This evolution reflects MATLAB's ongoing adaptation to meet the needs of its diverse user base across academia and industry.

1.1.4 STUDY METHODOLOGY

At the start, each member was given a task of making research about the assignment before our first meeting. The research concepts were obtained through watching tutorials on YouTube, reading the modules and consultations from other continuing students especially those in year three and four.

PART A

From the assignment of numerical methods make equivalent code based on recursive programming.

1.2 THE NEWTON RAPHSON METHOD

This is a fast and efficient technique for finding the roots of a real-valued function. It's widely used in mathematics, engineering, and computer science because of its rapid convergence—especially when the initial guess is close to the actual root.

NEWTON RASPHON METHOD USING RECURSION

```
function root_N=NRM_recursive(f,df,x0,tol,maxIter,iter)
if nargin<6
    iter=0;
end
x1=x0-f(x0)/df(x0);% new root approximation
if abs(x1-x0)<tol % checking for convergence
    root_N = x1;
    return;
else
    root_N=NRM_recursive(f,df,x1,tol,maxIter,iter + 1);% recursive call for
the function
end
end
f=@(x)x^3-x-2;% function whose root is required
df=@(x)3*x^2-1;% derivative of the function whose root is required
x0=1.5;
tol=10^-6;% tolerance for the absolute error
maxIter=50;
iter=0:50;
root_N=NRM_recursive(f,df,x0,tol,maxIter,iter)
```

```
root_N = 1.5214
```

1.2.1 BISECTION METHOD

This is a simple and reliable numerical technique used to find roots of a continuous function. It’s especially useful when you know the function changes sign over an interval—meaning there's a root somewhere between two points. Done as follows;

BISECTION METHOD

```
function root_B=bisect_recursive(f,a,b,tol,maxiter,iter)
if nargin<6
    iter=0;
end
% validate inputs
if iter>=maxiter
    error('maxiter exceeded.');
end
c=(a+b)/2; % midpoint
fc=f(c);
if abs(fc)<tol || (a+b)/2<tol % base case
    root_B=c;
    return
end
% recursive call for the function
if f(a)*f(c)<0
    root_B=bisect_recursive(f,a,c,tol,maxiter,iter+1);
else
   root_B=bisect_recursive(f,c,b,tol,maxiter,iter+1);
end
end
f=@(x)x^3-x-2;
a=1;
b=2;
tol=10^-6;
maxiter=50;
iter=0:50;
root_B=bisect_recursive(f,a,b,tol,maxiter,iter);
fprintf("root_B = %0.4f\n",root_B);
```

```
root_B = 1.5214
```

1.2.3 SECANT METHOD BY RECURSION PROGRAMMING

This is an iterative numerical technique used to find the root of a non linear equation. In recursive programming the same function calls itself repeatedly until the root is found within a given tolerance.

SECANT METHOD

```
function root_S=secant_recusive(f,x0,x1,tol,maxiter,iter)
if nargin<6
    iter=0;
end
f0=f(x0);
f1=f(x1);
if abs(f1-f0)==0 % validate inputs
    error('denominator is zero.');
end
x2=x1-f1*((x1-x0)/(f1-f0)); % new root
if abs(x2-x1)<tol % base case
    root_S=x2;
    return
end
    root_S=secant_recusive(f,x1,x2,tol,maxiter,iter+1); % recursive call for
the function
end
f=@(x)x^3-x-2;
x0=1;
x1=2;
tol=10^-6;
maxiter=50;
iter=0:50;
root_S=secant_recusive(f,x0,x1,tol,maxiter,iter);
fprintf("root_S = %0.4f\n",root_S);
```

```
root_S = 1.5214
```

1.2.4 FIXED POPINT ITERATION

This is a numerical technique used to find solutions to equations of the form: x=g(x)x = g(x)
It's one of the simplest iterative methods for solving nonlinear equations and is based on the idea of repeatedly applying a function to approximate a solution

FIXED ITERATION METHOD

```
function root_F=fixediter_recursive(g,x0,tol,toc,maxiter,iter)
tic;
if nargin<6
    iter=0;
end
x1=g(x0); % new root
if abs(x1-x0)<tol % base case
    root_F=x1;
    return
end
root_F=fixediter_recursive(g,x1,tol,toc,maxiter,iter+1); % recursive call
end
g=@(x)(x+2).^(1/3);
x0=1.5;
tol=10^-6;
maxiter=50;
iter=0:50;
root_F=fixediter_recursive(g,x0,tol,toc,maxiter,iter)
```

```
root_F = 1.5214
```

1 2 5. SOLVING ODES

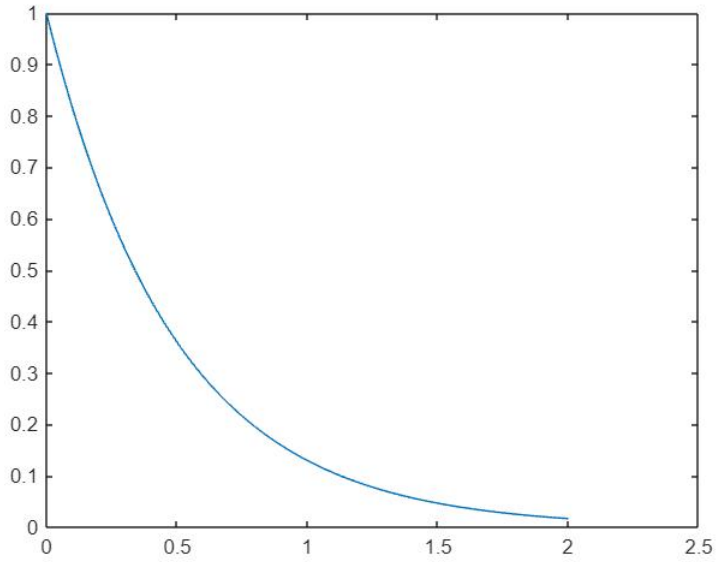
SOLVING ODEs USING RECURSION

EULER'S METHOD

```
function [T,Y] = euler_recursive(f, t0, tN, y0, N)
% f: function handle f(t,y)
% t0,tN: time interval
% y0: initial value (scalar or column vector)
% N: number of steps
h = (tN - t0)/N;
T = zeros(N+1,1);
Y = zeros(numel(y0), N+1);
T(1) = t0;
Y(:,1) = y0;

% recursive helper
function step(k)
    if k > N, return; end
    t = T(k);
    y = Y(:,k);
    ynext = y + h * f(t,y);
    T(k+1) = t + h;
    Y(:,k+1) = ynext;
    step(k+1);
end

step(1);
end
f = @(t,y) -2*y;
[T,Y] = euler_recursive(f, 0, 2, 1, 100);
plot(T, Y)
```



1.2.6 SOLVING ODEs USING RECURSION

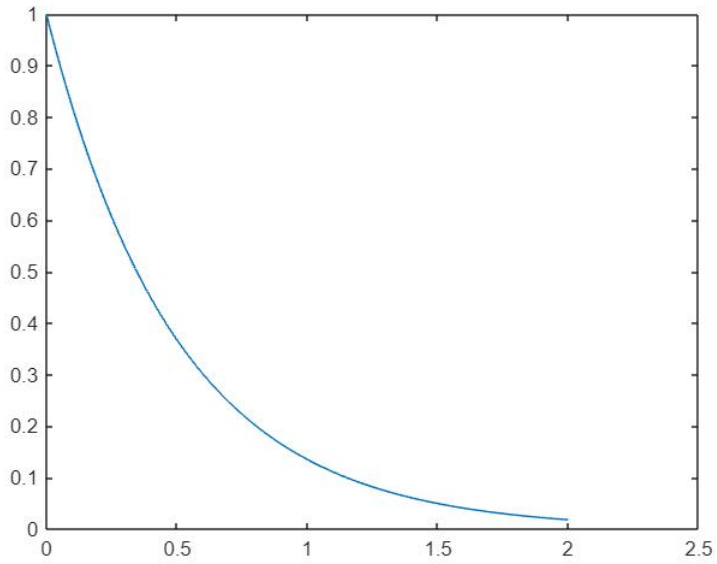
RUNGE KUTTA

```
function [T,Y] = rk4_recursive(f, t0, tN, y0, N)
% f: function handle f(t,y)
% t0,tN: time interval
% y0: initial value (scalar or column vector)
% N: number of steps
h = (tN - t0)/N;
T = zeros(N+1,1);
Y = zeros(numel(y0), N+1);
T(1) = t0;
Y(:,1) = y0;

function step(k)
    if k > N, return; end
    t = T(k);
    y = Y(:,k);
    k1 = f(t, y);
    k2 = f(t + 0.5*h, y + 0.5*h*k1);
    k3 = f(t + 0.5*h, y + 0.5*h*k2);
    k4 = f(t + h, y + h*k3);
    y_n = y + (h/6)*(k1 + 2*k2 + 2*k3 + k4);
    T(k+1) = t + h;
    Y(:,k+1) = y_n;
    step(k+1);
end

step(1);
```

```
end
f = @(t,y) -2*y;
[T,Y] = rk4_recursive(f, 0, 2, 1, 100);
plot(T, Y);
```



2. CHAPTER TWO

Use the concept of recursive and dynamic programming solve the following problems and make graphs to compare their computation time

a) knapsack problem

b) Fibonacci

2.1 KNAPSACKER PROBLEM BY RECURSION

The **0/1 Knapsack problem** can be formalized as follows: Given a set of n items, each with a value v_i and a weight w_i, and a knapsack with capacity W, determine the subset of items to include so the total value is maximized without exceeding the knapsack's capacity.

The problem exemplifies overlapping subproblems: Given fixed W and item subsets, many sub-configurations repeat, making it obedient to dynamic programming but challenging for naive recursion.

Algorithm Explanation

The recursive solution follows directly from the mathematical formulation. For each item, we consider two cases: include it or exclude it, and recursively compute the best solution for each sub-case. The base case is when there are no items left or the knapsack is full.

```
 function maxvalue=knapsack_recursive(weights,values,capacity,n) %wrapper
function to recall the recursive helper
 % base case, that is no item or no capacity left
 if n==0||capacity==0
     maxvalue=0;
     return;
 end
 % if weight of nth item is more than capacity then skip it
 if weights(n)>capacity
     maxvalue=knapsack_recursive(weights,values,capacity,n-1);
 else
     % inlude or exclude the nth item
     include=values(n)+knapsack_recursive(weights,values,capacity-
weights(n),n-1);
     exclude =knapsack_recursive(weights,values,capacity,n-1);
     maxvalue=max(include,exclude);
 end
 end

 weights=[95,64,23,73,50,22,6,57,40,98];
 values=[89,59,23,43,100,72,44,12,7,64];
```

```
capacity=300;
n=length(weights);
maxvalue=knapsack_recursive(weights,values,capacity,n);
fprintf("maximum value is %d\n",maxvalue);
```

```
maximum value is 394
```

2.1.2 KNAPSACK PROBLEM BY DYNAMIC PROGRAMMING (DP)

Dynamic programming overcomes recursion's inefficiency by storing results of subproblems in a table , thus avoiding redundant calculations. In MATLAB, this is typically implemented using bottom-up tabulation with arrays, but top-down tabulation is also possible.

```
function maxVal = knapsack_DP(W, weights, values)
n=length(values);
    K = zeros(n+1, W+1); % DP table

    for i = 1:n+1
        for w = 1:W+1
            if i == 1 || w == 1
                K(i, w) = 0;
            elseif weights(i-1) <= (w-1)
                K(i, w) = max(values(i-1) + K(i-1, w-weights(i-1)), K(i-1,
w));
            else
                K(i, w) = K(i-1, w);
            end
        end
    end
    maxVal = K(n+1, W+1);
end

weights=[95,64,23,73,50,22,6,57,40,98];
values=[89,59,23,43,100,72,44,12,7,64];
W=300;
maxVal=knapsack_DP(W,weights,values);
fprintf("maximum value is %d\n",maxVal);
```

```
maximum value is 394
```

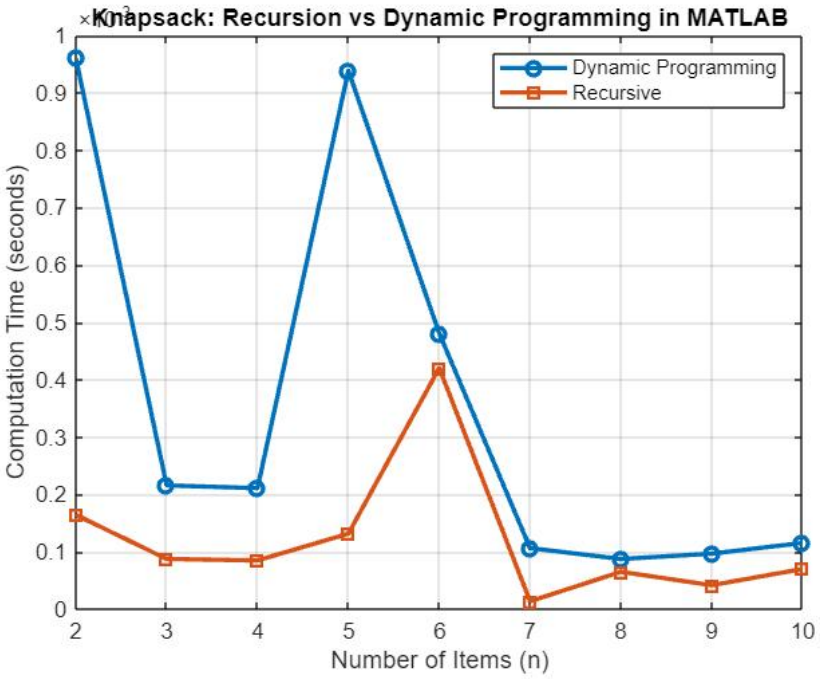
2.1.3 GRAPH FOR TIME COMPARISON (RECURSIVE VS DYNAMIC PROGRAMMING)

```
max_n = 10;
dp_time = zeros(1, max_n);
rec_time = zeros(1, max_n);
```

```
for n = 2:max_n
    tic;
    knapsack_DP(W, weights(1:n), values(1:n));
    dp_time(n) = toc;

    tic;
    knapsack_recursive(weights(1:n), values(1:n),capacity, n);
    rec_time(n) = toc;
end

figure;
plot(2:max_n, dp_time(2:end), '-o', 'LineWidth', 2);
hold on;
plot(2:max_n, rec_time(2:end), '-s', 'LineWidth', 2);
xlabel('Number of Items (n)');
ylabel('Computation Time (seconds)');
legend('Dynamic Programming', 'Recursive');
title('Knapsack: Recursion vs Dynamic Programming in MATLAB');
grid on;
```



2.1.4 FIBONACCI PROBLEM BY RECUSIVE

The **Fibonacci sequence** is defined recursively as: [F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)] The task is to compute the `n`th Fibonacci number or the sequence up to `n` using both recursive and dynamic programming approaches and compare their efficiency.

Algorithm Explanation

The naive recursive solution mirrors the mathematical definition, calling itself to compute `F(n-1)` and `F(n-2)` until the base case.

2.1.5 FIBONACCI PROBLEM BY RECUSIVE

```
function g=fibonacci_recursive(n)
% base case
if n==0
    g=0;
%Base case
elseif n==1
    g=1;
else
    g=fibonacci_recursive(n-1)+fibonacci_recursive(n-2); % recursive call
end
end
n=25;
g=fibonacci_recursive(n);
fprintf('value of g is %d',g)
```

```
value of g is 75025
```

2.1.6 FIBONACCI PROBLEM BY DYNAMIC PROGRAMMING

Dynamic programming computes and stores all Fibonacci numbers up to `n` in an array, either top-down or bottom-up (iteratively).

```
function f=fibonacci_DP(n)
% base case
if n==0
    f=0;
    return;
end
% DP table
T=zeros(1,n+1);
T(1)=0;
T(2)=1;
for k=3:n+1
    T(k)=T(k-1) + T(k-2);
```

```
end
f=T(n+1);
end
n=25;
f=fibonacci_DP(n);
fprintf("result of f = %d\n",f);
```

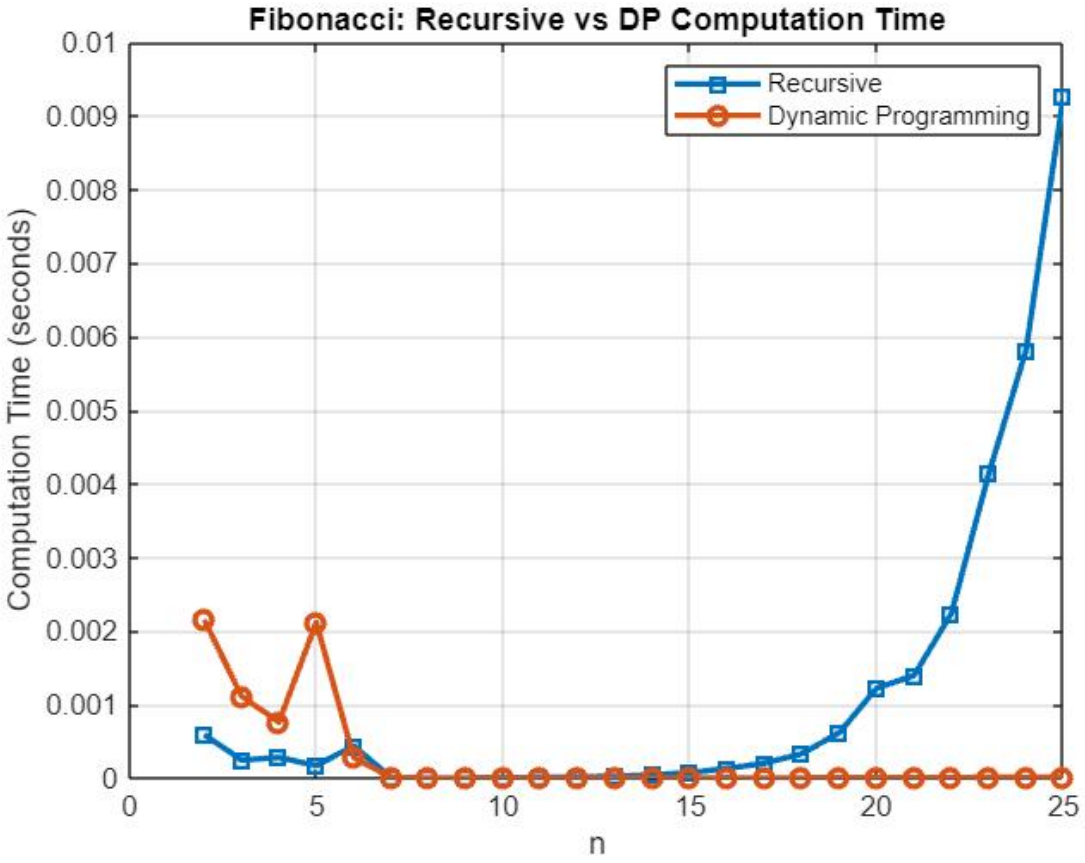
```
result of f = 75025
```

2.1.7 GRAPHICAL COMPARISON BETWEEN RECURSIVE AND DYNAMIC PROGRAMMING FOR FIBONACCI PROBLEM

```
max_n = 25;
rec_time = zeros(1, max_n);
dp_time = zeros(1, max_n);
for n = 2:max_n
    tic;
    fibonacci_recursive(n);
    rec_time(n) = toc;
    tic;
    fibonacci_DP(n);
    dp_time(n) = toc;
end

figure;
plot(2:max_n, rec_time(2:end), '-s', 'LineWidth', 2);
hold on;
plot(2:max_n, dp_time(2:end), '-o', 'LineWidth', 2);
xlabel('n');
ylabel('Computation Time (seconds)');
legend('Recursive', 'Dynamic Programming');
title('Fibonacci: Recursive vs DP Computation Time');
grid on;
```

```
value of g is 75025
result of f = 75025
```


Fibonacci: Recursive vs DP Computation Time
Recursive
Dynamic Programming
Computation Time (seconds)
n
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
5
10
15
20
25

Challenges

The challenge of recursive function is that it doesn’t perform efficiently with complex problems.

Recommendation

We recommend that for complex problems we use dynamic programming because it can handle even complex problems which is not the case with recursive function.

Conclusion

This comparative study demonstrates before in theory and practice why dynamic programming substantially outperforms classical recursion for the knapsack and Fibonacci problems.