****

**FACULTY OF ENGINEERING AND TECHNOLOGY**

**COURSE UNIT; COMPUTER PROGRAMMING**

**LECTURER: MR MASERUKA BENDICTO**

**COURSES; AMI, APE, WAR, MEB**

**GROUP 8**

GROUP 8 MEMBERS

|  |  |  |  |
| --- | --- | --- | --- |
| NAME | COURSE | REG NUMBER | SIGN |
| LWASAMPIJJA ASHIRAF | AMI | BU/UP/2024/5253 |  |
| AUJO EVERLYN | AMI | BU/UP/2024/3223 |  |
| RUBANGAKENE GEOFFREY | WAR | BU/UP/2024/3260 |  |
| BINYWERA NICHOLAS | WAR | Bu/up/2024/3249 |  |
| KAKUMBA ISSAC SSEMBOGO | WAR | BU/UP/2024/2593 |  |
| ADIKINI BEATRICE | WAR | BU/UP/2024/4090 |  |
| APIO MARY GRACE | WAR | BU/UP/2024/3247 |  |
| OJABAN NELSON | AMI | BU/UP/2024/3226 |  |
| KOMUHANGI AGATHA | AMI | BU/UP/2024/3735 |  |

DATE OF SUBMISSION: ………………………………

SUBMITTED TO: …………………………………

SIGNATURE: ……………………………………………..

ABSTRACT

This report represents a comprehensive analysis of various numerical methods employed to solve mathematical problems, their applications in real life and practical examples conducted by a collaborative group of six individuals. We employed both traditional hand calculations techniques and modern programming approaches like Matla to enhance our understanding of numerical analysis. The methods investigated include the Newton Raphson method, fixed point iteration, bisection method, secant and langrage method. Each member contributed to the development and implementation of these techniques, allowing for a comparative assessment of accuracy, efficiency and ease of use.

DECLARATION

We hereby declare that the work presented in this report is our original work and has not been submitted for any other academic purpose. This report is a result of our own research and understanding of the numerical methods, formulation, implementation and applications.

ACKNOWLEDGEMENT

We would like to begin by expressing our heartfelt gratitude to Almighty God for granting us wisdom, strength and perseverance needed to complete this report

We also appreciate the collaborative spirit of our team members and all those who contributed to the successful completion of this report on numerical analysis.

DEDICATION

We dedicate this report to all the inspiring mathematicians and engineers who strive to unlock the mysteries of numerical methods. May this inspire you to pursue knowledge with passion and determination.

LIST OF ABBREVIATIONS

AMI – Agricultural mechanization

WAR- Water resources engineering

APE – Agroproccessing engineering

MEB – Mining Engineering

INTRODUCTION

METHODOLOGY

The information in this group report was achieved and collected through different preparations. It aims to explore the various numerical methods used in programming and their practical applications in solving real-life engineering problems. By examining real-world examples, we will explain the effectiveness of these methods, providing insights into their role in advancing engineering practices. Through collaborative research and analysis, we hope to contribute to a deeper understanding of how numerical methods can be harnessed to tackle contemporary engineering challenges.

The implementation was done using MATLAB. Recursive functions were used for iterative numerical algorithms,   
while memoization and bottom-up dynamic programming techniques were applied to optimize recursion where possible.  
Each algorithm's execution time was measured using MATLAB’s built-in timing functions, and graphical plots were generated   
to compare their computational performances.

STUDY COVERAGE

The purpose of the study was to enable students appreciate how crucial numerical methods are in approximating solutions to mathematical problems that can not be solved analytically.

This is especially true in engineering, where real world scenarios often involve complex equations.

**. Recursive Numerical Methods**

The following numerical methods were implemented recursively for finding the root of the projectile motion function f(t) = v₀t - ½gt²:  
- Bisection Method  
- False Position Method  
- Newton-Raphson Method  
- Secant Method  
  
Each method terminates when the absolute error between successive approximations is below a tolerance value.

The projectile motion equation used:

f(t) = v₀t - ½gt², where v₀ = 20 m/s and g = 9.81 m/s².

**Recursive Numerical Methods for Differential Equations**

To extend recursion into differential equations, recursive versions of Euler’s method, Heun’s method,   
and the 4th Order Runge-Kutta (RK4) method were implemented to solve dy/dt = -2y with initial condition y(0) = 1.  
These were compared with the analytical solution y(t) = e⁻²ᵗ.

**Recursive and Dynamic Programming Problems**

Two classical algorithmic problems were solved using recursive and dynamic programming approaches:  
a) The 0/1 Knapsack Problem — optimizing the total value of items selected within a weight limit.  
b) The Fibonacci Sequence — computing the nth Fibonacci number using recursion, memoization, and iteration.

implemention

Implements **recursive versions** of: Bisection, False Position, Newton-Raphson, Secant (each with safe recursion limits).

* Implements **recursive (and memoized / dynamic)** versions of:
  + **Knapsack (0/1)**: naive recursion and top-down memoized dynamic programming.
  + **Fibonacci**: naive recursion, memoized recursion, and iterative DP.
* Implements **recursive stepping** variants for ODE methods (Euler, Heun, RK4 — implemented recursively).
* Measures computation times (averaged where appropriate) and generates comparison graphs.

Save the code below as a single file then run it in MATLAB. It will produce plots and print timings to the command window.

MATLAB code

% recursive\_numerical\_methods

% - Recursive root-finders: bisection, false-position, Newton-Raphson, secant

% - Knapsack: naive recursive and memoized DP

% - Fibonacci: naive recursive, memoized recursive, iterative DP

% - Time comparisons and plots

clear; close all; clc;

%% world problem : projectile motion root

v0 = 20; g = 9.81;

f = @(t) v0.\*t - 0.5.\*g.\*t.^2;

df = @(t) v0 - g.\*t; % derivative for Newton-Raphson

t\_exact = 2\*v0/g; % analytical impact time

% Root-search interval and tolerances

a = 0; b = 5;

tol = 1e-8;

maxRecDepth = 1000; % recursion safety

%% Recursive root finders

% Wrapper functions that measure time and call recursive implementations.

% 1) Recursive Bisection

tic; [root\_bis, iter\_bis] = recursive\_bisection(f, a, b, tol, 0, maxRecDepth); time\_bis = toc;

% 2) Recursive False Position (Regula-Falsi)

tic; [root\_false, iter\_false] = recursive\_false\_position(f, a, b, tol, 0, maxRecDepth); time\_false = toc;

% 3) Recursive Newton-Raphson (recursion on iterations)

x0\_nr = 2; tic; [root\_newton, iter\_newton] = recursive\_newton(f, df, x0\_nr, tol, 0, maxRecDepth); time\_newton = toc;

% 4) Recursive Secant

x0\_s = 0.1; x1\_s = 4.5; tic; [root\_secant, iter\_secant] = recursive\_secant(f, x0\_s, x1\_s, tol, 0, maxRecDepth); time\_secant = toc;

fprintf('Root-finding results (exact t = %.8f):\n', t\_exact);

fprintf('Bisection: root=%.10f, iter=%d, time=%.5e\n', root\_bis, iter\_bis, time\_bis);

fprintf('FalsePos: root=%.10f, iter=%d, time=%.5e\n', root\_false, iter\_false, time\_false);

fprintf('Newton: root=%.10f, iter=%d, time=%.5e\n', root\_newton, iter\_newton, time\_newton);

fprintf('Secant: root=%.10f, iter=%d, time=%.5e\n\n', root\_secant, iter\_secant, time\_secant);

%% Plot root-finder computation times

methods = {'Bisection','False Position','Newton-Raphson','Secant'};

times = [time\_bis, time\_false, time\_newton, time\_secant];

figure('Name','Root-finder times','NumberTitle','off');

bar(categorical(methods), times);

ylabel('Time (s)');

title('Computation times (recursive implementations)');

grid on;

%% Recursive ODE solvers (single-step recursion)

% Solve dy/dt = -2\*y with y(0)=1 on [0,2] for comparison with analytic solution y = exp(-2t)

ode\_f = @(t,y) -2\*y;

t0 = 0; tf = 2; N = 200; h = (tf - t0)/N;

t\_vec = t0:h:tf;

y\_exact = exp(-2.\*t\_vec);

% Euler (recursive)

tic; y\_euler = zeros(size(t\_vec)); y\_euler(1) = 1;

y\_euler = recursive\_euler\_step(ode\_f, t\_vec, y\_euler, 1, h); time\_euler = toc;

% Heun (recursive)

tic; y\_heun = zeros(size(t\_vec)); y\_heun(1) = 1;

y\_heun = recursive\_heun\_step(ode\_f, t\_vec, y\_heun, 1, h); time\_heun = toc;

% RK4 (recursive)

tic; y\_rk4 = zeros(size(t\_vec)); y\_rk4(1) = 1;

y\_rk4 = recursive\_rk4\_step(ode\_f, t\_vec, y\_rk4, 1, h); time\_rk4 = toc;

fprintf('ODE solver timings:\nEuler: %.5e, Heun: %.5e, RK4: %.5e\n\n', time\_euler, time\_heun, time\_rk4);

% Plot ODE results

figure('Name','ODE solvers vs exact','NumberTitle','off');

plot(t\_vec, y\_exact, 'k-', 'LineWidth', 2); hold on;

plot(t\_vec, y\_euler, '--r');

plot(t\_vec, y\_heun, '--b');

plot(t\_vec, y\_rk4, '--g');

legend('Exact', 'Recursive Euler', 'Recursive Heun', 'Recursive RK4','Location','northeast');

xlabel('t'); ylabel('y(t)'); title('Recursive ODE methods vs analytical');

grid on;

% Bar times

figure('Name','ODE times','NumberTitle','off');

bar(categorical({'Euler','Heun','RK4'}), [time\_euler,time\_heun,time\_rk4]);

ylabel('Time (s)'); title('ODE recursive integrators time (N=200)'); grid on;

%% Knapsack: recursive vs memoized DP

% Example knapsack instance (small-to-medium size for demonstration)

values = [60, 100, 120, 90, 50]; % values

weights = [10, 20, 30, 40, 10]; % weights

W = 50; % capacity

n = length(values);

% 1) Naive recursive (note: exponential time) -- measure time for small n only

tic;

[bestVal\_rec, ~] = knapsack\_recursive(n, W, weights, values);

time\_knap\_rec = toc;

% 2) Memoized DP (top-down)

tic;

memo = -ones(n+1, W+1); % initialize with -1 meaning unknown

[bestVal\_mem, memo\_used] = knapsack\_memo(n, W, weights, values, memo);

time\_knap\_memo = toc;

% 3) Bottom-up DP (iterative) for verification and faster time

tic;

bestVal\_dp = knapsack\_dp(weights, values, W);

time\_knap\_dp = toc;

fprintf('Knapsack results (W=%d, n=%d):\n', W, n);

fprintf('Naive recursion: best=%.2f, time=%.5e\n', bestVal\_rec, time\_knap\_rec);

fprintf('Memoized DP: best=%.2f, time=%.5e\n', bestVal\_mem, time\_knap\_memo);

fprintf('Bottom-up DP: best=%.2f, time=%.5e\n\n', bestVal\_dp, time\_knap\_dp);

% Plot knapsack times

figure('Name','Knapsack computation times','NumberTitle','off');

bar(categorical({'Recursive','Memoized','Bottom-up DP'}), [time\_knap\_rec, time\_knap\_memo, time\_knap\_dp]);

ylabel('Time (s)'); title('Knapsack methods timing'); grid on;

%% Fibonacci: naive vs memo vs iterative DP (timing & scaling)

% We'll measure for a range of n to show scaling

Ns = [5, 10, 15, 20, 25, 30]; % naive recursion becomes slow quickly (avoid >30)

times\_fib\_naive = zeros(size(Ns));

times\_fib\_memo = zeros(size(Ns));

times\_fib\_iter = zeros(size(Ns));

for idx = 1:length(Ns)

m = Ns(idx);

% naive recursion (may be slow for m>30)

tic; fn = fibonacci\_recursive(m); times\_fib\_naive(idx) = toc;

% memoized recursive

tic; fm = fibonacci\_memo(m); times\_fib\_memo(idx) = toc;

% iterative DP

tic; fi = fibonacci\_iter(m); times\_fib\_iter(idx) = toc;

end

% Display times

disp(table(Ns', times\_fib\_naive', times\_fib\_memo', times\_fib\_iter', ...

'VariableNames', {'n','Time\_Naive','Time\_Memo','Time\_Iter'}));

% Plot times (log scale for clarity)

figure('Name','Fibonacci computation times','NumberTitle','off');

semilogy(Ns, times\_fib\_naive, '-o', 'LineWidth', 1.5); hold on;

semilogy(Ns, times\_fib\_memo, '-o', 'LineWidth', 1.5);

semilogy(Ns, times\_fib\_iter, '-o', 'LineWidth', 1.5);

legend('Naive recursion','Memoized recursion','Iterative DP','Location','northwest');

xlabel('n'); ylabel('Time (s) [log scale]'); title('Fibonacci: naive vs memo vs iterative DP'); grid on;

%% Save figures

%to save figures to PNG files

saveas(figure(1),'root\_finder\_times.png');

saveas(figure(2),'ode\_vs\_exact.png');

saveas(figure(3),'ode\_times.png');

saveas(figure(4),'knapsack\_times.png');

saveas(figure(5),'fibonacci\_times.png');

%%Nested / Local function implementations

%% Recursive Bisection

function [c, iter] = recursive\_bisection(f, a, b, tol, depth, maxDepth)

if depth > maxDepth

error('recursive\_bisection: maximum recursion depth reached');

end

c = (a + b)/2;

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

iter = depth;

return;

end

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

[c, iter] = recursive\_bisection(f, a, c, tol, depth+1, maxDepth);

else

[c, iter] = recursive\_bisection(f, c, b, tol, depth+1, maxDepth);

end

end

%% Recursive False Position (Regula-Falsi)

function [c, iter] = recursive\_false\_position(f, a, b, tol, depth, maxDepth)

if depth > maxDepth

error('recursive\_false\_position: maximum recursion depth reached');

end

fa = f(a); fb = f(b);

c = (a\*fb - b\*fa) / (fb - fa); % regula-falsi formula

if abs(f(c)) < tol || abs(b-a) < tol

iter = depth;

return;

end

if fa \* f(c) < 0

[c, iter] = recursive\_false\_position(f, a, c, tol, depth+1, maxDepth);

else

[c, iter] = recursive\_false\_position(f, c, b, tol, depth+1, maxDepth);

end

end

%% Recursive Newton-Raphson (recursion by iteration)

function [x\_new, iter] = recursive\_newton(f, df, x, tol, depth, maxDepth)

if depth > maxDepth

error('recursive\_newton: maximum recursion depth reached');

end

dfx = df(x);

if dfx == 0

error('Derivative zero - Newton method fails at x = %g', x);

end

x\_new = x - f(x)/dfx;

if abs(x\_new - x) < tol

iter = depth;

return;

else

[x\_new, iter] = recursive\_newton(f, df, x\_new, tol, depth+1, maxDepth);

end

end

%% Recursive Secant

function [x2, iter] = recursive\_secant(f, x0, x1, tol, depth, maxDepth)

if depth > maxMaxDepthWrapper(maxDepth)

error('recursive\_secant: maximum recursion depth reached');

end

% Avoid division by zero

denom = f(x1)-f(x0);

if denom == 0

x2 = x1; iter = depth; return;

end

x2 = x1 - f(x1)\*(x1 - x0)/denom;

if abs(x2 - x1) < tol

iter = depth;

return;

else

[x2, iter] = recursive\_secant(f, x1, x2, tol, depth+1, maxMaxDepthWrapper(maxDepth));

end

end

function d = maxMaxDepthWrapper(maxDepth)

% small helper to avoid nested function name conflicts and keep semantics

d = maxDepth;

end

%% Recursive Euler integrator (fills y vector recursively)

function y = recursive\_euler\_step(f, t, y, idx, h)

% idx is the current index (1-based). Recursively compute y(idx+1) until end.

if idx >= length(t)

return;

end

y(idx+1) = y(idx) + h \* f(t(idx), y(idx));

y = recursive\_euler\_step(f, t, y, idx+1, h);

end

%% Recursive Heun (improved Euler)

function y = recursive\_heun\_step(f, t, y, idx, h)

if idx >= length(t)

return;

end

k1 = f(t(idx), y(idx));

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

y(idx+1) = y(idx) + h\*(k1 + k2)/2;

y = recursive\_heun\_step(f, t, y, idx+1, h);

end

%% Recursive RK4

function y = recursive\_rk4\_step(f, t, y, idx, h)

if idx >= length(t)

return;

end

k1 = f(t(idx), y(idx));

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

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

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

y(idx+1) = y(idx) + h\*(k1 + 2\*k2 + 2\*k3 + k4)/6;

y = recursive\_rk4\_step(f, t, y, idx+1, h);

end

%% Knapsack implementations

% Naive recursive 0/1 knapsack

function [bestVal, choice] = knapsack\_recursive(i, W, weights, values)

% i = number of items considered (1..n)

if i == 0 || W == 0

bestVal = 0;

choice = [];

return;

end

if weights(i) > W

[bestVal, choice] = knapsack\_recursive(i-1, W, weights, values);

return;

else

% Option 1: don't take item i

[val1, choice1] = knapsack\_recursive(i-1, W, weights, values);

% Option 2: take item i

[val2, choice2] = knapsack\_recursive(i-1, W - weights(i), weights, values);

val2 = val2 + values(i);

if val2 > val1

bestVal = val2;

choice = [choice2, i];

else

bestVal = val1;

choice = choice1;

end

end

end

% Top-down memoized knapsack

function [bestVal, memo] = knapsack\_memo(i, W, weights, values, memo)

% memo is matrix (i+1) x (W+1), -1 means unknown

if i == 0 || W == 0

bestVal = 0;

return;

end

if memo(i+1, W+1) ~= -1

bestVal = memo(i+1, W+1);

return;

end

if weights(i) > W

[bestVal, memo] = knapsack\_memo(i-1, W, weights, values, memo);

else

[val1, memo] = knapsack\_memo(i-1, W, weights, values, memo);

[val2, memo] = knapsack\_memo(i-1, W - weights(i), weights, values, memo);

val2 = val2 + values(i);

if val2 > val1

bestVal = val2;

else

bestVal = val1;

end

end

memo(i+1, W+1) = bestVal;

end

% Bottom-up DP knapsack (iterative)

function bestVal = knapsack\_dp(weights, values, W)

n = length(weights);

K = zeros(n+1, W+1);

for i = 1:n

for w = 0:W

if weights(i) <= w

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

else

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

end

end

end

bestVal = K(n+1, W+1);

end

%% Fibonacci implementations

% Naive recursive Fibonacci

function f = fibonacci\_recursive(n)

if n <= 0

f = 0; return;

elseif n == 1

f = 1; return;

end

f = fibonacci\_recursive(n-1) + fibonacci\_recursive(n-2);

end

% Memoized Fibonacci (top-down)

function f = fibonacci\_memo(n)

memo = -ones(1, n+1);

f = fib\_memo\_helper(n, memo);

end

function val = fib\_memo\_helper(n, memo)

if n <= 0

val = 0; return;

elseif n == 1

val = 1; return;

end

if memo(n+1) ~= -1

val = memo(n+1); return;

end

val = fib\_memo\_helper(n-1, memo) + fib\_memo\_helper(n-2, memo);

memo(n+1) = val; % note: MATLAB passes arrays by value, but it's OK for demonstration small n

end

% Iterative (bottom-up) fibonacci

function f = fibonacci\_iter(n)

if n <= 0

f = 0; return;

elseif n == 1

f = 1; return;

end

a = 0; b = 1;

for k = 2:n

c = a + b;

a = b; b = c;

end

f = b;

end

**Notes & Recommendations**

1. **Recursion depth**: MATLAB's recursion limit can be reached for very deep recursion (large N). I included maxRecDepth checks for root-finders (and a wrapper for secant) to avoid runaway recursion. If you plan to push recursion deep, either increase MATLAB's recursion limit (feature('DistributedArrayMaxNumWorkers', ...) is unrelated — instead use set(0,'RecursionLimit',N) if needed) or prefer iterative formulations.
2. **Performance**:
   * Naive recursive knapsack and naive Fibonacci have exponential runtime and are only practical for small sizes (n ≤ ~30 for Fibonacci; knapsack items ≤ ~25 is still heavy).
   * Memoization or bottom-up DP is strongly recommended for realistic sizes — timings in the script demonstrate that.
3. **Reproducibility**:
   * The script prints timings and shows several comparison plots. If you want, I can package graphs and commentary into a Word (.docx) report or export the figures as PNGs and a .mat file with results.
4. **Possible improvements**:
   * For knapsack, you might want to also return which items are selected in DP (I left that out to keep code focused; I gave naive recursion choice output).
   * For Fibonacci memoized helper, MATLAB's pass-by-value of arrays means the simple helper won't persist the memo across recursive calls efficiently — for large N you'd rework it to use persistent or a global vector, or implement the top-level wrapper to pass & return memo consistently