

FACULTY OF ENGINEERING & TECHNOLOGY

##### ASSIGNMENT REPORT BASED ON OBJECT-ORIENTED IMPLEMENTATION AND APPLICATION OF DIFFERENT CLASSES USING THE KNOWLEDGE OF MATLAB MODULES 1-5

COMPUTER PROGRAMMING

COURSE LECTURER: MR. MASERUKA BENDICTO

By GROUP 18

# ABSTRACT

This report looks at the application of different classes such as encapsulation, inheritance, polymorphism and abstraction to solve differential and integral problems and use them in the previously created codes in the previous report for numerical approximation methods for finding solutions to functions.

The above methods were designed in the MATLAB environment that is the Live Script. The project demonstrated fundamental skills in data handling, organization, and problem-solving within the MATLAB environment, providing practical experience in a complete data workflow.

# ACKNOWLEGEMENT

By the Grace of GOD we were able to work together as a group to complete the assignment and we acknowledge him for that.

We thank, Mr. Maseruka Bendicto our course lecturer for guiding us in this course which is a vital aspect for our engineering profession.

Appreciation goes to group members for the commitment and team spirit which simplified work and made it easy for us to complete the task and come up with this report.

# DECLARATION

We, Group 18 members hereby declare to the best of our knowledge, that this assignment report is a true record of our unending efforts in applying the knowledge we acquired from modules one through five. It is truly an original creation of our own and it has never been used by any other individual for any academic award in any learning institution.

**MEMBERS OF GROUP 18**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **NAME** | **REG NUMBER** | **PROGRAM** | **SIGN** |
| **1** | AUMA DIANA | BU/UP/2024/1020 | WAR |  |
| **2** | ENAMU REAGAN EGIMU | BU/UG/2024/2672 | APE |  |
| **3** | MUKHOOLI ELIJAH | BU/UG/2024/2586 | MEB |  |
| **4** | NABAWEESI CLAIRE | BU/UP/2024/1046 | WAR |  |
| **5** | NAKAWEESA LINNET | BU/UP/2024/4327 | APE |  |
| **6** | NANDAULA CATHERINE | BU/UP/2024/4322 | AMI |  |
| **7** | OLUK CHRISTIAN GLEN | BU/UP/2024/3842 | WAR |  |
| **8** | OMARA PASCAL KELLY | BU/UP/2024/1063 | WAR |  |
| **9** | TUMUHAISE SARAH | BU/UG/2024/2674 | AMI |  |
| **10** | UHURU DENISH BRIAN | BU/UP/2024/3841 | WAR |  |

# APPROVAL

This is to confirm that this report has been written and presented by Group 18, giving details of the assignment carried out.

Course Lecturer

|  |  |
| --- | --- |
| **Signature** | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |
| **Date** | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |
|  |  |

**Table of Contents**

[ABSTRACT ii](#_Toc211804480)

[ACKNOWLEGEMENT iii](#_Toc211804481)

[DECLARATION iv](#_Toc211804482)

[APPROVAL v](#_Toc211804483)

[CHAPTER 1 : INTRODUCTION 7](#_Toc211804484)

[CHAPTER 2 : CONCLUSION 24](#_Toc211804485)

[CHAPTER 3 : REFERENCES 25](#_Toc211804486)

# : INTRODUCTION

In this assignment we were required to use the previously created codes from the previous assignment within the MATLAB environment to develop and test high end or high level back end implementation of numerical methods, application for solving complication problems.

The assignment emphasized on ensuring the current class holds all abstract methods with two subclasses, one for differential and the other for integral problems.

The implementation builds upon previously developed code, enhancing it to support abstraction and polymorphism. Through this approach, the project not only demonstrates practical applications of OOP principles but also provides a robust computational tool capable of performing key numerical analysis efficiently and accurately.

**TASK GIVEN**

Whilst implementing the concept of classes encapsulation, inheritance, polymorphism and abstraction, develop and test high end or high level back end implementation of a numerical methods application for solving computational problems. For simplicity, apply the codes developed in the previous assignment and ensure the parent class holds all abstract methods with two subclasses, one for differential problem and the other for integral problems.

**Solution to the question**

**Using the Newton Raphson Method**

classdef (Abstract) NumericalMethod

properties (Access = protected)

f % Function handle

tol % Tolerance

maxIter % Maximum number of iterations

end

methods

function obj = NumericalMethod(f, tol, maxIter)

% Constructor method

obj.f = f;

obj.tol = tol;

obj.maxIter = maxIter;

end

end

classdef DifferentialProblems < NumericalMethod

properties (Access = private)

df % Derivative function

x0 % Initial guess

end

methods

function obj = DifferentialProblems(f, df, x0, tol, maxIter)

% Call parent constructor

obj@NumericalMethod(f, tol, maxIter);

obj.df = df;

obj.x0 = x0;

end

function root = solve(obj)

% Use the recursive Newton-Raphson method

root = obj.recursiveNewton(obj.f, obj.df, obj.x0, obj.tol, obj.maxIter);

fprintf('Recursive Newton-Raphson Root: %.6f\n', root);

end

end

methods (Access = private)

function root = recursiveNewton(obj, f, df, x0, tol, maxIter)

% call recursive newton raphson method

if maxIter == 0

root = x0;

return

end

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

if abs(x1 - x0) < tol

root = x1;

return

else

root = obj.recursiveNewton(f, df, x1, tol, maxIter - 1);

end

end

end

end

classdef IntegralProblems < NumericalMethod

properties (Access = private)

a % Lower limit

b % Upper limit

n % Number of intervals

end

methods

function obj = IntegralProblems(f, a, b, n)

% Call parent constructor with default tolerance and iteration values

obj@NumericalMethod(f, 1e-6, 100);

obj.a = a;

obj.b = b;

obj.n = n;

end

function I = solve(obj)

% Implement the Trapezoidal Rule

h = (obj.b - obj.a) / obj.n;

x = obj.a:h:obj.b;

y = obj.f(x);

I = (h/2) \* (y(1) + 2\*sum(y(2:end-1)) + y(end));

fprintf('Trapezoidal Integral: %.6f\n', I);

end

end

end

% TEST SCRIPT FOR NUMERICAL METHODS APPLICATION (OOP IMPLEMENTATION

clc; clear; close all;

% Differential Problem: Root Finding using Recursive Newton-Raphson

f = @(x) x.^3 - x - 2;

df = @(x) 3\*x.^2 - 1;

diffProb = DifferentialProblems(f, df, 1.5, 1e-6, 50);

root = diffProb.solve();

fprintf('Root found = %.6f\n\n', root);

% Integral Problem: Definite Integral using Trapezoidal Rule

g = @(x) x.^2; % Example: ∫₀³ x² dx = 9

intProb = IntegralProblems(g, 0, 3, 100);

area = intProb.solve();

fprintf('Approximate integral = %.6f\n', area);

g = @(x) x.^3 - x - 2;

intProb = IntegralProblems(g, 0, 3, 8);

area = intProb.solve();

fprintf('Recursive Trapezoidal Integral result = %.6f\n', area);

**SOLUTIONS TO THE ABOVE CODE FROM THE COMMAND WINDOW**

**DIFFERENTIAL TEST**

**Recursive Newton-Raphson Root: 1.521380**

**Root found = 1.521380**

**TEST ONE**

**Trapezoidal Integral: 9.000450**

**Approximate integral = 9.000450**

**TEST TWO**

**Trapezoidal Integral: 10.066406**

**Recursive Trapezoidal Integral result = 10.066406**

## ****Brief Description of the Codes****

## The developed MATLAB program demonstrates the use of **Object-Oriented Programming (OOP)** concepts including **abstraction, encapsulation, inheritance, and polymorphism** in implementing a **numerical methods application** for solving both differential and integral computational problems.

### ****Numerical methods (Abstract Parent Class)****

This is the **base (parent) class** that defines the general structure for all numerical method classes. It declares **properties** such as the function f, tolerance tol, and maximum iterations maxIter. It provides constructor to initialize these shared properties. It defines an **abstract method** that must be implemented by all subclasses.

**Differential problems (Subclass 1)**

This subclass inherits from newton Raphson method and focuses on **differential problems**, specifically **root-finding** using the **recursive Newton–Raphson method**.

It adds private properties for the derivative df and initial guess x0. The solve method overrides the abstract method from the parent class and uses the recursive function recursive newton to find the root of the given function.The recursion continues until the difference between successive approximations is less than the specified tolerance.  
This demonstrates **inheritance**, **encapsulation**, and **recursion.**

### ****Integral problem (Subclass 2)****

This subclass also inherits from numerical method but handles **integral problems** using the **recursive Trapezoidal Rule.** It has private properties for integration limits a, b, and the number of intervals n. The method implements the recursive trapezoidal integration formula by repeatedly subdividing the interval until only one trapezoid remains (base case).This shows solve **method overriding (polymorphism)** since both subclasses use solve differently.

Test

This script acts as the **driver program** to test both subclasses.A differential problem is created to solve the equation f(x)=x3−x−2=0f(x) = x^3 - x - 2 = 0 f(x)=x3−x−2=0 using the recursive Newton–Raphson method.An object of the integral problem is created to compute the integral over a specified range using recursive trapezoidal integration.The results are printed to the command window for verification.

**FIXED ITERATION METHOD**

classdef (Abstract) NumericalMethod

% Abstract parent class for all numerical methods

properties (Access = protected)

tol % Tolerance for convergence

maxIter % Maximum number of iterations

end

methods

function obj = NumericalMethod(tol, maxIter)

obj.tol = tol;

obj.maxIter = maxIter;

end

end

methods (Abstract)

result = solve(obj) % Abstract method to be implemented in subclasses

end

end

classdef DifferentialMethod < NumericalMethod

% Handles differential problems (e.g., root-finding)

properties (Access = private)

gFunc % function handle for iteration g(x)

x0 % initial guess

end

methods

function obj = DifferentialMethod(gFunc, x0, tol, maxIter)

% Call parent constructor

obj@NumericalMethod(tol, maxIter);

obj.gFunc = gFunc;

obj.x0 = x0;

end

function result = solve(obj)

fprintf('--- Solving Differential Problem (Fixed-Point Iteration) ---\n');

result = obj.recursiveFixedPoint(obj.gFunc, obj.x0, obj.tol, obj.maxIter);

fprintf('Recursive Fixed Point Root: %.6f\n', result);

end

end

methods (Access = private)

function root = recursiveFixedPoint(obj, g, x0, tol, maxIter)

% Recursive fixed-point iteration

if maxIter == 0

root = x0;

return;

end

x1 = g(x0);

if abs(x1 - x0) < tol

root = x1;

return;

else

% Recursive call through the same object

root = obj.recursiveFixedPoint(g, x1, tol, maxIter - 1);

end

end

end

end

classdef IntegralMethod < NumericalMethod

% Subclass handling integral computation problems

properties (Access = private)

fFunc % Function handle for f(x)

a % Lower limit

b % Upper limit

n % Number of subintervals

end

methods

function obj = IntegralMethod(fFunc, a, b, n, tol, maxIter)

% Constructor

obj@NumericalMethod(tol, maxIter);

obj.fFunc = fFunc;

obj.a = a;

obj.b = b;

obj.n = n;

end

function result = solve(obj)

% Trapezoidal rule for numerical integration

fprintf('--- Solving Integral Problem (Trapezoidal Rule) ---\n');

h = (obj.b - obj.a) / obj.n;

x = obj.a:h:obj.b;

y = obj.fFunc(x);

integral = (h/2) \* (y(1) + 2\*sum(y(2:end-1)) + y(end));

result = integral;

fprintf('Approximate Integral Value: %.6f\n', result);

end

end

end

clc; clear; close all;

% Differential problem (root finding)

g = @(x) (x + 2).^(1/3);

diffSolver = DifferentialMethod(g, 1.5, 1e-6, 50);

root = diffSolver.solve();

% Optional: Integral problem test

f = @(x) x.^2;

a = 0; b = 2; n = 100;

intSolver = IntegralMethod(f, a, b, n, 1e-6, 50);

I = intSolver.solve();

**SOLUTIONS TO THE ABOVE CODE FROM THE COMMAND WINDOW**

**DIFFERENTIAL TEST**

**Recursive Fixed Point Root: 1.521380**

**Root found = 1.521380**

**INTEGRAL TEST**

**Approximate Integral Value: 2.666800**

### ****Brief Description of the Code****

## The developed MATLAB program demonstrates the use of **Object-Oriented Programming (OOP)** concepts including **abstraction, encapsulation, inheritance, and polymorphism** in implementing a **numerical methods application** for solving both differential and integral computational problems.

### ****Numerical methods (Abstract Parent Class)****

This is the **base (parent) class** that defines the general structure for all numerical method classes. It declares **properties** such as the function f, tolerance tol, and maximum iterations maxIter. It provides constructor to initialize these shared properties. It defines an **abstract method** that must be implemented by all subclasses.

**Differential problems (Subclass 1)**

This subclass inherits from fixed iteration method and focuses on **differential problems**, specifically **root-finding** using the **recursive fixed iteration method**.

It adds private properties for the derivative df and initial guess x0. The solve method overrides the abstract method from the parent class and uses the recursive function recursive fixed point to find the root of the given function. The recursion continues until the difference between successive approximations is less than the specified tolerance.  
This demonstrates **inheritance**, **encapsulation**, and **recursion.**

### ****Integral problem (Subclass 2)****

This subclass also inherits from numerical method but handles **integral problems** using the **recursive Trapezoidal Rule.** It has private properties for integration limits a, b, and the number of intervals n. The method implements the recursive trapezoidal integration formula by repeatedly subdividing the interval until only one trapezoid remains (base case).This shows solve **method overriding (polymorphism)** since both subclasses use solve differently.

Test

This script acts as the **driver program** to test both subclasses. A differential problem is created to solve the equation f(x) = x^3 – x −2 = 0 f(x) = x^3 - x - 2 = 0 f(x)=x3−x−2=0 using the recursive fixed iteration. An object of the integral problem is created to compute the integral over a specified range using recursive trapezoidal integration . The results are printed to the command window for verification.

**BISECTION METHOD**

classdef (Abstract) NumericalMethod

% Abstract base class for all numerical methods

properties (Access = protected)

f % function handle

tol % tolerance

maxIter % maximum number of iterations

end

methods

function obj = NumericalMethod(f, tol, maxIter)

obj.f = f;

obj.tol = tol;

obj.maxIter = maxIter;

end

end

methods (Abstract)

result = solve(obj) % Abstract method to be implemented by subclasses

end

end

classdef DifferentialProblems < NumericalMethod

% Handles differential (root-finding) problems using recursive bisection

properties (Access = private)

a % lower limit

b % upper limit

end

methods

function obj = DifferentialProblems(f, a, b, tol, maxIter)

obj@NumericalMethod(f, tol, maxIter);

obj.a = a;

obj.b = b;

end

function root = solve(obj)

root = obj.recursiveBisection(obj.f, obj.a, obj.b, obj.tol, obj.maxIter);

end

end

methods (Access = private)

function root = recursiveBisection(obj, f, a, b, tol, maxIter)

% Recursive implementation of the Bisection Method

if maxIter == 0 || abs(b - a) < tol

root = (a + b)/2;

return;

end

c = (a + b)/2;

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

root = obj.recursiveBisection(f, a, c, tol, maxIter - 1);

else

root = obj.recursiveBisection(f, c, b, tol, maxIter - 1);

end

end

end

end

classdef IntegralProblems < NumericalMethod

% Handles integral problems using recursive trapezoidal integration

properties (Access = private)

a % lower limit

b % upper limit

n % number of intervals

end

methods

function obj = IntegralProblems(f, a, b, n, tol, maxIter)

obj@NumericalMethod(f, tol, maxIter);

obj.a = a;

obj.b = b;

obj.n = n;

end

function result = solve(obj)

result = obj.recursiveTrapezoid(obj.f, obj.a, obj.b, obj.n);

end

end

methods (Access = private)

function I = recursiveTrapezoid(obj, f, a, b, n)

% Recursive Trapezoidal Integration

if n == 1

I = (b - a) \* (f(a) + f(b)) / 2;

return;

else

mid = (a + b)/2;

left = obj.recursiveTrapezoid(f, a, mid, n/2);

right = obj.recursiveTrapezoid(f, mid, b, n/2);

I = left + right;

end

end

end

end

%% Differential Problem — Root Finding using Recursive Bisection

f1 = @(x) x.^3 - x - 2;

a = 1; b = 2; tol = 1e-6; maxIter = 50;

diffProb = DifferentialProblems(f1, a, b, tol, maxIter);

root = diffProb.solve();

fprintf('Root of f(x) = x^3 - x - 2 using Recursive Bisection: %.6f\n', root);

%% Integral Problem — Recursive Trapezoidal Rule

f2 = @(x) x.^2;

a = 0; b = 3; n = 4;

intProb = IntegralProblems(f2, a, b, n, 1e-6, 50);

integralValue = intProb.solve();

fprintf('Integral of f(x) = x^2 from %.1f to %.1f using Recursive Trapezoidal Rule: %.6f\n', a, b, integralValue);

**SOLUTION AFTER TESTING**

Root of f(x) = x^3 - x - 2 using Recursive Bisection: 1.521380

Integral of f(x) = x^2 from 0.0 to 3.0 using Recursive Trapezoidal Rule: 9.281250

**BRIEF DESCRIPTION OF THE CODE**

The program consists of a parent class and two subclasses, each implementing a different recursive numerical method.

Numerical method **(**Parent Class**):** this defines shared attributes such as the function handle, tolerance, and maximum iterations.

Differential problem (Subclass 1), it inherits from numerical methods and implements the Recursive Bisection Method to find the roots of nonlinear equations. It demonstrates inheritance and recursion, breaking down the interval until the root is approximated.

Integral problem **(**Subclass 2**):** inherits from numerical methods and implements the Recursive Trapezoidal Rule to compute definite integrals of continuous functions by recursively dividing the integration interval. This shows polymorphism.

The test scriptcreates objects from both subclasses and tests the implemented methods. It finds the root of f(x)=x3−x−2f(x) = x^3 - x - 2f(x)=x3−x−2 using recursive bisection and calculates the definite integral function, f(x)=x2f(x) = x^2f(x)=x2 between 0 and 3 using recursive trapezoidal integration.

RUNGE KUTTA 4TH ORDER(RK4) METHOD

% NumericalMethod.m

classdef (Abstract) NumericalMethod

properties (Access = protected)

tol

maxIter

end

methods (Abstract)

result = solve(obj)

end

methods

function obj = NumericalMethod(tol, maxIter)

if nargin < 2

tol = 1e-6; maxIter = 1000;

end

obj.tol = tol;

obj.maxIter = maxIter;

end

end

end

% DifferentialMethod.m

classdef DifferentialMethod < NumericalMethod

properties

r

y0

t

h

end

methods

function obj = DifferentialMethod(r, y0, t, h)

obj@NumericalMethod(); % Call parent constructor

obj.r = r;

obj.y0 = y0;

obj.t = t;

obj.h = h;

end

function y = solve(obj)

% Recursive Runge-Kutta 4th Order solver

y = obj.rk4\_recursive(obj.r, obj.y0, obj.t, obj.h, 1);

end

end

methods (Access = private)

function y = rk4\_recursive(obj, r, y0, t, h, i)

y = zeros(size(t));

y(i) = y0;

if i == length(t)

return;

else

k1 = r\*y0;

k2 = r\*(y0 + 0.5\*h\*k1);

k3 = r\*(y0 + 0.5\*h\*k2);

k4 = r\*(y0 + h\*k3);

y\_next = y0 + (h/6)\*(k1 + 2\*k2 + 2\*k3 + k4);

y\_rest = obj.rk4\_recursive(r, y\_next, t, h, i + 1);

y(1:i) = y0;

y(i+1:end) = y\_rest(i+1:end);

end

end

end

end

% IntegralMethod.m

classdef IntegralMethod < NumericalMethod

properties

f

a

b

n

end

methods

function obj = IntegralMethod(f, a, b, n)

obj@NumericalMethod();

obj.f = f;

obj.a = a;

obj.b = b;

obj.n = n;

end

function I = solve(obj)

% Trapezoidal rule implementation

h = (obj.b - obj.a) / obj.n;

x = obj.a:h:obj.b;

y = obj.f(x);

I = (h/2) \* (y(1) + 2\*sum(y(2:end-1)) + y(end));

end

end

end

% Differential Problem: dy/dt = r\*y

r = 0.5;

y0 = 100;

t0 = 0; tf = 10; h = 0.1;

t = t0:h:tf;

% Create object of DifferentialMethod

diffSolver = DifferentialMethod(r, y0, t, h);

y\_rk4 = diffSolver.solve();

% Exact solution

y\_exact = y0 \* exp(r\*t);

% Display results

figure;

plot(t, y\_rk4, 'bo-', t, y\_exact, 'r--');

xlabel('t'); ylabel('y');

legend('RK4 Recursive', 'Exact Solution');

title('Runge-Kutta 4th Order Method (Recursive)');

% Integral Problem Example: ∫(x^2) dx from 0 to 2

f = @(x) x.^2;

a = 0; b = 2; n = 20;

intSolver = IntegralMethod(f, a, b, n);

I = intSolver.solve();

fprintf('\nTrapezoidal Integration Result: %.4f\n', I);

RESULT FOR TEST

Trapezoidal Integration Result: 2.6700

**BRIEF DESCRIPTION OF THE CODE**

The Runge-Kutta 4th Order (RK4**)** method is an efficient and accurate numerical technique used to solve ordinary differential equations (ODEs). It improves the Euler’s method and then gives an approximate value. At each iteration, RK4 computes four slope and produces an output after combining them. The recursive RK4 implementation applies these steps repeatedly through a recursive function, computing the numerical solution step by step.

**EULER’S METHOD**

% NumericalMethod.m

classdef (Abstract) NumericalMethod

properties (Access = protected)

tol

maxIter

end

methods (Abstract)

result = solve(obj)

end

methods

function obj = NumericalMethod(tol, maxIter)

if nargin < 2

tol = 1e-6;

maxIter = 1000;

end

obj.tol = tol;

obj.maxIter = maxIter;

end

end

end

% DifferentialMethod.m

classdef DifferentialMethod < NumericalMethod

properties

r

y0

t

h

end

methods

% Constructor

function obj = DifferentialMethod(r, y0, t, h)

obj@NumericalMethod(); % Call parent constructor

obj.r = r;

obj.y0 = y0;

obj.t = t;

obj.h = h;

end

% Implementation of abstract method

function y = solve(obj)

y = obj.euler\_recursive(obj.r, obj.y0, obj.t, obj.h, 1);

end

end

% IntegralMethod.m

classdef IntegralMethod < NumericalMethod

properties

f

a

b

n

end

methods

function obj = IntegralMethod(f, a, b, n)

obj@NumericalMethod();

obj.f = f;

obj.a = a;

obj.b = b;

obj.n = n;

end

function I = solve(obj)

h = (obj.b - obj.a) / obj.n;

x = obj.a:h:obj.b;

y = obj.f(x);

I = (h/2) \* (y(1) + 2\*sum(y(2:end-1)) + y(end));

end

end

end

%Differential Problem (Euler Recursive)

r = 0.5;

y0 = 100;

t0 = 0; tf = 10; h = 0.1;

t = t0:h:tf;

% Create object of DifferentialMethod (Euler)

diffSolver = DifferentialMethod(r, y0, t, h);

y\_euler = diffSolver.solve();

% Exact analytical solution

y\_exact = y0 \* exp(r\*t);

% Plot results

figure;

plot(t, y\_euler, 'bo-', t, y\_exact, 'r--', 'LineWidth', 1.5);

xlabel('t'); ylabel('y');

legend('Euler Recursive', 'Exact');

title('Euler Recursive Method using OOP');

grid on;

%Integral Problem (Trapezoidal Rule)

f = @(x) x.^2; % Function to integrate

a = 0; b = 2; n = 20;

% Create object of IntegralMethod

intSolver = IntegralMethod(f, a, b, n);

I = intSolver.solve();

fprintf('\nApproximate Integral of x^2 from 0 to 2 = %.4f\n', I);

SOLUTION TO TEST

Approximate Integral of x^2 from 0 to 2 = 2.6700

**BRIEF DESCRIPTION TO THE CODE ABOVE**

The Euler’s Method is a simple numerical technique used to approximate solutions of first-order ordinary differential equations (ODEs**)**. It estimates the solution by taking small steps along the curve, using the slope at the current point to predict the next value. In this project, a recursive implementation of Euler’s method is used to iteratively compute each new value. The method demonstrates the fundamental concept of numerical approximation, and when structured in an OOP framework, identifies encapsulation, inheritance, and polymorphism in solving differential equations.

# : CONCLUSION

By the end of the assignment we realized that the implementation of the numerical methods application using MATLAB successfully demonstrated the effectiveness of applying object-oriented programming (OOP) principles such as encapsulation, inheritance, polymorphism, and abstraction in solving complex computational problems. By creating an abstract base class and extending it into two specialized subclasses for differential and integral problems, the project achieved a clear module and reusable design structure.

The implementation also provide a practical framework for extending numerical methods to handle more advanced and diverse problem types in the future.

# : REFERENCES

* MATLAB Documentation
* MATLAB lecture notes by Mr. Maseruka Bendicto
* You tube MATLAB tutorials