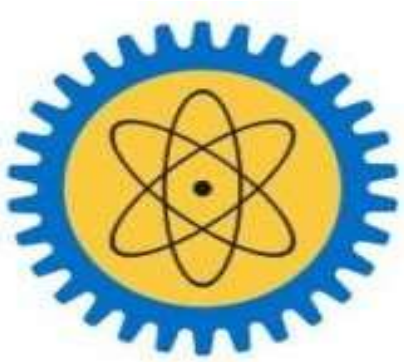

FACULTY OF ENGINEERING AND TECHNOLOGY

A REPORT ABOUT THE HIGH-LEVEL BACKEND DESIGN FOR NUMERICAL AND DIFFERENTIAL PROBLEM-SOLVING USING OBJECT ORIENTED PROGRAMMING

COURSE UNIT: COMPUTER PROGRAMMING

LECTURER: Mr. MASERUKA BENEDICTO

SUBMITTED BY: GROUP 16

E-MAIL: matlabgroup16@gmail.com

Submitted in partial fulfillment of the requirements of COMPUTER PROGRAMMING

DATE OF SUBMISSION...../...../.....

SUBMITTED TO: .....

DECLARATION

We, the undersigned members of group 16, do hereby declare that this report is the result of our own work carried out in partial fulfillment of the requirements of this course.

NAME	SIGNATURE
KABWERU ANDREW	.....
CHEMONGES MIKIRAR	.....
NAGASHA RITTA	.....
DIKITAL JOHN	.....
SANYU JOY	.....
OULE SADOCK	.....
WANGUSI DAVID	.....
SEBATIKA COLLINE	.....
ATYANG MILDRED	.....
KITUTU LEONARD	.....

APPROVAL

This report has been submitted and prepared by group 16 as part of the requirements for the completion of module 1to 5 under the guidance of our lecturer

Mr Maseruka Bendicto

Computer programming lecturer

Name: .....

Signature: .....

ACKNOWLEDGEMENT

We wish to express our sincere gratitude to our lecturer for the valuable guidance, encouragement and constructive criticism provided throughout the project. We extend our thanks to the fellow members of group 16 and friends for their constant support, discussions and suggestions during the development and testing of Matlab codes.

Finally, we give special thanks to our families for their patience, understanding and continuous encouragement throughout the course of this study

DEDICATION

This report is dedicated to our beloved families whose support, love, and encouragement have been the foundation of our academic achievements. We continue to dedicate this work to all students and researchers passionate about the application of computational and numerical methods in solving real world engineering and scientific problems.

ABSTRACT

This project presents an object-oriented programming approach to implementing numerical methods in Matlab. The main objective was to design a modular, reusable and extensible system that applies the OOP principles of abstraction, encapsulation, inheritance and polymorphism to solve both algebraic and differential computational problems.

An abstract parent class was created to define the structure for all numerical methods, with two subclasses.

Each subclass redefines the abstract solve method, showcasing polymorphism and ensuring method specific computation.

LIST OF ACRONYMS/ ABBREVIATIONS

1. MATLAB – Matrix Laboratory
2. abs - absolute

Table of Contents

CHAPTER ONE: INTRODUCTION

1.1 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.

1.2 Historical Development

- Initial Development: 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.
- Commercial Launch: MATLAB was officially launched as a commercial product in 1984 by MathWorks, a company founded by Moler along with Jack Little and Steve Bangert. This marked the transition from a simple calculator to a comprehensive programming environment. The software was reimplemented in C, enhancing its capabilities with the addition of user-defined functions, toolboxes, and graphical interfaces.
- Expansion and Toolboxes: Over the years, MATLAB has expanded significantly. By the late 1980s, it had introduced several specialized toolboxes for various applications, including control systems and signal processing. The introduction of the Simulink environment further allowed users to model and simulate dynamic systems graphically.
- Modern Enhancements: 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.

CHAPTER TWO: QUESTION ONE

2.1 Introduction

We were required to implement the concepts of classes, encapsulation, inheritance, polymorphism and abstraction.

To develop and test a high-end backend implementation of a numerical methods application for solving computational problems. While ensuring that the parent class holds all abstract methods with two subclasses, one for differential problems and the other for numerical problems.

2.2 SUPER CLASS CODE

2.2.1: steps taken to write the code

Step 1: defining and describing the class.

We started by defining the Matlab class as ***NumericalMethods*** which is the abstract super class for all numerical methods

```
classdef NumericalMethod
    %NUMERICALMETHOD Abstract Superclass for numerical computation problems.
    %   Encapsulates common parameters and defines abstract methods.

```

Step 2: variables stored in the class

We listed the class properties as shown below and declared them with access protected so that only the class and its sub classes can access these values

```
    properties (Access = protected)
        % Encapsulation: Parameters are protected, accessible by subclasses
        m = 80;
        g = 9.81;
        c = 0.25;
        mg; % m*g
    end

```

Step 3: methods

We designed the class as an abstract superclass that defines a common structure for all computation problems

```
    methods (Abstract)
        % Abstraction & Polymorphism: Subclasses must implement these specific
        % solution methods.
        solution = solveNRM(obj, initialGuess, tolerance, maxIter);
        solution = solveSecant(obj, x0, x1, tolerance, maxIter);
        solution = solveEuler(obj, initialValue, stepSize, timeEnd);
        solution = solveRungeKutta(obj, initialValue, stepSize, timeEnd);
    end
```

(i). Constructor

This was to initialise the property mg and ensuring that all subclasses start with the correct initialised constants

```
22        methods
23            % Constructor for the Superclass
24            function obj = NumericalMethod()
25                obj.mg = obj.m * obj.g;
26            end
27
```

(ii). Common function

This was to provide a common operation shared by all subclasses and compute the net force on an object under the influence of drag and gravity

```
28          % Encapsulation: Common functions used by subclasses
29          function val = f(obj, v)
30              % f(v) = m*g - c*v^2
31              val = obj.mg - obj.c * v^2;
32          end
33
34          function val = f_prime(obj, v)
35              % f'(v) = -2*c*v
36              val = -2 * obj.c * v;
37          end
38      end
```

(iii). Static function

This was to define a differential equation for use in integral methods i.e. Euler and Runge Kutta

```
40      % Define differential equation function for integral methods (Euler/RK)
41      % This represents the derivative dv/dt = f(v)
42      methods (Static)
43          function dvdt = differentialEq(t, v) %#ok<INUSD>
44              % The differential equation dv/dt is the function f(v) used
45              % in the NRM/Secant problems, representing the force balance.
46              % For this specific problem: f(v) = 784.8 - 0.25*v^2
47              m = 80; g = 9.81; c = 0.25;
48              dvdt = m*g - c*v^2;
49          end
50      end
51  end
```

2.3 SECTION 1: DIFFERENTIAL PROBLEMS

2.3.1 introduction

implement the concepts of classes, encapsulation, inheritance, polymorphism and abstraction.

To develop and test a high-end backend implementation of a numerical methods application for solving numerical method problems.

2.3.2 steps taken in writing the code

Step 1: defining the sub class

We defined the subclass as ***DifferentialProblems*** that inherits from the parent class ***NumericalMethods***

```
classdef DifferentialPoblem < NumericalMethod
    %ROOTFINDINGPROBLEM Solves differential problems (finding roots).
    %   Implements NRM and Secant methods.
```

(i) implementation of newton Raphson method

step 1: function definition and initialisation

we defined the function as ***solveNRM*** and set ***x*** as the starting value, made it run up to ***maxIter*** times and calculate ***fx*** and ***fpx***

```
    methods
        % Implementation of NRM (Polymorphism)
        function root = solveNRM(obj, initialGuess, tolerance, maxIter)
            fprintf('--- NRM Solution ---\n');
            x = initialGuess;
            for i = 1:maxIter
                fx = obj.f(x);
                fpx = obj.f_prime(x);
```

Step 2: checking for convergence and zero derivative

We set to print the root as x when ***abs(fx)*** is less than tolerance and display '***Zero derivative encountered.' NRM failed.*** When the derivative is not given

```
                if abs(fx) < tolerance
                    root = x;
                    fprintf('Converged in %d iterations.\n', i-1);
                    return;
                end

                if abs(fpx) < 1e-10
                    error('Zero derivative encountered. NRM failed.');
                end
```

Step 3: updating x and printing of root

We set it to update x to the next value of x and also print the root when maximum iterations are reached

```
            xNew = x - (fx / fpx);
            x = xNew;
        end
        root = x;
        fprintf('Max iterations reached. Approximate root: %.4f\n', root);
    end
```

(ii) implementation of secant method

step 1: function definition and initialisation

we defined the function as ***solveSecant*** and set ***x*** as the starting value, made it run up to ***maxIter*** times and calculate ***fx0*** and ***fx1***

```
% Implementation of Secant Method (Polymorphism)
function root = solveSecant(obj, x0, x1, tolerance, maxIter)
    fprintf('--- Secant Solution ---\n');
    for i = 1:maxIter
        fx0 = obj.f(x0);
        fx1 = obj.f(x1);
```

Step 2: checking for convergence and zero derivative

We set to print the root as x when ***abs(fx1)*** is less than tolerance and display ***'Secant method failed: zero denominator.'*** When ***abs(fx1-fx0)*** is less than 1e-10

```
        if abs(fx1) < tolerance
            root = x1;
            fprintf('Converged in %d iterations.\n', i);
            return;
        end

        if abs(fx1 - fx0) < 1e-10
            error('Secant method failed: zero denominator.');
        end
```

Step 3: updating x and printing of root

We set it to update x to the next value of x and also print the root when maximum iterations are reached

```
                    xNew = x1 - fx1 * (x1 - x0) / (fx1 - fx0);
                    x0 = x1;
                    x1 = xNew;
                end
                root = x1;
                fprintf('Max iterations reached. Approximate root: %.4f\n', root);
            end
```

Dumpy function

We wrote this code to output an error in case the problem provided cannot be implemented by the both secant and NRM i.e. Euler and Runge Kutta problems provided

```
        % FIX: Placeholder for Abstraction - Simply return NaN and a warning.
        % This completely removes the 'unset' warning.
        function solution = solveEuler(~, ~, ~, ~)
            warning('Euler method is intended for IntegrationProblem and is not implemented here.');
            % Set return value to satisfy the analyzer
            solution.t = NaN;
            solution.v = NaN;
        end

        function solution = solveRungeKutta(~, ~, ~, ~)
            warning('Runge Kutta method is intended for IntegrationProblem and is not implemented here.');
            % Set return value to satisfy the analyzer
            solution.t = NaN;
            solution.v = NaN;
        end
    end
end
```

2.4 SECTION 2: INTEGRATION PROBLEMS

2.4.1 introduction

implement the concepts of classes, encapsulation, inheritance, polymorphism and abstraction. To develop and test a high-end backend implementation of a numerical methods application for solving integration problems.

2.3.2 steps taken in writing the code

Step 1: defining the sub class

We defined the subclass as ***IntegrationProblems*** that inherits from the parent class ***NumericalMethods***

```
classdef IntegrationProblem < NumericalMethod
    %INTEGRATIONPROBLEM Solves integral problems (ordinary differential equations).
    %   Implements Euler and Runge Kutta methods.

```

step 2: Dumpy function

We wrote this code to output an error in case the problem provided cannot be implemented by the both secant and NRM i.e. Euler and Runge Kutta problems provided

```
    methods
        % FIX: Placeholder for Abstraction - Use a structure return to satisfy analyzer.
        function solution = solveNRM(~, ~, ~, ~)
            warning('NRM method is intended for RootFindingProblem and is not implemented here.');
            % Ensure solution is explicitly assigned a structure value.
            solution.result = NaN;
        end

        function solution = solveSecant(~, ~, ~, ~, ~)
            warning('Secant method is intended for RootFindingProblem and is not implemented here.');
            % Ensure solution is explicitly assigned a structure value.
            solution.result = NaN;
        end
```

(i) implementation of Euler method

step 1: defining the function and its initialisation

we defined the function as ***solveEuler*** and set ***v1*** as the starting value, made it run up to ***timeEnd*** times and calculate ***v(i+1)***

```
        % Implementation of Euler Method (Polymorphism)
        function [t, v] = solveEuler(obj, initialValue, stepSize, timeEnd)
            fprintf('--- Euler Method Solution ---\n');
            t = 0:stepSize:timeEnd;
            n = length(t);
            v = zeros(1, n);
            v(1) = initialValue;

            for i = 1:(n - 1)
                dvdt = obj.differentialEq(t(i), v(i));
                v(i+1) = v(i) + stepSize * dvdt;
            end
        end
```

(ii) implementation of Runge Kutta

step 1: defining the function and its initialisation

we defined the function as ***solveRungeKutta*** and set ***v1*** as the starting value, made it run up to ***timeEnd*** times and calculate ***k1, k2, k3, k4***, and then ***v(i+1)***

```
        % Implementation of Runge-Kutta 4th Order (RK4) (Polymorphism)
        function [t, v] = solveRungeKutta(obj, initialValue, stepSize, timeEnd)
            fprintf('--- Runge-Kutta 4th Order Solution ---\n');
            t = 0:stepSize:timeEnd;
            n = length(t);
            v = zeros(1, n);
            v(1) = initialValue;

            for i = 1:(n - 1)
                ti = t(i);
                vi = v(i);

                k1 = obj.differentialEq(ti, vi);
                k2 = obj.differentialEq(ti + 0.5*stepSize, vi + 0.5*stepSize*k1);
                k3 = obj.differentialEq(ti + 0.5*stepSize, vi + 0.5*stepSize*k2);
                k4 = obj.differentialEq(ti + stepSize, vi + stepSize*k3);

                v(i+1) = vi + (stepSize / 6) * (k1 + 2*k2 + 2*k3 + k4);
            end
        end
    end
end
```

SECTION THREE: TEST SCRIPT

The script begins by testing the ***DifferentialProblem*** subclass for root-finding problems using the Newton-Raphson and Secant methods. These functions determine the velocity at which the net force acting on a falling object equals zero, representing the terminal velocity.

```
% TestScript.m
clear; clc;

% --- PART 1: Differential Problem (Root Finding) ---
fprintf('*** Testing RootFindingProblem (NRM & Secant) ***\n');
% Create an instance of the RootFindingProblem subclass
rootSolver = DifferentialPoblem();

% NRM Test: Find the velocity v that makes f(v)=0 (terminal velocity)
initialGuess = 50;
tolerance = 1e-4;
maxIter = 100;
v_nrm = rootSolver.solveNRM(initialGuess, tolerance, maxIter);
fprintf('Terminal Velocity (NRM): %.4f m/s\n\n', v_nrm);

% Secant Test
x0 = 0;
x1 = 50;
v_secant = rootSolver.solveSecant(x0, x1, tolerance, maxIter);
fprintf('Terminal Velocity (Secant): %.4f m/s\n\n', v_secant);

```

In the second part, the ***IntegrationProblem*** subclass is tested for solving an ordinary differential equation representing motion under gravity and air resistance. Both Euler's Method and the Runge-Kutta 4th Order Method (RK4) are applied to compute velocity over time.

```
% --- PART 2: Integral Problem (ODE Solving) ---
fprintf('*** Testing IntegrationProblem (Euler & Runge Kutta 4th order) ***\n');
% Create an instance of the IntegrationProblem subclass
odeSolver = IntegrationProblem();

% Common Parameters for ODE:
initialVelocity = 0; % Start from rest
stepSize = 0.5;
timeEnd = 10; % Solve for time t = 0 to 10 seconds

% Euler Test: dv/dt = f(v)
[t_e, v_e] = odeSolver.solveEuler(initialVelocity, stepSize, timeEnd);
fprintf('Velocity after %.1f s (Euler): %.4f m/s\n\n', timeEnd, v_e(end));

% RK4 Test
[t_rk, v_rk] = odeSolver.solveRungeKutta(initialVelocity, stepSize, timeEnd);
fprintf('Velocity after %.1f s (Runge Kutta 4th Order): %.4f m/s\n\n', timeEnd, v_rk(end));
```

OUTPUT

The results are displayed in the MATLAB Command Window, showing convergence values and computed velocities. These tests verify that the designed OOP framework correctly encapsulates numerical methods within an extendable and maintainable structure.

```
*** Testing RootFindingProblem (NRM & Secant) ***
--- NRM Solution ---
Converged in 3 iterations.
Terminal Velocity (NRM): 56.0286 m/s

--- Secant Solution ---
Converged in 6 iterations.
Terminal Velocity (Secant): 56.0286 m/s

*** Testing IntegrationProblem (Euler & Runge Kutta 4th order) ***
--- Euler Method Solution ---
Velocity after 10.0 s (Euler): -Inf m/s

--- Runge-Kutta 4th Order Solution ---
Velocity after 10.0 s (Runge Kutta 4th Order): -Inf m/s
```

CHAPTER THREE: CONCLUSION AND RECOMMENDATION

3.1 CONCLUSION

This project successfully demonstrated the application of object-oriented programming (OOP) principles—abstraction, encapsulation, inheritance, and polymorphism—in implementing numerical methods using MATLAB.

By structuring the program into a parent abstract class and two subclasses, the design achieved modularity, code reusability, and clarity in handling different computational problems.

3.2 RECOMMENDATIONS

- Future work can include additional numerical methods such as Bisection, Gauss-Seidel, Adams-Bashforth, or Finite Difference methods within the same OOP framework.
- A MATLAB GUI can be developed to make the application user-friendly, allowing users to select methods, input functions, and view graphical results interactively.
- Future implementations may incorporate vectorization and parallel computing to improve computational efficiency, especially for large datasets or complex simulations.
- Introduce automatic error estimation and comparison with analytical or exact solutions to enhance result validation and learning outcomes.
- The developed framework can be adapted for teaching numerical methods, allowing students to visualize algorithm behaviour and understand OOP concepts in a practical setting.

CHAPTER FOUR: REFERENCES

- course Lecture notes (module 1 – 5)
- Matlab documentation

APPENDICES