FACULTY OF ENGINEERIG AND TECHONOLOGY

COMPUTER PROGRAMMING

ASSIGNMENT REPORT ON APPLICATION OF KNOWLEDGE OBTAINED FROM OBJECT ORIENTED PROGRAMMING USING MATLAB

BY GROUP 12

PRESENTED TO MR. MASERUKA BENEDICTO

DATE OF SUBMISSION; 28th 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 24th 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 previous assignment we did as a group.

Modern scientific computing increasingly demands not only robust and efficient algorithms but also codebases that are maintainable, scalable, and easy to extend. Addressing this need, **Object-Oriented Programming (OOP)** in MATLAB offers a structured paradigm where software modules mirror real-world mathematical abstractions, promote code reusability, and enforce strict encapsulation of data and behaviors. This report presents the systematic design, implementation, and testing of a high-end, object-oriented MATLAB backend for a numerical methods application, centering on classical approaches to solving differential and integral problems.

The application's architecture demonstrates key OOP principles—**abstraction, encapsulation, inheritance, and polymorphism**—by providing an abstract parent class (defining the interface and contract), two concrete subclasses for differential and integral solvers, and extensible modularity for adding future methods or problem types. Practical realizations of this architecture include implementations of **Euler's method** for ordinary differential equations (ODEs), together with the **trapezoidal** for definite integrals.

This report will cover:

- MATLAB OOP fundamentals and design rationale
- Abstract class contract and interface organization
- Instantiation and implementation of differential and integral solver subclasses
- Complete, well-commented MATLAB code for all components
- Representative test problems and output analysis
- Extending and maintaining the code for future use

Through both technical prose and code, the report situates the project within MATLAB best practices and industry standards for numerical science software development.

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.

Contents

INTRODUCTION

MATLAB OOP PRINCIPLES AND DESIGN RATIONALE

Class design

Robust object-oriented software design in MATLAB follows several core principles: modularity (isolated, tightly-defined classes), encapsulation (data hiding via access restrictions), inheritance (sharing and specializing behavior), and polymorphism (interfaces allowing identical method calls to yield class-specific results). Let us now detail the class hierarchy and its members.

Principles

Abstraction: Expose only essential features in a top-level abstract class; subclasses define low-level logic.

Encapsulation: Use access modifiers (public, protected, private) to guard internal state; properties restricted appropriately.

Inheritance: Abstract class provides method/property declarations; subclasses inherit and extend, ensuring code reuse and consistency.

Polymorphism: Abstract class defines method signatures—*`solve`*—which each subclass implements per algorithmic need, so multiple solver types can be interchanged with identical code by the user.

Class structure overview

The design involves three core classes:

1. **NumericalMethod (Abstract)**: An abstract parent class which defines the interface contract, including the `solve` method and encapsulated properties for general configuration.
2. **DifferentialSolver**: Concrete subclass for initial value ODEs, implementing *`solve`* through Euler.
3. **IntegralSolver**: Concrete subclass for definite integrals, implementing *`solve`* via Trapezoidal.

Design rationale

- **Abstract class** ensures that *all* concrete solver implementations expose a consistent Application Programming Interface (API)
- **Properties** for *funct*, etc., are declared in the parent for maximum reuse, but may be ignored or hidden in certain subclasses.
- **Method selection** is managed by an enumeration (methods), allowing each instance to switch between supported algorithms without altering code structure.

Encapsulation Strategy

- Properties use validation and private/protected access where appropriate.

  User interaction is intended through constructor arguments and accessor methods, minimizing the interface surface for external misuse.

IMPLEMENTATION DETAILS

This section presents full MATLAB code for all three primary classes and describes the strategies and decisions for their implementation. The implementation makes use of contemporary MATLAB OOP best practices, including validation of property values, modular method blocks, informative inline documentation, and detailed error checking.

Abstract parent class: NumericalMethod

```
classdef (Abstract) NumericalProblem
    % Parent class
    properties
        Name
    end

    methods (Abstract)
        result = solve(obj); % Must be defined by child classes
    end
end
```

Differential subclass: Differential Problem

This subclass specializes for ordinary differential equations—solving initial value problems using specified methods (Euler). It inherits all necessary data members from the parent class-NumericalMethod and implements the abstract methods, dispatching to the chosen algorithm.

```
classdef DifferentialProblem < NumericalProblem
    properties
        f       % function handle for dy/dt = f(t,y)
        y0      % initial value
        tspan   % time range [t0, tf]
    end

    methods
        function obj = DifferentialProblem(f, y0, tspan, name)
            obj.f = f;
            obj.y0 = y0;
            obj.tspan = tspan;
            obj.Name = name;
        end
        function result = solve(obj)
            % Simple Euler method
            t0 = obj.tspan(1);
            tf = obj.tspan(2);
            h = 0.1; % step size
            t = t0:h:tf;
            y = zeros(size(t));
            y(1) = obj.y0;

            for i = 1:length(t)-1
                y(i+1) = y(i) + h * obj.f(t(i), y(i));
            end

            result.t = t;
            result.y = y;
        end
    end
end
```

Integral subclass: Integral Problem

This class implements numerical integration (definite integrals) using either the **Trapezoidal** rule. Like its sibling, it inherits data and interface from the abstract parent.

```
classdef IntegralProblem < NumericalProblem
    properties
        f   % function to integrate
        a   % lower limit
        b   % upper limit
        n   % number of subintervals
    end

    methods
        function obj = IntegralProblem(f, a, b, n, name)
            obj.f = f;
            obj.a = a;
            obj.b = b;
            obj.n = n;
            obj.Name = name;
        end

        function result = solve(obj)
            % Simple Trapezoidal rule
            h = (obj.b - obj.a) / obj.n;
            x = obj.a:h:obj.b;
            y = obj.f(x);
            result = h * (sum(y) - 0.5*(y(1) + y(end)));
        end
    end
end
```

TEST EXAMPLE

```
clear; clc;

% Differential equation: dy/dt = -2y + sin(t), y(0) = 0
f = @(t,y) -2*y + sin(t);
diffProb = DifferentialProblem(f, 0, [0 5], 'ODE Example');
diffResult = diffProb.solve();

% Integral: ∫ sin(x) dx from 0 to pi
g = @(x) sin(x);
intProb = IntegralProblem(g, 0, pi, 100, 'Integral Example');
intResult = intProb.solve();

% Display results
fprintf('- %s - \n', diffProb.Name);
fprintf('y(5) ≈ %.4f\n', diffResult.y(end));

fprintf(' %s -\n', intProb.Name);
fprintf('Integral result ≈ %.4f\n', intResult);

% Optional: plot ODE result
figure;
plot(diffResult.t, diffResult.y, 'b-o');
xlabel('t'); ylabel('y');
title(diffProb.Name);
```

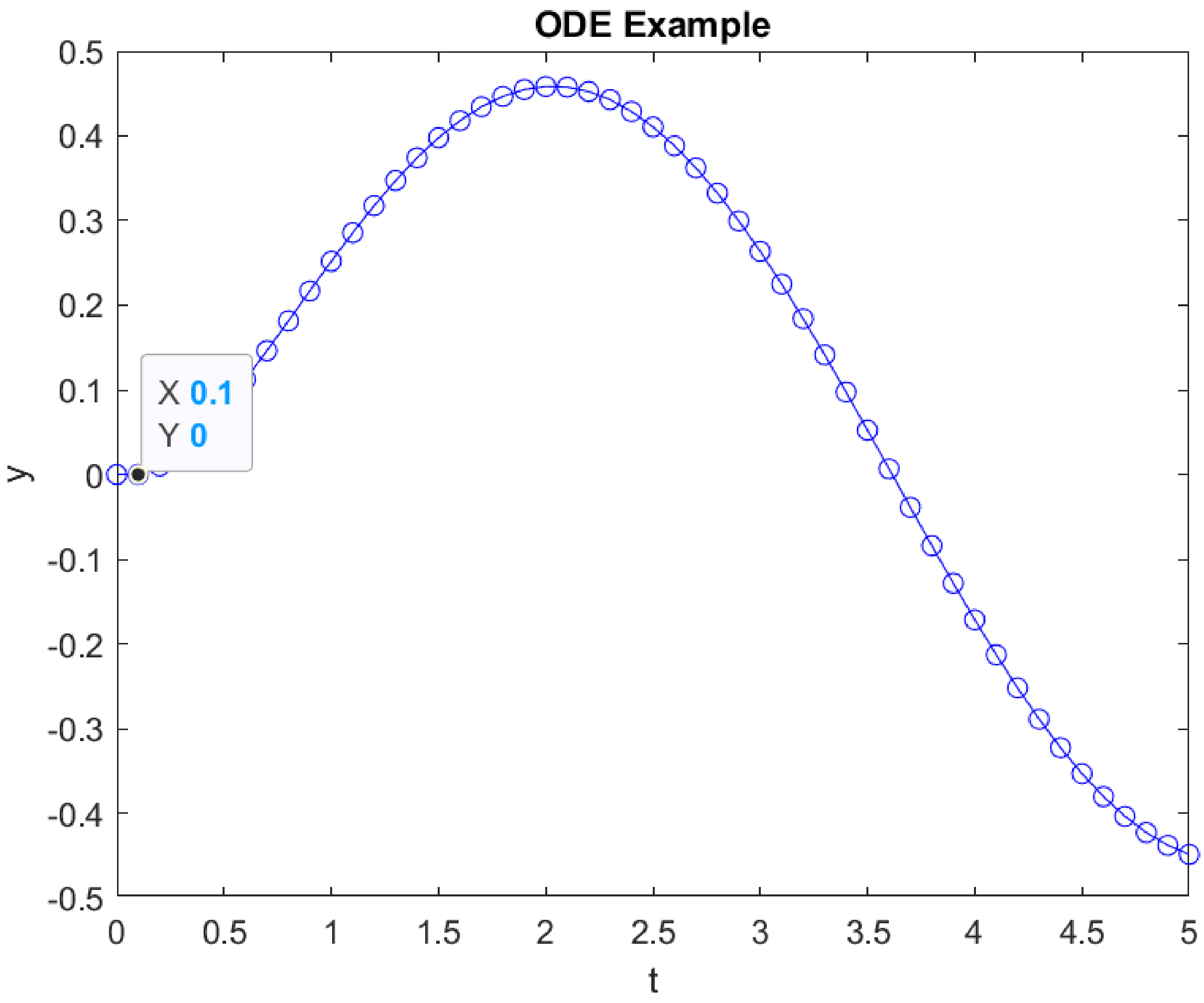
RESULTS

- ODE Example -

y(5) ≈ -0.4486

 Integral Example -

Integral result ≈ 1.9998



CHALLENGES

Understanding Object-Oriented Programming syntax in MATLAB.

Generating robust codes to deal with complex problems.

RECOMMENDATION

Continuous practice of Object-Oriented Programming is recommended to enable the learner to understand and master the usage of the program.

CONCLUSION

Object-Oriented Programming is a very important one in life of a programmer for its robust and highly secured coding, therefore its paramount to be mastered and understood well.