

FACULTY OF ENGINEERING AND TECHNOLOGY

COMPUTER PROGRAMMING

COURSE UNIT: MATLAB

**A REPORT ABOUT THE MATLAB ASSIGNMENT FIVE**

SUBMITTED BY:

NAME: GROUP 4

LECTURER: MR. BENDICTO MASERUKA

DATE OF SUBMISSION:………………………………………..

# ACKNOWLEDGEMENT

To begin with, we, Group 4, would like to thank The Almighty God for guiding and helping us to carry on with our assignment. We extend our heartfelt gratitude and appreciation to all member that gave a hand in the accomplishment of this assignment.

Secondly, special appreciation goes to our lecturer, Mr. Maseruka Bendicto for his guidance in this course unit. Your expertise and enthusiasm have greatly enhanced our understanding.

Lastly, we also appreciate the collaborative efforts and contributions of each group member, which enabled us to complete this assignment successfully

# DEDICATION

We dedicate this report to all individuals especially group 4 members, for their teamwork, dedication, and perseverance in completing this assignment

To our lecturer Mr. Maseruka Bendicto whose guidance and expertise have been priceless, mentorship and insightful feedback have diversified our understanding.

# APPROVAL

This is to confirm that this report has been written and presented by GROUP 4, giving the details of the MATLAB assignment carried out.

LECTURER;

NAME:…………………………………..

SIGNATURE:……………………………..

DATE:……………………………………

# DECLARATION

We hereby declare and certify that the information in this report is out of our own efforts, research as group 4 and it has never been used by any individual or submitted in any learning institution for any academic purposes

# ABSTRACT

As group 4 we met in the university library and discussed about our assignment which boosted our exposure to various cords in addition to the acquired knowledge from the previous lectures. Our discussion and research as a group helped us come up with a solution for the given assignment.

We hereby declare and certify that the information in this report is out of our own efforts, research as group 4 and it has never been used by any individual or submitted in any learning institution for any academic purposes.

Table of Contents

[ACKNOWLEDGEMENT 2](#_Toc212580003)

[DEDICATION 3](#_Toc212580004)

[APPROVAL 4](#_Toc212580005)

[DECLARATION 5](#_Toc212580006)

[ABSTRACT 6](#_Toc212580007)

[LIST OF GROUP 4 MEMBERS 8](#_Toc212580008)

[CHAPTER 1: MATLAB OVERVIEW 9](#_Toc212580009)

[BACKGROUND 9](#_Toc212580010)

[CHAPTER 2: STUDY OVERVIEW 10](#_Toc212580011)

[NEWTON RAPHSON SOLVER 10](#_Toc212580012)

[ROOT FINDING SOLVER 12](#_Toc212580013)

[NUMERICAL SOLVER 14](#_Toc212580014)

[BISECTION SOLVER 16](#_Toc212580015)

[SECANT SOLVER 19](#_Toc212580016)

[DIFFERENTIAL EQUATION SOLVER 21](#_Toc212580017)

[RUNGEKUTTA 2 SOLVER 25](#_Toc212580018)

[RUNGE KUTTA 4 SOLVER 27](#_Toc212580019)

[EULER SOLVER 29](#_Toc212580020)

[NUMERICAL METHODS APPLICATION 31](#_Toc212580021)

[CHAPTER 3:CONCLUSION 41](#_Toc212580022)

# LIST OF GROUP 4 MEMBERS

|  |  |  |
| --- | --- | --- |
| NAME | PROGRAMME | SIGNATURE |
| ASIMO MARTHA | MEB |  |
| BANGI REHEMAH FARIDAH | AMI |  |
| MBABAZI GILLIAN BLESSING | PTI |  |
| AMONGIN MARY LUCY | WAR |  |
| MWIDU PETER FREDRICK | WAR 3 |  |
| OCAA ELI | WAR |  |
| ADUPA GEORGE RICHARD | WAR |  |
| TUSIME JUNIOUR YUSUF | AMI |  |
| OKOT SANDRA | WAR |  |
| WAMBWA DERRICK | WAR |  |

# CHAPTER 1: MATLAB OVERVIEW

# BACKGROUND

MATLAB is a high level, multi paradigm programming language and environment developed by MathWorks. It is widely used in academia, research, and industry for numerical computation, data analysis, visualization and modelling.

# CHAPTER 2: STUDY OVERVIEW

Assignment question.

With the concepts of classes,encapsulation,polymorphism and abstrction.

Develop and test a high level back end implementation of the numerical methods applications for solving computational problems.

## NEWTON RAPHSON SOLVER

classdef NewtonRaphsonSolver < RootFindingSolver

% NEWTONRAPHSONSOLVER Concrete implementation of Newton-Raphson method

methods

function obj = NewtonRaphsonSolver(f, df, x0, tolerance, maxIterations)

obj = obj@RootFindingSolver('Newton-Raphson', f, df, x0, tolerance, maxIterations);

end

function result = solve(obj)

% Implement Newton-Raphson method recursively

tic;

obj.validateParameters();

[obj.root, obj.iterations, obj.history] = obj.recursiveNewton(...

obj.initialGuess, 1, []);

obj.functionValue = obj.problemFunction(obj.root);

obj.computationTime = toc;

result = struct('root', obj.root, 'iterations', obj.iterations, ...

'time', obj.computationTime, 'converged', obj.hasConverged);

end

function [root, iter, history] = recursiveNewton(obj, x0, currentIter, history)

% Recursive Newton-Raphson implementation

if currentIter > obj.maxIterations

root = x0;

iter = currentIter - 1;

return;

end

fx = obj.problemFunction(x0);

dfx = obj.derivativeFunction(x0);

if abs(dfx) < eps

error('Derivative too small at iteration %d', currentIter);

end

x1 = x0 - fx/dfx;

history(currentIter) = x1;

error\_val = abs(x1 - x0);

obj.recordHistory(x1, error\_val);

fprintf(' Iteration %d: x = %.8f, f(x) = %.2e, error = %.2e\n', ...

currentIter, x1, obj.problemFunction(x1), error\_val);

if obj.checkConvergence(error\_val, currentIter)

root = x1;

iter = currentIter;

else

[root, iter, history] = obj.recursiveNewton(x1, currentIter + 1, history);

end

end

end

end

## ROOT FINDING SOLVER

classdef (Abstract) RootFindingSolver < NumericalSolver

% ROOTFINDINGSOLVER Abstract class for root finding methods

% Inherits from NumericalSolver and adds root-specific functionality

properties (Constant)

SOLVER\_TYPE = 'Root Finding'

end

properties

root

initialGuess

functionValue

derivativeFunction

end

methods

function obj = RootFindingSolver(name, f, df, x0, tolerance, maxIterations)

% Call parent constructor

obj = obj@NumericalSolver(name, tolerance, maxIterations);

if nargin > 0

obj.problemFunction = f;

obj.derivativeFunction = df;

obj.initialGuess = x0;

end

end

function validateParameters(obj)

% Validate root finding parameters

if isempty(obj.problemFunction)

error('Problem function must be specified');

end

if obj.tolerance <= 0

error('Tolerance must be positive');

end

if obj.maxIterations <= 0

error('Maximum iterations must be positive');

end

fprintf('✓ %s parameters validated\n', obj.name);

end

function displayResults(obj)

% Display root finding results

fprintf('\n=== %s RESULTS ===\n', upper(obj.name));

fprintf('Method: %s\n', obj.name);

fprintf('Root: %.8f\n', obj.root);

fprintf('f(root): %.2e\n', obj.functionValue);

fprintf('Iterations: %d\n', obj.iterations);

fprintf('Computation Time: %.6f seconds\n', obj.computationTime);

fprintf('Converged: %s\n', string(obj.hasConverged));

if obj.hasConverged

fprintf('✓ Solution converged within tolerance\n');

else

fprintf('✗ Solution did not converge within maximum iterations\n');

end

end

end

end

## NUMERICAL SOLVER

classdef (Abstract) NumericalSolver

% NUMERICALSOLVER Abstract base class for all numerical solvers

% Implements abstraction and common functionality

properties (Abstract, Constant)

SOLVER\_TYPE % Must be defined by subclasses

end

properties

name

tolerance

maxIterations

computationTime

iterations

history

errorHistory

end

properties (Access = protected)

problemFunction

hasConverged

end

methods (Abstract)

% Abstract methods that must be implemented by subclasses

solve(obj)

validateParameters(obj)

displayResults(obj)

end

methods

function obj = NumericalSolver(name, tolerance, maxIterations)

% Constructor for base class

if nargin > 0

obj.name = name;

obj.tolerance = tolerance;

obj.maxIterations = maxIterations;

obj.computationTime = 0;

obj.iterations = 0;

obj.history = [];

obj.errorHistory = [];

obj.hasConverged = false;

end

end

function convergence = checkConvergence(obj, currentError, iteration)

% Common convergence checking method

convergence = (currentError < obj.tolerance) || (iteration >= obj.maxIterations);

obj.hasConverged = convergence;

end

function recordHistory(obj, value, error)

% Record iteration history

obj.history = [obj.history, value];

obj.errorHistory = [obj.errorHistory, error];

end

function plotConvergence(obj)

% Common convergence plotting method

if ~isempty(obj.history)

figure;

subplot(2,1,1);

plot(1:length(obj.history), obj.history, 'b-o', 'LineWidth', 2, 'MarkerSize', 4);

xlabel('Iteration');

ylabel('Solution Value');

title([obj.name ' - Solution History']);

grid on;

subplot(2,1,2);

semilogy(1:length(obj.errorHistory), abs(obj.errorHistory), 'r-s', 'LineWidth', 2, 'MarkerSize', 4);

xlabel('Iteration');

ylabel('Absolute Error');

title([obj.name ' - Error Convergence']);

grid on;

end

end

end

end

## BISECTION SOLVER

classdef BisectionSolver < RootFindingSolver

% BISECTIONSOLVER Concrete implementation of Bisection method

properties

lowerBound

upperBound

end

methods

function obj = BisectionSolver(f, a, b, tolerance, maxIterations)

obj = obj@RootFindingSolver('Bisection', f, [], (a+b)/2, tolerance, maxIterations);

obj.lowerBound = a;

obj.upperBound = b;

end

function result = solve(obj)

% Implement Bisection method recursively

tic;

obj.validateParameters();

if obj.problemFunction(obj.lowerBound) \* obj.problemFunction(obj.upperBound) > 0

error('Function must have opposite signs at endpoints');

end

[obj.root, obj.iterations, obj.history] = obj.recursiveBisection(...

obj.lowerBound, obj.upperBound, 1, []);

obj.functionValue = obj.problemFunction(obj.root);

obj.computationTime = toc;

result = struct('root', obj.root, 'iterations', obj.iterations, ...

'time', obj.computationTime, 'converged', obj.hasConverged);

end

function [root, iter, history] = recursiveBisection(obj, a, b, currentIter, history)

% Recursive Bisection implementation

if currentIter > obj.maxIterations

root = (a + b) / 2;

iter = currentIter - 1;

return;

end

c = (a + b) / 2;

fc = obj.problemFunction(c);

history(currentIter) = c;

error\_val = abs(fc);

obj.recordHistory(c, error\_val);

fprintf(' Iteration %d: a=%.6f, b=%.6f, c=%.8f, f(c)=%.2e\n', ...

currentIter, a, b, c, fc);

if obj.checkConvergence(error\_val, currentIter) || (b - a)/2 < obj.tolerance

root = c;

iter = currentIter;

obj.hasConverged = true;

else

if obj.problemFunction(a) \* fc < 0

[root, iter, history] = obj.recursiveBisection(a, c, currentIter + 1, history);

else

[root, iter, history] = obj.recursiveBisection(c, b, currentIter + 1, history);

end

end

end

end

end

## SECANT SOLVER

classdef SecantSolver < RootFindingSolver

% SECANTSOLVER Concrete implementation of Secant method

properties

secondGuess

end

methods

function obj = SecantSolver(f, x0, x1, tolerance, maxIterations)

obj = obj@RootFindingSolver('Secant', f, [], x0, tolerance, maxIterations);

obj.secondGuess = x1;

end

function result = solve(obj)

% Implement Secant method recursively

tic;

obj.validateParameters();

[obj.root, obj.iterations, obj.history] = obj.recursiveSecant(...

obj.initialGuess, obj.secondGuess, 1, []);

obj.functionValue = obj.problemFunction(obj.root);

obj.computationTime = toc;

result = struct('root', obj.root, 'iterations', obj.iterations, ...

'time', obj.computationTime, 'converged', obj.hasConverged);

end

function [root, iter, history] = recursiveSecant(obj, x0, x1, currentIter, history)

% Recursive Secant implementation

if currentIter > obj.maxIterations

root = x1;

iter = currentIter - 1;

fprintf(' Maximum iterations reached: %d\n', obj.maxIterations);

return;

end

fx0 = obj.problemFunction(x0);

fx1 = obj.problemFunction(x1);

if abs(fx1 - fx0) < eps

error('Division by zero avoided at iteration %d', currentIter);

end

x2 = x1 - fx1 \* (x1 - x0) / (fx1 - fx0);

history(currentIter) = x2;

error\_val = abs(x2 - x1);

obj.recordHistory(x2, error\_val);

fprintf(' Iteration %d: x = %.8f, f(x) = %.2e, error = %.2e\n', ...

currentIter, x2, obj.problemFunction(x2), error\_val);

if obj.checkConvergence(error\_val, currentIter)

root = x2;

iter = currentIter;

fprintf(' ✓ Converged in %d iterations\n', currentIter);

else

[root, iter, history] = obj.recursiveSecant(x1, x2, currentIter + 1, history);

end

end

function validateParameters(obj)

% Validate Secant method parameters

validateParameters@RootFindingSolver(obj);

if isempty(obj.secondGuess)

error('Second initial guess must be specified for Secant method');

end

if obj.initialGuess == obj.secondGuess

error('Initial guesses must be different for Secant method');

end

fprintf('✓ Secant method parameters validated\n');

end

end

end

## DIFFERENTIAL EQUATION SOLVER

classdef (Abstract) DifferentialEquationSolver < NumericalSolver

% DIFFERENTIALEQUATIONSOLVER Abstract class for ODE solvers

% Inherits from NumericalSolver and adds ODE-specific functionality

properties (Constant)

SOLVER\_TYPE = 'Differential Equation'

end

properties

timeVector

solution

analyticalSolution

initialCondition

stepSize

odeFunction

maxError

meanError

end

methods

function obj = DifferentialEquationSolver(name, odeFun, tSpan, h, y0, analyticalFun, tolerance, maxIterations)

% Call parent constructor

obj = obj@NumericalSolver(name, tolerance, maxIterations);

if nargin > 0

obj.odeFunction = odeFun;

obj.timeVector = tSpan(1):h:tSpan(2);

obj.stepSize = h;

obj.initialCondition = y0;

obj.analyticalSolution = analyticalFun;

end

end

function validateParameters(obj)

% Validate ODE solver parameters

if isempty(obj.odeFunction)

error('ODE function must be specified');

end

if obj.stepSize <= 0

error('Step size must be positive');

end

if isempty(obj.initialCondition)

error('Initial condition must be specified');

end

fprintf('✓ %s parameters validated\n', obj.name);

end

function calculateErrors(obj)

% Calculate various error metrics

if ~isempty(obj.analyticalSolution) && ~isempty(obj.solution)

analyticalValues = obj.analyticalSolution(obj.timeVector);

errors = abs(obj.solution - analyticalValues);

obj.maxError = max(errors);

obj.meanError = mean(errors);

obj.errorHistory = errors;

end

end

function displayResults(obj)

% Display ODE solver results

fprintf('\n=== %s RESULTS ===\n', upper(obj.name));

fprintf('Method: %s\n', obj.name);

fprintf('Step Size: %.4f\n', obj.stepSize);

fprintf('Time Steps: %d\n', length(obj.timeVector));

fprintf('Computation Time: %.6f seconds\n', obj.computationTime);

if ~isempty(obj.maxError)

fprintf('Maximum Error: %.2e\n', obj.maxError);

fprintf('Mean Error: %.2e\n', obj.meanError);

end

fprintf('Final Value: y(%.2f) = %.6f\n', obj.timeVector(end), obj.solution(end));

end

function plotComparison(obj)

% Plot numerical vs analytical solution

if ~isempty(obj.analyticalSolution)

figure;

t\_fine = linspace(obj.timeVector(1), obj.timeVector(end), 1000);

y\_analytical = obj.analyticalSolution(t\_fine);

plot(t\_fine, y\_analytical, 'k-', 'LineWidth', 3, 'DisplayName', 'Analytical');

hold on;

plot(obj.timeVector, obj.solution, 'r--o', 'LineWidth', 2, 'MarkerSize', 4, 'DisplayName', obj.name);

xlabel('Time');

ylabel('Solution');

title(['ODE Solution: ' obj.name ' vs Analytical']);

legend('Location', 'best');

grid on;

% Plot errors

figure;

errors = abs(obj.solution - obj.analyticalSolution(obj.timeVector));

plot(obj.timeVector, errors, 'b-s', 'LineWidth', 2, 'MarkerSize', 4);

xlabel('Time');

ylabel('Absolute Error');

title(['Error Analysis: ' obj.name]);

grid on;

end

end

end

end

## RUNGEKUTTA 2 SOLVER

classdef RungeKutta2Solver < DifferentialEquationSolver

% RUNGEKUTTA2SOLVER Concrete implementation of RK2 method (Heun's method)

methods

function obj = RungeKutta2Solver(odeFun, tSpan, h, y0, analyticalFun, tolerance, maxIterations)

obj = obj@DifferentialEquationSolver('Runge-Kutta 2', odeFun, tSpan, h, y0, analyticalFun, tolerance, maxIterations);

end

function result = solve(obj)

% Implement RK2 method recursively

tic;

obj.validateParameters();

[obj.timeVector, obj.solution] = obj.recursiveRK2(...

obj.timeVector(1), obj.initialCondition, 1, obj.timeVector(1), obj.initialCondition);

obj.calculateErrors();

obj.computationTime = toc;

result = struct('time', obj.timeVector, 'solution', obj.solution, ...

'max\_error', obj.maxError, 'time\_elapsed', obj.computationTime);

end

function [t\_array, y\_array] = recursiveRK2(obj, t\_current, y\_current, step, t\_array, y\_array)

% Recursive RK2 implementation (Heun's method)

if t\_current >= obj.timeVector(end)

return;

end

t\_next = t\_current + obj.stepSize;

if t\_next > obj.timeVector(end)

t\_next = obj.timeVector(end);

h\_actual = t\_next - t\_current;

else

h\_actual = obj.stepSize;

end

% RK2 calculations (Heun's method)

k1 = h\_actual \* obj.odeFunction(t\_current, y\_current);

k2 = h\_actual \* obj.odeFunction(t\_current + h\_actual, y\_current + k1);

y\_next = y\_current + (k1 + k2) / 2;

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

if mod(step, 10) == 0

fprintf(' Step %d: t = %.2f, y = %.6f\n', step, t\_next, y\_next);

end

% Recursive call

[t\_array, y\_array] = obj.recursiveRK2(t\_next, y\_next, step + 1, t\_array, y\_array);

end

end

end

## RUNGE KUTTA 4 SOLVER

classdef RungeKutta4Solver < DifferentialEquationSolver

% RUNGEKUTTA4SOLVER Concrete implementation of RK4 method

methods

function obj = RungeKutta4Solver(odeFun, tSpan, h, y0, analyticalFun, tolerance, maxIterations)

obj = obj@DifferentialEquationSolver('Runge-Kutta 4', odeFun, tSpan, h, y0, analyticalFun, tolerance, maxIterations);

end

function result = solve(obj)

% Implement RK4 method recursively

tic;

obj.validateParameters();

[obj.timeVector, obj.solution] = obj.recursiveRK4(...

obj.timeVector(1), obj.initialCondition, 1, obj.timeVector(1), obj.initialCondition);

obj.calculateErrors();

obj.computationTime = toc;

result = struct('time', obj.timeVector, 'solution', obj.solution, ...

'max\_error', obj.maxError, 'time\_elapsed', obj.computationTime);

end

function [t\_array, y\_array] = recursiveRK4(obj, t\_current, y\_current, step, t\_array, y\_array)

% Recursive RK4 implementation

if t\_current >= obj.timeVector(end)

return;

end

t\_next = t\_current + obj.stepSize;

if t\_next > obj.timeVector(end)

t\_next = obj.timeVector(end);

h\_actual = t\_next - t\_current;

else

h\_actual = obj.stepSize;

end

% RK4 calculations

k1 = h\_actual \* obj.odeFunction(t\_current, y\_current);

k2 = h\_actual \* obj.odeFunction(t\_current + h\_actual/2, y\_current + k1/2);

k3 = h\_actual \* obj.odeFunction(t\_current + h\_actual/2, y\_current + k2/2);

k4 = h\_actual \* obj.odeFunction(t\_current + h\_actual, y\_current + k3);

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

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

if mod(step, 10) == 0

fprintf(' Step %d: t = %.2f, y = %.6f\n', step, t\_next, y\_next);

end

% Recursive call

[t\_array, y\_array] = obj.recursiveRK4(t\_next, y\_next, step + 1, t\_array, y\_array);

end

end

end

## EULER SOLVER

classdef EulerSolver < DifferentialEquationSolver

% EULERSOLVER Concrete implementation of Euler method

methods

function obj = EulerSolver(odeFun, tSpan, h, y0, analyticalFun, tolerance, maxIterations)

obj = obj@DifferentialEquationSolver('Euler', odeFun, tSpan, h, y0, analyticalFun, tolerance, maxIterations);

end

function result = solve(obj)

% Implement Euler method recursively

tic;

obj.validateParameters();

[obj.timeVector, obj.solution] = obj.recursiveEuler(...

obj.timeVector(1), obj.initialCondition, 1, obj.timeVector(1), obj.initialCondition);

obj.calculateErrors();

obj.computationTime = toc;

result = struct('time', obj.timeVector, 'solution', obj.solution, ...

'max\_error', obj.maxError, 'time\_elapsed', obj.computationTime);

end

function [t\_array, y\_array] = recursiveEuler(obj, t\_current, y\_current, step, t\_array, y\_array)

% Recursive Euler implementation

if t\_current >= obj.timeVector(end)

return;

end

t\_next = t\_current + obj.stepSize;

if t\_next > obj.timeVector(end)

t\_next = obj.timeVector(end);

h\_actual = t\_next - t\_current;

else

h\_actual = obj.stepSize;

end

% Euler calculation

y\_next = y\_current + h\_actual \* obj.odeFunction(t\_current, y\_current);

% Store results

t\_array(step + 1) = t\_next;

y\_array(step + 1) = y\_next;

if mod(step, 10) == 0

fprintf(' Step %d: t = %.2f, y = %.6f\n', step, t\_next, y\_next);

end

% Recursive call

[t\_array, y\_array] = obj.recursiveEuler(t\_next, y\_next, step + 1, t\_array, y\_array);

end

end

end

## NUMERICAL METHODS APPLICATION

classdef NumericalMethodsApp

% NUMERICALMETHODSAPP Main application controller

% Demonstrates polymorphism and class hierarchy usage

properties

rootSolvers

odeSolvers

results

end

methods

function obj = NumericalMethodsApp()

obj.rootSolvers = {};

obj.odeSolvers = {};

obj.results = struct();

end

function obj = runRootFindingAnalysis(obj)

% Run root finding analysis using polymorphism

fprintf('\n=== ROOT FINDING ANALYSIS ===\n');

% Define the problem: f(r) = (8/3)\*pi\*r - 2000/r^2

f = @(r) (8/3)\*pi\*r - 2000./(r.^2);

df = @(r) (8/3)\*pi + 4000./(r.^3);

% Create different solvers (polymorphism in action)

newtonSolver = NewtonRaphsonSolver(f, df, 10, 1e-8, 100);

bisectionSolver = BisectionSolver(f, 5, 15, 1e-8, 100);

secantSolver = SecantSolver(f, 5, 15, 1e-8, 100);

obj.rootSolvers = {newtonSolver, bisectionSolver, secantSolver};

% Solve using each method (same interface, different implementations)

rootResults = [];

for i = 1:length(obj.rootSolvers)

solver = obj.rootSolvers{i};

fprintf('\n--- Solving with %s ---\n', solver.name);

try

result = solver.solve();

solver.displayResults();

solver.plotConvergence();

rootResults = [rootResults, result];

catch ME

fprintf(' ✗ Error in %s: %s\n', solver.name, ME.message);

end

end

obj.results.rootFinding = rootResults;

obj.plotRootFindingComparison();

end

function obj = runODEAnalysis(obj)

% Run ODE analysis using polymorphism

fprintf('\n=== ODE SOLVING ANALYSIS ===\n');

% Define the ODE problem: dy/dt = -k\*y, y(0) = y0

k = 0.5;

y0 = 100;

dydt = @(t, y) -k\*y;

analytical = @(t) y0 \* exp(-k\*t);

% Create different ODE solvers (polymorphism)

rk4Solver = RungeKutta4Solver(dydt, [0, 10], 0.1, y0, analytical, 1e-6, 1000);

rk2Solver = RungeKutta2Solver(dydt, [0, 10], 0.1, y0, analytical, 1e-6, 1000);

eulerSolver = EulerSolver(dydt, [0, 10], 0.1, y0, analytical, 1e-6, 1000);

obj.odeSolvers = {rk4Solver, rk2Solver, eulerSolver};

% Solve using each method

odeResults = [];

for i = 1:length(obj.odeSolvers)

solver = obj.odeSolvers{i};

fprintf('\n--- Solving with %s ---\n', solver.name);

try

result = solver.solve();

solver.displayResults();

solver.plotComparison();

odeResults = [odeResults, result];

catch ME

fprintf(' ✗ Error in %s: %s\n', solver.name, ME.message);

end

end

obj.results.odeSolving = odeResults;

obj.plotODEComparison();

end

function plotRootFindingComparison(obj)

% Compare root finding methods

if isfield(obj.results, 'rootFinding') && ~isempty(obj.results.rootFinding)

figure('Position', [100, 100, 1200, 800]);

% Get method names

methodNames = {};

for i = 1:length(obj.rootSolvers)

if i <= length(obj.results.rootFinding)

methodNames{end+1} = obj.rootSolvers{i}.name;

end

end

subplot(2,3,1);

roots = [obj.results.rootFinding.root];

bar(roots);

set(gca, 'XTickLabel', methodNames);

ylabel('Root Value');

title('Root Comparison');

grid on;

subplot(2,3,2);

iterations = [obj.results.rootFinding.iterations];

bar(iterations);

set(gca, 'XTickLabel', methodNames);

ylabel('Iterations');

title('Iteration Count');

grid on;

subplot(2,3,3);

times = [obj.results.rootFinding.time];

bar(times);

set(gca, 'XTickLabel', methodNames);

ylabel('Computation Time (s)');

title('Computation Time');

grid on;

subplot(2,3,4);

% Plot function and roots

r\_plot = linspace(5, 15, 1000);

f = @(r) (8/3)\*pi\*r - 2000./(r.^2);

plot(r\_plot, f(r\_plot), 'b-', 'LineWidth', 2);

hold on;

colors = ['r', 'g', 'm'];

for i = 1:length(obj.results.rootFinding)

root = obj.results.rootFinding(i).root;

plot(root, f(root), [colors(i) 'o'], 'MarkerSize', 10, 'LineWidth', 3, ...

'DisplayName', [methodNames{i} ' (r=' sprintf('%.6f', root) ')']);

end

xlabel('r');

ylabel('f(r)');

title('Function with Roots');

legend('Location', 'best');

grid on;

subplot(2,3,5);

% Plot convergence history

hold on;

colors = ['r', 'g', 'b'];

for i = 1:length(obj.rootSolvers)

if i <= length(obj.results.rootFinding)

solver = obj.rootSolvers{i};

if ~isempty(solver.history)

plot(1:length(solver.history), solver.history, [colors(i) '-o'], ...

'LineWidth', 1.5, 'MarkerSize', 4, 'DisplayName', solver.name);

end

end

end

xlabel('Iteration');

ylabel('Solution Value');

title('Convergence History');

legend('Location', 'best');

grid on;

subplot(2,3,6);

% Plot error convergence

hold on;

for i = 1:length(obj.rootSolvers)

if i <= length(obj.results.rootFinding)

solver = obj.rootSolvers{i};

if ~isempty(solver.errorHistory)

semilogy(1:length(solver.errorHistory), abs(solver.errorHistory), [colors(i) '-s'], ...

'LineWidth', 1.5, 'MarkerSize', 4, 'DisplayName', solver.name);

end

end

end

xlabel('Iteration');

ylabel('Absolute Error');

title('Error Convergence (log scale)');

legend('Location', 'best');

grid on;

end

end

function plotODEComparison(obj)

% Compare ODE solving methods

if isfield(obj.results, 'odeSolving') && ~isempty(obj.results.odeSolving)

figure('Position', [100, 100, 1200, 800]);

% Get method names

methodNames = {};

for i = 1:length(obj.odeSolvers)

if i <= length(obj.results.odeSolving)

methodNames{end+1} = obj.odeSolvers{i}.name;

end

end

subplot(2,3,1);

max\_errors = [obj.results.odeSolving.max\_error];

bar(max\_errors);

set(gca, 'XTickLabel', methodNames, 'YScale', 'log');

ylabel('Maximum Error');

title('Maximum Errors (log scale)');

grid on;

subplot(2,3,2);

times = [obj.results.odeSolving.time\_elapsed];

bar(times);

set(gca, 'XTickLabel', methodNames);

ylabel('Computation Time (s)');

title('Computation Time');

grid on;

subplot(2,3,3);

% Plot all solutions together

colors = ['r', 'g', 'b', 'm'];

hold on;

for i = 1:length(obj.odeSolvers)

if i <= length(obj.results.odeSolving)

solver = obj.odeSolvers{i};

plot(solver.timeVector, solver.solution, [colors(i) '-'], ...

'LineWidth', 2, 'DisplayName', solver.name);

end

end

% Add analytical solution

t\_fine = linspace(0, 10, 1000);

analytical = @(t) 100 \* exp(-0.5\*t);

plot(t\_fine, analytical(t\_fine), 'k--', 'LineWidth', 1, 'DisplayName', 'Analytical');

xlabel('Time');

ylabel('y(t)');

title('ODE Solutions Comparison');

legend('Location', 'best');

grid on;

subplot(2,3,4);

% Plot errors over time

hold on;

for i = 1:length(obj.odeSolvers)

if i <= length(obj.results.odeSolving)

solver = obj.odeSolvers{i};

if ~isempty(solver.errorHistory)

plot(solver.timeVector, solver.errorHistory, [colors(i) '-'], ...

'LineWidth', 1.5, 'DisplayName', [solver.name ' Error']);

end

end

end

xlabel('Time');

ylabel('Absolute Error');

title('Errors Over Time');

legend('Location', 'best');

grid on;

subplot(2,3,5);

% Plot error distributions

hold on;

for i = 1:length(obj.odeSolvers)

if i <= length(obj.results.odeSolving)

solver = obj.odeSolvers{i};

if ~isempty(solver.errorHistory)

histogram(solver.errorHistory, 20, 'FaceColor', colors(i), ...

'FaceAlpha', 0.6, 'DisplayName', solver.name);

end

end

end

xlabel('Error Magnitude');

ylabel('Frequency');

title('Error Distribution');

legend('Location', 'best');

grid on;

subplot(2,3,6);

% Plot relative errors

hold on;

for i = 1:length(obj.odeSolvers)

if i <= length(obj.results.odeSolving)

solver = obj.odeSolvers{i};

if ~isempty(solver.analyticalSolution) && ~isempty(solver.solution)

analytical\_values = solver.analyticalSolution(solver.timeVector);

relative\_errors = solver.errorHistory ./ abs(analytical\_values);

plot(solver.timeVector, relative\_errors, [colors(i) '-'], ...

'LineWidth', 1.5, 'DisplayName', [solver.name ' Relative Error']);

end

end

end

xlabel('Time');

ylabel('Relative Error');

title('Relative Errors Over Time');

legend('Location', 'best');

grid on;

end

end

function demonstratePolymorphism(obj)

% Demonstrate polymorphism with the class hierarchy

fprintf('\n=== POLYMORPHISM DEMONSTRATION ===\n');

allSolvers = [obj.rootSolvers, obj.odeSolvers];

for i = 1:length(allSolvers)

solver = allSolvers{i};

fprintf('\nSolver %d:\n', i);

fprintf(' Type: %s\n', class(solver));

fprintf(' Name: %s\n', solver.name);

fprintf(' Solver Type: %s\n', solver.SOLVER\_TYPE);

% Polymorphic method calls

solver.displayResults();

end

end

end

end

CHAPTER 3:CONCLUSION.

Developing and testing a high level back-end implementation of the numerical methods applications, required the concepts of classes,encapsulation,polymorphism and abstraction to solve the computational problems.