BUSITEMA UNIVERSITY
Pursuing Excellence

FACULTY OF ENGINNEERING AND TECHNOLOGY

REPORT ABOUT COMPUTER PROGRAMMING ASSIGNMENT ON MODULE 5

GROUP NAME: GROUP 9

COURSE UNIT: COMPUTER PROGRAMMING

GROUP LINK. https://github.com/Groupematlab/group-E.git

This assignment report is submitted to the lecturer of computer programming Mr. BENEDICTO MASERUKA by group 9 for the award of coursework marks.

Submitted on…..…/……..…/…….……

APPROVAL

This is to confirm that this report has been written and presented by GROUP 9 giving the details for the assignment.

LECTURER'SNAME:……
……..

SIGNITURE:……
……

DATE:……
……

DECLARATION

We, members of group 9, sincerely declare this report to all members who may need to use its content for approval or study. This is out of our own knowledge and research and is the content of our own writing and research.

Date of declaration……

Group representative signature……

GROUP MEMBER'S DETAILS

NAME	REG. NUMBER	COURSE	SIGNATURE
1. NAMARA ROMUS	BU/UG/2024/2596	WAR	
2. BIIRA EDITOR	BU/UG/2024/5058	WAR	
3. ABONGO CHRISTOPHER	BU/UP/2024/1002	WAR	
4. KANYANGE SHEEBAH M	BU/UG/2024/2630	MEB	
5. ATIM SARAH	BU/UP/2024/5473	WAR	
6. NUWAMANYA MUGISHA EVANS	BU/UP/2024/0877	APE	
7. NAMWANJE SAMALE	BU/UP/2024/3821	PTI	
8. MUHAIRWE VICTOR	BU/UP/2024/5254	AMI	
9. OBUA LOUIS	BU/UP/2024/0839	AMI	
10. KAWAASE JOHN KIZZA	BU/UP/2024/4661	AMI	

ACKNOWLEDGEMENT

We first of all thank GOD for the gift of understanding and unity among our group members from the start of the assignment to the point of accomplishment.

In addition, great thanks go to the lecturer for the teaching method he used to make us understand more techniques in MATLAB through giving us this assignment.

Lastly, we also appreciate each member for the support in researching and documenting the results of this assignment.

ABSTRACT

This report is about the assignment which was given to all groups in computer programming including our group 9 on October 16, 2025. We started with further research on addition to the knowledge which was given to us by our lecturer. We managed to succeed with the assignment by generating right codes that are matching to the assignment given.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In this assignment we were required to use the previous recreated codes from the previous assignments within the MATLAB environment to develop and tests high end or high level back end implementation of Numerical methods, application for solving applicable problems.

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

QUESTION

Whilst implementing the concepts of class encapsulation, inheritance, polymorphism and abstraction. Develop and test high end of high level back end implementation of a Numerical method 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 problems and the other for Integral problems.

SOLUTION

Executive Summary

The provided code implements an object oriented programming “Newton-Raphson method” for finding roots of mathematical functions. This report provides a comprehensive analysis of the code's functionality, performance, strengths, weaknesses, and potential improvements.

Newton Raphson Method.

Code Overview

Primary Components

1. Class definition “NewtonRaphson”: it encapsulates the root-finding algorithm with properties for the function and its derivative.

2. “constructor: Initializes the properties when an object of the class is created.

3. Find Root Method; implements the Newton-Raphson computation, iterating to find the root within the specified tolerance.

Detailed Class set up

```
classdef NewtonRaphson

 properties

       FunctionHandle
       DerivativeHande
       InitialGuess
       Tolerence
       MaxIterations
   end
 methods
        function obj = NewtonRaphson(func,deriv,initialGuess)
            obj.FunctionHandle = func;
            obj.DerivativeHande = deriv;
            obj.InitialGuess = initialGuess;
            obj.Tolerence = tolerance;
            obj.MaxIterations = maxIter;
    end
        function root = findRoot(obj)
            xCurrent = obj.InitialGuess;
            for i = 1:obj.MaxIterations
                fCurrent = obj.FunctionHandle(xCurrent);
                dfCurrent = obj.DerivativeHande(xCurrent);
                xNext = xCurrent - fCurrent/dfCurrent;
               if abs(xNext - xCurrent) < obj.Tolerence
                   root = xNext;
                   return
               end
               xCurrent = xNext;
            end
            error("Maximum iterations reached without convergence")
        end
    end
end
```

Using the above class

We created an instance (newton script) containing the function whose root was found using the above class.

```
f = @ (x) 2*x^.3 + 5*x - 8;
df = @(x) 6*x.^2 + 5;
InitialGuess = 1;
tolerance = 5e-3;
maxIter = 50;
nrSolver = NewtonRaphson(f,df,InitialGuess,tolerance,maxIter);
try
    root = nrSolver.findRoot();
    disp(['Root found:',num2str(root)]);
catch ME
    disp(['Error:',ME.message]);
end
Error Handling and Validation
The NewtonRaphson class includes error handling to notify when the maximum iteration
is reached without finding a root.
```

Secant Method Class Implementation

```
This class encapsulated the Secant Method functionality with properties for the
function and initial guesses.
Constructor: initialized the properties when an object of the class was created.
FindRoot Method: implemented the Secant Method iterating to find the root within the
specified tolerance as implemented by the class below.
classdef SecantMethod

    properties
        FunctionHandle
        InitialGuess1
        InitialGuess2
        Tolerance
        MaxIterations
    end

    methods
        function obj = SecantMethod(func,guess1,guess2,tolerance,maxIter)
            obj.FunctionHandle = func;
            obj.InitialGuess1 = guess1;
            obj.InitialGuess2 = guess2;
            obj.Tolerance = tolerance;
            obj.MaxIterations = maxIter;
        end

        function root = findRoot(obj)
            x2 = obj.InitialGuess1;
            x3 = obj.InitialGuess2;
            for i = 1:obj.MaxIterations
                f2 = obj.FunctionHandle(x2);
                f3 = obj.FunctionHandle(x3);

            xNext = x2 - f2*((x3-x2)/ (f3-f2));
```

```
            if abs(xNext-x2) < obj.Tolerance
                root = xNext;
                return;
            end
            x2 = x3;
            x3 = xNext;
            error('Maximum iterations reached without convergence.');
            end
        end
    end
end
```

The function script below (secantSolver) inherited the properties of the above class, hence coming up with root of the function.

```
f = @(x) sin(cos(exp(x)));
InitialGuess1 = 0;
InitialGuess2= 1;
tolerance = 1e-4;
maxIter = 10;
secantSolver = SecantMethod(f,InitialGuess1,InitialGuess2,tolerance,maxIter);
try
    root = secantSolver.findRoot();
    disp(['Root found:',num2str(root)]);
catch ME
    disp(['Error:',ME.message]);
end
```

Trapezoidal Method OOP Implementation

1. Class Definition(TrapezoidalMethod): this class encapsulated the functionality for numerical integration using the Trapezoidal rule.
2. Constructor: initialized the properties of the class when an object was instantiated. It included the function to integrate, integration limit ad the number of intervals.
3.Integrate method
Computed the integral using the trapezoidal rule. It calculated the step size,evaluated the function at defined intervals,and applied the trapezoidal formula.

```
classdef TrapezoidalMethod

    properties
        FunctionHandle
        LowerBound
        UpperBound
        NumIntervals
    end

    methods
        function obj = TrapzoidalMethod(func,a,b,n)
            obj.FunctionHandle= func;
            obj.LowerBound = a;
            obj.UpperBound = b;
            obj.NumIntervals = n;
        end
```

```
        function result = integrate(obj)
             h = (obj.UpperBound- obj.LowerBound)/obj.NumIntervals;
             x = obj.LowerBound:h:obj.UpperBound;
             fx = arrayfun(obj.FunctionHandle,x);
             result = (h/2)*(fx(1)+2*sum(fx(2:end-1)+fx(end)));
        end
    end
end
```

The function script below (trapezoidal solver) inherited the properties of the above class, hence coming up with the integral of the function.

```
f = @(x) sin(x);
lowerBound = 0;
upperBound = pi;
trapezoidalSolver = TrapezoidalMethod(f,lowerBound, upperBound, numIntervals);
result = trapezoidalsolver.integrate();
disp(["result of integraton of sin(x) from 0 to pi:",num2str(result)]);
```

Runge-Kutta Method Class Implementation

This class encapsulated the functionality for solving ODEs using the fourth-order Runge-Kutta method.
Constructor: initialized the properties when an object was created.
Solve method: Implemented the RungeKutta algorithm,iterating to compute the solution for the specified number of steps.
It calculated the intermediate slopes k1,k2,k3,k4 and updated the y-values and t-values.

```
classdef RungeKuttaMethod

    properties
        ODEFunction
        InitialTime
        InitialValues
        StepSize
        Steps
    end

    methods
        function obj = RungeKuttaMethod(odeFunc,y0,t0,h,steps)
            obj.ODEFunction = odeFunc;
            obj.InitialValues = y0;
            obj.InitialTime = t0;
            obj.StepSize = h;
            obj.Steps = steps;
        end

        function [tValues,yValues] = solve(obj)
            tValues = zeros(1,obj.Steps + 1);
            yValues = zeros(1,obj.Steps +1);
            tValues(1) = obj.InitialTime;
            yValues(1) = obj.InitialValues;
            for i = 1:obj.Steps
                t = tValues(1);
                y = yValues(1);
```

```
                k1 = obj.StepSize*obj.ODEFunction(t,y);
                k2 = obj.StepSize*obj.ODEFunction(t + obj.StepSize/2,y +k1/2);
                k3 = obj.StepSize*obj.ODEFunction(t + obj.StepSize/2,y + k2/2);
                k4 = obj.StepSize*obj.ODEFunction(t + obj.StepSize,y + k3);
                yValues(i + 1) = y + (1/6)*(k1 + 2*k2 + 2*k3 + k4);
                tValues(i + 1) = t + obj.StepSize;
            end
        end
    end
end
```

The function script below (RungeKutta solver) inherited the properties of the above class, thus solving the ODE below.

```
f = @(t,y) y^2 + sin(3*t);
y0= 1;
t0 = 0;
h = 0.2;
steps = 10;
rkSolver = RungeKuttaMethod(f,y0,t0,h,steps);
[tValues,yValues] = rkSolver.solve();
disp('Time values:');
disp(tValues);
disp('Computed values:');
disp(yValues);
```

Euler Method

1. class Definition: This class encapsulated the functionality of solving ODEs using the Euler Method.
2. Costructor: initialized the properties with the provided ODE function, initial values, step size, and number of steps when an object was instantiated.
3. solve Method: Implements the Euler Method iteratively to compute the solution for the specified number of steps.

It calculated the next y-value using the Euler formula and increments the t-value accordingly.

```
classdef EulerMethod

    properties
        ODEFunction
        InitialValue
        InitialTime
        StepSize
        Steps
    end

    methods
        function obj = EulerMethod(odeFunc,y0,t0,h,steps)
            if h <= 0 || steps<= 0
                error('Stepsize and number of steps must be positive.');
            end
            obj.ODEFunction = odeFunc;
            obj.InitialValue = y0;
            obj.InitialTime = t0;
            obj.StepSize = h;
```

```
            obj.Steps = steps;

        end

        function [tValues,yValues] = solve(obj)
            tValues = zeros(1,obj.Steps + 1);
            yValues = zeros(1,obj.Steps + 1);
            tValues(1) = obj.InitialTime;
            yValues(1) = obj.InitialValue;
            for i = 1:obj.Steps
                yValues(i +1) = yValues(i) +
obj.StepSize*obj.ODEFunction(tValues(i),yValues(i));
                tValues(i + 1) = tValues(i) + obj.StepSize;
            end
                figure;
                plot(tValues,yValues,'b.-','DisplayName','Euler Approximation');
                hold on;
                if nargin > 1
                    plot(tValues,yExact(tValues),'r--','DisplayName','Exact
Solution');
                end
                xlabel('Time');
                ylabel('Function Value');
                title('Euler Method Approximation vs Exact Solution');
                legend('show');
                grid on;
                hold off;

        end
    end
end
```

The function script below (Euler solver) inherited the properties of the above class, thus solving the ODE below.

CONCULSION

These implementations demonstrate how to leverage object oriented programming in MATLAB for solving Numerical approximation problems. The above classes can further be adapted and extended for various types of ODEs, integrations and differential advanced features as needed.