MATH301
Numerical Analysis
Final Project

Solving Numerical Methods Using MATLAB

Retal Nassar
S20106214
Renassar@effat.edu.sa

Mehreen Junaid
S20106424
mejunaid@effat.edu.sa

Wegdan Alshateri
S19105881
wealshateri@effat.edu.sa

Computer Science Department
Effat University
Jeddah, Saudi Arabia

MATH301
Numerical Analysis
Final Project

Abstract:

In this report we will go over the theoretical backgrounds of the numerical methods that we experienced during the semester. we provided the MATLAB code for the methods and provided some background information, examples and plots.

Keywords:

Bisection method, Newton’s method, Secant method, Euler’s method, Taylor’s method, Runge-Kutta Method

MATH301
Numerical Analysis
Final Project

Theoretical Background For The Methods:

1. Bisection Method:

Bisection or the binary search method is considered one of the most basic problems of numerical approximation, the root-finding problem. To determine the solution to a polynomial problem we apply the bisection technique. In this method we split the range where the solution of the calculation is located, it will operate gradually by closing the gap between the negative and positive sectors until an accurate solution is determined

2. Newton's Method:

Newton's method or the Newton-Raphson method is one of the most powerful and well-known numerical method for solving a root finding problem.It can be easily generalized to the problem of finding solutions of a system of non-linear equations, it is based on the simple idea of linear approximation.

3. Secant Method:

Secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. Newton's method is an extremely powerful technique but it has a major weakness the need to know the value of the derivative of f at each approximation. Frequently, f'' (x) is far more difficult and needs more arithmetic operations to calculate than f(x). To circumvent the problem of the derivative evaluation in Newton's method, we introduce a slight variation.

4. Euler Method:

Euler method is the most basic explicit method for numerical integration of ordinary differential equations. Euler's method uses the simple formula, to construct the tangent at the point x and obtain the value of y(x+h) , whose slope is, In Euler's method, you can approximate the curve of the solution by the tangent in each interval (that is, by a sequence of short line segments), at steps of h.

5. Taylor's Method:

Taylor series method involves use of higher order derivatives which may be difficult in case of complicated algebraic equations. The Taylor series can be used to calculate the value of an entire function at every point, if the value of the function, and of all of its derivatives, are known at a single point. It is the polynomial or a function of an infinite sum of terms. Each successive term will have a larger exponent or higher degree than the preceding term. f (a) + f ′ (a) 1 !

6. The Runge-Kutta Method

Runge–Kutta method is an effective and widely used method for solving the initial-value problems of differential equations. Runge–Kutta method can be used to construct high order accurate numerical method by functions' self without needing the high order derivatives of functions. They are easy to implement and are very stable unlike other methods.

MATH301
Numerical Analysis
Final Project

Bisection Method:

In numerical analysis, the halving method is a root-finding algorithm in which a period is halved iteratively and a sub-interval on which the root is located is selected in order to improve processing.

Input:

```
f = @(x) x^2 - 5*x + 3; % the function
n=10;      % numer of iteration
a=0;        %interval start value
b=3;        %interval end value
h=(b-a)/n;   % step size
e = 0.0001;

 x=a:h:b;
y=zeros(size(x));


y(1)=0;
```

algebraic Eq. $x^2 - 5x + 3 = 0$

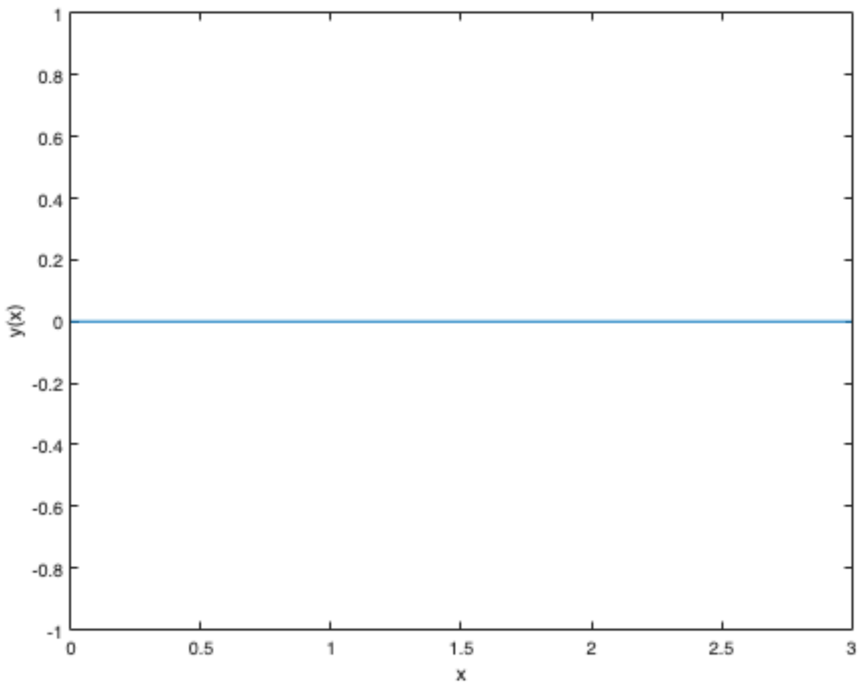
```
if f(a)*f(b)<0
    for i=1:n
        c = (a+b)/2;
        fprintf('P%d = %.4f\n',i,c)
        if abs(c-b) < e || abs(c-a) < e
            break
        end
        if f(a)*f(c)<0
            b = c;
```

```
        elseif f(b)*f(c)<0
            a = c;
        end
    end
else
    disp('No root between given brackets')
end
```

```
P1 = 1.5000
P2 = 0.7500
P3 = 0.3750
P4 = 0.5625
P5 = 0.6562
P6 = 0.7031
P7 = 0.6797
P8 = 0.6914
P9 = 0.6973
P10 = 0.6943
```

Visualization:

```
figure(1)
plot(x,y)
xlabel('x')
ylabel('y(x)')
```



Newton_method

In numerical analysis, the Newton method or the Newton-Raphson method is an efficient algorithm for finding the roots of a real function. So it is an example of root finding algorithms. It can be used to find upper and lower limits of such functions, by finding the roots of the first derivative of the function

Input

```
f = @(x) 2^x - 5*x +2;  % the function
df = @(x) log(2)*(2^x)-5;   % derivative of function
```

```
x0 = 0;                 % first interval
n = 2;                  % no of iterations
e=10^-4;                 % tolerance
```

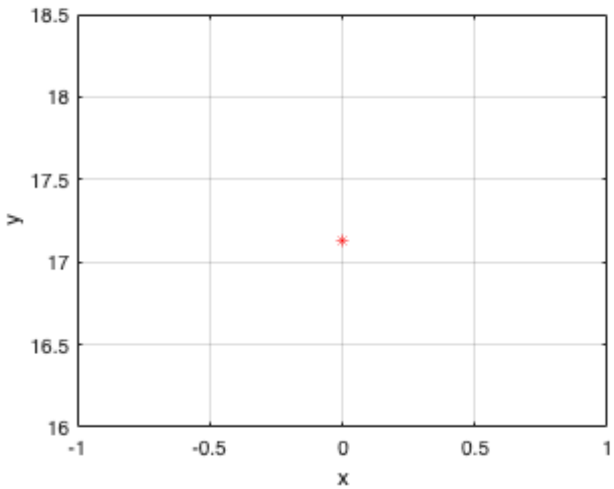
algebraic Eq. $2^x - 5x + 2$

```
if df(x0)~=0
    for i=1:n
        x1 = x0 -f(x0)/df(x0);
        fprintf('x%d = %.20f\n',i,x1)
        if abs (x1-x0)<e
            break
        end
        if df(x1) == 0
            disp('Newton failed')
        end
        x0 = x1;
    end
else
    disp('Newton failed');
end
```

```
x1 = 0.69656431871986701498
x2 = 0.73211534565952418596
```

Visualization

```
plot(a(i),b(i),'r*'); grid on;
xlabel('x'); ylabel('y');
hold on;
```



Secant_method

In numerical analysis, the secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. The secant method can be thought of as a finite-difference approximation of Newton's method.

Input

```
f = @(x) x^3 - 2*x -5; % the function
x0 = 1;                 % first interval
x1 = 3;                 % second interval
e = 10^-4;              % tolerance
n = 10;                 % no of iterations
```

```
h = (b-a)/n;            % Step size
```

algebraic Eq. $x^3 - 2x - 5$

```
for i=1:n
    x2 = (x0*f(x1)-x1*f(x0))/(f(x1)-f(x0))
    fprintf('x%d = %.10f\n',i,x2)
    if abs(x2-x1)<e
        break
    end
    x0 = x1;
    x1 = x2;
end
```

```
x2 = 1.5455
x1 = 1.5454545455
x2 = 1.8592
x2 = 1.8591632292
x2 = 2.2004
x3 = 2.2003500782
x2 = 2.0798
x4 = 2.0797991805
x2 = 2.0937
x5 = 2.0937042425
x2 = 2.0946
x6 = 2.0945585626
x2 = 2.0946
x7 = 2.0945514782
```

MATH301
Numerical Analysis
Final Project

Euler Method

Code:

```
 % Euler's Method
% Initial conditions and setup
h = (enter your step size here);  % step size
x = (enter the starting value of x here):h:(enter the ending value of x here);  % the range of x
y = zeros(size(x));  % allocate the result y
y(1) = (enter the starting value of y here);  % the initial y value
n = numel(y);  % the number of y values
% The loop to solve the DE
for i=1:n-1
    f = the expression for y' in your DE
    y(i+1) = y(i) + h * f;
end
```

Example:

```
% Euler's Method
% Initial conditions and setup
h = 0.2;  % step size
x = (0):h:(4);  % the range of x
y = zeros(size(x));  % allocate the result y
y(1) = (2);  % the initial y value
n = numel(y);  % the number of y values
% The loop to solve the DE
for i=1:n-1
    f = -sin(x(n))/(2*y(i))
    y(i+1) = y(i) + h * f;
end
```

```
Results:
f = 0.1892
f =0.1857
f =0.1824
f =0.1792
f =0.1762
```

```
f =0.1734
f =0.1707
f =0.1681
f =0.1656
f =0.1632
f =0.1610
f =0.1588
f =0.1567
f =0.1547
f =0.1528
f =0.1509
f =0.1491
f =0.1474
f =0.1457
f =0.1441
```

Taylor Method

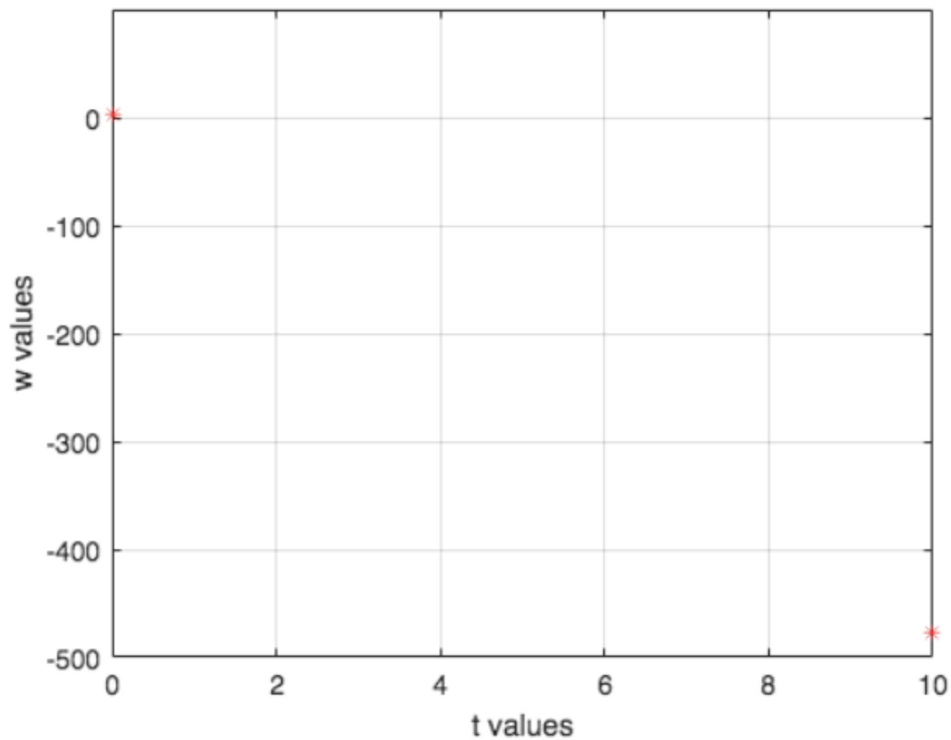
Code:

```
f = @(t,y) (-2*y + 3*exp(t));
fprime=@(t,y) (-4*y + 3*exp(t));
a = 0;
b = 10;
n = 1;
alpha = 3;

h = (b-a)/n;
t=[a zeros(1,n)];
w=[alpha zeros(1,n)];


for i = 1:n+1
```

```
t(i+1)=t(i)+h;
wprime=f(t(i),w(i))+(h/2)*fprime(t(i),w(i));
w(i+1)=w(i)+h*wprime;
fprintf('%5.4f %11.8f\n', t(i), w(i));
plot(t(i),w(i),'r*'); grid on;
xlabel('t values'); ylabel('w values');
hold on;
end
```

Result:

```
0.0000  3.00000000
10.0000 -477.00000000
```



MATH301
Numerical Analysis
Final Project

Runge-Kutta Method

Code:

```
h=0.5; % step size
x = 0:h:10; % Calculates upto y(3)
Y = zeros(1,length(x));
%y(1) = [-0.5;0.3;0.2];
y(1) = 3; % redo with other choices here.
% initial condition
F_xy = @(t,y) (-2*y)+(3*exp(t)); % change the function as you desire
for i=1:(length(x)-1) % calculation loop
k_1 = F_xy(x(i),y(i));
k_2 = F_xy(x(i)+0.5*h,y(i)+0.5*h*k_1);
k_3 = F_xy((x(i)+0.5*h),(y(i)+0.5*h*k_2));
k_4 = F_xy((x(i)+h),(y(i)+k_3*h));
y(i+1) = y(i) + (1/6)*(k_1+2*k_2+2*k_3+k_4)*h; % main equation
end
% validate using a decent ODE integrator
tspan = [0,10]; y0 = -0.5;
[tx, yx] = ode45(F_xy, tspan, y0)


plot(x,y,'o-', tx, yx, '--')
```

Result:

MATH301
Numerical Analysis
Final Project