Numerical Methods

Assignment 3

In this assignment, We will compare following properties of each method

1. Accuracy
2. Number of Iterations
3. Average CPU Cost per iteration

Contents

Student

- **Name:** Nauman Mustafa
- **CMS ID:** 111233
- **Reg No:** 32587

Question 1

In this question, we will compare three different methods for finding roots of equation:

1. False-Position Method
2. Secant Method
3. Newton's Method

```
% Lets Define Function
f = @(x) 230*x^4 + 18*x^3 + 9*x^2 - 221*x - 9; % Similar to Inline but symbolically
% Lets Define Derivative
df = @(x)  920*x^3 + 54*x^2 + 18*x - 221;
% Output Format
sFormat = '\tRoot: %g in %d Iterations\n\tWith Error %g and Time %gms per Iteration\n';
```

False Position Method

Following is Algorithm of False Position Method

```
function [x, n, e] = FalsePosition(f, a, b, err, nMax)
    n = 0;
    e = Inf;
    fa = f(a);
    fb = f(b);
    c0 = 0;
    while n<nMax && e>err
        c1 = b - fb*(a-b)/(fa-fb);
        e = abs(c1-c0);
        fc = f(c1);
        test = fc*fa;
        if test<0
```

```
            b = c1;
            fb = fc;
        elseif test>0
            a = c1;
            fa = fc;
        else
            e = 0;
        end
        c0 = c1;
        n = n + 1;
    end
    x = c0;
end
```

```
disp('=====================');
disp('False Position Method');
disp('=====================');
% Solution 1 of Function
disp('Solution of Function in Interval [-1, 0]:')
t = cputime;
[x, n, e] = FalsePosition(f, -1, 0, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
% Solution 2 of Function
disp('Solution of Function in Interval [0, 1]:')
t = cputime;
[x, n, e] = FalsePosition(f, 0, 1, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
```

```
=====================
False Position Method
=====================
Solution of Function in Interval [-1, 0]:
        Root: -0.0406585 in 16 Iterations
        With Error 7.61975e-07 and Time 0ms per Iteration
Solution of Function in Interval [0, 1]:
        Root: 0.962398 in 8 Iterations
        With Error 4.45344e-07 and Time 0ms per Iteration
```

Secant Method

Following is Algorithm for Secant Method

```
function [x, n, e] = SecantMethod(f, a, b, err, nMax)
    n = 0;
    e = Inf;
    fa = f(a);
    fb = f(b);
    c0 = 0;
    while n<nMax && e>err
        c1 = b - fb*(a-b)/(fa-fb);
        e = abs(c1-c0);
        fc = f(c1);

        a = b;
        b = c1;
        fa = fb;
        fb = fc;

        c0 = c1;
        n = n + 1;
```

```
    end
    x = c0;
end
```

```
disp('=====================');
disp('====Secant Method====');
disp('=====================');
% Solution 1 of Function
disp('Solution of Function For Interval [-1, 0]:')
t = cputime;
[x, n, e] = SecantMethod(f, -1, 0, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
% Solution 2 of Function
disp('Solution of Function For Interval [0, 1]:')
t = cputime;
[x, n, e] = SecantMethod(f, 0, 1, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
% Since for interval [0,1] we got same solution so we change our interval
disp('Solution of Function For Interval [1, 2]:')
t = cputime;
[x, n, e] = SecantMethod(f, 1, 2, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
```

```
=====================
====Secant Method====
=====================
Solution of Function For Interval [-1, 0]:
        Root: -0.0406593 in 4 Iterations
        With Error 2.5738e-08 and Time 0ms per Iteration
Solution of Function For Interval [0, 1]:
        Root: -0.0406593 in 11 Iterations
        With Error 6.00724e-08 and Time 1.42045ms per Iteration
Solution of Function For Interval [1, 2]:
        Root: 0.962398 in 6 Iterations
        With Error 1.54228e-07 and Time 0ms per Iteration
```

Newton's Method

Following is Algorithm for Newton's Method

```
function [x, n, e] = NewtonMethod(f, df, xi, err, nMax)
    n = 0;
    e = Inf;
    x = 0;

    while n<nMax && e>err
        x = xi - f(xi)/df(xi);

        e = abs(x-xi);

        xi = x;
        n = n + 1;
    end
end
```

```
disp('=====================');
disp('====Newton Method====');
```

```
disp('=====================');
% Solution 1 of Function
disp('Solution of Function For Initial Value -0.5:')
t = cputime;
[x, n, e] = NewtonMethod(f, df, -0.5, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
% Solution 2 of Function
disp('Solution of Function For Initial Value 0.5:')
t = cputime;
[x, n, e] = NewtonMethod(f, df, 0.5, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
% Since for initial guess 0.5 we got same solution so we change our guess
disp('Solution of Function For Initial guess:')
t = cputime;
[x, n, e] = NewtonMethod(f, df, 1.5, 10^-6, Inf);
t = cputime - t;
fprintf(sFormat,x,n,e,t*1000/n);
```

```
=====================
====Newton Method====
=====================
Solution of Function For Initial Value -0.5:
        Root: -0.0406593 in 4 Iterations
        With Error 5.51816e-08 and Time 0ms per Iteration
Solution of Function For Initial Value 0.5:
        Root: -0.0406593 in 6 Iterations
        With Error 2.95761e-11 and Time 0ms per Iteration
Solution of Function For Initial guess:
        Root: 0.962398 in 6 Iterations
        With Error 3.78143e-08 and Time 0ms per Iteration
```

Thus we can compare results as following

- CPU time per iteration is too small to be measured
- *False Position Method* provides reasonable result within interval as compared to other methods which don't remain in interval
- Although *Secant Method* for first solution converges quicker then *Newton's Method* but in general **Newton's Method** is more robust and has high convergance rate as compared to *Secant Method*

Question 2

In this question we will compare two methods specifically simpson's methods for finding integral of function on given interval:

1. Simpson's 1/3 Rule
2. Simpson's 3/8 Rule

```
% Let's Deffine Function of Integration
f = @(x)sin(x).^2-2*x.*sin(x)+1;
```

Simpson's 1/3 Rule

Following is Optimized Algorithm for Simpson's 1/3 Rule:

```
function int = Simpson13(f, nSubInt, a, b)
    n = nSubInt * 2;
    h = (b-a)/n;
    x = a:h:b;
    fx = f(x);
    int = fx(1)+fx(n+1);
```

```
    int = int+4*sum(fx(2:2:(n)))+2*sum(fx(3:2:(n-1)));
    int = int * (b-a)/(3*n);
end
```

```
% Since n is 12 but in Simpson's 1/3 Rule we only have n/2 sub intervals so
int1 = Simpson13(f,6,0.75,1.75);
fprintf('Simpson 1/3 Result: %g\n',int1);
```

```
Simpson 1/3 Result: -0.489018
```

Simpson's 3/8 Rule

```
function int = Simpson38(f, nSubInt, a, b)
    n = nSubInt * 3;
    h = (b-a)/n;
    x = a:h:b;
    fx = f(x);
    int = fx(1)+fx(n+1);
    int = int + 3*sum(fx(2:3:(n-1)))+3*sum(fx(3:3:n))+2*sum(fx(4:3:(n-2)));
    int = int * (3*h/8);
end
```

```
% Since n is 12 but in Simpson's 3/8 Rule we only have n/3 sub intervals so
int2 = Simpson38(f,6,0.75,1.75);
fprintf('Simpson 3/8 Result: %g\n',int2);
```

```
Simpson 3/8 Result: -0.489019
```

Following Comparision shows there is a little difference between result of *Simpson's 1/3 Rule* and *Simpson 3/8 Rule*.

```
disp(int2-int1)
```

```
  -8.1960e-07
```

- Error of both rules is of order of 4
- Only difference in error is factor 8/27
- Thus **Simpson's 1/3 Rule** error is 8/27 times the error of 3/8 Rule
- From Code Point of view, Simpson's 3/8 Rule is better for serial code execution whereas Simpson's 3/8 is useful when parallel execution is required to accelerate computations

Question 3

In this question, we will solve three equations each function of three variable to achieve accuracy of 10^-6 using newton's method for solution of multiple equations

```
% F is cell array of anonymous functions which are to be zeroed
% Number of elements in F is same as number of variables in each f
% xi is initial guess column vector with length equal to F's len.
function [x, n, e] = NewtonMethodMulti(F, xi, err, nMax)
    n = 0;
    e = Inf;
    x = 0;
```

```
    l = length(F);

    f = zeros(l,1);
    J = zeros(l,l);

    while n<nMax && e>err

        x0 = num2cell(xi);
        parfor i=1:l
            Fi = F{i};
            f(i)=Fi(x0{:});
            for j=1:l
                J(i,j)=Partial(Fi,x0,j);
            end
        end

        x = xi - (pinv(J)*f);

        e = max(abs(x-xi));

        xi = x;
        n = n + 1;
    end
end
function y = Partial(f, x, i)
    epsilon = 1e-10;
    z = x;
    z{i}=z{i}+epsilon;
    y = (f(z{:})-f(x{:}))/epsilon;
end
```

```
% Lets Define Equations
F = {@(x1,x2,x3)6*x1-2*cos(x2*x3)-1,
    @(x1,x2,x3)9*x2+sqrt(x1^2+sin(x3)+1.06)+0.9,
    @(x1,x2,x3)60*x3+3*exp(-x1*x2)+10*pi-3};
xi = [1;2;3]; % MUST be A COLUMN VECTOR
[x,n,e]=NewtonMethodMulti(F,xi,1e-10,Inf);
fprintf('Got Solution with \n\tx1=%g\tx2=%g\tx3=%g\n\tWithin %d Iterations with accuracy of %g\n',x(1),x(2),x(3),n,e);
```

```
Got Solution with
        x1=0.498145     x2=-0.199606    x3=-0.528826
        Within 6 Iterations with accuracy of 1.4988e-15
```

Question 4

In this Question, we are going to solve 3rd order ordinary differential equation using Runge-Kutta Method of second order.

```
% RK Method of Order 2 for 3rd order Equation & Single Iteration
% Not very generic implementation of RK method
% f := f(t,y,p,q)
function [y1, p1, q1] = RK32(t0, y0, p0, q0, h, f)
    k1 = h * p0;
    l1 = h * q0;
    m1 = h * f(t0, y0, p0, q0);

    k2 = h * (p0 + 3*l1/2);
    l2 = h * (q0 + 3*m1/2);
    m2 = h * f(t0 + 3*h/2, y0 + 3*k1/2, p0 + 3*l1/2, q0 + 3*m1/2);

    y1 = y0 + (2*k1+k2)/3;
    p1 = p0 + (2*l1+l2)/3;
    q1 = q0 + (2*m1+m2)/3;
```

```
end
```

```
% Lets ODE define function:
f = @(t,y,p,q)q/t - 3*p/t^2 + 4*y/t^3 + 5*log(t)+9;
% Lets Define Real solution
g = @(t)-t^2+t*cos(log(t))+t*sin(log(t))+t^3*log(t);

% Initial Conditions
y = 0; p = 1; q = 3;



fprintf('Time\tReal Solution\tApproximate Solution\n');
for t=1:0.1:2
    yr = g(t);
    fprintf('%g\t%f\t%f\n',t, yr, y);
    [y, p, q] = RK32(t, y, p, q, 0.1, f);
end
```

```
Time    Real Solution   Approximate Solution
1       0.000000        0.000000
1.1     0.116548        0.115000
1.2     0.272738        0.269264
1.3     0.479102        0.473361
1.4     0.746998        0.738679
1.5     1.088493        1.077306
1.6     1.516265        1.501943
1.7     2.043536        2.025825
1.8     2.684015        2.662673
1.9     3.451846        3.426641
2       4.361578        4.332285
```

Disclaimer

This Assignment contains implementation of few *Numerical Methods* some of which are highly optimized and vectorized to provide maximum performance without loss of accuracy. These methods have not copied from any internet source but made by myself to test my knowledge of matlab.

Published with MATLAB® R2016b