CS / MATH 4334 : Numerical Analysis
Homework Assignment 2

Matthew McMillian
mgm160130@utdallas.edu

September 26, 2018

MatLab Problems

Problem 1 : ffalpos.m

```
function c = ffalpos(a, b)
%FFALPOS Summary of this function goes here
%   Detailed explanation goes here
fa = (a-2)^2 + 1 - (1 - (a-2)^2 / 4)^0.5 - 1;
fb = (b-2)^2 + 1 - (1 - (b-2)^2 / 4)^0.5 - 1;
c = b - fb*((b-a)/(fb-fa));
end
```

Problem 1 : falpos.m

```
%find root of f(x) = 0
%using Bisection Method

format long e

% chosen error tolerance (TOL)
TOL = .000001;

% choose max number of iterations
MAXIT = 50;

% initial bracket
% Calcuated from simplifying (f-g)(x)
% There exists a root since f(a)f(b) < 0, and
% f(a) > 0, f(b) < 0;
a = 1;
b = 2;

%keep track of number of iterations
count = 0;

%record iterates - a col vector of MAXIT length
cits = zeros(MAXIT,1);

%evaluate func. at a and b
fa = 4*a^4 - 32*a^3 + 97*a^2 - 132*a + 64;
fb = 4*b^4 - 32*b^3 + 97*b^2 - 132*b + 64;

%stop if not appropriate interval
if sign(fa)*sign(fb) >= 0
    fprintf('Not appropiate interval\n')
    return
```

```
33
34 end
35
36 %stop loop when error less than TOL or MAXIT reached
37 while abs(b-a)/2 >= TOL && count < MAXIT
38
39     %get midpoint(root estimate)
40     c = ffalpos(a, b);
41
42     %eval. func at midpoint
43     fc = 4*c^4 - 32*c^3 + 97*c^2 - 132*c + 64;
44
45     %stop if f(c)=0
46     if fc == 0
47         break
48     end
49
50     %update count
51     count = count + 1;
52
53     %add to list of iterates
54     cits(count) = c;
55
56     %if sign change between a and c make c the new right endpt
57     if sign(fa)*sign(fc)<0
58
59         b = c;
60
61     %if sign chg betw c and b make c the new left endpt
62     else
63
64         a = c;
65
66     end
67
68 end
69
70 %update count
71     count = count + 1;
72
73 %get final midpoint(root estimate)
74     c = ffalpos(a,b);
75
76 %add to vector of iterates
77 cits(count) = c;
```

```
78
79 %display error estimate
80 error = abs(b-a)/2
81
82 %display vector of iterates
83 cits
84
85 %display number of iterates
86 count
```

>> falpos.m

```
cits =

1.118146029604788e+00
1.062353732855704e+00
1.060497138511485e+00
1.060437095603502e+00
1.060435155605902e+00
1.060435092926107e+00
1.060435090900974e+00
1.060435090835543e+00
1.060435090833429e+00
1.060435090833361e+00
1.060435090833359e+00


count =

11
```

Problem 2 : newt1.m

```
format long e

% Tolerance Level
TOL = 10^(-6);

% Initial Guess
xp = 2;

iterates = [];

% While the relative error is less than the give tolerance, we
    will
% continue to use newton's method.
while 1

    xf = xp - fnewt(xp) / fpnewt(xp);
    iterates = [iterates, xf];

    if(abs((xf-xp)/(xf)) < TOL)
        break
    end

     xp = xf;

end

% Ouprinting the resulting iterates
clc
fprintf("Iterates::")
iterates'
```

Problem 2 : newt2.m

```
format long e

clc

% Tolerance Level
TOL = 10^(-6);

% Initial Guess
xp = 2;
```

```
10
11 % Arrays to store the iterates and error values required
12 iterates = [];
13 relerr = [];
14 relerr2 = [];
15
16 % ep is 0 for the first iteration
17 ep = 1;
18
19 % While the relative error is less than the give t  olerance,
     we will
20 % continue to use newton's method.
21 while 1
22
23     % Newton's Algorithm
24     xf = xp - fnewt(xp) / fpnewt(xp);
25     iterates = [iterates, xf];
26
27      % Error and Tolerance
28     ef = abs((xf-xp)/(xf));
29     fprintf("%f\n", ef)
30     if(ef < TOL)
31         break
32     end
33
34     relerr = [relerr, ef/ep];
35     relerr2 = [relerr2, ef/ep^2];
36
37     xp = xf;
38     ep = ef;
39
40 end
41
42 % Ouprinting the resulting iterates
43 clc
44 fprintf("Iterates::\n")
45 iterates'
46
47 fprintf("Error 1::\n")
48 relerr'
49
50 fprintf("Error 2::\n")
51 relerr2'
52
53 fprintf("Since the derivative of the function is non-zero, we
```

```
    can determine that there are multiple roots. Also, we can
    determine that there is linear convergence since the error
    decreases at a somewhat linear rate.\n")

v = round(iterates(length(iterates)), 3);
mult = fnewt(v)/2*fpnewt(v);

rofc = (mult-1)/mult;

fprintf("Iterates:: %d\n", v)
fprintf("Multiplicity:: %d\n", mult)
fprintf("Rate of Convergence:: %d\n", rofc)

fprintf("We have determined that we have a rate of convergence
    of approx: 1, therefore we have linear convergence for
    this newton's method.\n")
```

Problem 2 : fnewt.m

```
function [f] = fnewt(I)

f = 6000 * (1 + (I/12))^60 - 60000*I - 6000;

end
```

Problem 2 : fpnewt.m

```
function [f] = fpnewt(I)

f = 30000 * (1 + (I/12))^59 - 60000;

end
```

>> newt2.m

```
Iterates::

ans =

1.767085782124950e+00
1.538624098431182e+00
1.315274752139667e+00
1.098538133852653e+00
8.915239987248459e-01
```

6.999922093683941e-01
5.329629816329148e-01
4.012834512079965e-01
3.132961399003459e-01
2.696854759245819e-01
2.582949829973730e-01
2.575434526842937e-01
2.575402738874424e-01
2.575402738307063e-01

Error 1::

ans =

1.318069672854064e-01
1.126529257184075e+00
1.143634792453416e+00
1.161847033275790e+00
1.176927760096816e+00
1.178367340962222e+00
1.145375182200481e+00
1.047060053412795e+00
8.558506476282967e-01
5.757979815791652e-01
2.727039364615275e-01
6.617126557971360e-02
4.229817522440887e-03

Error 2::

ans =

1.318069672854064e-01
8.546811146521263e+00
7.702053241592080e+00
6.841963471527189e+00
5.965304965421256e+00
5.074739283541284e+00
4.186008531733245e+00
3.340997594141816e+00
2.608140388188166e+00
2.050242044110255e+00
1.686383195908350e+00
1.500523816401236e+00
1.449524776306588e+00

Since the derivative of the function is non-zero, we can determine that there are multiple roots. Also, we can determine that there is linear convergence since the error decreases at a somewhat linear rate.
Iterates:: 2.580000e-01
Multiplicity:: 4.692627e+05
Rate of Convergence:: 9.999979e-01
We have determined that we have a rate of convergence of approx: 1, therefore we have linear convergence for this newton's method.

Problem 3 : hw3.m

```
format long e

clc

% Defining Function and Starting Points
x1 = 5;
x2 = 5 + 10^(-10);

fun = @(x)exp(x-1) - 1;
actual = 1;

fprintf("fzero(function1, x) results::\n")
%Computing the root and calculating the corresponding fwrd and
    bwkd error
%of the 1st root
[rootest, fval] = fzero(fun, x1);
backerr = abs(fun(rootest));
forerr = abs(actual - rootest);
fprintf("(x, rootest, fval): %d, %d, %d\n", x1, rootest, fval)
fprintf("(Backward error, Forward error): %d, %d\n", backerr,
   forerr)

% Computing the root and calculating the fwrd adn bwkd error
   of the 2nd
% root
[rootest, fval] = fzero(fun, x2);
backerr = abs(fun(rootest));
forerr = abs(actual - rootest);
fprintf("(x, rootest, fval): %d, %d, %d\n", x2, rootest, fval)
fprintf("(Backward error, Forward error): %d, %d\n\n", backerr
   , forerr)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Second Function
fun = @(x)exp(4*x-4) - 2*exp(3*x-3) + 2*exp(x-1) - 1;

fprintf("fzero(function2, x) results::\n")
% Error and root for second function root 1
[rootest, fval] = fzero(fun, x1);
backerr = abs(fun(rootest));
forerr = abs(actual - rootest);
```

```
39 fprintf("(x, rootest, fval): %d, %d, %d\n", x1, rootest, fval)
40 fprintf("(Backward error, Forward error): %d, %d\n", backerr,
       forerr)
41
42 % Errors and root for seconds function root 2
43 [rootest, fval] = fzero(fun, x2);
44 backerr = abs(fun(rootest));
45 forerr = abs(actual - rootest);
46 fprintf("(x, rootest, fval): %d, %d, %d\n", x2, rootest, fval)
47 fprintf("(Backward error, Forward error): %d, %d\n\n", backerr
       , forerr)
48
49 syms f(x)
50 f(x) = exp(x-1) - 1;
51
52 % Calculating the derivative of the function F until its not
       0, which
53 % defines its multiplicity
54 mult1 = 0;
55 while f(actual) == 0
56     f = diff(f, x);
57     mult1 = mult1 + 1;
58 end
59
60 f(x) = exp(4*x-4) - 2*exp(3*x-3) + 2*exp(x-1) - 1;
61
62 % Calculating the multiplicity of the 2nd funciton
63 mult2 = 0;
64 while f(actual) == 0
65     f = diff(f, x);
66     mult2 = mult2 + 1;
67 end
68
69 %multiplicity outprint
70 fprintf("Multiplicity f1: %d\n", mult1)
71 fprintf("Multiplicity f2: %d\n", mult2)
72 fprintf("As the multiplicity increases, stability decreases.\n
       Thus, with multiplicity 1 for the first function, we\n
      have a pretty stable algorithm. However, when we \n
      introduce higher multiplicity in function 2, we start to \n
       lose some of our stability and it begins to slightly
      affect \n our approximations. The inital guess doesn't
      affect the first \n function very much, but you can see the
       increased variablity in \n the seconds function from the
      root difference.\n")
```

>> hw3.m

fzero(function1, x) results:: (x, rootest, fval): 5, 1, 0
(Backward error, Forward error): 0, 0
(x, rootest, fval): 5.000000e+00, 1, 0
(Backward error, Forward error): 0, 0

fzero(function2, x) results::
(x, rootest, fval): 5, 9.999996e-01, 0
(Backward error, Forward error): 0, 4.312810e-07
(x, rootest, fval): 5.000000e+00, 1.000003e+00, 0
(Backward error, Forward error): 0, 3.095297e-06

Multiplicity f1: 1
Multiplicity f2: 3
As the multiplicity increases, stability decreases. Thus, with multiplicity 1 for the first function, we have a pretty stable algorithm. However, when we introduce higher multiplicity in function 2, we start to lose some of our stability and it begins to slightly affect our approximations. The inital guess doesn't affect the first function very much, but you can see the increased variablity in the seconds function from the root difference.