CS / MATH 4334 : Numerical Analysis
Homework Assignment 1

Matthew McMillian
mgm160130@utdallas.edu

September 12, 2018

MatLab Problems

Problem 1 : nest.m

```
%Program 0.1 Nested multiplication
%Evaluates polynomial from nested form using Horner's method
%Input: degree d of polynomial,
%       array of d+1 coefficients (constant term first),
%       x-coordinate x at which to evaluate, and
%       array of d base points b, if needed
%Output: value y of polynomial at x
function y=nest(d,c,x,b)
if nargin<4, b=zeros(d,1); end
y=c(d+1);
for i=d:-1:1
  y = y.*(x-b(i))+c(i);
end
```

Problem 1 : nest-script.m

```
format long e

% The degree of the greatest polynomial in the given function p(x).
degree = 25;

% Defining the coefficients of the function p(x) in accordance
% with horner-form, with the constant defined as the first
% value. Each other coefficient is placed by adding is exponent
% value to the previous index.
% i.e.  4x^5, the smallest exponent in the function p(x), has an exponent
% value of '5', so we move '5' indicies from the constant term, filling
% in the spaces in between with 0's in accordance with horner-form.
coef = [-1,0,0,0,0,4,0,0,0,0,-1,0,0,0,0,7,0,0,0,0,0,0,0,0,0,2];

% The value at which the function p(x) is evaluated at.
x = -2;

% C code that prints / formats the answer for readability.
clc
fprintf('Evaluation of p(%i): %f\n', x, nest(degree,coef,x))
```

>> nest-script.m

```
Evaluation of p(-2): -67339393.000000
```

Problem 2 : arctanseries.m

```
function [sum, k] = arctanseries(cur_val)

% Boolean variable to determine when the series will no longer
% produce a new terms due to swamping.
hasSolved = 0;

% k is our iterater value, that is, it iterates through the summation.
k = 0;

% x is our input the the pseduo 'f(x) = arctan(x)' function.
x = single(1);

% This variable stores the previous value added to the series. We use this
% to determine if the summation has succumbed to swamping.
prev_val = single(0);

% Pseudocode:
% while (the series has not begun to swamp)
%    store the previous value of the series
%    calculate the new value of the series
%    if (the series has swamped)
%        stop looping
%    end
%    go to the next term in the series
%  end
while ~hasSolved
    prev_val = cur_val;

    % You can remove the power(x,2*k+1) since this always evaluates to 1
    % for increased performance, however I have left this in here for \
    % semantic reasons.
    cur_val = cur_val + power(-1,k) * power(x,2*k+1) / (2*k+1);

    if cur_val == prev_val
        break
    end

```

```
    k=k+1;
end

sum = cur_val;

end
```

Problem 2 : expseries-script.m

```
format long e

% This variable is the current value of the summation after iteration 'k'.
cur_val = single(0);

% k is our iterater value, that is, it iterates through the summation.
k = 0;

% Running the infinite series until the sum no longer changes
[cur_val, k] = arctanseries(cur_val);

% Equivelant to 4*arctan(1), the approx. value of pi.
pi_approx = 4*cur_val;

% The actial value of pi.
pi_actual = pi;

% Calculating the absolute error.
abs_err = abs(pi_actual - pi_approx);

% Calculating the relative error.
rel_err = abs((pi_actual - pi_approx) / pi_actual);

% Formatting.
% Clears the console, then prints out the required informatWion.
clc
fprintf("a)\n")
fprintf("  1.a) Actual. value of pi from MatLab = %e\n", pi_actual)
fprintf("  1.b) Approx. value of pi using Maclaurin series: 4 arctan(1) = %e\n", pi_approx)
fprintf("  2) Absolute Error: %e\n", abs_err)
```

```
fprintf("   3) Relative Error: %e\n", rel_err)
fprintf("   4) Number of 'k' terms needed to approx. in single percision: %i\n",k)

fprintf("b)\n")
fprintf("   Eventually the next value in the series becomes extremely small (since k is constantly increasing in the denominator, the next value to be added in the series will be small). This is the result of the phenomenon SWAMPING, since we are trying to add two numbers whose sizes are very different (one large and one extremely small). Therefore, the percision will eventually lose track of the very small values computed due to the rounding and computational limitations.\n")
```

>> expseries-script.m

a)
1.a) Actual. value of pi from MatLab = 3.141593e+00
1.b) Approx. value of pi using Maclaurin series: 4arctan(1) = 3.141597e+00
2) Absolute Error: 4.140539e-06
3) Relative Error: 1.317974e-06
4) Number of 'k' terms needed to approx. in single percision: 16777216

b)
Eventually the next value in the series becomes extremely small (since k is constantly increasing in the denominator, the next value to be added in the series will be small). This is the result of the phenomenon SWAMPING, since we are trying to add two numbers whose sizes are very different (one large and one extremely small). Therefore, the percision will eventually lose track of the very small values computed due to the rounding and computational limitations.

Problem 3 : quadroots.m

```
function [x1, x2] = quadroots(a,b,c)

b_sq = power(b,2);
ac4  = 4*a*c;

if b > 0
    x1 = (-2*c) / (b + sqrt(b_sq - ac4));
    x2 = (-1) * (b + sqrt(b_sq - ac4)) / (2*a);
else
    x1 = (-b + sqrt(b_sq - ac4)) / (2*a);
    x2 = (2*c) / (-b + sqrt(b_sq - ac4));
end

end
```

Problem 3 : quadscript.m

```
format long e

% Defining the variables 'a,b,c' as a list for simplicity.
function1 = [1, power(-10,5), 1];
function2 = [1, power(10,5), 1];

% Running the coefficients of both functions into quadroots and storing
% their results.
[function1_x1, function1_x2] = quadroots(function1(1), function1(2), function1(3));
[function2_x1, function2_x2] = quadroots(function2(1), function2(2), function2(3));

% Clearing the console and outputting the results.
clc
fprintf("Function Roots (Computational):\n")
fprintf("function1-x1: %0.15e\nfunction1-x2: %0.15e\nfunction2-x1: %0.15e\nfunction2-x2: %0.15e\n", function1_x1, function1_x2, function2_x1, function2_x2)

% Calculating the real soulutions for function 1 and function 2 using the
% roots obtained in the previous part.
```

```
function1_solution1 = abs(power(function1_x1,2) + power(-10,5)*(function1_x1) + 1);
function1_solution2 = abs(power(function1_x2,2) + power(-10,5)*(function1_x2) + 1);

function2_solution1 = abs(power(function2_x1,2) + power(10,5)*(function2_x1) + 1);
function2_solution2 = abs(power(function2_x2,2) + power(10,5)*(function2_x2) + 1);

% Printing the results.
fprintf("\nFunction Actuals (With Computational Roots):\n")
fprintf("function1(x1): %0.15e\nfunction1(x2): %0.15e\n", function1_solution1, function1_solution2);
fprintf("function2(x1): %0.15e\nfunction2(x2): %0.15e\n", function2_solution1, function2_solution2);

% Calculating the roots of the functions 'bad' computational way.
[function1_ncomp_x1, function1_ncomp_x2] = noncompquadroots(function1(1), function1(2), function1(3));
[function2_ncomp_x1, function2_ncomp_x2] = noncompquadroots(function2(1), function2(2), function2(3));

% Printing the restults.
fprintf("\nFunction Roots (Non-Computational):\n")
fprintf("function1-x1: %0.15e\nfunction1-x2: %0.15e\nfunction2-x1: %0.15e\nfunction2-x2: %0.15e\n", function1_ncomp_x1, function1_ncomp_x2, function2_ncomp_x1, function2_ncomp_x2)

% Calculating the solutions of the 'non' computational roots and
% retrieving their errors.
function1_ncomp_solution1 = abs(power(function1_ncomp_x1,2) + power(-10,5)*(function1_ncomp_x1) + 1);
function1_ncomp_solution2 = abs(power(function1_ncomp_x2,2) + power(-10,5)*(function1_ncomp_x2) + 1);

function2_ncomp_solution1 = abs(power(function2_ncomp_x1,2) + power(10,5)*(function2_ncomp_x1) + 1);
function2_ncomp_solution2 = abs(power(function2_ncomp_x2,2) + power(10,5)*(function2_ncomp_x2) + 1);

% Printing the results.
fprintf("\nFunction Actuals (Backwards Errors With Non-
```

```
    Computational Roots):\n")
fprintf("function1(x1): %0.15e\nfunction1(x2): %0.15e\n", function1_ncomp_solution1, function1_ncomp_solution2);
fprintf("function2(x1): %0.15e\nfunction2(x2): %0.15e\n", function2_ncomp_solution1, function2_ncomp_solution2);
```

>> quadscript.m

```
Function Roots (Computational):
function1-x1: 9.999999999000000e+04
function1-x2: 1.000000000100000e-05
function2-x1: -1.000000000100000e-05
function2-x2: -9.999999999000000e+04

Function Actuals (With Computational Roots):
function1(x1): 0.000000000000000e+00
function1(x2): 0.000000000000000e+00
function2(x1): 0.000000000000000e+00
function2(x2): 0.000000000000000e+00

Function Roots (Non-Computational):
function1-x1: 9.999999999000000e+04
function1-x2: 1.000000338535756e-05
function2-x1: -1.000000338535756e-05
function2-x2: -9.999999999000000e+04

Function Actuals (Backwards Errors With Non-Computational Roots):
function1(x1): 0.000000000000000e+00
function1(x2): 3.384357558644524e-07
function2(x1): 3.384357558644524e-07
function2(x2): 0.000000000000000e+00
```