CS / MATH 4334 : Numerical Analysis
Homework Assignment 1

Matthew McMillian
mgm160130@utdallas.edu

September 5, 2018

Theoretical Problems

1. Show how to evaluate the polynomial $p(x) = 2x^{25} + 7x^{15} - x^{10} + 4x^5 - 1$ using as few arithmetic operations as possible.

   To begin to solve this problem, we must first break down the polynomial p(x). I will start by storing some of the possible polynomial values ahead of time to conserve arithmetic operations.

$x = x$	- identity	(0+0 = 0 multiplications)
$x^2 = x * x$	- 1 multiplication	(0+1 = 1 multiplications)
$x^4 = x^2 * x^2$	- 1 multiplication	(1+1 = 2 multiplications)
$x^5 = x^4 * x$	- 1 multiplication	(2+1 = 3 multiplications)

   In total, storing values until we have stored an x^5 will net us a total of 3 multiplications. Next, we will apply a Horner's expansion to the function $p(x)$, given by:

   $$p(x) = 2x^{25} + 7x^{15} - x^{10} + 4x^5 - 1$$
   $$p(x) = x^5 * (2x^{20} + 7x^{10} - x^5 + 4) - 1$$
   $$p(x) = x^5 * (x^5 * (2x^{15} + 7x^5 - 1) + 4) - 1$$
   $$p(x) = x^5 * (x^5 * (x^5 * (2x^{10} + 7) - 1) + 4) - 1$$
   $$p(x) = x^5 * (x^5 * (x^5 * (x^5 * (2x^5 + 0) + 7) - 1) + 4) - 1$$
   $$p(x) = x^5 * (x^5 * (x^5 * (x^5 * (x^5 * (2) + 0) + 7) - 1) + 4) - 1$$

   Expanding $p(x)$ with Horner's method and storing the calculations prior to the functions evaluation nets us a total of $\boxed{\text{8 multiplications and 5 additions or subtractions}}$.

2. Convert the binary number $101101.000\overline{1011}$ to decimal form.

To convert this binary number to a decimal number, we must convert the items on both the LEFT and RIGHT of the radix point. From the LEFT, we use the algorithm to multiply the binary digits by their respective 2^x compliments:

$$101101_2 = (1 * 2^5) + (0 * 2^4) + (1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0) = 45_{10}$$

From the RIGHT, we could normally multiply by a base 2 number algorithm. However, the repeating decimal makes it tricky. In this case, we must normalize our decimal number and apply some algebra tricks to obtain and approximate value for the decimal.

$x = .000\overline{1011}$
$2^3x = .\overline{1011}$
$2^7x = 1011.\overline{1011}$
$2^7x - 2^3x = (1011)_2 = (11)_{10}$
$x * (2^7 - 2^3) = 111$
$x * (120) = 11$
$x = \frac{11}{120} \approx .091\overline{6}$

Thus our final base 10 decimal number is a concatenation of our LEFT and RIGHT numbers, which is $\boxed{45.091\overline{6}}$.

3. Consider the decimal number -26.1. Convert this number to binary form, then determine the machine representation of this number in double precision. Give the entire set in hexadecimal form.

To convert this decimal number to binary, we must convert both sides of the radix point using the divide and multiply algorithm:

Left of Radix Point	Right of Radix Point
$26 \div 2 = 13r0$	$.1 * 2 = .2r0$
$13 \div 2 = 6r1$	$.2 * 2 = .4r0$
$6 \div 2 = 3r0$	$.4 * 2 = .8r0$
$3 \div 2 = 1r1$	$.8 * 2 = 1.6r1$
$1 \div 2 = 0r1$	$.6 * 2 = 1.2r1$
	$.2 * 2 = 4r0$
	continuing...

Thus given by applying the algorithm, we obtain a binary number of -11010.$0\overline{0011}$ for part (a). Now, to determine the machine representation of this binary number in double precision, we must find out 3 things; the sign bit, the exponent bits, and the mantissa.

- The sign bit is easy. Since we have a negative number our sign bit is set to 1.
- For the exponent, we must first normalize the binary number, and add an offset of 1023 to our exponent to obtain the correct number for the computer:

  Given, e = $11010.0\overline{0011} * 2^0$

  Then we normalize, e = $1.10100\overline{0011} * 2^4$

  Then we add an offset to our exponent, $1023 + 4 = 1027$

  Then we convert to binary, $(1027)_{10} = (10000000011)_2$
- For the mantissa, we put in the numbers after our normalized radix point and check for rounding:

  Let m be the mantissa

  Given $m = |_0$ 1010 | 0001 | 1001 | 1001 | 1001 | ... | 1001 $_{52}|_{53}$ 1001 | ...

  Then we round m. Since the 53^{nd} bit is a 1, and there are non-zero bits after it, we add a 1 to the 52^{nd} bit.

  Thus our $m = |_0$ 1010 | 0001 | 1001 | 1001 | 1001 | ... | 1010 $_{52}|$
- To finalize the problem, we must concatenate our sign bit, exponent bits, and mantissa bits and convert the entire double precision expression to hexadecimal:

  Let $s, e, m =$ the sign bit, exponent bits, and mantissa respectively.

  Let $d =$ the concatenation of s, e, and m, equivalently $d = s + e + m$.

  $d =$ 1100|0000|0011|1010|0001|1001|1001|1001|1001|1001|1001|1001|1001|1001|1010

  Converting to hexadecimal, $d =$ C03A1999999999A

Thus, our final binary number and double precision hexadecimal number are $\boxed{\text{-11010.}0\overline{0011}}$ and $\boxed{\text{C03A1999999999A}}$ respectively.

4. Suppose a certain computer stores decimal numbers (instead of binary) by chopping (instead of rounding) each normalized decimal number to 6 significant digits, i.e., $d_0.d_1d_2d_3d_4d_5$, where $d_0 \neq 0$. Find an upper bound on the relative error from this chopping.

   To find the relative error for this computer, we define relative error to be $|\frac{x - x_c}{x}|$, where x, x_c are an exact number and an approximate number respectively.

   Let $x = d_0.d_1d_2d_3d_4d_5 * 10^p$ be exact.

   Let $x_c = fl(x) = d_0.d_1d_2d_3d_4d_5 * 10^p$ be approximate.

   Then, our relative error becomes $|\frac{x - x_c}{x}| = |\frac{0.d_1d_2d_3d_4d_5}{9.d_1d_2d_3d_4d_5}|$.

   Then, we try to find an upper bound to our error by finding the maximum value of the numerator n_{max}, and minimum value of the denominator d_{min}, which are $n_{max} = 0.00009$ and $d_{min} = 9.00000$.

This results in the relative error of $|\frac{0.00009}{9}| = 0.00001 = 10^{-6}$.

Therefore, similarly to the example in class, we determine that the upper bound of relative error for chopping roundoff error is $\boxed{10^{-6}}$, or more generally, 10^{-p}, where p is the significant digits of your system.

5. Show how roundoff error can propagate through multiplication, given x_c, y_c are approximate values and x, y are exact values. Also, let $x_c = x + \epsilon$ and $y_c = y + \epsilon$.

   To find the roundoff error propagation, we must substitute in the values for x_c and y_c and solve to find a representation that has an error of $\frac{1}{2}\epsilon_{mach}$.

   Given $|\frac{x_c y_c - xy}{xy}| = |\frac{(x+\epsilon_x)(y+\epsilon_y) - xy}{xy}| = |\frac{xy + x\epsilon_y + y\epsilon_x + \epsilon_x\epsilon_y - xy}{xy}| =$

   $|\frac{x\epsilon_y + y\epsilon_x + \epsilon_x\epsilon_y}{xy}| = |\frac{\epsilon_y}{y} + \frac{\epsilon_x}{x} + \frac{\epsilon_x\epsilon_y}{xy}| = |\frac{\epsilon_y}{y}| + |\frac{\epsilon_x}{x}| + |\frac{\epsilon_x\epsilon_y}{xy}| =$

   $2^{-53} + 2^{-53} + (2^{-53})^2 \approx 2^{-52} + 2^{-106} \leq \frac{1}{2}\epsilon_{mach} = 2^{-53}$

   Thus we have proven that the multiplication roundoff error has a bounded error of $\boxed{2^{-52} + 2^{-106}} \leq \frac{1}{2}\epsilon_{mach}$.

6. Let $x = 12345$, $y = 777.76$, and $z = 0.000321123$. Compute $(x - y) * z$ using floating point arithmetic while rounding to 4 significant digits with the usual rounding rules. We evaluate this expression using PEMDAS rules:

   $(x - y)$:

   Given $x = 12345$ and $y = 777.76$, we normalize each variable.

   $x_{norm} = 1.235 * 10^4$, $y_{norm} = 7.778 * 10^2$.

   $(x_{norm} - y_{norm}) = (1.245 * 10^4 - 7.778 * 10^2) = 1.157 * 10^4 = \gamma$

   $(\gamma) * z$:

   Given $\gamma = 1.157 * 10^4$ and $z = 0.0003213$

   $\gamma = \gamma_{norm} = 1.158 * 10^4$, $z_{norm} = 0.000 * 10^4$

   $(\gamma_{norm}) * (z_{norm}) = (1.157 * 10^4 * 0.000 * 10^4) = 0.000.$

   Thus, our resulting answer of $(x - y) * z = \boxed{0.000}$.

7. Consider the function $f(x) = \dfrac{cos(2x) - 1}{2x^2}$. Explain why it is problematic for computational math of x near certain values, use algebra or trigonometric identities to fix the function, then find the first three terms of an approximation of $cos(x) = \Sigma_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!}$.

a.) The original function of $f(x)$ is problematic for specific values of x since it can introduce loss of significance. Consider if we get an x value such that $cos(x)$ is extremely close to 1, but not exactly 1. This would introduce $\boxed{\text{loss of significance}}$ since we could lose important bit information due to rounding in the subtraction operation.

b.) To fix this error, we can substitute $cos(x)$ with a trig identity equivalent to get rid of the -1.

By trig identity, $cos(x) \equiv 1 - 2sin^2(x)$.

Substituting this in, we obtain $f(x) = \dfrac{1 - 2sin^2(x) - 1}{2x^2} = \boxed{\dfrac{-sin^2(x)}{x^2}}$.

We have eliminated the subtraction operator in our equation, thus we can now safely use $f(x)$ computationally.

c.) To find an approximation of $f(x)$ that is less error prone, we can find the first 3 non-zero terms of the Maclaurin series $cos(x) = \Sigma_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!}$.

$$k = 0: \quad \frac{(-1)^0 x^{(2*0)}}{(2*0)!} = 1.$$

$$k = 1: \quad \frac{(-1)^1 x^{(2*1)}}{(2*1)!} = \frac{-x^2}{2}.$$

$$k = 2: \quad \frac{(-1)^2 x^{(2*2)}}{(2*2)!} = \frac{x^4}{24}.$$

Thus, assuming $x \neq 0$, the first 3 terms are the first non-zero terms of $f(x)$ are $\boxed{1, \dfrac{-x^2}{2}, \text{ and } \dfrac{x^4}{24}}$.

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.
```

```
19 clc
20 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

```
1  function [sum, k] = arctanseries(cur_val)
2
3  % Boolean variable to determine when the series will no longer
4  % produce a new terms due to swamping.
5  hasSolved = 0;
6
7  % k is our iterater value, that is, it iterates through the
       summation.
8  k = 0;
9
10 % x is our input the the pseduo 'f(x) = arctan(x)' function.
11 x = single(1);
12
13 % This variable stores the previous value added to the series.
       We use this
14 % to determine if the summation has succumbed to swamping.
15 prev_val = single(0);
16
17 % Pseudocode:
18 % while (the series has not begun to swamp)
19 %    store the previous value of the series
20 %    calculate the new value of the series
21 %    if (the series has swamped)
22 %        stop looping
23 %    end
24 %    go to the next term in the series
25 %  end
26 while ~hasSolved
27     prev_val = cur_val;
28
29     % You can remove the power(x,2*k+1) since this always
           evaluates to 1
30     % for increased performance, however I have left this in
           here for \
31     % semantic reasons.
32     cur_val = cur_val + power(-1,k) * power(x,2*k+1) / (2*k+1)
           ;
33
34     if cur_val == prev_val
35         break
36     end
37
```

```
38      k=k+1;
39  end
40
41  sum = cur_val;
42
43  end
```

Problem 2 : expseries-script.m

```
1   format long e
2
3   % This variable is the current value of the summation after
        iteration 'k'.
4   cur_val = single(0);
5
6   % k is our iterater value, that is, it iterates through the
        summation.
7   k = 0;
8
9   % Running the infinite series until the sum no longer changes
10  [cur_val, k] = arctanseries(cur_val);
11
12  % Equivelant to 4*arctan(1), the approx. value of pi.
13  pi_approx = 4*cur_val;
14
15  % The actial value of pi.
16  pi_actual = pi;
17
18  % Calculating the absolute error.
19  abs_err = abs(pi_actual - pi_approx);
20
21  % Calculating the relative error.
22  rel_err = abs((pi_actual - pi_approx) / pi_actual);
23
24  % Formatting.
25  % Clears the console, then prints out the required
        informatWion.
26  clc
27  fprintf("a)\n")
28  fprintf("  1.a) Actual. value of pi from MatLab = %e\n",
        pi_actual)
29  fprintf("  1.b) Approx. value of pi using Maclaurin series: 4
        arctan(1) = %e\n", pi_approx)
30  fprintf("  2) Absolute Error: %e\n", abs_err)
```

```
31 fprintf("  3) Relative Error: %e\n", rel_err)
32 fprintf("  4) Number of 'k' terms needed to approx. in single
     percision: %i\n",k)
33
34 fprintf("b)\n")
35 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.
```

```
19 function1_solution1 = abs(power(function1_x1,2) + power(-10,5)
     *(function1_x1) + 1);
20 function1_solution2 = abs(power(function1_x2,2) + power(-10,5)
     *(function1_x2) + 1);
21
22 function2_solution1 = abs(power(function2_x1,2) + power(10,5)
     *(function2_x1) + 1);
23 function2_solution2 = abs(power(function2_x2,2) + power(10,5)
     *(function2_x2) + 1);
24
25 % Printing the results.
26 fprintf("\nFunction Actuals (With Computational Roots):\n")
27 fprintf("function1(x1): %0.15e\nfunction1(x2): %0.15e\n",
     function1_solution1, function1_solution2);
28 fprintf("function2(x1): %0.15e\nfunction2(x2): %0.15e\n",
     function2_solution1, function2_solution2);
29
30 % Calculating the roots of the functions 'bad' computational
     way.
31 [function1_ncomp_x1, function1_ncomp_x2] = noncompquadroots(
     function1(1), function1(2), function1(3));
32 [function2_ncomp_x1, function2_ncomp_x2] = noncompquadroots(
     function2(1), function2(2), function2(3));
33
34 % Printing the restults.
35 fprintf("\nFunction Roots (Non-Computational):\n")
36 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)
37
38 % Calculating the solutions of the 'non' computational roots
     and
39 % retrieving their errors.
40 function1_ncomp_solution1 = abs(power(function1_ncomp_x1,2) +
     power(-10,5)*(function1_ncomp_x1) + 1);
41 function1_ncomp_solution2 = abs(power(function1_ncomp_x2,2) +
     power(-10,5)*(function1_ncomp_x2) + 1);
42
43 function2_ncomp_solution1 = abs(power(function2_ncomp_x1,2) +
     power(10,5)*(function2_ncomp_x1) + 1);
44 function2_ncomp_solution2 = abs(power(function2_ncomp_x2,2) +
     power(10,5)*(function2_ncomp_x2) + 1);
45
46 % Printing the results.
```

```
47 fprintf("\nFunction Actuals (Backwards Errors With Non-
       Computational Roots):\n")
48 fprintf("function1(x1): %0.15e\nfunction1(x2): %0.15e\n",
       function1_ncomp_solution1, function1_ncomp_solution2);
49 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
```