Table of Contents

Assignment 2, Problem 1. c) and d)

```
clc; close all; clear all;
epsilon = 1;

while 1
    if (epsilon+1) <= 1
       break;
    end
    epsilon = epsilon/2;
end

epsilon = epsilon*2;
fprintf('Machine Epsilon for my computer is %f\n', epsilon);

%{
In this implementation, the value of epsilon is divided by 2 through
each pass in the otherwise infinite while loop.
After each division, the if statement checks to see if adding 1 to
 epsilon
would have it be less than or equal to 1. Adding a very small number
 to 1
would result in the processor still equating that to 1 in the case
 that
epsilon is so small, that the only value stored in the mantissa would
 be
equal to 1. This kind of roundoff error can occur when adding an
extremely small number to a relatively larger error.
If epsilon+1 does equal 1, that means we divided by 2 one too many
 times;
so, we break out of the while loop, and multiply by 2. The resulting
 value
of epsilon is then the value of Machine Epsilon for this specific
 computer.
%}
```

```
Machine Epsilon for my computer is 0.000000
```

Problem 2. Calculations only

```
clc; close all; clear all;
format long;
upper_bound_error = (.5 * 10^-6);
```

```
x = 0.3*pi;
true = sin(0.3*pi);
ml_approx = 0;
ml = [];
true_error = [];
approx_error = [-100000]; %placeholder since first approx. error = N/A
n = 0;

% I used the vectors stored in the workspace variables to fill out my
 table
% in the handwritten assignment.

while 1
    ml_approx = ml_approx + ((-1)^n)*(x^(2*n + 1)/(factorial(2*n+1)));
    ml(n+1) = ml_approx;
    true_error(n+1) = abs((true - ml_approx)/true * 100);
    if n ~= 0
        approx_error(n+1) = abs((ml(n+1) - ml(n))/ml(n+1) * 100);
        if approx_error(n+1) <= upper_bound_error
            break
        end
    end
    n = n+1;
end
fprintf('The number of iterations to approximate to 8 sig figs is %.0f
\n', n+1);
%Total iterations = n+1, since n starts at 0.
```

The number of iterations to approximate to 8 sig figs is 7

Problem 3

```
clc; close all; clear all;
format long;
sumofe = 0;
sum1 = [];
for n = 0:10
    sumofe = single(sumofe + 1/factorial(n));
    sum1(n+1) = sumofe;
end
te1 = abs((exp(1) - sumofe)/exp(1))*100;
fprintf('The true percent error for e, going from 0 to 10 is %.8f
 percent\n', te1);

sumofe = 0;
sum2 = [];
for n = 10:-1:0
    sumofe = single(sumofe + 1/factorial(n));
    sum2(-n+11) = sumofe;
end
te2 = abs((exp(1) - sumofe)/exp(1))*100;
fprintf('The true percent error for e, going from 10 to 0 is %.8f
 percent\n', te2);
```

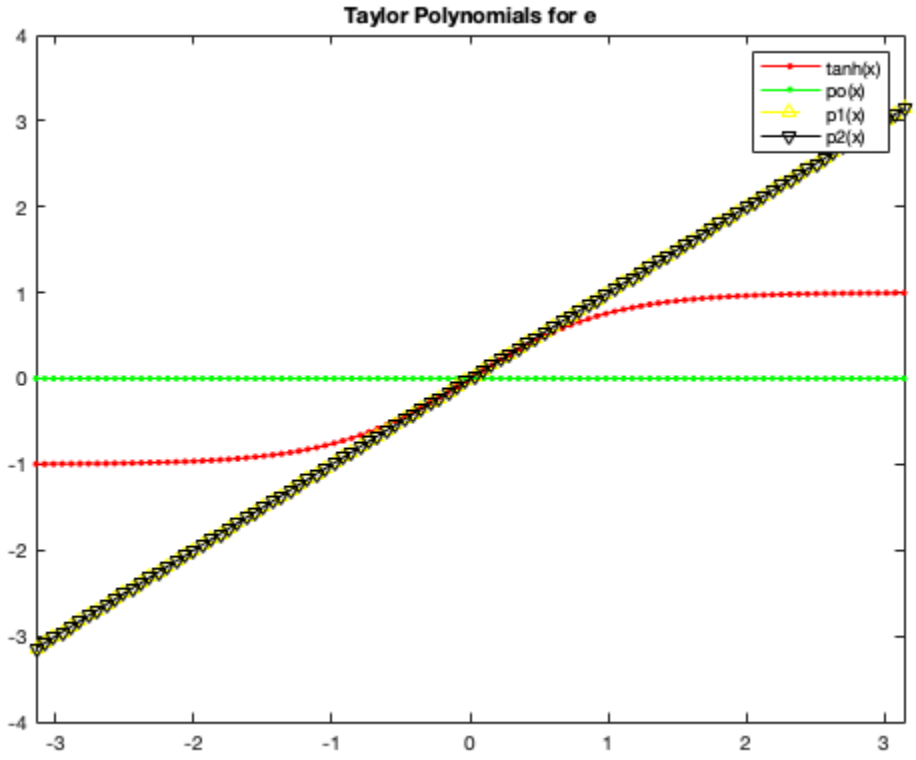
```
%{
The difference between these two percent errors can be explained if
 you
consider how a computer is processing these terms. Taking a look at
 the
vectors sum1 and sum2 will show that sum2 starts with numbers to the
magnitude of 10^-7. The computer can store a certain number of
 significant
digits in its mantissa, and it does not have to store all the leading
 zeros
in the miniscule number, and instead saves it in scientific notation.
 As
sumofe gets gradually larger, the summation is more precise, as the
computer is saving as many significant digits as possible before
 rounding
off.
On the other hand, lookking at sum1, we can see that from the first
 term,
the computer has a digit before the leading zeros to store as well,
 which
decreases the number of significant digits the computer can hold per
 term,
an issue that is only exacerbated as we try and add more and more
 small
yet significant terms to the sum. An example of this can be seen by
 looking
at sum1(3), which chould hold 1.6666666666666 as long as it can before
 the
last digit is changed to a 7. However, the computer is actually
 storing
this as 1.666666626930237. As we add more and more terms to sumofe,
 the
roundoff error compounds, to give a slightly larger error.
%}
```

The true percent error for e, going from 0 to 10 is 0.00000573 percent
The true percent error for e, going from 10 to 0 is 0.00000304 percent

Problem 4, part b)

```
clc; close all; clear all;
x = linspace(-pi, pi, 100);
e = exp(1);
tanhx = (e.^x - e.^(-x))./(e.^x + e.^(-x));

p0 = zeros([1,100]);
p1 = p0 + x;
p2 = p1 + 0.*x.^2;

plot(x,tanhx, 'r.-');
hold on
plot(x, p0, 'g.-');
```

```
plot(x,p1, 'y^--');
plot(x,p2, 'kv-');

title('Taylor Polynomials for e');
xlim([-pi pi]);

legend('tanh(x)', 'po(x)', 'p1(x)', 'p2(x)');
```



Published with MATLAB® R2020a