Selected Solutions to Assignment #2

2. Suppose that $0 < q < p$ and that $\alpha_n = \alpha + O(n^{-p})$ (as $n \to \infty$). Then $\alpha_n = \alpha + O(n^{-q})$.

PROOF. We know that there is a $K > 0$ so that

$$|\alpha_n - \alpha| \le K \frac{1}{n^p}$$

for large n. But, because $p > q$ we know that

$$n^p > n^q$$

for any n which is at least two. But then

$$\frac{1}{n^p} < \frac{1}{n^q}.$$

Therefore

$$|\alpha_n - \alpha| \le K \frac{1}{n^p} < K \frac{1}{n^q}$$

so $\alpha_n = \alpha + O(n^{-q})$. □

3. This exercise requires writing a code. The basic procedure that I follow is this: First open MATLAB/OCTAVE, then also open an editor, and make sure that MATLAB/OCTAVE can "find" the file that is being edited. Once the file is saved, it becomes a command in the language, available at that command line. Note that comments inserted at the beginning of the code become the "help file" for the new command. Note that when you run a code like the ones below, the variables defined and changed by the script become available at the command line. That is, the codes below are *scripts* without the "`function`" keyword at the start, which localizes the scope of the variables.

a. We compute

$$P_{99}(0.3) = \sum_{i=1}^{99} (-1)^{i+1} \frac{(0.3)^{2i-1}}{2i-1}.$$

This sum is computed by a `for` loop, starting with a sum of zero and adding on each term:

atansum.m

```
% ATANSUM  Computes a partial sum of the Maclaurin series for
%   arctan(x).  Evaluates at x=0.3, and compares to value from
%   built-in function.

x = 0.3;
y = 0.0;
for i = 1:99
  y = y + (-1)^(i+1) * x^(2*i-1) / (2*i-1);  % y is partial sum
end

% print out partial sum, built-in value of arctan, and actual error:
y
atan(x)
err = abs(y - atan(x))
```

Note that the name of the arctan function in MATLAB/OCTAVE is "`atan`"; this is common to most programming languages. Once this file is saved as "`atansum.m`", we can do the following at the MATLAB/OCTAVE command line:

```
>> help atansum
'atansum' is a script from the file /home/bueler/Desktop/M310_F10/atansum.m

 ATANSUM  Computes a partial sum of the Maclaurin series for arctan(x).
          Evaluates at x=0.3, and compares to built-in function.
...
>> atansum
y =     0.291456794477867
ans =   0.291456794477867
err = 5.55111512312578e-17
```

Note that this estimate of $\arctan(0.3)$ is apparently *very* accurate.

b. First, to understand the question, note that

$$\tan(\pi/4) = 1 \qquad \Longleftrightarrow \qquad \frac{\pi}{4} = \arctan(1) \qquad \Longleftrightarrow \qquad \pi = 4\arctan(1).$$

Partial sums of the Maclaurin series given in part **a** of the question are approximations of $\arctan(x)$. Thus

$$\pi = 4\arctan(1) \approx 4P_n(1).$$

So here is a program that computes the partial sums, namely $P_n(1)$ for higher and higher n, and then compares to the built-in value of π; note the use of `break` to leave the `for` loop when the error $|4P_n(1) - \pi|$ is the desired smallness:

atan4pi.m

```
% ATAN4PI  Computes partial sums of the Maclaurin series for
%   arctan(x), at x=1, until we are close to the value of pi.

format long g
x = 1.0;
y = 0;
for i = 1:9999                    % goes well past where needed
  y = y + (-1)^(i+1) * x^(2*i-1) / (2*i-1);   %  y = P_n(1)  when i=n
  err_vs_pi = abs(4 * y - pi);  % distance between  4P_n(1)  and  pi
  [i 4*y err_vs_pi]             % show intermediate results
  if err_vs_pi < 1.0e-3
    break                       % break out of loop if  4P_n(1)  is
  end                           %   close to  pi
end
```

Running it gives this output, which shows the counter `i`, the approximation $4P_i(1)$, and the error $|4P_i(1) - \pi|$:

```
>> atan4pi
ans =
                   1                   4     0.858407346410207
ans =
                   2     2.66666666666667     0.474925986923126
ans =
```

```
                     3      3.46666666666667     0.325074013076874
...
ans =
                   998      3.14059064983328   0.00100200375650905
ans =
                   999      3.14259365434004    0.001001000750251
ans =
                  1000      3.14059265383979  0.000999999749998981
```

Yes, amazingly enough, the error goes below 10^{-3} exactly on the 1000th term:

$$|4P_{1000}(1) - \pi| < 10^{-3}.$$

Note the much slower convergence of the series for arctan(1) than for arctan(0.3). In fact, the radius of convergence for the Maclaurin series for arctan(x) is $R = 1$, and the convergence of the series for arctan(1) is merely conditional, not absolute.

The above method computes the actual error by using the MATLAB/OCTAVE built-in value for π. That is, it "unrealistically" uses the exact answer to evaluate the quality of the approximation. As suggested in the question, however, one may know in advance how many terms the partial sum must have to be close to π. In fact, for an alternating series, the difference between the sum and a partial sum is less than the $(n+1)$st term. Abstractly:

$$\left| \sum_{j=1}^{\infty} (-1)^j a_j - \left(\sum_{j=1}^{n} (-1)^j a_j \right) \right| \leq a_{n+1},$$

assuming that $a_j > 0$ for all j and that the a_j decrease, so that the series is genuinely alternating and is convergent. (See any calculus book for a proof of this fact.) Thus we find the first n for which

$$\frac{4}{2(n+1)-1} \leq 10^{-3},$$

which gives $2n + 1 \geq 4000$ or $n \geq 3999/2 = 1999.5$. It follows that the partial sum $P_{2000}(1)$ is as close as we want, which is true. And, as we now know, mildly conservative.

4. Again I wrote a program, a script "m-file", and saved it as `bisect4a.m`. One step in making the program relatively clean is the line

```
f = @(x) x.^4 - 2 * x.^3 - 4 * x.^2 + 4 * x + 4;
```

This saves the function itself. It allows easy evaluation of the function when needed. This syntax for defining functions is called an "anonymous" function, because the expression with "`@(x)`" defines a function already, even though we then give it the name "`f`". (The command "`inline`" is similar, but use of anonymous functions is recommended while `inline` is deprecated.)

a. The initial bracket is $[-2, -1]$ which has length 1. Thus we know that *halving this bracket 7 times* reduces the length of the bracket to $1/128 < 10^{-2}$. That is, $2^7 > 100$.

bisect4a.m

```
% BISECT4A  Solve polynomial root problem by bisection.

f = @(x) x.^4 - 2 * x.^3 - 4 * x.^2 + 4 * x + 4;  % define the fcn

a = -2;                  % set initial bracket
b = -1;
[f(a) f(b)]              % print out f values; show we have a bracket
```

```
for n = 1:7
  c = (a+b)/2;
  if f(c) > 0           % note that  f(a)>0,  so if  f(c)>0  then
    a = c;              %   c is the new a
  else
    b = c;
  end
end

[a b]                   % show:  final bracket
(b - a)/2               %        half-length of bracket (=max err)
[c f(c)]                %        estimate of root and f value
```

Running shows several outputs. First we see that $f(a) = f(-2) = 12 > 0$ and $f(b) = f(-1) = -1 < 0$, so we have a bracket. Then we see the final bracket, and its half-length, and the final estimate $c = -1.4140625$. So we know that there is a solution within 0.0039 of the number $c = -1.4140625$.

```
>> format long g
>> bisect4a
ans =              12                  -1
ans =       -1.421875           -1.4140625
ans =      0.00390625
ans =      -1.4140625  -0.00120812281966209
```

b. The program here is nearly identical, so I don't even show it. Note that, again, $f(a) = f(0) > 0$, but the initial bracket length is 2 so it makes sense to do 8 halvings. The run looks like:

```
>> bisect4b
ans =               4                  -4
ans =       1.4140625            1.421875
ans =      0.00390625
ans =       1.4140625   0.00120848789811134
```

5. This time I used `fprintf`, which does formatted printing of strings, including numbers computed inside the program. Some numbers should be printed as integers and some as floating-point numbers, and sometimes we want scientific notation, and so on. Programmers who have used the C language will already recognize how `fprintf` works, but for the rest of us see:

`http://www.mathworks.com/help/techdoc/ref/fprintf.html`

The program also checks the sign of $f(x)$ at the left end of the bracket, and updates the bracket the right way whether $f(a)$ is positive or negative. The program is otherwise similar to the last problem:

bisect5.m

```
% BISECT5  Solve polynomial root problem by bisection.

f = @(x) 3 * x - exp(x);  % define the function

a = 1;  b = 2;            % set initial bracket
s = sign(f(a));           % either +1 or -1 according to sign of f(a)
for n = 0:19
  fprintf('  [a_%2d, b_%2d] = [%f, %f]\n', n, n, a, b)
  c = (a+b)/2;
  if sign(f(c)) == s      % now this works if  f(a)>0  *or* if  f(a)<0
    a = c;
```

```
  else
    b = c;
  end
end
fprintf('final estimate  c = %f  has maximum error %e\n',c,(b-a)/2)
```

The output is clear and informative:

```
>> bisect5
  [a_ 0, b_ 0] = [1.000000, 2.000000]
  [a_ 1, b_ 1] = [1.500000, 2.000000]
  [a_ 2, b_ 2] = [1.500000, 1.750000]
  [a_ 3, b_ 3] = [1.500000, 1.625000]
  [a_ 4, b_ 4] = [1.500000, 1.562500]
  [a_ 5, b_ 5] = [1.500000, 1.531250]
  [a_ 6, b_ 6] = [1.500000, 1.515625]
  [a_ 7, b_ 7] = [1.507812, 1.515625]
  [a_ 8, b_ 8] = [1.511719, 1.515625]
  [a_ 9, b_ 9] = [1.511719, 1.513672]
  [a_10, b_10] = [1.511719, 1.512695]
  [a_11, b_11] = [1.511719, 1.512207]
  [a_12, b_12] = [1.511963, 1.512207]
  [a_13, b_13] = [1.512085, 1.512207]
  [a_14, b_14] = [1.512085, 1.512146]
  [a_15, b_15] = [1.512115, 1.512146]
  [a_16, b_16] = [1.512131, 1.512146]
  [a_17, b_17] = [1.512131, 1.512138]
  [a_18, b_18] = [1.512131, 1.512135]
  [a_19, b_19] = [1.512133, 1.512135]
final estimate  c = 1.512134  has maximum error 4.768372e-07
```

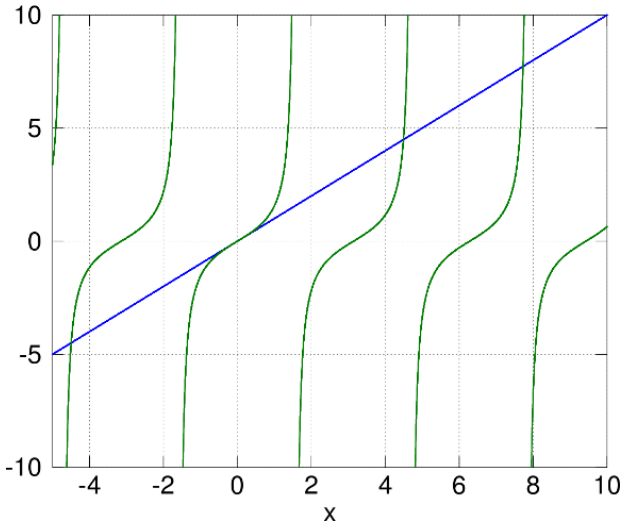

FIGURE 1. Superimposed plots of $y = x$ and $y = \tan x$. The first place they cross to the right of $x = 0$, at about $x = 4.3$, is the first positive solution of $x = \tan x$.

6. **a.** I made a single figure showing what I wanted:

xtanx.m

```
% XTANX plots of y=x and y=tan(x), superimposed

x = -5:.0001:10;
```

```
plot(x,x,'.',x,tan(x),'.')
grid on, xlabel('x')
axis([-5 10 -10 10])
```

The result is Figure 1. In `xtanx.m`, note the use of `axis` to scale the y-axis; the default scaling will show the huge values of $\tan x$, and hide the crossings we seek. Also, the plot style uses isolated dots, and not the default lines-connecting-dots. This means there are no vertical lines showing where $\tan x$ "jumps" from $+\infty$ to $-\infty$.

b. Define the function $f(x) = x - \tan x$. We want to solve $f(x) = 0$. From the graph, the solution we want is in the interval $[4, 3\pi/2]$, so we choose initial bracket $[4, 4.7]$. Here's the code:

solvextanx.m

```
% SOLVEXTANX  Solve x = tan(x) for first positive root using bisection.

f = @(x) x - tan(x);      % define the fcn

a = 4; b = 4.7;           % set initial bracket
s = sign(f(a));           % either +1 or -1 according to sign of f(a)
n = 0;
while (b-a)/2 > 1.0e-5
  fprintf('  [a_%2d, b_%2d] = [%f, %f]\n', n, n, a, b)
  c = (a+b)/2;
  if sign(f(c)) == s      % now this works if  f(a)>0  *or* if  f(a)<0
    a = c;
  else
    b = c;
  end
  n = n+1;
end
fprintf('final estimate  c = %f  has maximum error %e\n',c,(b-a)/2)
```

And here's the output, showing 4.49342 is the first positive solution of $x = \tan x$:

```
>> solvextanx
  [a_ 0, b_ 0] = [4.000000, 4.700000]
  [a_ 1, b_ 1] = [4.350000, 4.700000]
  [a_ 2, b_ 2] = [4.350000, 4.525000]
  [a_ 3, b_ 3] = [4.437500, 4.525000]
  [a_ 4, b_ 4] = [4.481250, 4.525000]
  [a_ 5, b_ 5] = [4.481250, 4.503125]
  [a_ 6, b_ 6] = [4.492188, 4.503125]
  [a_ 7, b_ 7] = [4.492188, 4.497656]
  [a_ 8, b_ 8] = [4.492188, 4.494922]
  [a_ 9, b_ 9] = [4.492188, 4.493555]
  [a_10, b_10] = [4.492871, 4.493555]
  [a_11, b_11] = [4.493213, 4.493555]
  [a_12, b_12] = [4.493384, 4.493555]
  [a_13, b_13] = [4.493384, 4.493469]
  [a_14, b_14] = [4.493384, 4.493427]
  [a_15, b_15] = [4.493405, 4.493427]
final estimate  c = 4.493416  has maximum error 5.340576e-06
```