Problem 1.

In an attempt to have Matlab compute the sum $S = 0.1 + 0.2 + \cdots + 0.9$, someone writes the following code:

```
s = 0
x = 0
while x~= 1.0
        s = s + x
        x = x + 0.1
end
S = s
```

a) Test this code in Matlab. Why does the program keep running indefinitely?

b) What should be changed in the code to make it stop?

Solution

a) The variable x is stored in IEEE binary floating point format. In exact binary system, $0.1 = (0.000\overline{1100})_2$. To represent this number in IEEE format, rounding off must be performed to terminate the infinite sequence of digits. Therefore, x is no longer represented by a number of value exactly 0.1. In fact, x is represented by

$$(1.1001100110011001100110011001100110011001100110011010)_2 \times 2^{-4}$$

which is about 0.1000000000000000055511151231257 in decimal system. As a consequence, the value of variable x in the program after 9 loops is very close to 1.0, but is not exactly 1.0. As such, the exit condition $x \neq 1.0$ never occurs, resulting in an infinite loop. One way to see why x can never reach exactly 1 is by printing out the difference between 1.0 and x at each step by adding the command `disp(x - 1)`inside the while loop. The iterates displayed to the terminal will always be nonzero.

b) We should change the stopping condition for the while loop. One way to do so is by changing the logical condition on the third line from $x \sim= 1$ to $x < 0.95$.

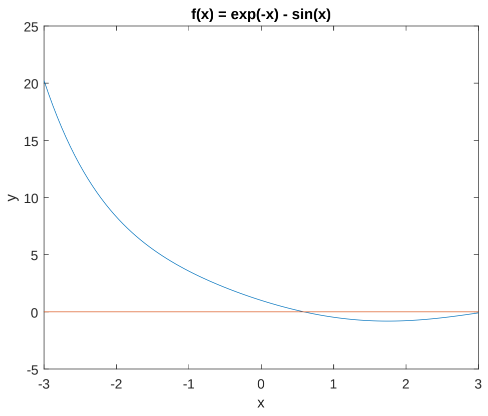
Problem 2.

We would like to compute approximately a solution to the equation $e^{-x} = \sin(x)$.

a) Plot the function $f(x) = e^{-x} - \sin(x)$. What is an interval $[a, b]$ that contains only the smallest positive root of f?

Solution

a)



We notice that the function is decreasing for $x < 0$. The interval $[0, 1]$ contains the smallest root of f and no other roots.

b) We can construct a table to describe the iterates

a	b	$f((a+b)/2)$
0	1	0.1271
$\frac{1}{2}$	1	-0.2093
$\frac{2}{4}$	$\frac{3}{4}$	-0.0498
$\frac{2}{4}$	$\frac{5}{8}$	0.365
$\frac{9}{16}$	$\frac{5}{8}$	

Our fourth midpoint is $\frac{9}{16}$. The sequence of estimates is $1/2,\ 3/4,\ 5/8,\ 9/16$. Our estimation after 4 iterates is $\frac{9}{16} = 0.5625$.

c) We can modify the sample code available on the course website. We can insert a general function call to allow us to more easily modify the code for problem 3. We need to change the direction of the inequality on line 7. This program produces an estimate of 0.588532924652100, close to our previous estimate of 9/16.

```
err = 10^(-6);
a = 0;
b = 1;
c = (a+b)/2;
while (b-a>err)
        if sign((problem_func(b)))*sign(problem_func(c))>0
                b = c;
        else
                a = c;
        end
                c = (a+b)/2
end
c

function out = problem_func(x)
out = exp(-x) - sin(x);
end
```

Problem 3.

The following functions are theoretically the same.

$$\begin{aligned} f_1(x) &= (x-1)^3 \\ f_2(x) &= -1 + (x)(3 + x(-3 + x)) \\ f_3(x) &= x^3 - 3x^2 + 3x - 1 \end{aligned}$$

However their computations in floating-point format are different. Let us do some experiments on finding roots of each function by bisection method. (Note that $x = 1$ is the only root).

a) Modify slightly the program in Problem 2, Part (c) to a program that computes approximately the root of f_1, f_2, and f_3 with error tolerance of $\epsilon = 10^{-6}$.

b) Try the following initial intervals $[a, b] = [0, 1.5], [0.5, 2.0], [0.5, 1.1]$. Explain the results.

Solution

a) We make several small modifications to the code to obtain where n is used to select the desired function

```
err = 10^(-6);
a = 0;
b = 2;
c = (a+b)/2;
n = 3; % which function do we want?
while (b-a>err)
        if sign((problem_func(b,n)))*sign(problem_func(c,n))>0
                b = c;
        else
                a = c;
        end
                c = (a+b)/2
end
c

function out = problem_func(x,n)
switch n
        case 1
                out = (x-1).^3;
        case 2
                out = -1 + x.*(3+x.*(-3+x));
        case 3
                out = x^3 - 3*x.^2 +3.*x -1;
end
end
```

to evaluate.

b) We use n to denote the particular f_n function ($n = 1, 2, 3$). First for $[a, b] = [0, 1.5]$.

n	root
1	1.000000119209290
2	1.000005125999451
3	1.000007271766663

Secondly for $[a, b] = [0.5, 2.0]$.

n	root
1	0.999999880790710
2	1.000004887580872
3	1.000007033348083

Lastly for $[a, b] = [0.5, 1.1]$.

n	root
1	1.000000095367432
2	1.000004100799561
3	1.000006961822510

We see that for any of the three choices of the interval, the root obtained from f_1 is closest to the true root. When evaluating these functions, we notice that f_1 requires the least number of operations. At each step, the arithmetic error caused by evaluating f_1 is, in general, slightly smaller than those caused by f_2 and f_3. After many steps, these seemingly insignificant errors at each step accumulate, resulting in a 'visible' difference in final results.