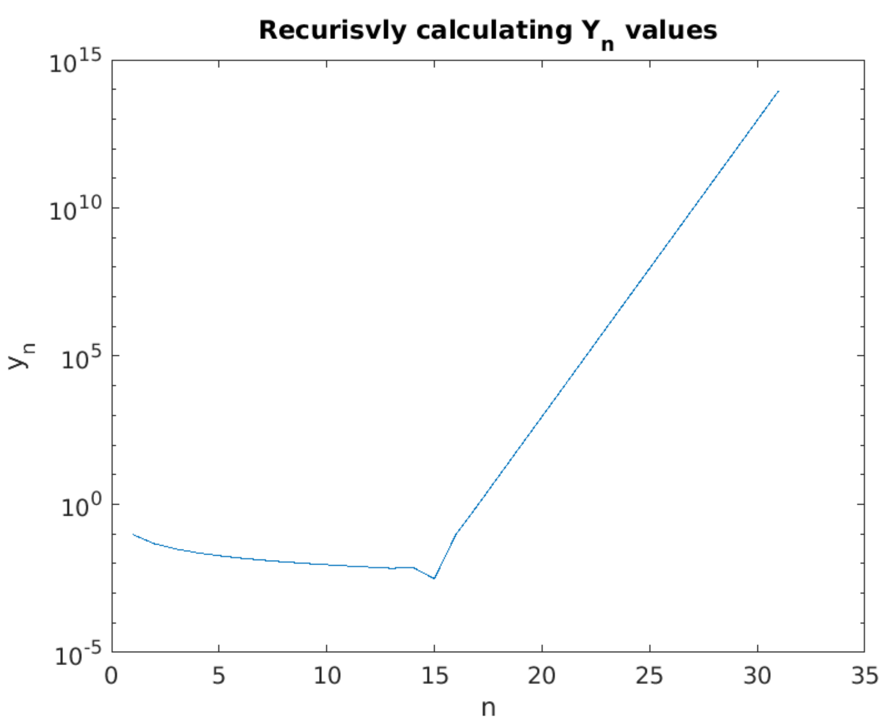
CX 4640 Assignment 1

Wesley Ford

August 24th, 2020

1. Example 1.6 on page 13 of our textbook discusses a recursive formula for computing the integral y_n for $n = 1, 2, ..., 30$.
   i. Compute the thirty values of y_n using the recursive formula and verify the exponential growth of the errors. Plot the computed values in a graph with a logarithmic y-axis scale

n	yn
0	9.53E-02
1	4.69E-02
2	3.10E-02
3	2.32E-02
4	1.85E-02
5	1.54E-02
10	8.33E-03
15	9.74E-02
20	-9.17E+03
25	9.17E+08
30	-9.17E+13



As we can see from the table and from plotting the magnitude of the Y_n values, we can identify an exponential error revealing itself around $n = 15$.

ii. In our first lecture, we mentioned it was possible to approximate the area under a curve(an integral) by the sum of the areas of a number of rectangles. Implement such a method (using Matlab or a language of your choice) and compute y_n for several values of n with your method. You should show the results when a very large number of rectangles is used, as well as when a modest (smaller) number of rectangles is used. Show the key parts of your code in your submitted solutions.

When only using a small number of rectangles (n=10), significant portions of the curve are estimated. At large n, this causes the integral to be severely underestimated. Using larger numbers of n fixes this discretion error by using smaller rectangles.

n	y_n $(n = 10)$	y_n $(n = 1e6)$
0	$9.57661e{-2}$	$9.53106e{-2}$
5	$1.11786e{-2}$	$1.53483e{-2}$
10	$4.52449e{-3}$	$8.32342e{-3}$
15	$2.26379e{-3}$	$5.70796e{-3}$
20	$1.22994e{-3}$	$4.34249e{-3}$

```
num_rect = 1e6;
minX = 0;
maxX = 1;
step = (maxX-minX) / num_rect;
steps = minX:step:maxX
n=0:30;
riemann = zeros(1,length(n));
for i=n
    func = getIntegrand(i);
    for x=steps(1:end-1)
        riemann(i+1) = riemann(i+1) + (func(x));
    end
end
riemann = riemann .* step;
```

An implementation a Riemann summation approximation for estimating y_n.

iii. What would you consider to be the most accurate way to compute the integrals y_n? Implement your method and compare the results to those in part (b).

To compute accurately the integrals y_n, we can manipulate the recursive formula $y_n = \frac{1}{n} - 10y_{n-1}$, into $y_{n-1} = \frac{1}{10}(\frac{1}{n} - y_n)$. After each recursive iteration, any error that exists in our initial estimation of y_n is reduced by about a factor of 10. Using this method we can start at some estimation of an arbitrary y_n value, and recursively calculate y_{n-1}.

```
%An alternative recursive algorithm for computing yn.
%Given an estimation for y30 (in this %case 0), the values
%for y0-y29 very quickly converge to an accurate approximation.
%The error in the initial estimation is divided by 10 each iteration.

y_alt = zeros(1,30);
for i=length(y_alt):-1:2
    y_alt(i-1) = 0.1 * ((1/(i)) - y_alt(i));
end
```

```
y0_alt = 0.1 * (1 - y_alt(1))
y_alt = [y0_alt, y_alt];
```

Compared with the Riemann sum method in part b, this method does not require hundreds or thousands of iterations to get a good estimate. Comparing this method and a large Riemann Sum to Matlab's integral function, the alternative recursive method approaches 5 digits of accuracy within 6 iterations (y_{24}), even with a relatively bad starting approximation of $y_{30} = 0$. To increase the accuracy of values like y_30, a large n value could be used.

n	Matlab Integral Yn	Alternative Recursive Yn	Reimann Sum (n=1e4)
0	9.53102E-02	9.53102E-02	9.53106E-02
5	1.53529E-02	1.53529E-02	1.53484E-02
10	8.32797E-03	8.32797E-03	8.32342E-03
15	5.71251E-03	5.71251E-03	5.70797E-03
20	4.34704E-03	4.34704E-03	4.34249E-03
24	3.64916E-03	3.64916E-03	3.64462E-03
25	3.50835E-03	3.50838E-03	3.50381E-03
26	3.37800E-03	3.37771E-03	3.37346E-03
27	3.25699E-03	3.25993E-03	3.25245E-03
28	3.14435E-03	3.11494E-03	3.13981E-03
29	3.03924E-03	3.33333E-03	3.03470E-03
30	2.94093E-03	0	2.93639E-03

2. Perhaps a surprising property of finite precision floating-point arithmetic is that it is not associative, due to roundoff errors.

   i. Find three numbers a, b, and c that can be represented in IEEE double precision such that

   $$(a+b)+c \neq a+(b+c)$$

   Explain how you found these numbers, and show using Matlab that equality does not hold.

   To find three numbers a, b, and c which are non-associative, all we need to do is find numbers such that $a+b=b$, and then set $c=-b$. If pick $b=2^{1023}$, then the smallest precision number would be $2^{1023}*2^{-52}=2^{971}$. Any number less than 2^{971} can not be recorded, as the mantissa does not have enough bits to record to a smaller precision. Setting $a=2^{970Flo}$, and $c=-2^{1023}$, we find that $(a+b)+c=0$, but $a+(b+c)=2^{970}$. Using Matlab, we can confirm this.

```
a = 2^970;
b= 2^1023;
c= -2^1023;

d = (a+b) + c
e = a + (b+c)
```

```
d = 0
e = 9.9792e+291
```

   ii. Associativity does not hold either for finite precision multiplication. Again using IEEE double precision, explain *how common* you think it is to find that

   $$(a*b)*c \neq a*(b*c)$$

   for arbitrary values of a, b, and c.

   In IEEE double precision, non-associativity can happen realtivly often. There are the situations where we might get an overflow or and underflow, for example when $a=2^{-1023}, b=2^{1023}, c=2^2$. We know $a*b*c=2^2$, however, when $b*c$ is done first, we get an overflow error, resulting in $(a*b)*c=4, a*(b*c)=Inf$. Another occurrence of these errors would come from when a rounding error occurs in $(a*b)$ and not $(b*c)$. A good example of this is $a=0.1, b=0.2, c=0.3$. Using Matlab, we can see that $(0.1*0.2)*0.3 \neq 0.1*(0.2*0.3)$. Generalizing, we can create a Matlab script which randomly generates a mantissa as well as an exponent.

```
a=2^-1023;
b=2^1023;
c=2^2;
(a*b)*c
a*(b*c)

a=0.1;
b=0.2;
c=0.3;
(a*b)*c == a*(b*c)

tmp = 0
range = -128:128;
for e1=range
    for e2=range
        for e3=range
            m1 = rand(1,1);
            m2 = rand(1,1);
            m3 = rand(1,1);
            a=m1*2^e1;
            b=m2*2^e2;
            c=m3*2^e3;
            tmp = tmp + ((a*b)*c ~= a*(b*c));
        end
    end
end
tmp / (length(range))^3
```

```
ans = 4
ans = Inf

ans = logical
   0

tmp = 0

ans = 0.3484
```

   Sampleing random values from exponents ranging from -128 to 128, we approximate that with 3 random IEEE double precision values a, b, c, about 35% of the time $(a*b)*c \neq a*(b*c)$.