FP_density_instructor_final

September 17, 2019

```
In [1]: import numpy as np
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn as sns
        sns.set(font_scale=1.5)
        plt.style.use('seaborn-whitegrid')
        mpl.rcParams['figure.figsize'] = (10.0, 8.0)
```

1 Distribution of Floating Point Numbers

Recall that any positive number x in a normalized binary floating-point system is of the form:

$x = 1.a_1a_2a_3 \ldots a_n \times 2^m$,

where n is the number of bits stored in the fractional part of the significand, $a_i \in \{0, 1\}$ are the bits themselves, and $m \in [L, U]$ is the exponent.

Suppose we have a binary floating point system with $n = 3$, an exponent range of $[L, U] = [-4, 4]$, and we want to generate a list of all the **positive normalized** numbers in this floating point system in ascending order. We would like to store their representation in the standard **decimal** system.

That is, our list should look like this:

- $1.000 \times 2^{-4} = .0625$
- $1.001 \times 2^{-4} = .0703125$
- $1.010 \times 2^{-4} = .078125$
- $1.011 \times 2^{-4} = .0859375$
- $\ldots$
- $1.111 \times 2^{-4} = .1171875$
- $1.000 \times 2^{-3} = .125$
- $1.001 \times 2^{-3} = .140625$
- $\ldots$
- $1.110 \times 2^{4} = 28$
- $1.111 \times 2^{4} = 30$

In order to do so, let's recall that the binary representation means:

$$1.a_1a_2a_3 \times 2^m = \left(1 + \frac{a_1}{2} + \frac{a_2}{2^2} + \frac{a_3}{2^3}\right) \times 2^m \tag{1}$$

We'll write a nested loop to generate all the numbers in the list in decimal format using this expression:

```
In [2]: base = 2
        exponent_min = -4
        exponent_max = 4

        fp_3 = []
        for m in range(exponent_min,exponent_max + 1):
            for a_1 in range(0,base):
                for a_2 in range(0,base):
                    for a_3 in range(0,base):
                        significand = 1 + a_1*base**-1 + a_2*base**-2 + a_3*base**-3
                        new_number = significand * base**m
                        fp_3.append(new_number)
```

```
In [3]: print(fp_3[:4]) # first four number
        print(fp_3[-2:]) # last two numbers
```

```
[0.0625, 0.0703125, 0.078125, 0.0859375]
[28.0, 30.0]
```

Ok, it seems to be working. However, we used 4 loops: 1 for the exponent, and 1 for each bit in the fraction. So if we want to do the same thing for a floating point system for $n = 23$, like IEEE single precision, we would need 24 loops! That's way too much!

There's a simpler way to express these floating point numbers. Let's examine the first 4 numbers:

$$1.000 \times 2^{-4} = \left(1 + \frac{0}{2^1} + \frac{0}{2^2} + \frac{0}{2^3}\right) \times 2^{-4} = \left(1 + \frac{0}{2^3}\right) \times 2^{-4}$$

$$1.001 \times 2^{-4} = \left(1 + \frac{0}{2^1} + \frac{0}{2^2} + \frac{1}{2^3}\right) \times 2^{-4} = \left(1 + \frac{1}{2^3}\right) \times 2^{-4}$$

$$1.010 \times 2^{-4} = \left(1 + \frac{0}{2^1} + \frac{1}{2^2} + \frac{0}{2^3}\right) \times 2^{-4} = \left(1 + \frac{1}{2^2}\right) \times 2^{-4} = \left(1 + \frac{2}{2^3}\right) \times 2^{-4}$$

$$1.011 \times 2^{-4} = \left(1 + \frac{0}{2^1} + \frac{1}{2^2} + \frac{1}{2^3}\right) \times 2^{-4} = \left(1 + \frac{1}{2^2} + \frac{1}{2^3}\right) \times 2^{-4} = \left(1 + \frac{2}{2^3} + \frac{1}{2^3}\right) \times 2^{-4} = \left(1 + \frac{3}{2^3}\right) \times 2^{-4}$$

Are you seeing a pattern? Use this to generate all the positive floating point numbers in a binary system with $n = 4$ stored bits, and exponent range $[L, U] = [-5, 5]$. You should only need two loops.

Write a code snippet to that creates the variable list `fp_numbers`, storing all the floating numbers in ascending order

```
In [4]: #clear
        n = 4
        p = n+1
```

```
exponent_min = -5
exponent_max = 5

fp_numbers = []
for m in range(exponent_min,exponent_max + 1):
    for sigs in range(2**n):
        fp_numbers.append( (1+sigs/(2**n))*2**(m) )
```

Print the resulting numbers you stored in `fp_numbers`. You should check the following:

- What are the largest and smallest positive number in your floating point system?
- What is the range of integer numbers that you can represent **exactly**?

- Do the answers to these questions make sense? Think about the expressions you derived in previous lectures. Do you think your implementation is correct?

In [5]:
```
#clearn
print(fp_numbers)
p=n+1
print("Integer range:", 2**p)
print("Smallest positive number:", 2**(exponent_min) )
print("Largest positive number:", 2**(exponent_max+1)*(1-2**(-p)) )
```

```
[0.03125, 0.033203125, 0.03515625, 0.037109375, 0.0390625, 0.041015625, 0.04296875, 0.044921875,
Integer range: 32
Smallest positive number: 0.03125
Largest positive number: 62.0
```

1.0.1 What is machine epsilon for this floating point system?

The index() method returns the index of the element `x` in the list `fp_numbers`:

```
ind = fp_numbers.index(x)
```

Recall that machine epsilon is the gap between 1 and the **next larger** floating point number. How can you use the method above to obtain machine epsilon?

In [6]:
```
# Write some code here
eps_machine = ...
print('Machine Epsilon is:', eps_machine)
```

```
Machine Epsilon is: Ellipsis
```

In [7]:
```
#clear
ind = fp_numbers.index(1.0)
eps_machine = fp_numbers[ind+1] - fp_numbers[ind]
print('Machine Epsilon is:', eps_machine)
```

```
Machine Epsilon is: 0.0625
```

What is the underflow level for this floating point system?

```
In [8]: UFL = ...
        print('Underflow Level is:', UFL)
```

```
Underflow Level is: Ellipsis
```

```
In [9]: #clear
        UFL = fp_numbers[0]
        print('Underflow Level is:', UFL)
```

```
Underflow Level is: 0.03125
```

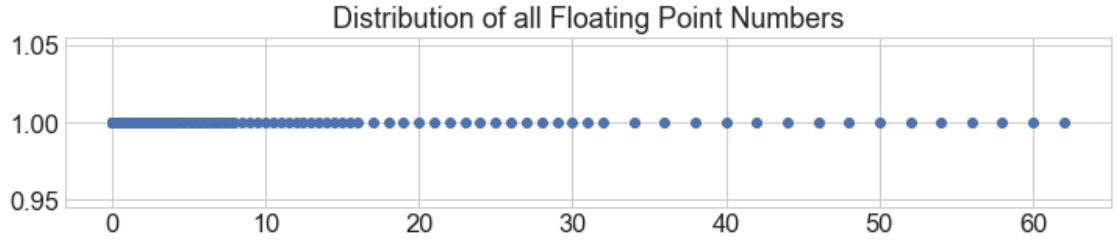
Check if the numbers you obtained above are consistent with the definitions you obtained in previous lectures.

1.0.2 Display a plot of the density of the floating point numbers in this system

Use the plotting function below to see the distribution of the floating point numbers you obtained above:

```
In [10]: plt.figure(figsize=(12,2))
         plt.plot(fp_numbers, np.ones_like(fp_numbers), "o");
         plt.title('Distribution of all Floating Point Numbers')
```

```
Out[10]: Text(0.5, 1.0, 'Distribution of all Floating Point Numbers')
```



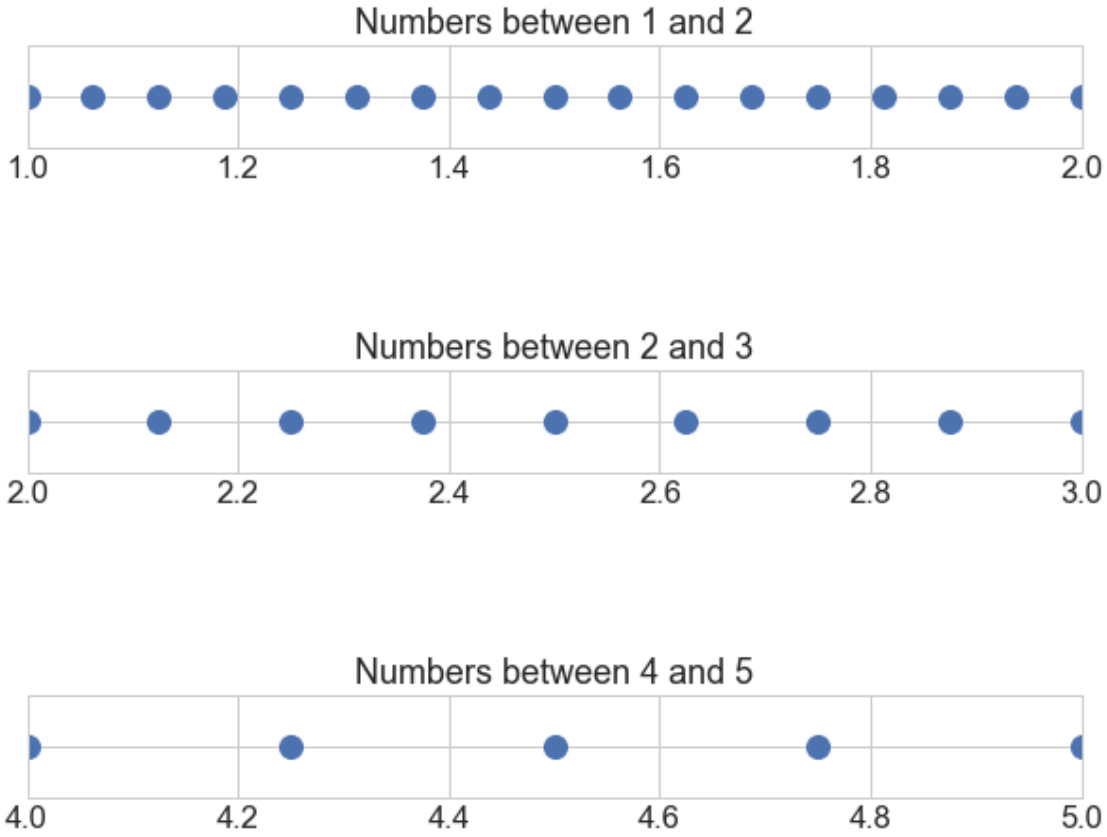
1.0.3 Let's check the gaps between two adjacent floating point numbers.

We will be using the helper function `number_scale` to plot the floating point numbers `fp_numbers` for a given interval `[start,end]`.

```
In [11]: def number_scale(fp_numbers,start,end):
             plt.figure(figsize=(10,1))
             plt.plot(fp_numbers, np.ones_like(fp_numbers), "o",clip_on = True, markersize=12);
             plt.axis([start,end, 0.95, 1.05])
             plt.title('Numbers between ' + str(start)+ ' and ' +  str(end))
             cur_axes = plt.gca()
             cur_axes.get_yaxis().set_ticklabels([]);
```

Use the function `number_scale` to plot FP numbers between integers: Try the intervals $[1,2]$, $[2,3]$ and $[4,5]$

```
In [12]: #clear
         number_scale(fp_numbers,1,2)
         number_scale(fp_numbers,2,3)
         number_scale(fp_numbers,4,5)
```



What do you observe about the gap between these numbers? - What is the gap between FP numbers in the interval $[1,2]$? - What is the gap between FP numbers in the interval $[2,3]$? - What is the gap between FP numbers in the interval $[4,5]$?

Answer the questions above using the "gap definition" you learned in class, and also by evaluating the difference between to consecutive FP numbers on that interval. You should be getting the same thing (but using the definition is a lot more convenient :-))

```
In [13]: #clear
         print("gap in $[1,2]$ interval is: ", eps_machine*2**0)
         print("or also in this way", fp_numbers[91]-fp_numbers[90])

         print("gap in $[2,3]$ interval is: ", eps_machine*2**1)
         print("or also in this way", fp_numbers[100]-fp_numbers[99])

         print("gap in $[4,5]$ interval is: ", eps_machine*2**2)
         print("or also in this way", fp_numbers[114]-fp_numbers[113])
```

```
gap in $[1,2]$ interval is:  0.0625
or also in this way 0.0625
gap in $[2,3]$ interval is:  0.125
or also in this way 0.125
gap in $[4,5]$ interval is:  0.25
or also in this way 0.25
```

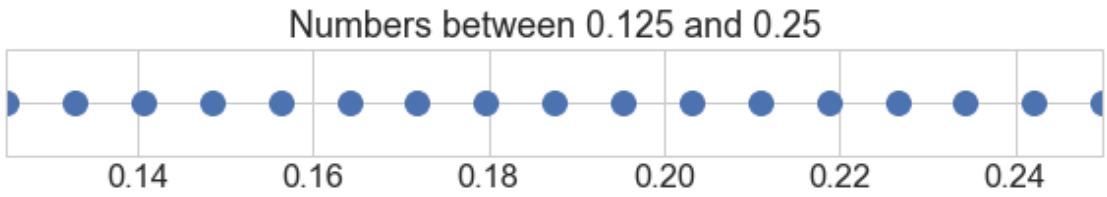
When using this system with n=4 and exponent range $[L,U] = [-5,5]$, what is the bound for the absolute error due to rounding? Think of the more general answer, and check how it applies in the 3 intervals illustrated above.

For example, suppose you want to represent the number `4.6`. Without doing any calculation to obtain the exact error due to rounding, can you give a bound for that error?

```
In [14]: #clear
         # The answer should be 0.25.
         # If they were to evaluate this, they would see they can only represent 4.5 and 4.75,
         # and confirm the absolute error is indeed less than 0.25
```

Use the function `number_scale` to plot FP numbers between powers of two: Try the intervals $[2^{-5},2^{-4}]$, $[2^{-3},2^{-2}]$ and $[2^{2},2^{3}]$

```
In [15]: #clear
         number_scale(fp_numbers,2**(-5),2**(-4))
         number_scale(fp_numbers,2**(-3),2**(-2))
         number_scale(fp_numbers,2**2,2**3)
```



It looks like the gap is the same... Don't be fooled! Remember that we are now plotting ranges that are different! Use the definition of the gap to determine the distance between two consecutive numbers in each one of the intervals above.

```
In [16]: #clear
         print("gap in $[2^{-5},2^{-4}]$ interval is: ", eps_machine*2**-5)

         print("gap in $[2^{-3},2^{-2}]$ interval is: ", eps_machine*2**-3)

         print("gap in $[2^{2},2^{3}]$ interval is: ", eps_machine*2**2)
```

```
gap in $[2^{-5},2^{-4}]$ interval is:  0.001953125
gap in $[2^{-3},2^{-2}]$ interval is:  0.0078125
gap in $[2^{2},2^{3}]$ interval is:  0.25
```

When using this system with n=4 and exponent range $[L, U] = [-5, 5]$, what is the bound for the **relative** error due to rounding?

Suppose you want to represent the number `5.1`. Determine the relative error (use rounding to the nearest). Does it satisfy the bound above?

```
In [17]: #clear
         # er = 0.1/5 = 0.02 <= 0.0625 YES!
```

You can test your finding that the relative error between two consecutive floating point numbers is smaller than machine epsilon.

Write a code snippet that loops over `fp_numbers` and calculates the relative errors between two consecutive numbers in the list. Check if the error is <= machine epsilon. You should be getting True values for the entire list.

```
In [18]: #clear
         success = []
         for i in range(len(fp_numbers)-1):
             success.append( (fp_numbers[i+1]-fp_numbers[i])/fp_numbers[i+1] <=eps_machine )
             if success[-1]:
                 continue
             else:
                 print("Ops!")
                 break
         print(success)
```

```
[True, True, True, True, True, True, True, True, True, True, True, True, True, True, True,
```

2 Ternary Floating Point System

The Setun Computer was a computer built in the Soviet Union in 1958 by Ukrainian computer scientist Nikolay Brusentsov. It used *ternary* logic rather than binary. In Brusentov's own words: "Computer science cannot limit itself to the universally-accepted binary system; the base should be ternary". He argued that ternary logic was a better representation of the way humans think; unfortunately for him, it never caught on.

2.0.1 Base 3 floating point system

This means floating-point arithmetic was done in base 3, not base 2, as is standard today. Rather than using "bits", the architecture of this computer dependend on "trits" that can take three values instead of two.

We'll will implement a standardized ternary floating point system, where any positive number x can be written as

$x = a_0.a_1a_2 \ldots a_{n-1} \times 3^m$.

The exponent $m \in [L, U]$ is essentially identical to the exponent in the binary case. However, there are some important differences between here and the normalized binary representation.

1. The base is now equal to 3, not 2. Hence the 3^m, instead of 2^m.
2. Each of the "trits" $a_1, a_2, \ldots, a_{n-1}$ can now take any of the 3 values in $\{0, 1, 2\}$.
3. In a normalized binary system, the leading bit is always equal to 1. But in normalized ternary system the leading "trit" a_0 can take either the value 1 or 2. It cannot be zero, because this is a **normalized** floating point system
4. Because a_0 can vary, the leading trit must be explicitly stored. This is not the case with normalized binary systems. That's also why we stopped at a_{n-1} instead of going to a_n. In binary, we get one extra bit of precision because we don't need to store the leading one.

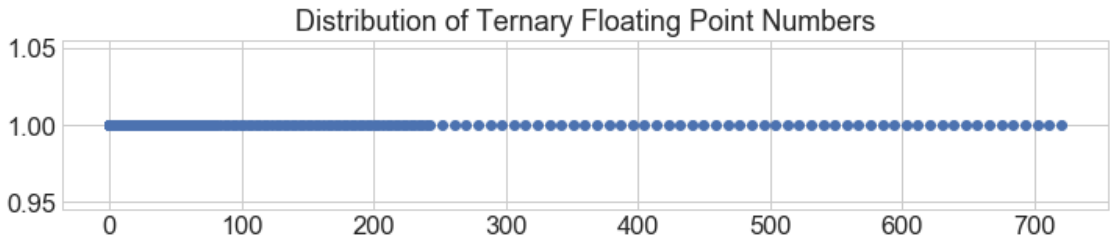
Just like for the binary system beforehand, write a loop that collects all floating point numbers in a given ternary floating point system. We'll use comparable parameters: i.e we use $n = 4$ trits, and have an exponent range of $m \in [-5, 5]$.

Your loop will be similar to the one you wrote above, but not exactly. Make sure you take into account the differences between the two systems!

```
In [19]: #clear
         base = 3
         n = 4
         exponent_min = -5
         exponent_max = 5

         fp_ternary = []
         for m in range(exponent_min,exponent_max + 1):
             for a0 in [1,2]:
                 for sigs in range(base**(n-1)):
                     s = a0 + (sigs/(base**(n-1)))
                     fp_ternary.append( s*base**(m) )
```

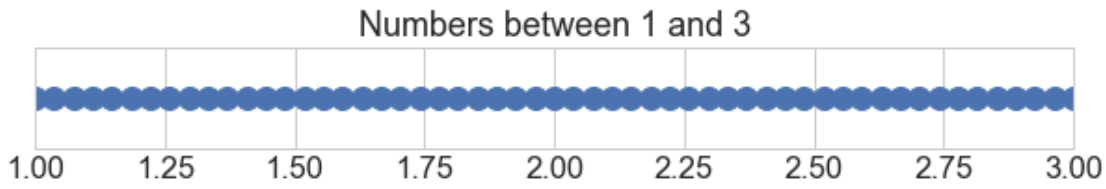
2.0.2 Density of Ternary Floating Point Numbers

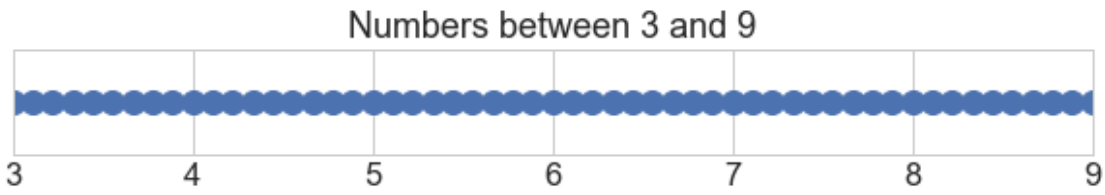
```
In [20]: plt.figure(figsize=(12,2))
         plt.plot(fp_ternary, np.ones_like(fp_ternary), "o");
         plt.title('Distribution of Ternary Floating Point Numbers')

Out[20]: Text(0.5, 1.0, 'Distribution of Ternary Floating Point Numbers')
```

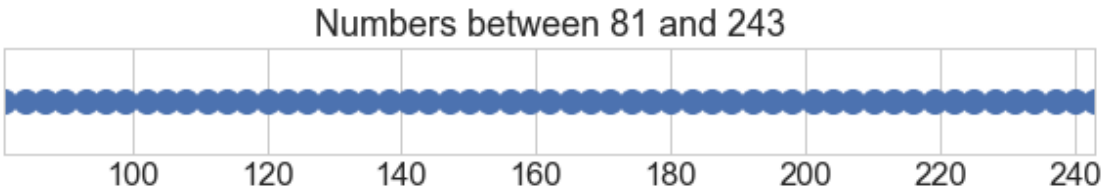


2.0.3 Plot different ranges of this system. How are the numbers spaced?

```
In [21]: number_scale(fp_ternary,3**0,3**1)
         number_scale(fp_ternary,3**1,3**2)
         number_scale(fp_ternary,3**4,3**5)
```







2.0.4 Compare to the normalized binary system

1. How many numbers does each floating point system have?
2. How do the overflow and underflow levels compare?
3. How do each system's machine epsilon compare?

```
In [22]: #clear
         print('Binary system has %d numbers' % len(fp_numbers))
         print('Ternary system has %d numbers' % len(fp_ternary))
         print()
         print('Binary Overflow: ', fp_numbers[-1])
         print('Ternary Overflow: ', fp_ternary[-1])
         print()
         print('Binary Underflow: ', fp_numbers[0])
         print('Ternary Underflow: ',fp_ternary[0])

         ind_b = fp_numbers.index(1.0)
         ind_t = fp_ternary.index(1.0)
         eps_b = fp_numbers[ind_b+1] - 1
         eps_t = fp_ternary[ind_t+1] - 1
         print()
         print('Binary Machine Epsilon: ', eps_b)
         print('Ternary Machine Epsilon: ',eps_t)
```

```
Binary system has 176 numbers
Ternary system has 594 numbers

Binary Overflow:  62.0
Ternary Overflow:  720.0
```

```
Binary Underflow:  0.03125
Ternary Underflow:  0.00411522633744856

Binary Machine Epsilon:  0.0625
Ternary Machine Epsilon:  0.03703703703703698
```

2.0.5 Machine Epsilon for IEEE

Compare it with machine epsilon in standard IEEE double precision arithmetic (what Python uses). We don't need to make a list of all floating point numbers in IEEE double precision to do this however.

Another way of thinking about machine epsilon is that it is the smallest positive number ϵ such that

$1 + \epsilon \neq 1$.

Use this relationship to write a loop to find machine epsilon for the IEEE double precision floating point system

```
In [23]: # come up with a while loop and an appropriate testing statement
         my_statement = False
         while (my_statement):
             ...

In [24]: #clear
         a = 1
         while 1+a > 1:
             a = a/2
         print(a*2)
```

```
2.220446049250313e-16
```

Numpy can actually tell you directly:

```
In [25]: import numpy as np
         np.finfo(float).eps
```

```
Out[25]: 2.220446049250313e-16
```