Numbers

When numbers get serious They leave a mark on your door

– Paul Simon

One of the themes of this course will be shifting between algebraic and computations views of various concepts.

Today we need to talk about why the answers we get from computers can be different from the answers we get mathematically – for the same problem!

A number is a mathematical concept – an abstract idea.

God made the integers, all else is the work of man.

– Leopold Kronecker (1823 - 1891)

In a computer we assign **bit patterns** to correspond to certain numbers.

We say the bit pattern is the number's *representation.*

For example the number '3.14' might have the representation '01000000010010001111010111000011'.

For reasons of efficiency, we use a fixed number of bits for these representations. In most computers nowadays we use **64 bits** to represent a number.

Integers

For the most part, using integers is not complicated.

Integer representation is essentially the same as binary numerals.

For example, in a 64-bit computer, the representation of the concept of 'seven' would be '0..0111' (with 61 zeros in the front).

There is a size limit on the largest value that can be stored as an integer, but it's so big we don't need to concern ourselves with it in this course.

So for our purposes, an integer can be stored exactly.

In other words, there is an 1-1 correspondence between every representation and the corresponding integer.

So, what happens when we compute with integers?

For (reasonably sized) integers, computation is **exact** .... as long as it only involves **addition, subtraction, and multiplication.**

In other words, there are no errors introduced when adding, subtracting or multiplying integers.

However, it is a different story when we come to division, because the integers are not closed under division.

For example, 2/3 is not an integer. ... It is, however, a **real** number.

Real Numbers and Floating-Point Representations

Representing a real number in a computer is a **much** more complicated matter.

In fact, for many decades after electronic computers were developed, there was no accepted "best" way to do this!

Eventually (in the 1980s) a widely accepted standard emerged, called IEEE-754. This is what almost all computers now use.

The style of representation used is called **floating point.**

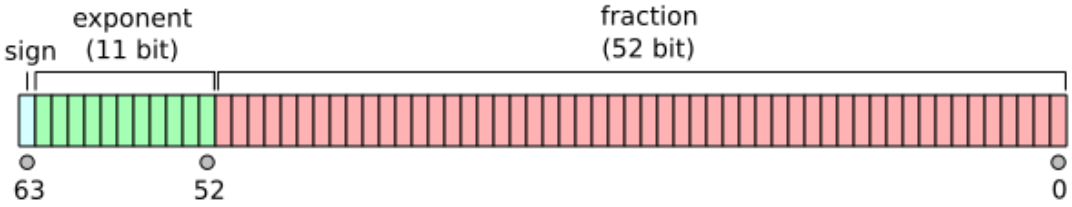
Conceptually, it is similar to "scientific notation."

$$123456 = \underbrace{1.23456}_{\text{significand}} \times \underbrace{10}_{\text{base}} \overbrace{5}^{\text{exponent}}$$

Except that it is encoded in binary:

$$17 = \underbrace{1.0001}_{significand} \times \underbrace{2}_{base}^{\overbrace{4}^{exponent}}$$

The sign, significand, and exponent are all contained within the 64 bits.



By Codekaizen (Own work) [GFDL or CC BY-SA 4.0-3.0-2.5-2.0-1.0], via Wikimedia Commons

Because only a fixed number of bits are used, **most real numbers cannot be represented exactly in a computer.**

Another way of saying this is that, usually, a floating point number is an approximation of some particular real number.

Generally when we try to store a real number in a computer, **what we wind up storing is the closest floating point number that the computer can represent.**

The Relative Error of a Real Number stored in a Computer

The way to think about working with floating point (in fact, how the hardware actually does it) is:

1. Represent each input as the **nearest** representable floating point number.
2. Compute the result exactly from the floating point representations.
3. Return the **nearest** representable floating point number to the result.

What does "**nearest**" mean? Long story short, it means "round to the nearest representable value."

Let's say we have a particular real number r and we represent it as a floating point value f.

Then $r = f + \epsilon$ where ϵ is the amount that r was rounded when represented as f.

How big can ϵ be? Let's say f is

$$f = \underbrace{1.010...01}_{53\ bits} \times 2^n$$

Then $|\epsilon|$ must be smaller than

$$|\epsilon| < \underbrace{0.000...01}_{53\ bits} \times 2^n$$

.

So as a *relative error*,

$$\text{relative error} = \frac{|\epsilon|}{f} < \frac{0.000...01 \times 2^n}{\underbrace{1.000...00}_{53\ bits} \times 2^n} = 2^{-52} \approx 10^{-16}$$

This value 10^{-16} is an important one to remember.

It is approximately **the relative error that can be introduced any time a real number is stored in a computer.**

Another way of thinking about this is that you **only have about 16 digits of accuracy** in a floating point number.

Implications of Representation Error

Problems arise when we work with floating point numbers and confuse them with real numbers, thereby forgetting that most of the time we are not storing the real number exactly, but only a floating point number that is close to it.

Let's look at some examples. First:

```
[3]: # ((1/8)*8)-1
     a = 1/8
     b = 8
     c = 1
     (a*b)-c
```

```
0.0
```

It turns out that 1/8, 8, and 1 can all be stored exactly in IEEE-754 floating point format.

So, we are * storing the inputs exactly (1/8, 8 and 1) * computing the results exactly (by definition of IEEE-754), yielding $(1/8) * 8 = 1$ * and representing the result exactly (zero)

OK, here is another example:

```
[4]: # ((1/7)*7)-1
     a = 1/7
     b = 7
     c = 1
     a * b - c
```

```
0.0
```

Here the situation is different.

1/7 can **not** be stored exactly in IEEE-754 floating point format.

In binary, 1/7 is $0.001\overline{001}$, an infinitely repeating pattern that clearly cannot be represented in a finite sequence of bits.

Nonetheless, the computation $(1/7) * 7$ still yields exactly 1.0.

Why? Because the rounding of $0.001\overline{001}$ to its closest floating point representation, when multiplied by 7, yields a value whose closest floating point representation is 1.0.

Now, let's do something that seems very similar:

```
[5]: # ((1/70)*7)-0.1
     a = 1/70
     b = 7
     c = 0.1
     a * b - c
```

```
-1.3877787807814457e-17
```

In this case, both 1/70 and 0.1 **cannot** be stored exactly.

More importantly, the process of rounding 1/70 to its closest floating point representation, then multiplying by 7, yields a number whose closest floating point representation is **not** 0.1

However, that floating point representation is very **close** to 0.1.

Let's look at the difference: -1.3877787807814457e-17.

This is about $-1 \cdot 10^{-17}$.

In other words, -0.0000000000000001

Compared to 0.1, this is a very small number. The relative error abs(-0.0000000000000001 / 0.1) is about 10^{-16}.

This suggests that when a floating point calculation is not exact, the error (in a relative sense) is usually very small.

Notice also that in our example the size of the relative error is about 10^{-16}.

Recall that the significand in IEEE-754 uses 52 bits and that $2^{-52} \approx 10^{-16}$.

There's our "sixteen digits of accuracy" principle again.

Special Values

There are three kinds of special values defined by IEEE-754: 1. NaN, which means "Not a Number" 2. Infinity – both positive and negative 3. Zero – both positive and negative.

NaN and **Inf** behave about as you'd expect. If you get one of these values in a computation you should be able to reason about how it happened. Note that these are values, and can be assigned to variables.

```
[6]: np.sqrt(-1)
```

```
<ipython-input-6-597592b72a04>:1: RuntimeWarning: invalid value encountered in sqrt
  np.sqrt(-1)
```

```
nan
```

```
[7]: var = np.log(0)
     var
```

```
<ipython-input-7-49e412a30c50>:1: RuntimeWarning: divide by zero encountered in log
  var = np.log(0)
```

```
-inf
```

```
[8]: 1/var
```

```
-0.0
```

As far as we are concerned, there is no difference between positive and negative zero. You can ignore the minus sign in front of a negative zero.

```
[9]: var = np.nan
     var + 7
```

```
nan
```

```
[10]: var = np.inf
      var + 7
```

```
inf
```

Mathematical Computation vs. *Mechanical* Computation

In a mathematical theorem, working with (idealized) numbers, it is always true that:
If $c = 1/a$, then $abc = b$.
In other words, $(ab)/a = b$.
Let's test whether this is always true in actual computation.

```
[11]: a = 7
      b = 1/10
      c = 1/a
      a*c*b
```

```
0.1
```

```
[12]: b*c*a
```

```
0.09999999999999999
```

```
[13]: a*c*b == b*c*a
```

```
False
```

Here is another example:

```
[14]: 0.1 + 0.1 + 0.1
```

```
0.30000000000000004
```

```
[15]: 3 * (0.1) - 0.3
```

```
5.551115123125783e-17
```

What does all this mean for us in practice?
I will now give you three principles to keep in mind when computing with floating point numbers.

Principle 1: Do not compare floating point numbers for equality

Two floating point computations that *should* yield the same result mathematically, may not do so due to rounding error.

However, in general, if two numbers should be equal, the relative error of the difference in the floating point should be small.

So, instead of asking whether two floating numbers are equal, we should ask whether the relative error of their difference is small.

```
[16]: r1 = a * b * c
      r2 = b * c * a
      np.abs(r1-r2)/r1
```

```
1.3877787807814457e-16
```

```
[17]: np.finfo('float')
```

```
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=float64)
```

```
[18]: print(r1 == r2)
```

```
False
```

```
[19]: print(np.abs(r1 -  r2)/np.max([r1, r2]) < np.finfo('float').resolution)
```

```
True
```

Next, we will generalize this idea a bit:
beyond the fact that numbers that should be equal, may not be in practice,
we can also observe that it can be hard to be accurate about the **difference** between two numbers that are **nearly** equal. This leads to the next two principles.

Principle 2: Beware of ill-conditioned problems

An **ill-conditioned** problem is one in which the outputs depend in a very sensitive manner on the inputs.
That is, a small change in the inputs can yield a very large change in the outputs.
The simplest example is computing $1/(a-b)$.

```
[20]: print(f'r1 is {r1}')
      print(f'r2 is very close to r1')
      r3 = r1 + 0.0001
      print(f'r3 is 0.1001')
      print(f'1/(r1 - r2) = {1/(r1 - r2)}')
      print(f'1/(r3 - r2) = {1/(r3 - r2)}')
```

```
r1 is 0.1
r2 is very close to r1
r3 is 0.1001
1/(r1 - r2) = 7.205759403792794e+16
1/(r3 - r2) = 9999.999999998327
```

If a is close to b, small changes in either make a big difference in the output.
Because the inputs to your problem may not be exact, if the problem is ill-conditioned, the outputs may be wrong by a large amount.
Later on we will see that the notion of ill-conditioning applies to matrix problems too, and in particular comes up when we solve certain problems involving matrices.

Principle 3: Relative error can be magnified during subtractions

Two numbers, each with small relative error, can yield a value with large relative error if subtracted.
Let's say we represent a = 1.2345 as 1.2345002 – the relative error is 0.0000002.
Let's say we represent b = 1.234 as 1.2340001 – the relative error is 0.0000001.
Now, subtract a - b: the result is .0005001.
What is the relative error? 0.005001 - 0.005 / 0.005 = 0.0002
The relative error of the result is 1000 times larger than the relative error of the inputs.
Here's an example in practice:

```
[21]: a = 1.23456789
      b = 1.2345678
      print(0.00000009)
      print(a-b)
      print(np.abs(a-b-0.00000009)/ 0.00000009)
```

```
9e-08
8.999999989711682e-08
1.1431464011915431e-09
```

We know the relative error in the inputs is on the order of 10^{-16}, but the relative error of the output is on the order of 10^{-9} – i.e., a million times larger.

A good summary that covers additional issues is at https://docs.python.org/2/tutorial/floatingpoint.html.