WHAT EVERY PHYSICIST SHOULD KNOW ABOUT FLOATING-POINT ARITHMETIC

FALL 2023

https://www.phys.uconn.edu/~rozman/Courses/P2200_23F/

Last modified: September 20, 2023

1 Why don't my numbers add up?

So you've written some simple test code, say for example:

```
0.1 + 0.2 == 0.3
```

and got a really unexpected result:

```
false
```

Or, you decided to raise an integer, say 10, into an integer power:

```
10^19
```

and got something very wrong:

```
-8446744073709551616
```

This document is here to:

- Explain why you get that unexpected result
- Tell you how to deal with this problem

2 Number formats

2.1 Binary Fractions

As a programmer, you should be familiar with the concept of binary integers, i.e. the representation of integer numbers as a series of bits:

$$123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0,$$

$$456_8 = 4 \cdot 8^2 + 5 \cdot 8^1 + 6 \cdot 8^0 = 302_{10}$$

$$\begin{aligned} 1001001_2 &= 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 64_{10} + 8_{10} + 1 = 73_{10} \end{aligned}$$

This is how computers store integer numbers internally. And for fractional numbers, they do the same thing:

$$0.123_{10} = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 3 \cdot 10^{-3}$$

$$\begin{aligned} 0.100101 &= 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6} \\ &= \frac{1}{2} + \frac{1}{16} + \frac{1}{64} = \frac{37}{64} = .578125_{10} \end{aligned}$$

While they work the same in principle, binary fractions are different from decimal fractions in what numbers they can accurately represent with a given number of digits, and thus also in what numbers result in rounding errors. Specifically, binary can only represent those numbers as a finite fraction where the denominator is a power of 2. Unfortunately, this does not include most of the numbers that can be represented as finite fraction in base 10, like 0.1.

Fraction	Base	Positional Notation	Rounded to 4 digits
1/10	10	0.1	0.1
1/3	10	0.3333...	0.3333
1/2	2	0.1	0.1
1/10	2	0.00011...	0.0001

And this is how you already get a rounding error when you just *write down* a number like 0.1 and run it through your interpreter or compiler. It's not as big as 3/80 and may be invisible because computers cut off after 23 or 52 binary digits rather than 4. But the error is there and *will* cause problems eventually if you just ignore it.

2.2 Why use Binary?

At the lowest level, computers are based on billions of electrical elements that have only two states, (usually low and high voltage). By interpreting these as 0 and 1, it's very easy to build circuits for storing binary numbers and doing calculations with them.

While it's possible to simulate the behavior of decimal numbers with binary circuits as well, it's less efficient. If computers used decimal numbers internally, they'd have less memory and be slower at the same level of technology.

Since the difference in behavior between binary and decimal numbers is not important for most applications, the logical choice is to build computers based on binary numbers and live with the fact that some extra care and effort are necessary for applications that require decimal-like behavior.

3 Floating Point Numbers

3.1 Why floating-point numbers are needed

Since computer memory is limited, you cannot store numbers with infinite precision, no matter whether you use binary fractions or decimal ones: at some point you have to cut off. But how much accuracy is needed? And *where* is it needed? How many integer digits and how many fraction digits?

- To an engineer building a highway, it does not matter whether it's 10 meters or 10.0001 meters wide - his measurements are probably not that accurate in the first place.
- To someone designing a microchip, 0.0001 meters (a tenth of a millimeter) is a *huge* difference - But he'll never have to deal with a distance larger than 0.1 meters.
- A physicist needs to use the speed of light (about 300000000 in SI units) and Newton's gravitational constant (about 0.0000000000667 in SI units) together in the same calculation.

To satisfy the engineer and the chip designer, a number format has to provide accuracy for numbers at very different magnitudes. However, only *relative* accuracy is needed. To satisfy the physicist, it must be possible to do calculations that involve numbers with different magnitudes.

Basically, having a fixed number of integer and fractional digits is not useful - and the solution is a format with a *floating point*.

3.2 How floating-point numbers work

The idea is to compose a number of two main parts:

- A **significand** that contains the number's digits. Negative significands represent negative numbers.
- An **exponent** that says where the decimal (or binary) point is placed relative to the beginning of the significand. Negative exponents represent numbers that are very small (i.e. close to zero).

Such a format satisfies all the requirements:

- It can represent numbers at wildly different magnitudes (limited by the length of the exponent)
- It provides the same relative accuracy at all magnitudes (limited by the length of the significand)
- allows calculations across magnitudes: multiplying a very large and a very small number preserves the accuracy of both in the result.

Decimal floating-point numbers usually take the form of scientific notation with an explicit point always between the 1st and 2nd digits. The exponent is either written explicitly including the base, or an **e** is used to separate it from the significand.

Significand	Exponent	Scientific notation	Fixed-point value
1.5	4	$1.5\ 10^4$	15000
-2.001	2	$-2.001\ 10^2$	-200.1
5	-3	$5\ 10^{-3}$	0.005
6.667	-11	6.667e-11	0.00000000006667

3.3 The standard

Nearly all hardware and programming languages use floating-point numbers in the same binary formats, which are defined in the IEEE 754 standard. The usual formats are 32 or 64 bits in total length:

	Single precision	Double precision
Total bits	32	64
Sign bits	1	1

	Single precision	Double precision
Significand bits	23	52
Exponent bits	8	11
Smallest number	$2^{-126} \approx 1.2 \times 10^{-38}$	$2^{-1022} \approx 2.2 \times 10^{-308}$
Largest number	ca. $2 \times 2^{127} \approx 3.4 \times 10^{38}$	ca. $2 \times 2^{1023} \approx 1.8 \times 10^{308}$

Note that there are some peculiarities:

- The **actual bit sequence** is the sign bit first, followed by the exponent and finally the significand bits.
- The exponent does not have a sign; instead an **exponent bias** is subtracted from it (127 for single and 1023 for double precision). This, and the bit sequence, allows floating-point numbers to be compared and sorted correctly even when interpreting them as integers.
- The significand's most significant bit is assumed to be 1 and omitted, except for special cases.
- There are separate **positive and a negative zero** values, differing in the sign bit, where all other bits are 0. These must be considered equal even though their bit patterns are different.
- There are special **positive and negative infinity** values, where the exponent is all 1-bits and the significand is all 0-bits. These are the results of calculations where the positive range of the exponent is exceeded, or division of a regular number by zero.
- There are special **not a number** (or NaN) values where the exponent is all 1-bits and the significand is *not* all 0-bits. These represent the result of various undefined calculations (like multiplying 0 and infinity, any calculation involving a NaN value, or application-specific cases). Even bit-identical NaN values must *not* be considered equal.

4 Errors

4.1 Rounding Errors

Because floating-point numbers have a limited number of digits, they cannot represent all real numbers accurately: when there are more digits than the format allows, the leftover ones are omitted - the number is *rounded*.

4.2 Comparing floating-point numbers

Due to rounding errors, most floating-point numbers end up being slightly imprecise. As long as this imprecision stays small, it can usually be ignored. However, it also means that numbers expected to be equal (e.g. when calculating the same result through different correct methods) often differ slightly, and a simple equality test fails. For example:

```
a = 0.15 + 0.15;
b = 0.1 + 0.2;
a == b # should be false!
isapprox(a, b) # should be true
```

4.3 Error Propagation

While the errors in single floating-point numbers are very small, even simple calculations on them can contain pitfalls that increase the error in the result way beyond just having the individual errors "add up".

In general:

- Multiplication and division are "safe" operations
- Addition and subtraction are dangerous, because when numbers of different magnitudes are involved, digits of the smaller-magnitude number are lost.
- This loss of digits can be inevitable and benign (when the lost digits also insignificant for the final result) or catastrophic (when the loss is magnified and distorts the result strongly).
- A method of calculation can be *stable* (meaning that it tends to reduce rounding errors) or *unstable* (meaning that rounding errors are magnified). Very often, there are both stable and unstable solutions for a problem.