Math 405: 08: Floating point arithmetic

Book: Overton, "Numerical Computing with IEEE Floating Point Arithmetic".

Book: Trefethen & Bau, "Numerical Linear Algebra", Chapter 13.

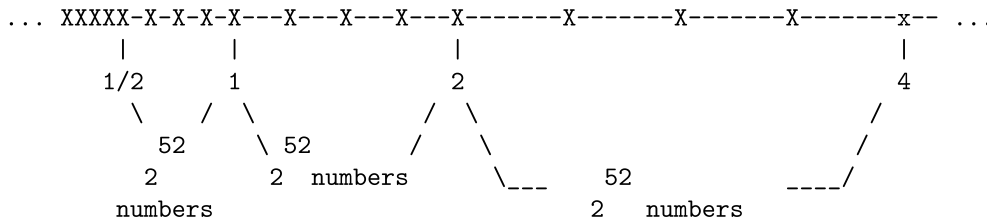
Other articles: Goldberg, "What every computer scientist should know about floating point arithmetic".

(Physics analogy based on a lecture of Nick Trefethen).

Representation of real numbers

Computers represent the real line by a discrete approximation. The numbers are evenly spaced in $[1, 2)$, where we find exactly $2^{52} \approx 4.5e15$ "floating point numbers".

Everything is scale-invariant, so we find the same picture, rescaled, in $[1/2, 1)$ or in $[8, 16)$, etc.:



This picture persists far out to the *underflow* and *overflow* limits, around $2^{-1024} \approx$ 1e-308 and $2^{1024} \approx$ 1e308.

Also have representations for ∞, $-\infty$ and "NaN" (not-a-number) for exceptions or ambiguities (e.g., "0/0").

Analogy to physics

The laws of classical physics describe the motion and deformation of fluids and solids. They involve quantities such as density, pressure, and temperature, and they are typically written as PDEs.

But this is an approximation: world is made of discrete atoms and molecules. The "continuous" quantities (density, pressure) are averages. Yet the continuous level works for most applications in science and engineering. We ignore the atoms and regard the world as continuous.

Molecules and their interactions are "implementation details" of the laws of physics.

How fine is the physical continuum?

Avogadro's number: 6e23 molecules in a mole of a substance. There are 50 moles of gas in a cubic meter at ordinary conditions, so 3e25 molecules per cubic meter. Cube root of 3e25 is 3e8. So 3e8 moles per meter in an ordinary gas. In a solid, about 10 times higher 3e9.

Summing up roughly: A gas or solid has around 10ˆ9 particles per meter.

Thus in some sense: *Computer arithmetic is a million times finer than physics.*

If we gave *floating point* arithmetic coordinates to this room, we'd find there were a million or more coordinate points between each adjacent pair of molecules.

IEEE standard

IEEE standard: specification to replace each hardware manufacturer doing their own thing. Driving force was Vel Vel Kahan, 1985. Controversial at the time, but now well- and widely-regarded. Almost all computer hardware uses this (GPUs were a temporary(?) exception).

1 floating point number = 64 bits or 8 bytes

1 bit for sign 11 bits for exponent (base 2) 53 bits for fraction (52 stored, 1 implicit)

Distance between 1 and the next floating point number: $2^{-52} \approx 2.2 \times 10^{-16}$: "machine epsilon".

Details are in Overton's book.

Computing

That was just representation of numbers. Need to compute with them.

Principle: if x and y are floating point numbers and $x + y$, $x - y$, x/y and xy is computed, then the result is the exact answer, correctly rounded to the floating point representation.

Corollary: Each operation gives:

```
computed(x <op> y) = (x <op> y)*(1+eps)
```

for some with $|\epsilon| \leq \epsilon_{mach}/2 = 2^{-53} \approx 1.1 \times 10^{-16}$. Note this is half the distance between 1 and the next floating point number. ($\epsilon_{mach}/2$ because "correctly rounded").

Note: a series of multiple calculations is probably not giving the exact solution, rounded to the floating point representation. But for the basic arithmetic operators, IEEE requires this of the hardware (see famous "Pentium bug").

Big picture At every step of your algorithm, a small *rounding error* introduced at 16th (floating point) decimal digit. Assuming we avoid overflow, underflow (not hard to do in practice), that's really all we need to know about the underlying system.

Forward Error and Backward Error

Consider $f(y)$, a mathematical *problem* f which depends on *data* y.

We have an *algorithm*/implementation $\hat{f}(y)$.

Forward error: $f(y) - \hat{f}(y)$

Backward error: $\hat{y} = y + \delta y$ such that $f(\hat{y}) = \hat{f}(y)$.

[Diagram!]

Backward stable algorithm

Definition: A *backward stable algorithm* gives the exact solution of a nearby problem.

Used most often in analysis of implementations of special functions (library/hardware that implements sin, cos, besselj, etc). Also in numerical linear algebra.

For ODEs/PDEs we are usually concerned with errors larger than ϵ_{mach} (e.g., LTE) and its growth or decay (which leads to the "higher-level" concepts of stability we discussed earlier such as A-stability). But important to keep in mind that even if everything else is perfect, the steps will still be perturbed by 10^{-16}.