Lecture 2. Floating point representation.

Last class, we saw the effects of computing the solution to

$$0.1036\,x + 0.2122\,y = 0.7381 \quad (1)$$

$$0.2081\,x + 0.4247\,y = 0.9327 \quad (2)$$

in n decimal place arithmetic. Roughly, the problem was that after subtracting a multiple of (1) from (2) to elimate x, the coefficient of y was quite small:

$$-\frac{1599}{1{,}036{,}000} = 0.00154343629\ldots$$

In n-place arithmetic, we only preserved $(n-3)$ significant digits of this constant.

Computer systems face a similar problem: one has n bits available to represent a number, but it seems wasteful to spend a lot of bits storing zeros if the number is small.

Idea: Use scientific notation.

Definition. The <u>normalized floating point representation</u> of a real x is

$$x = \pm 0.d_1 d_2 d_3 \ldots \times 10^n$$

where $d_1 \neq 0$ and $n \in \mathbb{Z}$. We also write this

$$x = \pm r \times 10^n \quad (1/10 \leq r < 1)$$

where r is the <u>normalized mantissa</u> and n is the exponent.

Now we can also have floating point numbers which are <u>not</u> normalized.

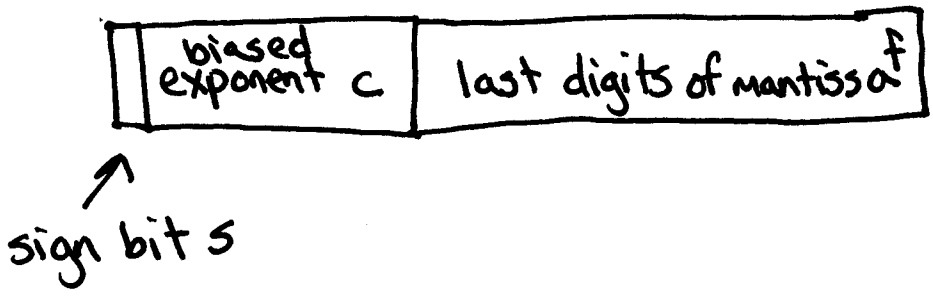
(machine_numbers.nb)

We have now seen some ordinary and normalized floating point numbers. This is not quite what happens in a computer, for obvious reasons (what about 0? ∞?).

Definition. An overflow (underflow) error is produced when a number too large (small) for the set of machine numbers is produced in a computation.

Most (all?) actual computers use the "IEEE 754" standard for floating point numbers.

single precision or double precision, we have



so that the number is

$$(-1)^s \times 2^{c-\text{Off}} \times (1.f)_2 \leftarrow \text{in binary}$$

and "off" is an offset depending on the number of digits, used for c. ~~For "single precision", the~~

single precision = 1 + 8 + 23 = 32 bits off = 127

double precision = 1 + 11 + 52 = 64 bits off = 1023

"long double" precision = 1 + 15 + 112 = 128 bits off = 16383

Note: long double means different things to different companies and is <u>not</u> guaranteed to be this good.

There are some (nonobvious) conventions:

all 0's (except possibly the sign bit) represents the value 0 (and not $1 \times 2^{-\text{off}}$).

all 1's in the exponent is reserved for special cases.

So we have

$$0 \leq c \leq 2^{\#\text{ of exponent bits}} - 1$$

$$< (11\cdots1)_2$$

(# of exponent bits)

and the actual exponent is between

$$-\text{off} < c - \text{off} < (11\cdots1)_2 - \text{off}$$

or

$$-126 < c - \text{off} \leq 127 \quad \text{(single)}$$

$$-1022 < c - \text{off} \leq 1023 \quad \text{(double)}$$

$$-16382 < c - \text{off} \leq 16383 \quad \text{(long double)}$$

Our conventions for the mantissa mean it's between

$$1 \le (1.f)_2 \le 2 - 2^{-\#\text{ of bits in mantissa}}$$

$$= (1.\underbrace{11\ldots1}_{\#\text{ of bits in mantissa}})_2$$

in our various formats that means that

$$(1.\underbrace{00\ldots01}_{\#\text{ of bits in mantissa}})_2 = \epsilon$$

is the smallest number different from 1.0.

This number is called the machine epsilon for the given format. In our formats

$\epsilon = 2^{-23}$	1.2×10^{-7}	6 digits	(single)
2^{-52}	2.2×10^{-16}	15 digits	(double)
2^{-112}	1.92×10^{-34}	33 digits	(long double)

The largest and smallest numbers in these formats are

$$\left(2-2^{-\#\text{ mantissa bits}}\right)2^{\text{max exponent}}$$

and

$$2^{\text{min exponent}}$$

or

$$\left(2-2^{-23}\right)2^{127} \approx 2^{128} \approx 3.4\times10^{38} \quad \text{(single)}$$

$$2^{-126} \approx 1.2\times10^{-38}$$

$$\left(2-2^{-52}\right)2^{1023} \approx 2^{1024} \approx 1.8\times10^{308}$$

$$2^{-1022} \approx 2.2\times10^{-308} \quad \text{(double)}$$

$$\left(2-2^{-112}\right)2^{16383} \approx 2^{16384} \approx 1.19\times10^{4932}$$

$$2^{-16382} \approx 3.36\times10^{-4932} \quad \text{(long double)}$$

We can now try (on a computer) converting various decimals to these formats.

How much error is invdved in converting to floating point form?

When the ~~normalized~~ binary representation of the mantissa of x has > # mantissa bits, we must drop some bits to represent x as a machine number.

Example.

$1/10$ is the repeating (binary) ~~fraction~~ decimal

$= (0.0\,0011\,0011\,0011\,0011\ldots)_2$

so it cannot be represented perfectly in <u>any</u> floating point binary format

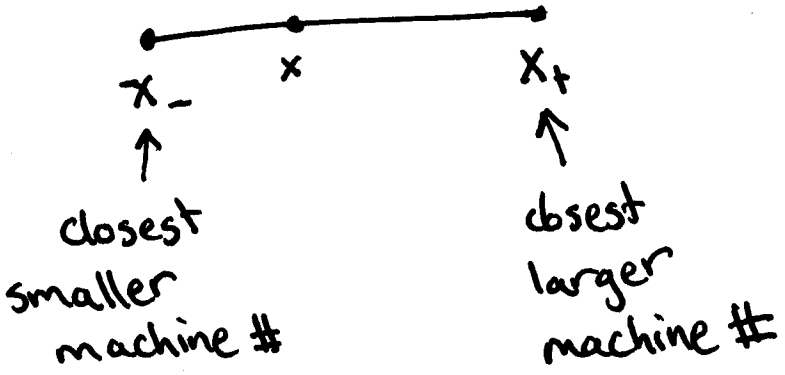
Checking the calculator for

1.1, we see 1.0999999 (single)

~~[illegible]~~ (double)

1.100 000 000 000 000 888·2

What is the maximum size of this roundoff error? We have diagram



~~Now we know that if $x = q \times 2^m$, then if $x_- = q_- \times 2^m$, $x_+ = q_+ \times 2^m$, we have that q_- differs from q in the (# mantissa bits) + 1 place and~~

Now we obtain x_- and x_+ by rounding down or up. These numbers differ by at most $2^{-(\#\text{ mantissa bits})} \times 2^{\text{exponent of } x}$.

so the max absolute roundoff error is half of this or

$$2^{\text{exponent of } x \,-\, \#\text{mantissa bits} \,-\, 1}$$

the max relative error is bounded by

$$2^{-(\#\text{mantissa bits} + 1)}$$

Example. The Patriot missile defense system has a clock which measures time in 0.1 sec intervals. The time is stored as a 24 bit floating point number, with roundoff error

$$(1.1001100\ldots)_2 \times 2^{-24} \approx 0.95 \times 10^{-7}$$

The system fails to track an incoming missile and intercept it if the accumulated error is > 0.00687 seconds. (translates to 11.51 meters at SCUD cruising speed of Mach 5, 3750 mph. A SCUD is 11 m long.)

On February 25, 1991 a Patriot system in Dharan, Saudia Arabia operating for > 100 hours failed to intercept a SCUD. The missile killed 28 americans when it hit an Army barracks.

Was roundoff error the cause?
(According to the GAO, yes.)

Example 2. Suppose we have a 5 digit decimal machine and add

$$0.37218 \times 10^4 + 0.71422 \times 10^{-1}$$

in a double-length (or single length) work area. Find roundoff errors.

$$\begin{array}{r} 0.37218\ 00000 \times 10^4 \\ 0.00000\ 71422 \times 10^4 \\ \hline 0.37218\ 71422 \times 10^4 \end{array}$$

This is rounded to 0.37219×10^4 with

error given by $0.00000\,28578 \times 10^4$.

The relative error is

$$\frac{0.00000\,28578 \times 10^4}{0.37218\,71422 \times 10^4} \approx 0.77 \times 10^{-5}$$

In a single-length accumulator, we get

$$\begin{array}{r} 0.37218 \times 10^4 \\ 0.00000 \times 10^4 \\ \hline 0.37218 \times 10^4 \end{array}$$

and relative error

$$\frac{0.71422 \times 10^{4-1}}{0.37218\,71422 \times 10^4} \approx 0.19 \times 10^{-4}$$

(which is dreadful!).

In general,

$fl(x)$ = closest floating point # to x

for a 32 bit single precision floating format,

$$\frac{|x - fl(x)|}{|x|} \leq u = 2^{-24}$$

or if

$$\delta = \frac{fl(x) - x}{x},$$

$$x\delta = fl(x) - x$$

$$x(1+\delta) = fl(x) \quad \text{and} \quad |\delta| < 2^{-24}.$$

~~In this case, we see that if~~

~~$fl(1+\epsilon) = 1,$~~

~~then~~

~~$(1+\epsilon)(1+\delta) = fl(1+\epsilon) = 1.$~~

~~or~~

~~$1 + \epsilon\delta + \epsilon + \delta = 1$~~

~~or~~

~~$\epsilon\delta + \epsilon + \delta = 0.$~~

Computers are designed so that if $\odot$ is $+, -, \times$, or $\div$, then for any machine numbers x and y,

$$fl(x \odot y)$$

is produced (instead of $x \odot y$). So

$$fl(x+y) = (x+y)(1+\delta)$$

$$= x(1+\delta) + y(1+\delta).$$

This means that the approximate answer to the calculation $x+y$ is the exact answer to a slightly different question. This is called backwards error analysis.

Example. What is the max roundoff error in computing $z(x+y)$ if x, y, z are machine numbers?

We really compute

$$fl[z\, fl(x+y)] = z\, fl(x+y)(1+d_2)$$

$$= z(x+y)(1+d_1)(1+d_2)$$

$$\approx z(x+y)(1+d_1+d_2+d_1d_2)$$

$$\approx z(x+y)(1+d_1+d_2).$$

Since $|d_1|, |d_2| < 2^{-24}$, $|d_1+d_2| < 2^{-23}$, and that's our relative error bound.

Example. Criticize the following attempt to calculate roundoff error in the calculation

$$z = x + y.$$

in single precision arithmetic.

$$z = fl(fl(x) + fl(y))$$

$$= (x(1+\delta) + y(1+\delta))(1+\delta)$$

$$= (x+y)(1+\delta)^2$$

$$\approx (x+y)(1+2\delta).$$

So relative error is

$$\left| \frac{(x+y) - z}{(x+y)} \right| = \left| \frac{2\delta(x+y)}{(x+y)} \right| = |2\delta| \leq 2^{-23}.$$

The problem is that while we're used to writing

$$fl(x) = x(1+\delta)$$

this doesn't mean that if

$$fl(x) = x(1+\delta)$$

and

$$fl(y) = y(1+\delta)$$

that these are the same δ! This is a common mistake, and it can lead to dangerous cancellations. In the example above, we should have written

$$\begin{aligned} z &= fl(fl(x) + fl(y)) \\ &= (x(1+\delta_1) + y(1+\delta_2))(1+\delta_3) \\ &= (x+y+\delta_1 x + \delta_2 y)(1+\delta_3) \\ &= x + y + (\delta_1+\delta_3)x + (\delta_2+\delta_3)y + \text{terms with a product } \delta_i\delta_j \end{aligned}$$

so roundoff error should have been

$$\left|\frac{(x+y)-z}{(x+y)}\right| = \left|\frac{x(\delta_1+\delta_3)+y(\delta_2+\delta_3)}{x+y}\right|$$

$$= \left|\frac{(x+y)\delta_3 + \delta_1 x + \delta_2 y}{x+y}\right|$$

$$= \left|\delta_3 + \frac{\delta_1 x + \delta_2 y}{x+y}\right|.$$

This second term could be very large!
For example, if x and y are large, but $x+y$ is very small, the second term could be ~~as big as you want~~ close to 1.

For example, x is a machine number, so $\delta_1 = 0$, $y \approx$ [redacted] ≈ 1. ~~And $x+y \approx \delta_1$~~

Then if $-x$ is the closest machine number to y, $x+y \approx d_2 \approx d_y$, and the relative error is close to 1.

Homework.

13. 18. 25. 33. 36.

Challenge (coding) problems.

5. 10.