How do computers represent numbers?

⇒ stored in a finite amount of memory!

A computer can't directly store a base 10 number — each cell is either on or off

∴ Binary!

0 = 0
1 = 1
2 = 10
3 = 11
11 = 1011
etc...

each binary digit is a bit

8 bits = 1 byte

single precision = 32 bits (4 bytes)

For integers you can store them exactly in a finite range!

$$-2^{31} < N < 2^{31} - 1$$

(one bit used for sign)

2^{31} isn't very big!

$2^{31} = 2,147,483,648$

(less than the amount the gov't spends in one day...)

Double precision (default for MatHab) doubles the # of bits to 64

To store larger or non-integer #'s, you use floating point scientific notation!

In single precision, use 24 bits mantissa, 8 bits for exponent

one of each needed for signs!

So the largest single precision # $= 1.111\ldots \times 2^{+127} \approx 10^{38}$

(base 2)

Any thing larger leads to overflow!

The 24 bit mantissa means that you only get ~7 sig digits (base 10).

For double precision, you do much better - about 16 sig digits, and a max ~~#~~ of $\sim 10^{308}$

This seems great - 10^{308} is huge (as a ref, there are $\sim 10^{50}$ atoms in earth)

You can still have problems!

In probability formulas you often need to use factorials

(to determine ~~#~~ possible permutations)

$171! \sim 10^{309} = \text{Inf}$

not a big ~~#~~!

The smallest number in Matlab is $\sim 10^{-308}$

Anything smaller is taken as zero!

How do we avoid underflow & overflow errors?

=> well-designed algorithms!

Example: Probability!

Suppose the probability of some outcome occuring is p

(coin flip, heads prob is p=0.5)

Now suppose you do this N times.

What is the probability of this occuring exactly k times?

$$X(p, k, N) = \binom{N}{k} p^k (1-p)^{N-k}$$

The quantity $\binom{N}{k}$ is the binomial coefficient:

$$\binom{N}{K} \equiv \frac{N!}{K!(N-K)!} = \frac{N(N-1)\cdots(N-K+1)}{K(K-1)\cdots 2\cdot 1}$$

A natural way to do this is to use the Matlab command factorial.m

You could also iterate over K:

$$\binom{N}{K} = \frac{\lambda_2}{\lambda_1}; \quad \lambda_2 = \prod_{i=N-K+1}^{N} i$$

$$\lambda_1 = \prod_{i=1}^{K} i$$

If N is larger than 170 you are toast!

In single precision N has to be much smaller!

Take coinflips - flip it 50 times, what is the prob it comes up heads exactly 25 times?

So: $N = 50$, $K = 25$, $p = 0.5$

$$X = \frac{50!}{25!\,(50-25)!}\,(0.5)^{25}\,(0.5)^{50-25}$$

If we calc. λ_1 & λ_2 we have trouble in single precision:

$$\lambda_2 = \prod_{i=K+1}^{N} i = \frac{N!}{K!} = 1.96 \times 10^{39}$$

(larger than max in single)

But the answer is just

$$X = 0.112 \text{ or } \sim 11\%\,!$$

How can we avoid the error? Simple! As we go through the calculation, we just add lines of code that check when the # is too large or small, then mult or divide by a large #. If we

keep track of the ~~**~~ of times we do it, we can recover the correct result at the end!

(Run example)

So much for the exponent, what about the mantissa? ⇒ finite precision! (round off error!)

Single Precision $\sim 10^{-7}$

Double Precision $\sim 10^{-16}$

These are small ; single ~ 13ft error in distance to moon

Double ~ 0.1 μm

But you can still have trouble!

Most common: Algorithms which subtract large #s to get small ones!

Example: Suppose we want to calc $f(x) = e^{-x}$

We can use the Taylor Series

$$f(x) \approx f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2} f''(x_0)(x-x_0)^2 + \frac{1}{3!} f'''(x_0)(x-x_0)^3 + \ldots$$

For e^{-x} if we take $x_0 = 0$ this is just:

$$f(x) = \sum_{i=0}^{\infty} \frac{(-x)^i}{i!} = 1 - x + \frac{x^2}{2} - \frac{x^3}{6} + \ldots$$

Suppose each term has some precision ε (rel. to mag)

The summed error is thus:

$$\text{error} \sim \varepsilon \cdot 1 + \varepsilon \cdot x + \varepsilon \frac{x^2}{2} + \cdots$$

(errors are additive!)

$$\therefore \text{error} \sim \varepsilon e^x$$

$$\text{so } \left(e^{-x}\right)_{num} = e^{-x} \pm \varepsilon e^x$$

$$\text{or } = e^{-x}\left(1 \pm \varepsilon e^{2x}\right)$$

Even for double precision, error is $O(1)$ for $x \sim 18$!

Note that the expansion was exact as it converges (eventually) for all x, but due to round off errors it fails for large (ish) x!

Never design an algorithm (or experiment!) where you subtract large #s to get small ones! This kills your precision!

How do we fix this? Simple!

$$e^{-x} = \frac{1}{e^x} = \frac{1}{\sum_{i=0}^{\infty} \frac{x^i}{i!}}$$

error is always $O(\varepsilon)$ for $x > 0$!

Another example of error is the combination of algorithm & round-off (precision) errors!

Suppose we have some $f(x)$ and we want the derivative $\left.\frac{df}{dx}\right|_{x_0}$

We can get this using the forward difference approximation:

$$\left.\frac{\partial f}{\partial x}\right|_{x_0} = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

What is the error in this formula?

Algorithm error is from h being too large, while numerical error is from h being too small!

Algorithm error:

We use the Taylor Series:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0^2) + \ldots$$

We can truncate the series if we introduce some point ξ:

$$\xi \in [x_0, x]$$

So ξ lies between x_0 & x

So: $f(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2}(x-x_0)^2 f''(\xi)$

↑ unknown point

From the forward difference equation:

$$\left(\frac{df}{dx}\Big|_{x_0}\right)_{num} \equiv \frac{f(x_0+h) - f(x_0)}{h}$$

$$= \frac{f(x_0) + h f'(x_0) + \frac{1}{2}h^2 f''(\xi) - f(x_0)}{h}$$

$$= f'(x_0) + \frac{1}{2} h f''(\xi)$$

So the algorithm error is $\sim h$

What about the numerical error?

Both $f(x_0)$ and $f(x_0+h)$ have error of $O(\varepsilon f(x_0))$ due to round off error!

Thus, round off error $\sim O\left(\frac{2\varepsilon f(x_0)}{h}\right)$

This blows up as $h \to 0$

The total error is a minimum for some intermediate h, about where both are the same magnitude.

So: $$\frac{2\varepsilon f(x_0)}{h_{opt}} \approx \frac{1}{2} h_{opt} f''(\xi)$$

So we have: $$h_{opt} \sim O\left(4\varepsilon \frac{f(x_0)}{f''(\xi)}\right)^{1/2}$$

(for small h_{opt}, $\xi \approx x_0$)

Thus, the best you can do with this formula is:

$$\left(\frac{\partial f}{\partial x}\bigg|_{x_0}\right)_{num} = f'(x_0)\left[1 + O\left(\varepsilon \frac{ff''}{f'^2}\right)^{1/2}\right]$$

So if $\varepsilon \sim 10^{-7}$, error in derivative is $O(\varepsilon^{1/2}) \sim 0.03\%$

double precision effectively becomes single precision in accuracy...

Run the Matlab example.

You can do better with better algorithms. A center difference approx uses points on either side of x_0:

$$\left(\frac{df}{dx}\Big|_{x_0}\right)_{num} = \frac{f(x_0+h) - f(x_0-h)}{2h}$$

The round off error is still $O(h^{-1})$, but the algorithm error is now $O(h^2)$ ⇒ Use the Taylor Series to prove this! (also plug into example)

This means the optimum h is larger, and the minimum total error is $\sim O(\varepsilon^{2/3})$.