Floating point numbers

FP numbers are what we use to approximate real numbers in numerical computations. Now that we've seen some basic numerical methods, let's look in more detail into what these numbers are and how they work (and don't work).

→ Must always remember that FP numbers are <u>not</u> real #'s.

We've already said that FP numbers are basically real numbers expressed approximately using scientific notation. For example:

"normalized" = one digit before decimal point.

5.4213×10^6 or $5.4213 E+6$

[Why do we like to represent numbers this way?

- can approximate numbers of any magnitude with an amount of ink that's proportional to the <u>relative precision</u>, or number of significant figures.
- it's easy to do arithmetic on numbers in this form.

Some alternatives:

- fixed point = keep a fixed number of digits on either side of the decimal point. One million is 1 000 000. 000 000; one millionth is 0 000 000. 000 001 Limited range (10^d where d = # digits) and constant <u>absolute</u> precision but variable relative precision
- store logarithm in fixed point. This gives perfectly constant relative precision and a large range. But you can't easily add and subtract.

FP makes a good engineering compromise.]

Before diving into real machine formats, I will first look at a "computer-like" fixed size human readable format, to establish the basic ideas. Then I will present the half-precision (16-bit) form of IEEE standard FP numbers, because it fits on the board better than the more standard longer formats. Finally I'll get to the single and double precision formats.

Let's say we are doing hand computation in decimal and decide to retain four significant figures in all intermediate results, and to allow two-digit exponents. Then we can write numbers in a fixed amount of space like this:

+5.421E+06	-1.186E+02	3.210E-5
(5,421,000)	(-118.6)	(0.0000321)

In FP terms, I have standardized on a four-digit mantissa (or significand) and a two-digit exponent.

What is the range of representable numbers in this scheme?

largest pos. number: +9.999E+99 (almost 10^{100})
smallest pos. number: +0.001E-99 (10^{-102})
(similar for negative)

Notice that I can represent numbers quite a bit smaller than the smallest exponent might suggest.

There is some waste here. How to write zero?

+0.000E+00 =0
+0.000E+12 =0
(any exponent is still zero.)

I can also write a non-normalized mantissa:

+0.032E+05 = 3200 } equal but this one is less precise.
+3.200E+03 = 3200

So for every exponent there are a bunch of "useless" digit patterns, in the sense that we can express the same number by another pattern.

What is the precision of these numbers?

A good way to answer this question is to look at the distance (difference) between adjacent representable numbers. For instance, the number after 1.0:

+1.000E+00
is +1.001E+00 } difference is 10^{-3}.

So in this range we have an absolute precision of 10^{-3} and a relative precision of 1:1000.

Looking at the number <u>before</u> 1.0:

+1.000E+00
+9.999E-01 } difference is 10^{-4}

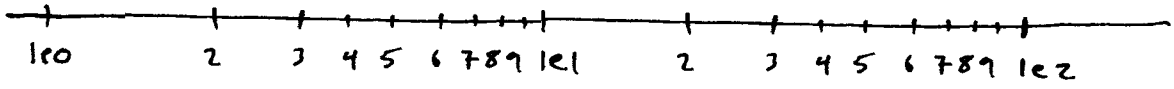
So in this range (just below 1.0) we have an absolute precision of 10^{-4} and a relative precision of 1:10,000. For most numbers the precision (relative) is between these two: when we've just switched to a new exponent the precision is lowest, and when we're about to switch (lots of 9s) it is highest.

Some numbers very close to ~~zero~~ have lower relative prec. because they can't be written with normalized mantissa:

+0.012E-99
+0.013E-99 } diff = 10^{-102}. val = 1.2×10^{-101}
rel prec = 1:12

These are known as "denormalized" numbers, and they behave like fixed point numbers because the exponent can't adjust any further.

[Interesting connection: plot the representable numbers with one digit mantissa as tic marks on a line in log scale:



these are familiar as the conventional tic marks for a log-scaled plot.]

Half-precision

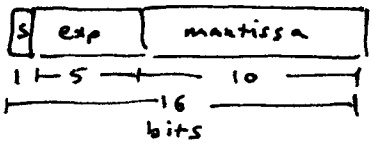
Of course, in the computer we use binary, not decimal, numbers. There's no problem with using scientific notation in binary.

$$123 = 111\,1011 = 1.111011 \times 2^6$$
$$-118.625 = -\underbrace{111\,0110}_{118}.\underbrace{101}_{5/8} = -1.110110 \times 2^6$$

There's one interesting difference: for normalized mantissas, the number to the left of the binary point ("radix pt.") carries no information and does not need to be written down.

The encoding of numbers like this into strings of bits is governed by a standard: IEEE 754. The standard specifies a 32-bit (single) and 64-bit (double) format, but I'll present a 16-bit (half) format because it is more blackboard friendly. This format is in widespread use for image data in graphics and will make its way into a future revision of IEEE 754.

The format looks like this:



One bit is for the sign, 5 bits are for the exponent, and 10 are for the mantissa. Note lack of sign for the exponent: instead of using signed or 2s complement we bias the exponent by adding about half the range to it — in this case 15. (More on why later.)

$$-118.625 \rightarrow -|6|1.110110 \rightarrow 0\cdot10101\cdot1101\,1000\,00$$
$$123 \rightarrow +|6|1.111011 \rightarrow 0\cdot10101\cdot1110\,1100\,00$$

Notice that I omitted the leading 1 in the mantissa.

There is one problem with this leading -1 convention : we can no longer represent zero!

Fix : introduce rule that exponent 0 (which you'd think is 2^{-15}) is instead 2^{-14} (same as exponent 1) but has an implied leading zero rather than an implied leading one. This leads to the number range :

0|00000|0000 0000 00	= 0
0|00000|0000 0000 01	= 2^{-24} : smallest denormalized
0|00000|1111 1111 11	≈ 2^{-14} : largest denormalized
0|00001|0000 0000 00	= 2^{-14} : smallest normalized
0|01111|0000 0000 00	= 1.0
0|11110|1111 1111 11	≈ 2^{16} : largest normalized

And obviously a similar pattern for negative sign.

A remarkable thing to note: these numbers are in the same order as the positive integers represented by the same bits! The largest value for a particular exponent has mantissa 1.11…1, and the smallest value for the next larger exponent has value 1.00…0 (represented by all zeros). The effect is to carry one into the exponent, just as if you were adding one to an integer.

Note that I reserved the exponent 11111. More on this later.

Just as with the decimal numbers we can look at the distance between adjacent numbers to get an idea of the precision. For numbers with exponent e, the difference is

$$0.0000\ 0000\ 01 \times 2^{e} = 2^{e-10}$$

The relative precision ranges from $2^{e-10}/2^{e} = 2^{-10}$ to $2^{e-10}/2^{e+1} = 2^{-11}$. That is 1:1024 to 1:2048. Denormalized numbers, again, have lower precision, all the way down to the first one, which has relative precision 1:2. Denormalized numbers are a kind of graceful underflow: less precise but better than rounding to zero.

The bit patterns with exponent 11111 don't represent regular numbers; instead they are used for special values. If the mantissa is zero we have an infinity (conveniently placed in sequence just after the last regular number). There are positive and negative infinities depending on the sign bit.

```
0 11111 0000 0000 00    = +Inf
1 11111 0000 0000 00    = -Inf
```

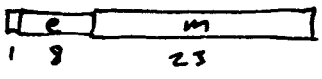
These are catch-all values to cover real numbers that are too large in magnitude to represent, as well as the concept of infinity.

If the mantissa is nonzero we have NaN, or "not-a-number". This is a placeholder for the result of a computation that doesn't have an answer.

One more note: there are two bit patterns to represent the real number 0: they are all zeros with two different values for the sign bit. The distinction is rarely, but occasionally, significant.

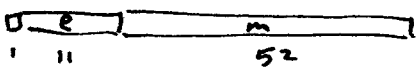
Real FP formats (or at least more common ones):

IEEE single precision (C/Java "float"; Matlab "single")



rel. prec (normalized): $1:2^{23}$ to $1:2^{24}$
rng (norm) $\sim 10^{\pm 38}$ $\approx$ 1:10Million / 7sig figs.

IEEE double precision (C/Java/Matlab "double")



rel. prec. $1:2^{52}$ to $1:2^{53}$
rng (norm) $\sim 10^{\pm 308}$ $\approx$ 1:1Quadrillion / 15 sig figs.

Arithmetic with FP

Now we know what values are available — but how are they used in computation?

basically, it works exactly as you'd expect. The system computes the exact answer (based on the necessarily quantized input values) and then returns the nearest representable number. (∃ different rounding modes that affect the details)

If the answer is smaller than the smallest (in magnitude) representable number, the calculation underflows and the result is a zero with the same sign as the answer.

If the answer is larger than the largest (in magnitude) representable number, the calculation overflows and the result is an infinity with the same sign as the answer.

numbers can be compared for equality and order, and the result is defined by the real numbers they represent. This means that -0 and $+0$ compare equal. $+Inf > x$ for all finite x. $-Inf < x$ for all finite x. $NaN \neq x$ for all x (even $NaN \neq NaN$).

Finally there is the question of what happens when you do arithmetic on these special values. The rules are sensible if you think of $\pm Inf$ as just really big numbers:

$\pm Inf \pm x$	for finite x	$\longrightarrow \pm Inf$	
$\pm Inf * x$	for finite x	$\longrightarrow \pm Inf$	
$\pm x / \pm 0$	for finite x	$\longrightarrow \pm Inf$	and all ops involving NaN result in NaN.
$\pm Inf / \pm 0$		$\longrightarrow \pm Inf$	
$\pm x / \pm Inf$	for finite x	$\longrightarrow \pm 0$	
$\pm 0 / \pm 0$		$\longrightarrow NaN$	
$\pm Inf / \pm Inf$		$\longrightarrow NaN$	

Infs make good sentinel values or special return values. E.g. $\log(0) = -Inf$. In C/Java, inverse trig functions return NaN when there is no real answer (In Matlab it returns complex results...)

In Matlab it's easy to use Inf and NaN because there are functions to return them. Try it yourself!

In Matlab there are also functions isfinite, isinf, and isnan. Exactly one will return true for any input.