

Floating Point Demystified, Part 1

September 15, 2014

Author's note: *this article used to be called "What Every Computer Programmer Should Know About Floating Point, part 1". Some people noted that this was perhaps too similar to the name of the existing article "What Every Computer Scientist Should Know About Floating Point". Since I have a bunch of other articles called "X Demystified", I thought I'd rename this one to remove confusion.*

The subject of floating-point numbers can strike vague uncertainty into all but the hardest of programmers. The first time a programmer gets bitten by the fact that `0.1 + 0.2` is *not quite equal* to `0.3`, the whole thing can seem like an inscrutable mess where nothing behaves like it should.

But lying amidst all of this seeming insanity are a lot of things that make perfect sense if you think about them in the right way. There is an existing article called [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#), but it is very math-heavy and focuses on subtle issues that face data scientists and CPU designers. This article is aimed at the general population of programmers. I'm focusing on simple and practical results that you can use to build your intuition for how to think about floating-point numbers.

As a practical guide I'm concerning myself only with the IEEE 754 floating point formats `single (float)` and `double` that are implemented on current CPUs and that most programmers will come into contact with, and not other topics like decimal floating point, arbitrary precision, etc. Also my goal is to build intuition and show the shapes of things, not prove theorems, so my math may not be fully precise all the time. That said, I don't want to be misleading, so please let me know of any material errors!

Articles like this one are often written in a style that is designed to make you question everything you thought you knew about the subject, but I want to do the opposite: I want to give you confidence that floating-point numbers actually make sense. So to kick things off, I'm going to start with some good news.

Integers are exact! As long as they're not too big.

It's true that `0.1 + 0.2 != 0.3`. But this lack of exactness does not apply to integer values! As long as they are small enough, floating point numbers can represent integers exactly.

```
1.0 == integer(1) (exactly)
5.0 == integer(5) (exactly)
2.0 == integer(2) (exactly)
```

This exactness also extends to operations over integer values:

```
1.0 + 2.0 == 3.0 (exactly)
5.0 - 1.0 == 4.0 (exactly)
2.0 * 3.0 == 6.0 (exactly)
```

Mathematical operations like these will give you exact results as long as all of the values are integers smaller than 2^{53} (for `double`) or 2^{24} (for `float`).

So if you're in a language like JavaScript that has no integer types (all numbers are double-precision floating point), and you have an application that wants to do precise integer arithmetic, you can treat JS numbers as 53-bit integers, and everything will be perfectly exact. Though of course if you do something inherently non-integral, like `8.0 / 7.0`, this exactness guarantee doesn't apply.

And what if you exceed 2^{53} for a double, or 2^{24} for a float? Will that give you strange dreaded numbers like `16777220.99999999` when you really wanted `16777221`?

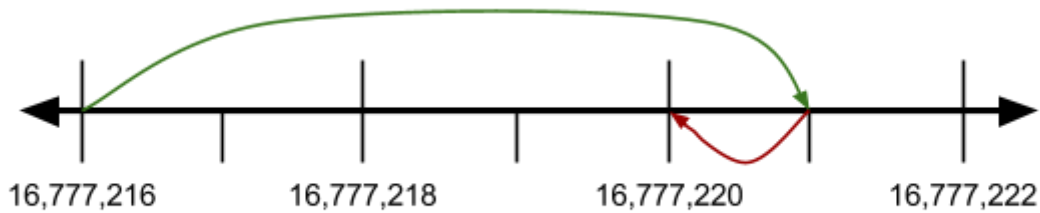
No — again for integers the news is much less dire. Between 2^{24} and 2^{25} a float can exactly represent half of the integers: specifically the **even** integers. So any mathematical operation that *would have* resulted in an odd number in this range will instead be rounded to one of the even numbers around it. But the result will still be an integer.

For example, let's add:

```
16,777,216 (2^24)
+           5
-----
16,777,221 (exact result)
16,777,220 (rounded to nearest representable float)
```

You can generally think of floating point operations this way. It's as if they computed exactly the correct answer with infinite precision, but then rounded the result to the nearest representable value. It's not implemented this way of course (putting infinite precision arithmetic in silicon would be expensive), but the results are generally the same as if it had.

We can also represent this concept visually, using a number line:



The green line represents the addition and the red line represents the rounding to the nearest representable value. The tick marks above the number line indicate which numbers are representable and which are not; because these values are in the range $[2^{24}, 2^{25}]$, only the even numbers are representable as float.

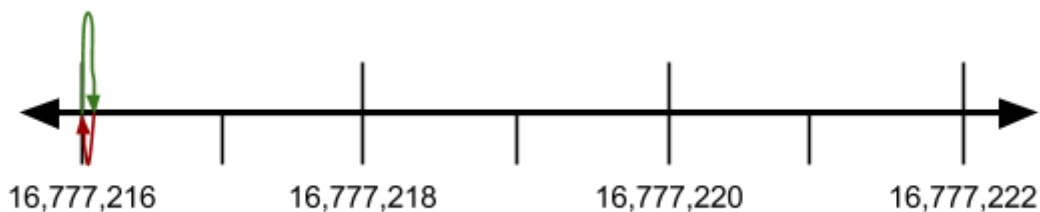
This model can also explain why adding two numbers that differ wildly in magnitude can make the smaller one get lost completely:

```

16,777,216
+      0.0001
-----
16,777,216.0001 (exact result)
16,777,216      (rounded to nearest representable float)

```

Or in the number line model:



The smaller number was not nearly big enough to get close to the next largest representable value (16777218), so the rounding caused the smaller value to get lost completely.

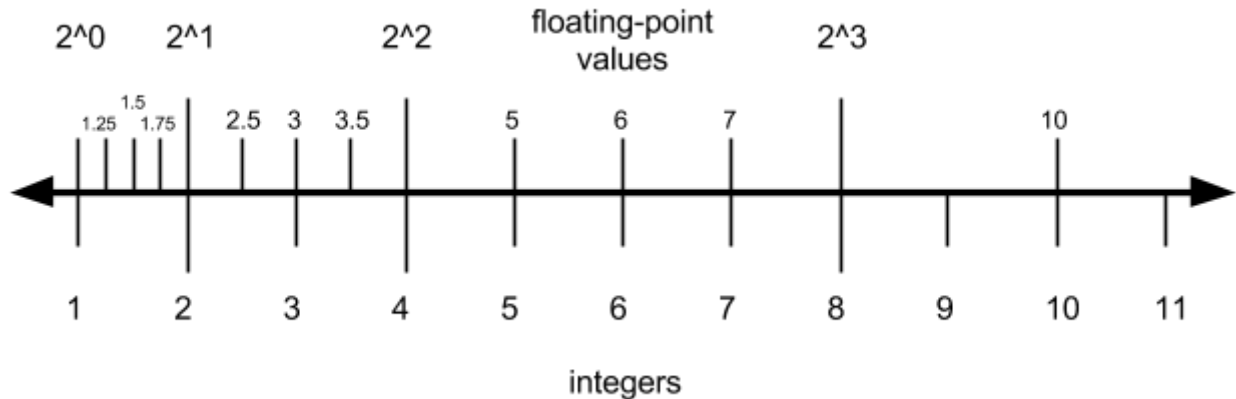
This rounding behavior also explains the answer to question number 4 in Ridiculous Fish's excellent article [Will It Optimize?](#) It's tempting to have floating-point anxiety and think that transforming `(float)x * 2.0f` into `(float)x + (float)x` must be imprecise somehow, but in fact it's perfectly safe. The same rule applies as our previous examples: compute the exact result with infinite precision and then round to the nearest representable number. Since the `x + x` and `x * 2` are mathematically exactly the same, they will also get rounded to exactly the same value.

So far we've discovered that a `float` can represent:

- all integers $[0, 2^{24}]$ exactly
- half of integers $[2^{24}, 2^{25}]$ exactly (the even ones)

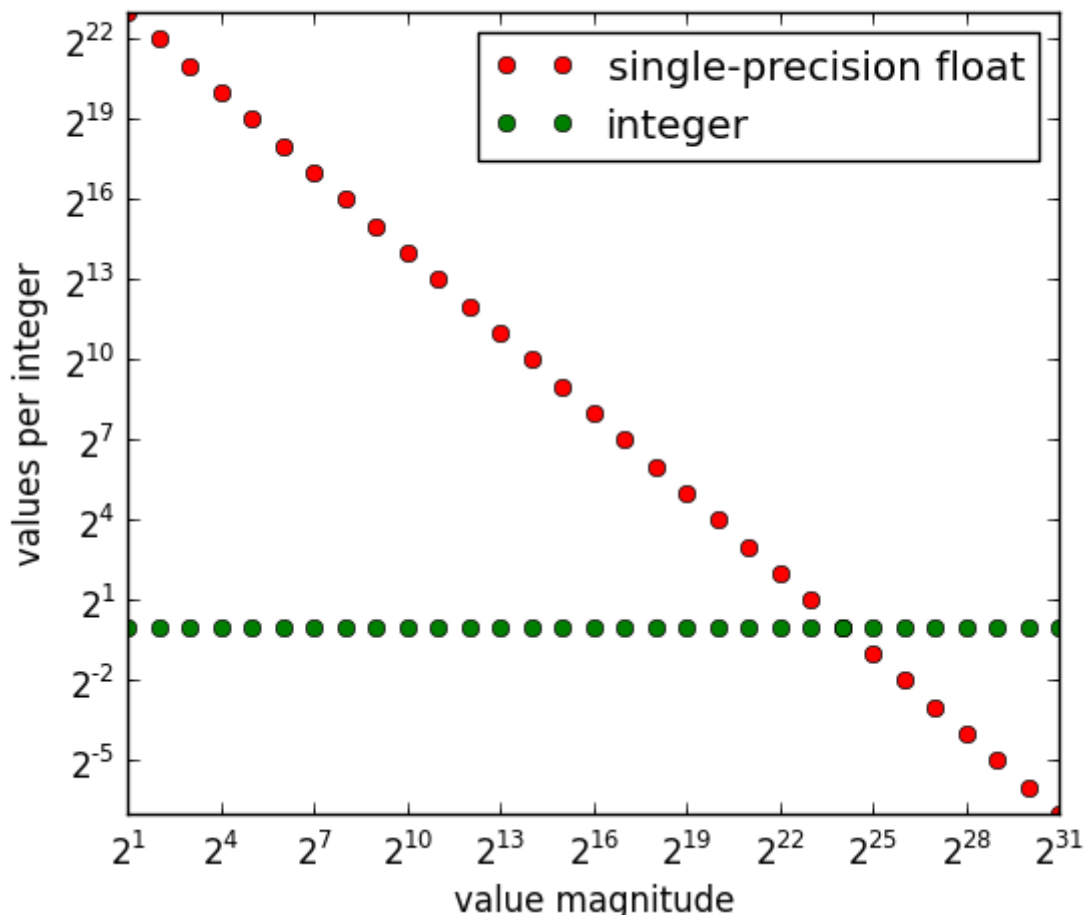
Why is this? Why do things change at 2^{24} ?

It turns out that this is part of a bigger pattern, which is that floating-point numbers are more precise the closer they are to zero. We can visualize this pattern again with a number line. This illustration isn't a real floating-point format (it has only two bits of precision, *much* less than `float` or `double`) but it follows the same pattern as real floating-point formats:



This diagram gets to the essence of the relationship between floating point values and integers. Up to a certain point (4 in this case), there are multiple floating point values per integer, representing numbers between the integers. Then at a certain point (here between 4 and 8) the set of floating point and integer values are the same. Once you get larger than that, the floating point values skip some integer values.

We can diagram this relationship to get a better sense and intuition for what numbers floats can represent compared to integers:



This plot is just a continuation of what we've said already. The green dots are boring and only appear for reference: they are saying that no matter how large or small your values are for an integer representation like `int32`, they can represent exactly one value per integer. That's a complicated way of saying that integer representations exactly represent the integers.

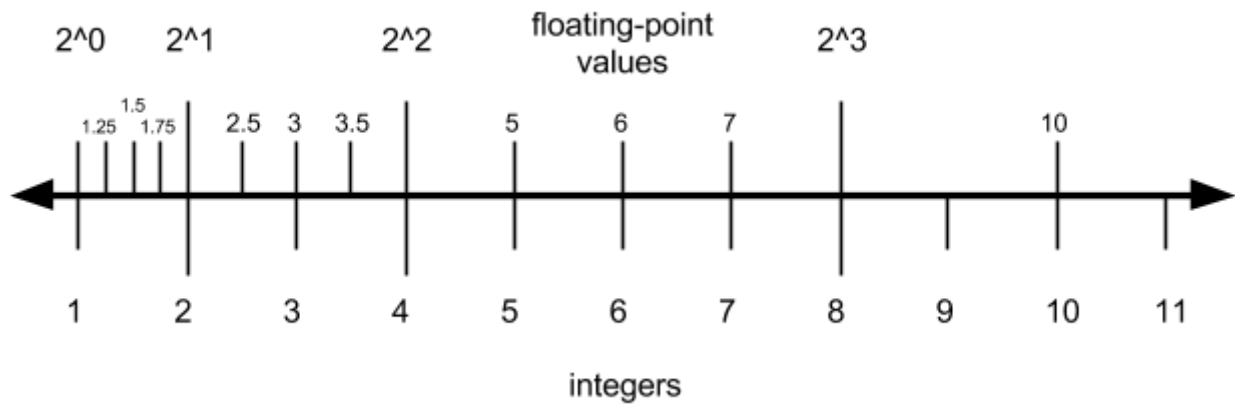
But where it gets interesting is when we compare integers to floats, which appear as red dots. The green and red dots intersect at 2^{24} ; we've already identified this as the largest value for which floats can represent every integer. If we go larger than this, to 2^{25} , then floats can represent half of all integers, (2^{-1} on the graph), which again is what we have said already.

The graph shows that the trend continues in both directions. For values in the range $[2^{25}, 2^{26}]$, floats can represent 1/4 of all integers (the ones divisible by 4). And if we go *smaller*, in the range $[2^{23}, 2^{24}]$, floats can represent 2 values per integer. This means that in addition to the integers themselves, a float can represent one value *in between* each integer, that being $x.5$ for any integer x .

So the closer you get to zero, the more values a float can stuff between consecutive integers. If you extrapolate this all the way to 1, we see that `float` can represent 2^{23} unique values between 1 and 2. (Between 0 and 1 the story is more complicated).

Range and Precision

I want to revisit this diagram from before, which depicts a floating-point representation with two bits of precision:



A useful observation in this diagram is that there are always 4 floating-point values between consecutive powers of two. For each increasing power of two, the number of integers doubles but the number of floating-point values is constant.

This is also true for `float` (2^{23} values per power of two) and `double` (2^{52} values per power of two). For any two powers-of-two that are in range, there will always be a constant number of values in between them.

This gets to the heart of how *range* and *precision* work for floating-point values. The concepts of range and precision can be applied to any numeric type; comparing and contrasting how integers and floating-point values differ with respect to range and precision will give us a deep intuition for how floating-point works.

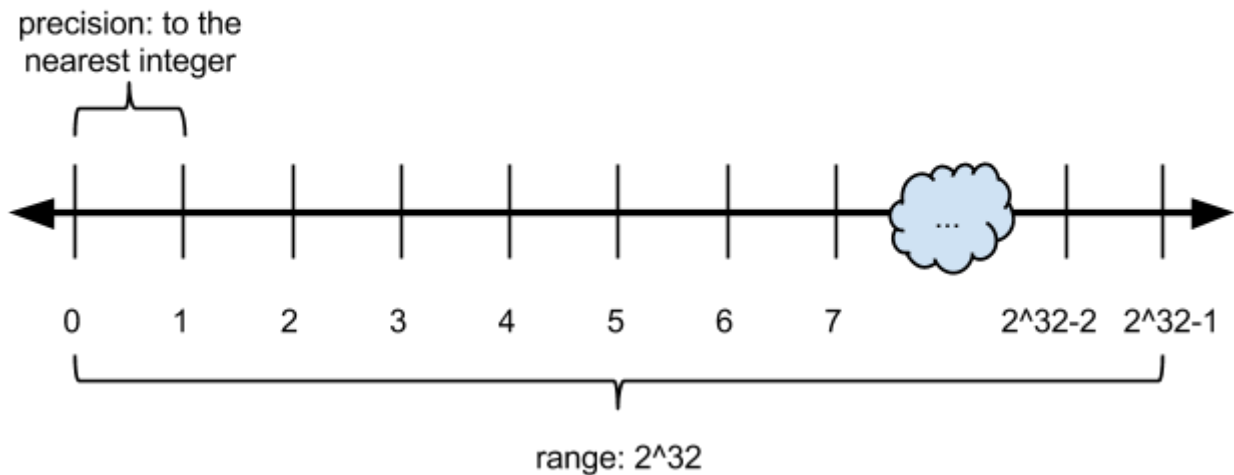
Range/precision for integers and fixed-point numbers

For an integer format, the range and precision are straightforward. Given an integer format with n bits:

- every value is precise to the nearest integer, regardless of the magnitude of the value.
- range is always 2^n between the highest and lowest value (for unsigned types the lowest value is 0 and for signed types the lowest value is $-(2^{n-1})$).

If we depict this visually, it looks something like:

range/precision of 32-bit integer

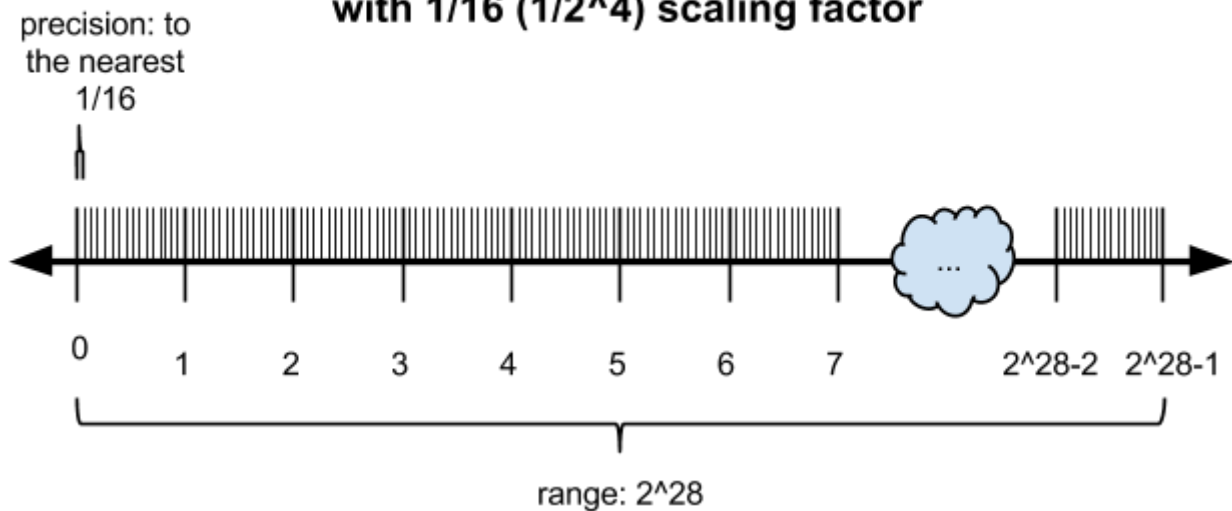


If you ever come across fixed point math, for example the [fixed-point support in the Allegro game programming library](#), fixed point has a similar range/precision analysis as integers. Fixed-point is a numerical representation similar to integers, except that each value is multiplied by a constant *scaling factor* to get its true value. For example, for a $1/16$ scaling factor:

integers	equivalent fixed point value
1	$1 * 1/16 = 0.0625$
2	$2 * 1/16 = 0.125$
3	$3 * 1/16 = 0.1875$
4	$4 * 1/16 = 0.25$
...	...
16	$16 * 1/16 = 1$
...	...

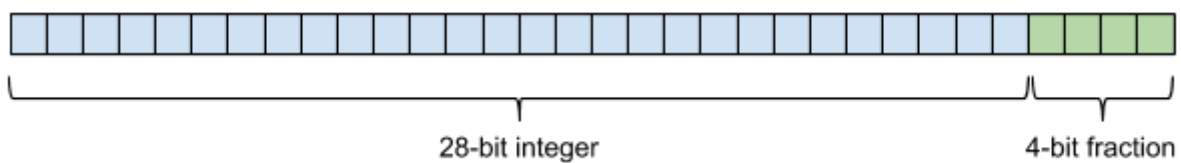
Like integers, fixed point values have a constant precision regardless of magnitude. But instead of a constant precision of 1, the precision is based on the scaling factor. Here is a visual depiction of a 32-bit fixed point value that uses a $1/16$ ($1/2^4$) scaling factor. Compared with a 32-bit integer, it has 16x the precision, but only $1/16$ the range:

range/precision of 32-bit fixed point with $1/16$ ($1/2^4$) scaling factor

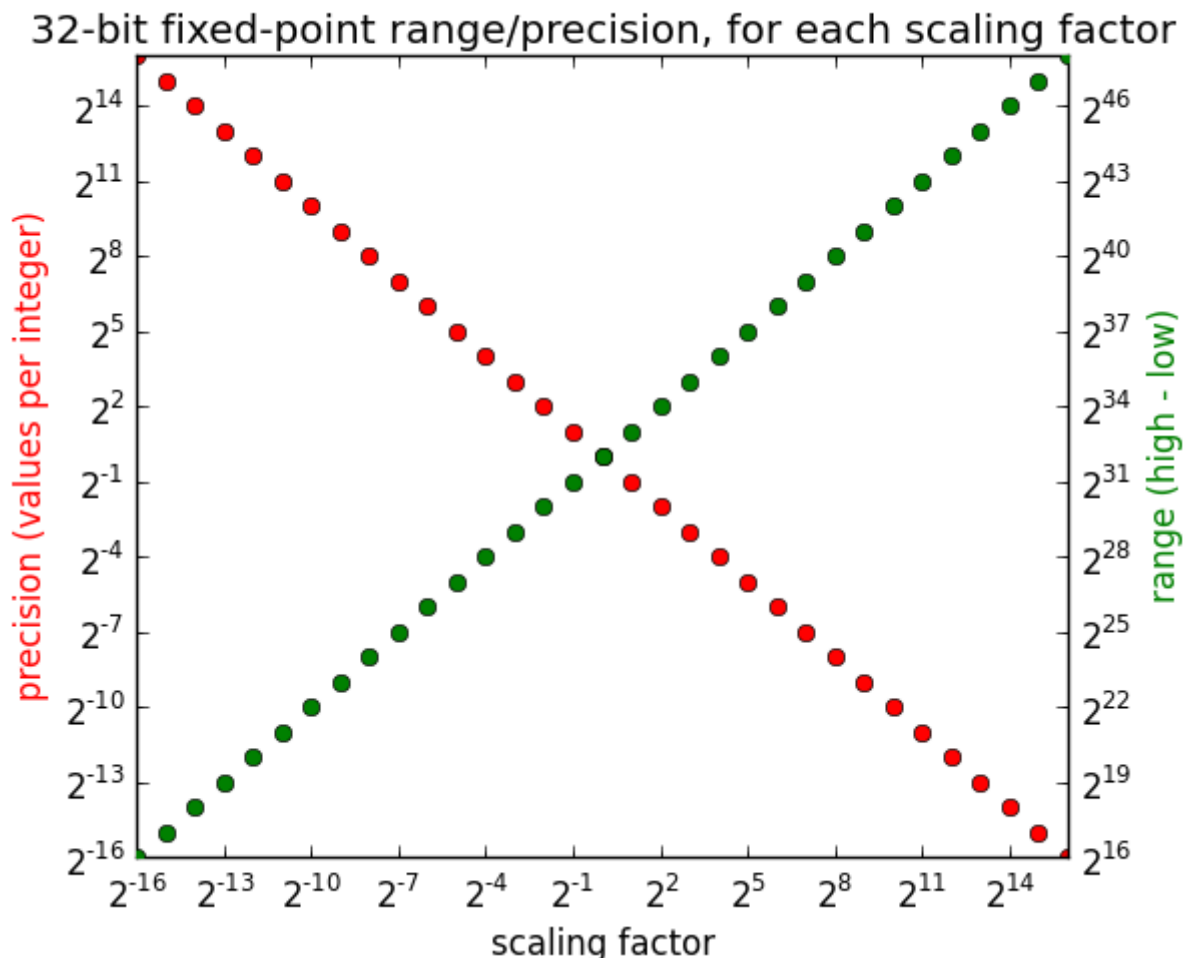


The fixed-point scaling factor is usually a fractional power of two in (ie. $1/2^n$ for some n), since this makes it possible to use simple bit shifts for conversion. In this case we can say that n bits of the value are dedicated to the fraction.

32-bit fixed point with $1/16$ ($1/2^4$) scaling factor



The more bits you spend on the integer part, the greater the range. The more bits you spend on the fractional part, the greater the precision. We can graph this relationship: given a scaling factor, what is the resulting range and precision?



Looking at the first value on the left, for scaling factor 2^{-16} (ie. dedicating 16 bits to the fraction), we get a precision of 2^{16} values per integer, but a range of only 2^{16} . Increasing the scaling factor increases the range but decreases the precision.

At scaling factor $2^0 = 1$ where the two lines meet, the precision is 1 value per integer and the range is 2^{32} — this is exactly the same as a regular 32-bit integer. In this way, **you can think of regular integer types as a generalization of fixed point**. And we can even use positive scaling factors: for example with a scaling factor of 2, we can double the range but can only represent half the integers in that range (the even integers).

The key takeaway from our analysis of integers and fixed point is that we can trade off range and precision, but given a scaling factor the precision is always constant, regardless of how big or small the values are.

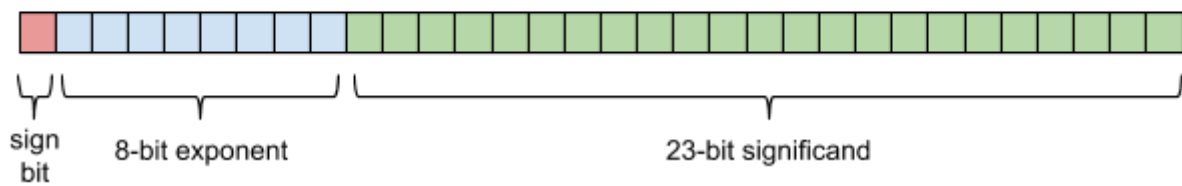
Range/precision for floating-point numbers

Like fixed-point, floating-point representations let you trade-off range and precision. But unlike fixed point or integers, the precision is proportional to the size of the value.

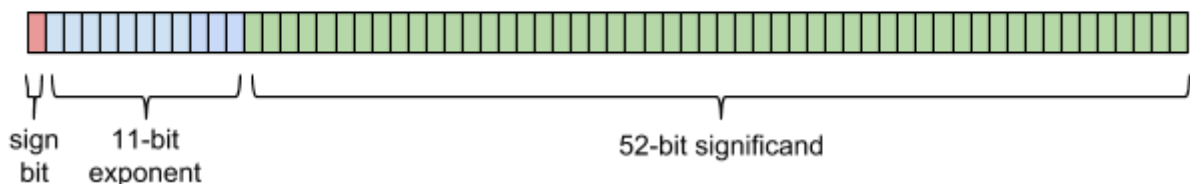
Floating-point numbers divide the representation into the *exponent* and the *significand* (the latter is also called the *mantissa* or *coefficient*). The number of bits dedicated to the

exponent dictates the range, and the number of bits dedicated to the significand determines the precision.

Single-precision floating point



Double-precision floating point



We will discuss the precise meanings of the exponent and significand in the next installment, but for now we will just discuss the general patterns of range and precision.

Range works a little bit differently in floating-point than in fixed point or integers. Have you ever noticed that `FLT_MIN` and `DBL_MIN` in C are not negative numbers like `INT_MIN` and `LONG_MIN`? Instead they are very small positive numbers:

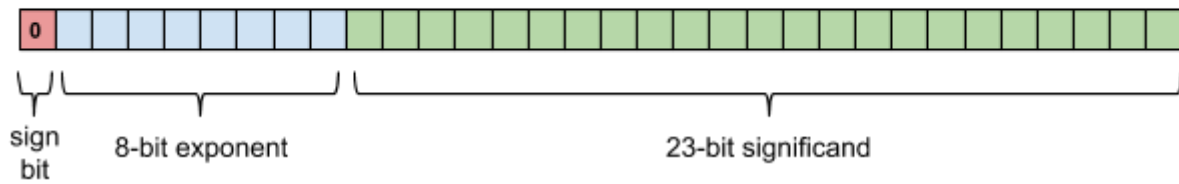
```
#define FLT_MIN      1.17549435E-38F
#define DBL_MIN      2.2250738585072014E-308
```

Why is this?

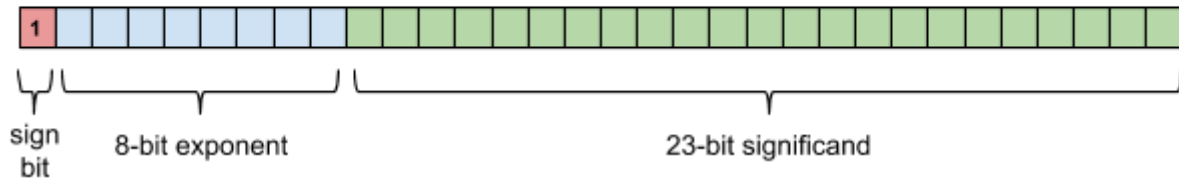
The answer is that floating point numbers, because they are based on exponents, can never actually reach zero or negative numbers “natively”. Every time you decrease the exponent you get closer to zero but you can never actually reach it. So the smallest number you *can* reach is `FLT_MIN` for `float` and `DBL_MIN` for `double`. (*denormalized* numbers can go smaller, but they are considered special-case and are not always enabled. `FLT_MIN` and `DBL_MIN` are the smallest *normalized* numbers.)

You may protest that `float` and `double` can clearly represent zero and negative numbers, and this is true, but only because they are special-cased. There is a sign bit that indicates a negative number when set.

Positive float (sign bit unset)

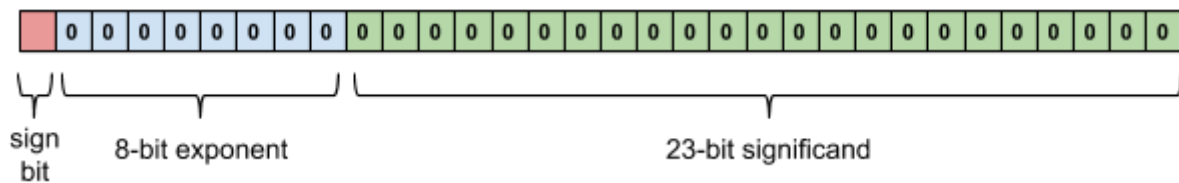


Negative float (sign bit set)



And when the exponent and significand are both zero, this is special-cased to be the value zero. (If the exponent is zero but the significand is non-zero, this is a denormalized number; a special topic for another day.)

Zero float (sign bit indicates +/-0)



Put these two special cases together and you can see why positive zero and negative zero are two distinct values (though they compare equal).

Because floating-point numbers are based on exponents, and can never truly reach zero, the range is defined not as an absolute number, but as a *ratio* between the largest and smallest representable value. That range ratio is entirely determined by the number of bits allotted to the exponent.

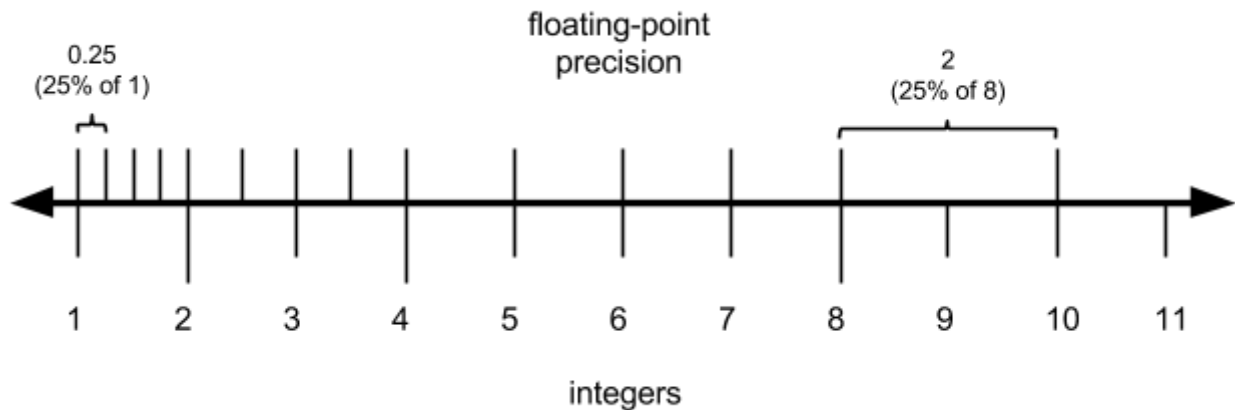
If there are n bits in the exponent, the ratio of the largest to the smallest value is roughly 2^{2^n} . Because the n -bit number can represent 2^n distinct values, and since those values are themselves exponents we raise 2 to that value.

We can use this formula to determine that `float` has a range ratio of roughly 2^{256} , and `double` has a range ratio of roughly 2^{2048} . (In practice the ranges are not quite this big, because IEEE floating point reserves a few exponents for zero and `NaN`).

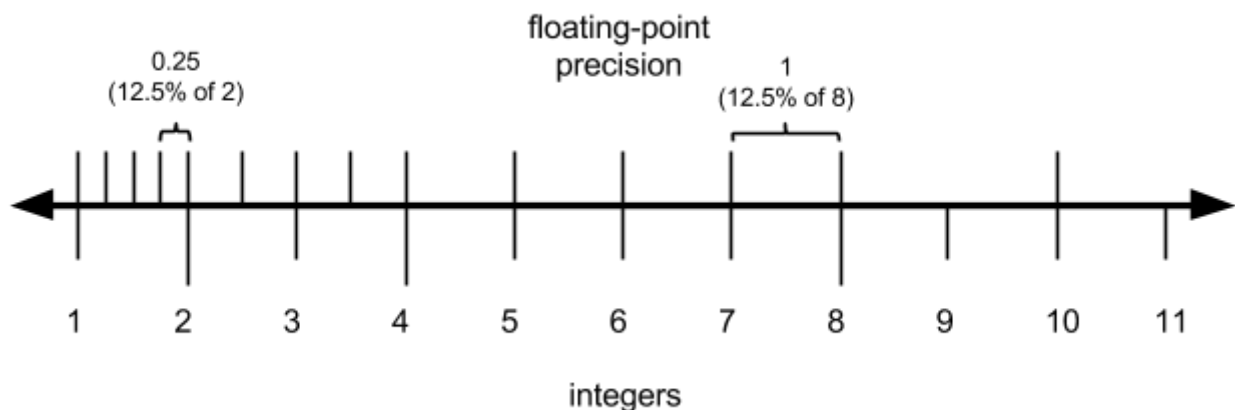
This alone doesn't say what the largest and smallest values actually are, because the format designer gets to choose what the smallest value is. If `FLT_MIN` had been chosen as $2^0 = 1$, then the largest representable value would be $2^{256} \approx 10^{77}$.

But instead `FLT_MIN` was chosen as $2^{-126} \approx 10^{-37}$, and `FLT_MAX` is $\approx 2^{128} \approx 3.4 \times 10^{38}$. This gives a true range ratio of $\approx 2^{254}$, which roughly lines up with our previous analysis that yielded 2^{256} (reality is a bit smaller because two exponents are stolen for special cases: zero and NaN/infinity).

What about precision? We have said several times that the precision of a floating-point value is proportional to its magnitude. So instead of saying that the number is precise to the nearest integer (like we do for integer formats), **we say that a floating-point value is precise to $X\%$ of its value**. Using our sample from before of an imaginary floating point format with a two-bit significand, we can see:



So at the low end of each power of two, the precision is always 25% of the value. And at the high end it looks more like:



So for a two-bit significand, the precision is always between 12.5% and 25% of the value. We can generalize this and say that for an n -bit significand, the precision is between $1/2^n$ and $1/(2^{n+1})$ of the value (ie. between $\frac{100}{2^n}\%$ and $\frac{100}{2^{n+1}}\%$ of the value. But since $1/2^n$ is the worst case, we'll talk about that because that's the figure you can count on.

We have finally explored enough to be able to fully compare/contrast fixed-point and integer values with floating point!

	range	precision
--	-------	-----------

fixed point and integer	<i>scalar</i> (high - low) $2^n \times \text{scaling factor}$	<i>absolute/constant</i> equal to the scaling factor
floating point	<i>ratio</i> (high / low) 2^{2^e}	<i>relative</i> (X%) $\frac{100}{2^n} \%$ (worst case)

If we apply these formulas to single-precision floating point vs. 32-bit unsigned integers, we get:

	range	precision
integer	2^{32}	1
floating point	$2^{256}/1$	0.00001% (worst case)

Practical trade-offs between fixed/floating point

Let's step back for a second and contemplate what all this really means, for us humans here in real life as opposed to abstract-math-land.

Say you're representing lengths in kilometers. If you choose a 32-bit integer, the shortest length you can measure is 1 kilometer, and the longest length you can measure is 4,294,967,296 km (measured from the Sun this is somewhere between Neptune and Pluto).

On the other hand, if you choose a single-precision float, the shortest length you can measure is 10^{-26} nanometers — a length so small that a single atom's radius is 10^{24} times greater. And the longest length you can measure is 10^{25} light years.

The float's range is almost unimaginably wider than the int32. And what's more, the float is also *more accurate* until we reach the magic inflection point of 2^{24} that we have mentioned several times in this article.

So if you choose int32 over float, you are giving up an unimaginable amount of range, *and* precision in the range $[0, 2^{24}]$, all to get better precision in the range $[2^{24}, 2^{32}]$. In other words, the int32's sole benefit is that it lets you talk about distances greater than 16 million km to kilometer precision. But how many instruments are even that accurate?

So why does anyone use fixed point or integer representations?

To turn things around, think about `time_t`. `time_t` is a type defined to represent the number of seconds since the epoch of `1970-01-01 00:00 UTC`. It has traditionally been defined as a 32-bit signed integer (which means that [it will overflow in the year 2038](#)). Imagine that a 32-bit single-precision float had been chosen instead.


With a `float time_t`, there would be no overflow until the year 5395141535403007094485264579465 AD, long after the Sun has swallowed up the Earth as a Red Giant, and turned into a Black Dwarf. However! With this scheme the granularity of timekeeping would get worse and worse the farther we got from 1970. Unlike the `int32` which gives second granularity all the way until 2038, with a `float time_t` we would already in 2014 be down to a precision of 128 seconds — far too coarse to be useful.

So clearly floating point and fixed point / integers all have a place. Integers are still ideal for when you are counting things, like iterations of a loop, or for situations like a time counter where you really do want a constant precision over its range. Integer results can also be more predictable since the precision doesn't vary based on magnitude. For example, integers will always hold the identity `x + 1 - 1 == x`, as long as `x` doesn't overflow. The same can't be said for floating point.

Conclusion

There is more still to cover, but this article has grown too long already. I hope this has helped build your intuition for how floating point numbers work. In the next article(s) in the series, we'll cover: the precise way in which the value is calculated from exponent and significand, fractional floating point numbers, and the subtleties of printing floating-point numbers.

Josh Haberman
jhaberman@gmail.com

 haberman
 JoshHaberman

Parsing, performance, and low-level programming.