Floating point number representation

Floating point representations vary from machine to machine, as I've implied. Fortunately one is by far the most common these days: the IEEE-754 standard. This standard is prevalent enough that it's worthwhile to look at it in depth; chances are good you'd be able to use this information on your platform (look for ieee754.h).

An IEEE-754 float (4 bytes) or double (8 bytes) has three components (there is also an analogous 96-bit extended-precision format under IEEE-854): a sign bit telling whether the number is positive or negative, an exponent giving its order of magnitude, and a mantissa specifying the actual digits of the number. Using single-precision floats as an example, here is the bit layout:

```
 seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm    meaning
31                               0    bit #

s = sign bit, e = exponent, m = mantissa
```

The value of the number is the mantissa times 2^x, where x is the exponent. Notice that we are dealing with binary fractions, so that 0.1 (the leftmost mantissa bit) means 1/2 (the place values to the right of the decimal point are 2^-1, 2^-2, etc., just as we have 10^-1, 10^-2, etc. in decimal).

Notice further that there's a potential problem with storing both a mantissa and an exponent: 2x10^-1 = 0.2x10^0 = 0.02x10^1 and so on. This would correspond to lots of different bit patterns representing the same quantity, which would be a huge waste (it would probably also make it harder and slower to implement math operations in hardware). This problem is circumvented by interpreting the whole mantissa as being to the right of the decimal point, with an implied "1" always present to the left of the decimal. I'll refer to this as a "1.m" representation. "But wait!" you cry. "What if I don't want a 1 there?" Think of it is as follows: imagine writing a real number in binary. Unless it's zero, it's gotta have a 1 somewhere. Shift your decimal point to just after the first 1, then don't bother to store that 1 since we know it's always implied to be there. Now all you have to do is set the exponent correctly to reproduce the original quantity.

But what if the number *is* zero? The good people at the IEEE standards committee solve this by making zero a special case: if every bit is zero (the sign bit being irrelevant), then the number is considered zero.

Oh dear. Take a moment to think about that last sentence. Now it would seem we have no way to represent humble 1.0, which would have to be 1.0x2^0 (an exponent of zero, times the implied one)! The way out of this is that the interpretation of the exponent bits is not straightforward either. The exponent of a single-precision float is "shift-127" encoded, meaning that the actual exponent is eeeeeeee minus 127. So thankfully, we can get an exponent of zero by storing 127 (0x7f). Of course simply shifting the range of the exponent is not a panacea; something still has to give somewhere. We yield instead at the low extreme of the spectrum of representable magnitudes, which should be 2^-127. Due to shift-127, the lowest possible exponent is actually -126 (1 - 127). It seems wise, to me, to give up the smallest exponent instead of giving up the ability to represent 1 or zero!

Zero is not the only "special case" float. There are also representations for positive and negative infinity, and for a not-a-number (NaN) value, for results that do not make sense (for example, non-real numbers, or the result of an operation like infinity times zero). How do these work? A number is infinite if every bit of the exponent is set (yep, we lose another one), and is NaN if every bit of the exponent is set plus any mantissa bits are set. The sign bit still distinguishes +/-inf and +/-NaN.

To review, here are some sample floating point representations:

```
0      0x00000000
1.0    0x3f800000
0.5    0x3f000000
3      0x40400000
+inf   0x7f800000
-inf   0xff800000
+NaN   0x7fc00000 or 0x7ff00000
in general: number = (sign ? -1:1) * 2^(exponent) * 1.(mantissa bits)
```

As a programmer, it is important to know certain characteristics of your FP representation. These are listed below, with example values for both single- and double-precision IEEE floating point numbers:

Property	**Value for float**	**Value for double**
Largest representable number	3.402823466e+38	1.7976931348623157e+308
Smallest number without losing precision	1.175494351e-38	2.2250738585072014e-308
Smallest representable number(*)	1.401298464e-45	5e-324
Mantissa bits	23	52
Exponent bits	8	11
Epsilon(**)	1.1929093e-7	2.220446049250313e-16

Note that all numbers in the text of this article assume single-precision floats; doubles are included above for comparison and reference purposes.

(*)
Just to make life interesting, here we have yet another special case. It turns out that if you set the exponent bits to zero, you can represent numbers other than zero by setting mantissa bits. As long as we have an implied leading 1, the smallest number we can get is clearly 2^-126, so to get these lower values we make an exception. The "1.m" interpretation disappears, and the number's magnitude is determined only by bit positions; if you shift the mantissa to the right, the apparent exponent will change (try it!). It may help clarify matters to point out that 1.401298464e-45 = 2^(-126-23), in other words the smallest exponent minus the number of mantissa bits.

However, as I have implied in the above table, when using these extra-small numbers you sacrifice precision. When there is no implied 1, all bits to the left of the lowest set bit are leading zeros, which add no information to a number (as you know, you can write zeros to the left of any number all day long if you want). Therefore the absolute smallest representable number (1.401298464e-45, with only the lowest bit of the FP word set) has an appalling mere single bit of precision!

(**)
Epsilon is the smallest x such that 1+x > 1. It is the place value of the least significant bit when the exponent is zero (i.e., stored as 0x7f).

III. Effective FP Programming

Numerical programming is a huge area; if you need to develop sophisticated numerical algorithms then this article will not be sufficient. Asking how to compute with ideal accuracy and precision is like asking how to write the fastest program, or asking how every piece of software should be designed—the answer depends on the application, and might require a book or two to communicate. Here I'll just try to cover what I think every programmer should know.

Equality

First let's tackle that pesky issue of equality: why is it so hard to know when two floats are equal? In one sense, it really isn't that hard; the == operator will, in fact, tell you if two floats are exactly equal (i.e., match bit for bit). You will agree, however, that it usually makes little sense to compare bits when some of those bits might be incorrect anyway, and that's the situation we have with the limited accuracy of floats. Results have to be rounded to fit in a finite word, and if the CPU and/or software does not round as you expected, your equality tests might fail.

Worse still, it often isn't the inherent inaccuracy of floats that bites you, but the fact that many operations commonly done on floats are themselves inaccurate. For example, the standard C library trig functions (sin, cos, etc.) are implemented as polynomial approximations. It might be too much to hope for that every bit of the cosine of pi/2 would be 0.

So the question of equality spits another question back at you: "What do *you* mean by equality?" For most people, equality means "close enough". In this spirit, programmers usually learn to test equality by defining some small distance as "close enough" and seeing if two numbers are that close. It goes something like this:

```
#define EPSILON 1.0e-7

#define flt_equals(a, b) (fabs((a)-(b)) < EPSILON)
```

People usually call this distance EPSILON, even though it is not the epsilon of the FP representation. I'll use EPSILON (all caps) to refer to such a constant, and epsilon (lower case) to refer to the actual epsilon of FP numbers.

This technique sometimes works, so it has caught on and become idiomatic. In reality this method can be very bad, and you should be aware of whether it is appropriate for your application or not. The problem is that it does not take the exponents of the two numbers into account; it assumes that the exponents are close to zero. How is that? It is because the precision of a float is not determined by magnitude ("On this CPU, results are always within 1.0e-7 of the answer!") but by the *number of correct bits*. The EPSILON above is a tolerance; it is a statement of how much precision you expect in your results. And precision is measured in significant digits, not in magnitude; it makes no sense to talk of "1.0e-7 of precision". A quick example makes this obvious: say we have the numbers 1.25e-20 and 2.25e-20. Their difference is 1e-20, much less than EPSILON, but clearly we do not mean them to be equal. If, however, the numbers were 1.2500000e-20 and 1.2500001e-20, then we might intend to call them equal.

The take-home message is that when you're defining how close is close enough, you need to talk about how many significant digits you want to match. Answering this question might require some experimentation; try out your algorithm and see how close "equal" results can get.

Overflow

Let's move on. Due to that pesky finite-ness of real computers, numerical overflow is one of a programmer's most common concerns. If you add one to the largest possible unsigned integer, the number rolls back to zero. Annoyingly, you can't tell that this integer overflowed just by looking at it; it looks the same as any zero. Most CPUs will actually set a flag bit whenever an operation overflows, and checking this bit is one of the few hand-coded assembly language optimizations that are not obsolete.

However, one of the truly nice things about floats is that when they overflow, you are conveniently left with +/-inf. These quantities tend to behave as expected: +inf is greater than any other number, -inf is less than any other number, inf+1 equals inf, and so on. This property makes floats useful for checking overflow in integer math as well. You can do a calculation in floating point, then simply compare the result to something like INT_MAX before casting back to integer.

Casting opens up its own can of worms. You have to be careful, because your float might not have enough precision to preserve an entire integer. A 32-bit integer can represent any 9-digit decimal number, but a 32-bit float only offers about 7 digits of precision. So if you have large integers, making this conversion will clobber them. Thankfully, doubles have enough precision to preserve a whole 32-bit integer (notice, again, the analogy between floating point precision and integer dynamic range). Also, there is some overhead associated with converting between numeric types, going from float to int or between float and double.

Whether you're using integers or not, sometimes a result is simply too big and that's all there is to it. However, you must try to avoid overflowing results needlessly. Often the final result of a computation is smaller than some of the intermediate values involved; even though your final result is representable, you might overflow during an intermediate step. Avoid this numerical faux pas! The classic example (from "Numerical Recipes in C") is computing the magnitude of a complex number. The naive implementation is:

```
double magnitude(double re, double im)
{
    return sqrt(re*re + im*im);
}
```

Let's say both components are 1e200. The magnitude is 1.4142135e200, well within the range of a double. However, squaring 1e200 yields 1e400, which is outside the range—you get infinity, whose square root is still infinity. Here is a much better way to write this function:

```
double magnitude(double re, double im)
{
    double r;

    re = fabs(re);
    im = fabs(im);
    if (re > im) {
        r = im/re;
        return re*sqrt(1.0+r*r);
    }
    if (im == 0.0)
        return 0.0;
    r = re/im;
    return im*sqrt(1.0+r*r);
}
```

All we did was rearrange the formula by bringing an re or im outside the square root. Which one we bring out depends on which one is bigger; if we square im/re when im is larger, we still risk overflow. If im is 1e200 and re is 1, clearly we don't want to square im/re, but squaring re/im is ok since it is 1e-400 which is rounded to zero—close enough to get the right answer. Notice the asymmetry: large magnitudes can get you lost at +inf, but small magnitudes end up as zero (not -inf), which is a good approximation.

Loss of significance

Finally, we come to issues of "getting the right answer". Uncertain equality is only the tip of the iceberg of problems caused by limited accuracy and precision. Having your decimals cut off at some point wreaks a surprising amount of havoc with mathematics. "Loss of significance" refers to a class of situations where you end up inadvertently losing precision (discarding information) and potentially ending up with laughably bad results.

As we have seen, the 1.m representation prevents waste by ensuring that nearly all floats have full precision. Even if only the rightmost bit of the mantissa is set (assuming a garden-variety exponent), all the zeros before it count as significant figures because of that implied 1. However, if we were to *subtract*

two numbers that were very close to each other, the implied ones would cancel, along with whatever mantissa digits matched. If the two numbers differed only in their last bit, our answer would be accurate to only one bit! Ouch!

Just like we avoided overflow in the complex magnitude function, there is essentially always a way to rearrange a computation to avoid subtracting very close quantities (I cover myself by saying "essentially always", since the math behind this is way beyond the scope of this article). Naturally there is no general method for doing this; my advice would be to just go through and take a hard look at all your subtractions any time you start getting suspicious results. An example of a technique that might work would be changing polynomials to be functions of 1/x instead of x (this can help when computing the quadratic formula, for one).

A related problem comes up when summing a series of numbers. If some terms of your series are around an epsilonth of other terms, their contribution is effectively lost if the bigger terms are added first. For example, if we start with 1.0 (single precision float) and try to add 1e-8, the result will be 1.0 since 1e-8 is less than epsilon. In this case the small term is swallowed completely. In less extreme cases (with terms closer in magnitude), the smaller term will be swallowed partially—you will lose precision.

If you're lucky and the small terms of your series don't amount to much anyway, then this problem will not bite you. However, often a large number of small terms can make a significant contribution to a sum. In these cases, if you're not careful you will keep losing precision until you are left with a mess. Sometimes people literally sort the terms of a series from smallest to largest before summing if this problem is a major concern.

A rule of thumb

An overwhelming amount of information is available describing numerical gotchas and their fixes—far more than all but the dedicated scientific programmer wants to deal with. To simplify things, the way we often think about loss of precision problems is that a float gradually gets "corrupted" as you do more and more operations on it. Take the aforementioned cosine of pi/2, 6.12303e-17. By itself it's not so bad, it's pretty close to zero. But if our next step was to divide by 1e-17, then we're left with about 6, which is a far cry from the zero we would have expected.

This makes algorithms with lots of "feedback" (taking previous outputs as inputs) suspect. Often you have a choice between modifying some quantity incrementally or explicitly; you could say "x += inc" on each iteration of a loop, or you could use "x = n*inc" instead. Incremental approaches tend to be faster, and in this simple case there isn't likely to be a problem, but for numerical stability "refreshing" a value by setting it in terms of stable quantities is preferred. Unfortunately, feedback is a powerful technique that can provide fast solutions to many important problems. All I can say here is that you should avoid it if it is clearly unnecessary; when you need a good algorithm for something like solving nonlinear equations, you'll need to look for specialized advice.

Don't forget about integers

Lastly, a reminder not to forget the humble integer: its accuracy can be a useful tool. Sometimes a program needs to keep track of a changing fraction of some kind, a scaling factor perhaps. In this situation you know that the number you are storing is rational, so you can avoid all the problems of floating point math by storing it as an integer numerator and denominator. This is particularly easy for unit fractions; if you need to move around among 1/2, 1/3, 1/4, etc., you should clearly be storing only the denominator and regenerating 1.0/denom whenever you need the fraction as a float.

Next: Cleanly Printing Floating Point Numbers