Comparing Floating-Point Numbers Is Tricky

Matt Kline
matt@bitbashing.io

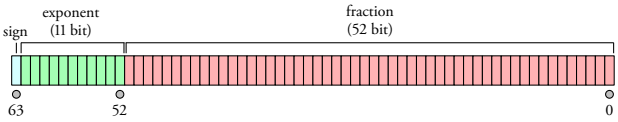
March 30, 2017

Abstract

Floating-point math is fraught with subtle gotchas, and comparing values properly is no exception. Here we discuss common pitfalls, examine some possible solutions, and try to beat Boost.

Things you probably know about floats

If you need to represent a non-integer in a mainstream programming language, you'll probably end up using IEEE 754 floating-point values. Since their standardization in 1985, they've become ubiquitous. Nearly all modern CPUs—and many microprocessors—contain special hardware (called *floating-point units*, or FPUs) to handle them.



The layout of a 64-bit IEEE 754 float
(from Wikipedia)

Each float consists of a sign bit, some bits representing an exponent, and bits representing a fraction, also called the *mantissa*. Under most circumstances, the value of a float is:

$$(-1)^s \times 1.m \times 2^{e-c}$$

where s is our sign bit, m is some fraction represented by the mantissa bits, e is an unsigned integer represented by the exponent bits, and c is half the maximum value of e, i.e., 127 for a 32-bit float and 1023 for a 64-bit float.

There are also some special cases. For example, when all exponent bits are zero, the formula changes to:

$$(-1)^s \times 0.m \times 2^{-c+1}$$

Note the lack of an implicit 1 preceding the mantissa—this allows us to store small values close to zero, called *denormal* or *subnormal* values. And when all exponent bits are one, certain mantissa values represent $+\infty$, $-\infty$, and "Not a Number" (NaN), the result of undefined or unrepresentable operations such as dividing by zero.

We'll make two observations that will prove themselves useful shortly:

1. Floats cannot store arbitrary real numbers, or even arbitrary rational numbers. They can only store numbers representable by the equations shown before. For example, if I declare some variable,

   ```
   float f = 0.1f;
   ```

   `f` becomes 0.100000001490116119384765625, the closest 32-bit float value to 0.1.

2. Since the equations are exponential, the distance on the number line between adjacent values increases (exponentially!) as you move away from zero. The distance between 1.0 and the next possible value is about 1.19×10^{-7}, but the distance between adjacent floats near 6.022×10^{23} is roughly 3.6×10^{16}. This will prove to be our greatest challenge: when comparing floats, we want to handle inputs close to zero as well as we handle ones close to the Avogadro constant.

What is equality?

Since the result of every floating-point operation must be rounded to the nearest possible value, math doesn't behave like it does with real numbers. Depending on your hardware, compiler, and compiler flags, 0.1×10 may produce a different result than $\sum_{n=1}^{10} 0.1$.* Whenever we compare calculated values to each other, we should provide some leeway to account for this. Comparing their exact values with == won't cut it.

*On my setup, the former gives 1.0 and the latter gives 1.000000119209290.

Instead, we should consider two distinct values a and b equal if $|a - b| \leq \epsilon$ for some sufficiently small ϵ. As luck would have it, the C standard library contains a `FLT_EPSILON`. Let's try it out!

```
bool almostEqual(float a, float b)
{
    return fabs(a - b) <= FLT_EPSILON;
}
```

We would hope that we're done here, but we would be wrong. A look at the language standards reveals that `FLT_EPSILON` is equal to the difference between 1.0 and the value that follows it. But as we noted before, float values aren't equidistant! For values less than 1, `FLT_EPSILON` quickly becomes too large to be useful. For values greater than 2, `FLT_EPSILON` is smaller than the distance between adjacent values, so `fabs(a - b) <= FLT_EPSILON` will always be false.

To address these problems, what if we scaled ϵ proportionally to our inputs?

```
bool relativelyEqual(float a, float b,
    float maxRelativeDiff = FLT_EPSILON)
{
    const float difference = fabs(a - b);

    // Scale to the largest value.
    a = fabs(a);
    b = fabs(b);
    const float scaledEpsilon =
        maxRelativeDiff * max(a, b);

    return difference <= scaledEpsilon;
}
```

This works better than our initial solution, but it's not immediately obvious what values of `maxRelativeDiff` we might want for different cases. The fact that we scale it by arbitrary inputs also means it can fall prey to the same rounding we're worried about in the first place.

What about Boost?

Boost, the popular collection of C++ libraries, provides functions for similar purposes.* After removing template boilerplate and edge case handling for $\pm\infty$ and NaNs, they resemble:

```
float relative_difference(float a, float b)
{
    return fabs((a - b) / min(a, b));
}

float epsilon_difference(float a, float b)
{
    return relative_difference(a, b) /
           FLT_EPSILON;
}
```

Unfortunately, these functions don't seem to solve our problems.† Since the division in `relative_difference` often makes its result quite small,‡ how do we know what a good threshold might be? By dividing that result by `FLT_EPSILON`, `epsilon_difference` attempts to give an easier value to reason about. But we just saw the dangers of `FLT_EPSILON`! This scheme becomes increasingly questionable as inputs move away from one.

What about ULPs?

It would be nice to define comparisons in terms of something more concrete than arbitrary thresholds. Ideally, we would like to know the number of possible floating-point values—sometimes called *units of least precision*, or ULPs—between inputs. If I have some value a, and another value b is only two or three ULPs away, we can probably consider them equal, assuming some rounding error. Most importantly, this is true regardless of the distance between a and b on the number line.

Boost offers a function called `float_distance` to get the distance between values in ULPs, but it's about an order of magnitude slower than the approaches discussed so far. With some bit-fiddling, we can do better.

Consider some positive float x where every mantissa bit is one. $x + 1\text{ULP}$ must use the next largest exponent, and all its mantissa bits must be zero. As an example, consider 1.99999988 and 2:

*See *Floating-point Comparison* in the floating-point utilities section of Boost's Math toolkit.

†Boost libraries are usually high-quality and thoroughly reviewed, so please contact me if I've missed some critical observation.

‡For example, the `relative_difference` between 42 and the next float value is about 9.08×10^{-8}.

Value	Bits	Exponent	Mantissa bits
1.99999988	`0x3FFFFFFF`	127	`0x7FFFFF`
2.0	`0x40000000`	128	`0x000000`

The property holds for denormals, even though they have a different value equation. Consider the largest denormal value and the smallest normal one:

Value	Bits	Exp.	Man. bits
$1.1754942 \times 10^{-38}$	`0x007FFFFF`	–126	`0x7FFFFF`
$1.17549435 \times 10^{-38}$	`0x00800000`	–126	`0x000000`

Notice an interesting corollary: adjacent floats (of the same sign) have adjacent integer values when reinterpreted as such. This reinterpretation is sometimes called *type punning*, and we can use it to calculate the distance between values in ULPs.

Traditionally in C and C++, one used a union trick:

```
union FloatPun {
    float f;
    int32_t i;
};

FloatPun fp;
fp.f = 25.624f;
// Read the same value as an integer.
printf("%x", fp.i);
```

This still works in C, but can run afoul of strict aliasing rules in C++.* A better approach is to use `memcpy`. Given the usual use of the function, one might assume that it would be less efficient, but

```
int32_t floatToInt(float f)
{
    int32_t r;
    memcpy(&r, &f, sizeof(float));
    return r;
}
```

compiles to a single instruction that moves the value from a floating-point register to an integer one. This is exactly what we want.

With that problem solved, calculating the ULPs between values becomes quite straightforward:

```
int32_t ulpsDistance(const float a, const float b)
{
    // Save work if the floats are equal.
    // Also handles +0 == -0.
    if (a == b) return 0;

    const auto max =
        std::numeric_limits<int32_t>::max();

    // Max distance for NaN
    if (isnan(a) || isnan(b)) return max;

    // If one's infinite and they're not equal,
    // max distance.
    if (isinf(a) || isinf(b)) return max;

    int32_t ia, ib;
    memcpy(&ia, &a, sizeof(float));
    memcpy(&ib, &b, sizeof(float));

    // Don't compare differently-signed floats.
    if ((ia < 0) != (ib < 0)) return max;

    // Return the absolute value of
    // the distance in ULPs.
    int32_t distance = ia - ib;
    if (distance < 0) distance = -distance;
    return distance;
}
```

This code is quite portable—it only assumes that the platform supports 32-bit integers and that floats are stored in accordance with IEEE 754.† We avoid comparing differently-signed values for a few reasons:

1. ULPs are the wrong tool to compare values near or across zero, as we'll see below.

2. Almost all modern CPUs use two's complement arithmetic, while floats use signed magnitude. Converting one format to the other in order to meaningfully add or subtract differently-signed values requires some extra work. For the same reason, the sign of our result might not be what we expect, so we take its absolute value. We only care about the distance between our two inputs.

3. If the subtraction overflows or underflows, we get undefined behavior with signed integers and modular arithmetic with unsigned ones. Neither is desirable here.

*See http://stackoverflow.com/q/11639947 and http://stackoverflow.com/q/17789928.

†The format of `float` is implementation-defined according to the C++ standard, and not necessarily adherent to IEEE 754. (See http://stackoverflow.com/a/24157568.) Perhaps this is why Boost's `float_distance` is implemented the way it is.

We calculate the absolute value ourselves instead of using `std::abs` for two reaons. First, the integer versions of `std::abs` only take types—such as `int`, `long`, and `long long`—whose sizes are platform-specific. We want to avoid assumptions about implicit conversions between those types and `int32_t`.* The second is a strange pitfall related to the placement of `std::abs` overloads in the C++ standard library. If you include `<cmath>` but not `<cstdlib>`, only the floating-point versions of `std::abs` are provided. Several toolchains I tested then promote the `int32_t` value to a `double`, even if your target only has a 32-bit FPU and must emulate `double` using integer registers. (As one might guess, this is *terrible* for performance.) Warning flags such as `-Wconversion` can help us notice this happening, or we can just avoid all these gotchas by calculating the absolute value directly. At any rate, this is a trivial detail.

No silver bullets

Relative epsilons—including ULPs-based ones—don't make sense around zero. The exponential nature of floats means that many more values are gathered there than anywhere else on the number line. Despite being a fairly small value in the context of many calculations, 0.1 is over one billion ULPs away from zero! Consequently, fixed epsilons are probably the best choice when you expect the results to be small. What particular ϵ you want is entirely dependent on the calculations performed.

Armed with this knowledge, you may be tempted to write some end-all comparison function along the lines of:

```
bool nearlyEqual(float a, float b,
        float fixedEpsilon, int ulpsEpsilon)
{
    // Handle the near-zero case.
    const float difference = fabs(a - b);
    if (difference <= fixedEpsilon) return true;

    return ulpsDistance(a, b) <= ulpsEpsilon;
}
```

But using it meaningfully is difficult without understanding the theory we've discussed.

Brief aside: Other ULPs-based functions

We can use the same techniques to write other useful functions, such as one that increments a float by some number of ULPs. Boost offers a similar family of functions (`float_next`, `float_advance`, etc.), but like its `float_distance`, they pay a performance cost to avoid type punning.

One would hope we could simply get our ULPs, perform our addition, and pun the result back, e.g.,

```
/// Increases f by the given number of ulps
float ulpsIncrement(float f, int32_t ulps)
{
    if (isnan(f) || isinf(f)) return f;
    int32_t i;
    memcpy(&i, &f, sizeof(float));
    i += ulps;
    memcpy(&f, &i, sizeof(float));
    return f;
}
```

This naïve solution works for positive values, but on most hardware, "incrementing" a negative float by a positive number of ULPs will move us away from zero! This is probably not what we want. We mentioned before that floats use a signed magnitude scheme, whereas most CPUs use two's complement. So, to operate on negative values, we need to convert from the former to the CPU's native integer format.†

```
static const int32_t int32SignBit =
        (int32_t)1 << 31;

int32_t floatToNativeSignedUlps(float f)
{
    int32_t i;
    memcpy(&i, &f, sizeof(float));

    // Positive values are the same in both
    // two's complement and signed magnitude.
    // For negative values, remove the sign bit
    // and negate the result (subtract from 0).
    return i >= 0 ? i : -(i & ~int32SignBit);
}
```

After operating on the ULPs, we must convert back to signed magnitude:

*Granted, this is borderline paranoia—`int32_t` is one of those types on nearly every relevant platform.

†On esoteric hardware, the native format may also be signed magnitude. In those cases, we trust the compiler to elide the needless work we do here.

```
float nativeSignedUlpsToFloat(int32_t ulps)
{
    if (ulps < 0) {
        ulps = -ulps;
        ulps |= int32SignBit;
    }
    float f;
    memcpy(&f, &ulps, sizeof(float));
    return f;
}
```

With those functions defined, we can return to our goal:

```
float ulpsIncrement(float f, int32_t ulps)
{
    if (isnan(f) || isinf(f)) return f;
    int32_t i = floatToNativeSignedUlps(f);
    i += ulps;
    return nativeSignedUlpsToFloat(i);
}
```

Takeaways

When comparing floating-point values, remember:

- `FLT_EPSILON`… isn't float epsilon, except in the ranges $[-2, -1]$ and $[1, 2]$. The distance between adjacent values depends on the values in question.
- When comparing to some known value—especially zero or values near it—use a fixed ϵ that makes sense for your calculations.
- When comparing non-zero values, some ULPs-based comparison is probably the best choice.
- When values could be anywhere on the number line, some hybrid of the two is needed. Choose epsilons carefully based on expected outputs.

Acknowledgments

Much of this was adapted from Bruce Dawson's *fantastic* exploration of the topic on his blog, Random ASCII. Thanks also to coworkers Evan Thompson and Matt Drees for their input.

Appendix: Performance concerns

The relatively poor performance of `boost::float_distance` was a large motivation for implementing our own `ulpsDistance`. For the sake of completeness, the following is a benchmark (using Google's benchmark library) comparing the two with a handful of inputs.

```cpp
#include <cstring> // For memcpy
#include <limits> // for numeric_limits<float>::infinity
#include <random>

#include <benchmark/benchmark.h>
#include <boost/math/special_functions/next.hpp>
#include <boost/math/special_functions/relative_difference.hpp>

using namespace std;
using namespace boost::math;

std::pair<float, float> pickInput()
{
    static auto re = mt19937(random_device()());
    static auto coinFlip = bernoulli_distribution(0.5);
    static auto inputPicker = uniform_int_distribution<int>(1, 10);

    const float infinity = numeric_limits<float>::infinity();

    switch(inputPicker(re)) {
        // Let's say there's a 5% chance our values are denormal.
        // (This is probably more pessimal than our actual data.)
        case 1:
            if (coinFlip(re)) return {1e-38f, float_advance(1e-38f, 3)};
            // Intentional fall-through

        // Let's throw in some huge numbers
        case 2:
        case 3:
        case 4:
        case 5:
            return {6.022e23f, 2.998e8f};
            break;

        // And so not-so-huge ones.
        case 6:
        case 7:
        case 8:
        case 9:
            return {1.0f, 11.0f};

        // Let's say there's a 5% chance we have NaNs
        // and another 5% chance they're infinity
        case 10:
            if (coinFlip(re)) return {42, numeric_limits<float>::quiet_NaN()};
            else return {42, infinity};

        default: assert(0);
    }
}
```

```
__attribute__((noinline)) // For visibility when benchmarking
int32_t ulpsDistance(const float a, const float b)
{
    // We can skip all the following work if they're equal.
    if (a == b) return 0;

    const auto max = numeric_limits<int32_t>::max();

    // We first check if the values are NaN.
    // If this is the case, they're inherently unequal;
    // return the maximum distance between the two.
    if (isnan(a) || isnan(b)) return max;

    // If one's infinite, and they're not equal,
    // return the max distance between the two.
    if (isinf(a) || isinf(b)) return max;

    // At this point we know that the floating-point values aren't equal and
    // aren't special values (infinity/NaN).
    // Because of how IEEE754 floats are laid out
    // (sign bit, then exponent, then mantissa), we can examine the bits
    // as if they were integers to get the distance between them in units
    // of least precision (ULPs).
    static_assert(sizeof(float) == sizeof(int32_t), "What size is float?");

    // memcpy to get around the strict aliasing rule.
    // The compiler knows what we're doing and will just transfer the float
    // values into integer registers.
    int32_t ia, ib;
    memcpy(&ia, &a, sizeof(float));
    memcpy(&ib, &b, sizeof(float));

    // If the signs of the two values aren't the same,
    // return the maximum distance between the two.
    // This is done to avoid integer overflow, and because the bit layout of
    // floats is closer to sign-magnitude than it is to two's complement.
    // This *also* means that if you're checking if a value is close to zero,
    // you should probably just use a fixed epsilon instead of this function.
    if ((ia < 0) != (ib < 0)) return max;

    // If we've satisfied all our caveats above, just subtract the values.
    // The result is the distance between the values in ULPs.
    int32_t distance = ia - ib;
    if (distance < 0) distance = -distance;
    return distance;
}

void benchFloatDistance(benchmark::State& state)
{
    while (state.KeepRunning()) {
        state.PauseTiming();
        float a, b;
        std::tie(a, b) = pickInput();
        state.ResumeTiming();
        // float_distance can't handle NaN and Infs.
        if (!isnan(a) && !isnan(b) && !isinf(a) && !isinf(b)) {
            benchmark::DoNotOptimize(float_distance(a, b));
        }
```

```
    }
}
BENCHMARK(benchFloatDistance);

void benchUlps(benchmark::State& state)
{
    while (state.KeepRunning()) {
        state.PauseTiming();
        float a, b;
        std::tie(a, b) = pickInput();
        state.ResumeTiming();
        benchmark::DoNotOptimize(ulpsDistance(a, b));
    }
}
BENCHMARK(benchUlps);


BENCHMARK_MAIN();
```

On my laptop (an Intel Core i7 Skylake), I get:

```
Benchmark                   Time           CPU Iterations
------------------------------------------------------------
benchFloatDistance        717 ns        836 ns     850424
benchUlps                 157 ns        176 ns    3914780
```

And on an ARMv6 board we use at work for embedded Linux platforms, I get:

```
Benchmark                   Time           CPU Iterations
------------------------------------------------------------
benchFloatDistance      43674 ns      42609 ns      16646
benchUlps                4748 ns       4602 ns     151382
```

Actual timing values obviously depend on the type of inputs, the uniformity of the inputs (which influences branch prediction), and *many* other factors, but our function seems to outperform Boost alternatives in the general case.