Geometric Robustness

jodavis42@gmail.com

The focus of this presentation is on robust geometry computations, primarily in the light of floating point numbers. The main takeaways I want you to get from this presentation is why computational geometry has to go to great lengths to deal with the issues of representing real numbers on computers. Mixed in with this I want you to see several scenarios that can break if you aren't careful, many of which we have already run into.

Geometric Robustness (With Floats)

Why do floating point numbers cause problems?

Why does point vs. plane actually need an epsilon?

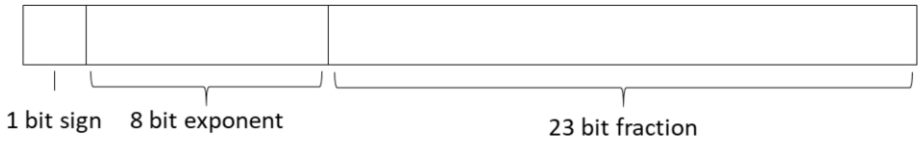
Sure, floating point numbers have limits but why do they cause the problems we've seen so far and how do we fix them?

Have you ever stopped to wonder why various floating point operations break? For instance, why do we need an epsilon with point vs. plane? It might make a bit of intuitive sense that it's really hard to get a point to be exactly on the plane, but is the reason even deeper than this?

To show this I first need to go into how floating point numbers work and some of the ramifications of this behavior.

Floating Point Numbers

IEEE754 Standard – 32 bit floats



Numbers are give by:

$$(-1)^{sign} * (1.fraction) * 2^{exponent-127}$$

Most examples will be in base 10 to make them easier:

$$(-1)^{sign} * (fraction) * 10^{exponent} = (-1)^{sign} * fraction * e^{exponent}$$

We've probably all seen the iee754 standard for 32 bit floats before. We use 32 bits to represent 3 different parts, the sign, exponent and fraction (often incorrectly called the mantissa).

With these bits we can represent a number in any base as: $(-1)^{sign} * (1.fraction) * B^{exponent-127}$ where B is the target base. Note here the implicit 1 in the fraction (which isn't always there but for simplicity let's assume so). As we can use any base I'll be using base 10 for most examples as it makes understanding the math significantly easier.

Floating Point Numbers

Basic examples:

$1{,}300{,}000 = 1.3e^{6}$

$0.000154 = 1.54e^{-4}$

$1{,}234{,}567 = 1.234567e^{6}$

Etc…

I won't be going into the specifics of converting base-10 to floats but I will show some basic examples to make sure it all roughly makes sense.

Floating Point Precision

What if we can only store a limited number of digits (4)?

$1{,}300{,}000 = 1.3e^{6}$

$0.000154 = 1.54e^{-4}$

$1{,}234{,}567 = 1.234567e^{6} = 1.235e^{6}$

We rounded since we could only store a limited number of digits

Unfortunately we don't have an infinite amount of memory to store digits so let's pretend we can only store 4 digits total (1 before the decimal place and 3 after). Now let's look at our previous examples again. The first two don't change but we have to shorten the third one. I chose to round here as that's the default method used by most processors. Note that we can store big numbers or small numbers but we can't store many digits!

Floating Point Precision – Arithmetic

Look at some basic arithmetic operations:

$$1{,}300{,}000 + 1{,}300{,}000 = 1.3e^{6} + 1.3e^{6} = 2.6e^{6}$$

$$0.000154 + 0.000154 = 1.54e^{-4} + 1.54e^{-4} = 3.08e^{-4}$$

Nothing odd so far…

$$\begin{aligned} 1{,}300{,}000 + 0.000154 &= 1.3e^{6} + 1.54e^{-4} \\ &= 1.300000000154e^{6} \\ &= 1.300e^{6} \end{aligned}$$

What?

Now let's do some basic arithmetic operations. You can see that are first two tests of adding numbers together worked out just fine, the third however seems a bit strange. In fact, if pay attention you'll notice that our result is actually the same as the first operand, we didn't actually add anything at all!!

Floating Point Precision – Arithmetic

The precision of floating point numbers is relative!

We can store large or small numbers, just not together

Adding large and small numbers together breaks!

So let's think a bit about what happened. We can store really large numbers just fine and we can also store really small numbers with no issues but as we've already seen we can't store a lot of precision with either. When we added our two numbers together we got a number that required too much precision to store. After storing the most significant digits and chopping off the rest it turns it that the operation produced no change whatsoever.

This is one of the first key things to realize with floating point numbers, most operations don't work well when large and small numbers mix together!

Floating Point Precision – Arithmetic

Multiplication has the same problem

$$1.54e^2 \times 1.54e^2 = 2.3716e^4 = 2.372e^4$$

In general multiplication requires twice the precision for the result

As should be obvious, multiplication has the exact same problem. Multiplication has it worse as multiplication of two similar magnitude values can often require twice as much precision.

Floating Point Precision - Associativity

This means arithmetic is not associative!

$$(A + B) + C \neq A + (B + C)$$

$= (1.3e^5 + 5.0e^1) + 5.0e^1$
$= 1.30050e^5 + 5.0e^1$
$= 1.300e^5 + 5.0e^1$
$= 1.30050e^5$
$= 1.300e^5$

$= 1.3e^5 + (5.0e^1 + 5.0e^1)$
$= 1.3e^5 + 1.0e^2$
$= 1.301e^5$

We get either 130,000 or 130,100 depending on grouping!

This unfortunately means that there's a number of common arithmetic properties that we take for granite that don't hold, the simplest one is associativity. Due to loss of precision of intermediary results we can have completely different answers!

Floating Point Precision - Associativity

Multiplication inverse also doesn't hold: $\frac{A}{B} \neq A * \left(\frac{1}{B}\right)$

Let's try: 3000/3

$$= \frac{3.0 * 10^3}{3.0 * 10^0}$$
$$= 1.0 * 10^3$$

$$= 3.0 * 10^3 \times \frac{1 * 10^0}{3.0 * 10^0}$$
$$= 3.0 * 10^3 \times 3.333 * 10^{-1}$$
$$= 9.999 * 10^3$$

We get either 1000 or 999!

The property of having a multiplicative inverse doesn't hold as well as there is a difference between dividing by a number and multiplying by that number's inverse!

Floating Point Precision – Properties

Some various properties that don't hold:

- Associativity
- Distributivity
- Multiplicative inverse
- Transitivity (with epsilons)
- It gets worse at the end…

This unfortunately means a number of our every day expectations fall short. There are a few more properties I can't go into until I get to the end so just wait and see it get even worse!

Binary Representation

Some ramifications of a base 2 system:

Not all numbers can be represented properly:

0.3 is actually 0.300000012…

We can only represent so many unique decimal digits

A 23 bit mantissa can only represent about 7

I said I was doing all examples in base 10 instead of base 2 because it all played out the same, but that was a slight lie. There's a few small things to talk about with the base 2 representation. The simplest is that not all numbers can be represented. We're already used to this as $\frac{1}{3}$ is represented as 0.333… in base 10. Similar problems show up in base 2 just not one's we're used to.

The other interesting property is that of unique representation of decimal numbers. While floating point numbers can represent a ton of precision, it can only uniquely represent about 7 decimal digits. We have precision beyond that it's just we won't be able to represent all numbers that require 8 digits.

Geometry Issues

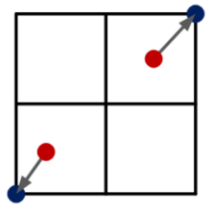
Now we know what floats do and why

How and why does this break our geometry calculations?

Now that we've taken a look at the underlying structure of floating point numbers we can take a look at how it breaks our geometry calculations. This is a random collection of things we've already seen and had to deal with so far but now we can learn why.

Geometry Issues

First think of the numbers as a uniform grid (this is wrong, more later)



Points snap to the nearest grid value

The simplest place to start is to look at what happens when we try to represent a point. To give some spatial meaning to this think of the number lines as defining a uniform grid (due to the limited precision of floats). If we try to define a point on this grid it'll have to snap to the nearest value, causing some loss of information.

Geometry Issues

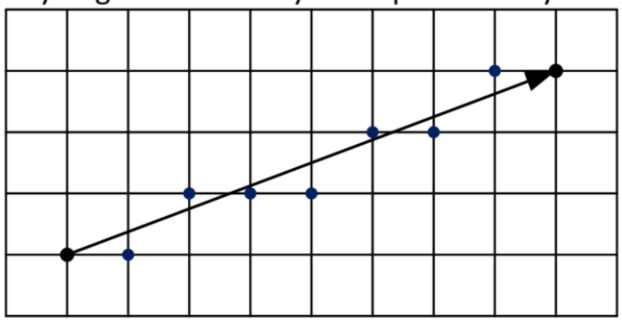
This only happens when we convert from decimal to floats right?

What simple cases break despite having exact float representations?

The previous case doesn't actually come up much directly as it's effectively when we load decimal values into floating point registers. If we ignore this issue and we assume we're always working with grid aligned values do more issues show up? But of course!

Geometry Issues

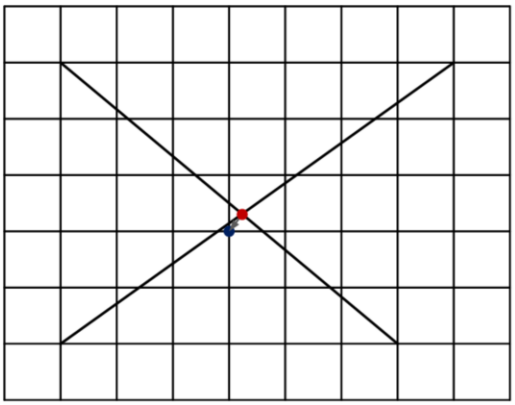
Values of a ray might not actually end up on the ray



The simplest example to think of is walking a line segment where both end points have exact floating point representations. Note that every possible value between the two points is not actually on the floating point grid. Any in-between value we try to compute will not be on the original line!

Geometry Issues

The intersection point of two lines also is incorrect



Another common example is the intersection point of two lines. This scenario is even more likely to not have an answer that lies on the number grid.

Geometry Issues – Irregular Spacing

Are floats actually a uniform grid?

What is the distance between two floats? Constant or changing?

Simple trick to check: reinterpret as int, add 1, cast back to float

Value	Epsilon
0	$1.4e^{-45}$
1	$1.2e^{-7}$
1,000	$6.1e^{-5}$
1,000,000	0.0625
1,000,000,000	64

This is how we find a number one ULP (units in last place) away

Let's take a step back and think, is it correct to think of the floating point number line as a uniform grid? How might we determine this? With integers it makes sense to define a grid where the value between 0 and 1 is 1, between 1 and 2 is 1 and so on. Is this true for floats? Is the distance between two floats constant (uniform) or not?

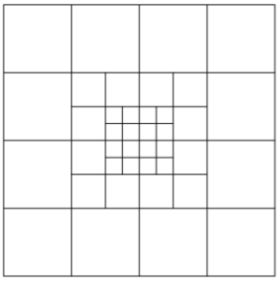
Well there's a neat trick that you can do to find the closest float to another. All floating point numbers (positive) increase with their binary representation. We can simply convert our floating point number to an integer, add 1, and convert it back to determine the nearest neighbor. We can do the opposite (find the difference between the two integer representations) to measure what is known as ulps or units in the last place. This is one measure of how close two floating point numbers are.

If we do this we can see how the epsilon between the nearest number changes as we move over the number line. It should be obvious now that the distance between two floats is anything but constant. In fact at a certain point we are no longer able to represent units of less than 1 (at around 8 million).

Geometry Issues – Irregular Spacing

The number line is actually a non-uniform grid!

1D representation



2D representation

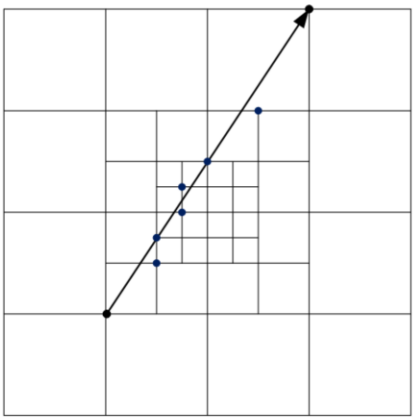
How are the previous examples affected by this?

So what does this irregular spacing mean for our visual representation? Well in 1-d it means we just have a set of non-uniformly spaced points. The real important one to think about is the 2d grid. The 2d grid is a non-uniform grid that is denser near the origin and gets larger the further away we go.

With this information we need to investigate our previous examples to see what this changes!

Geometry Issues – Irregular Spacing

The further from the origin the more off points get!

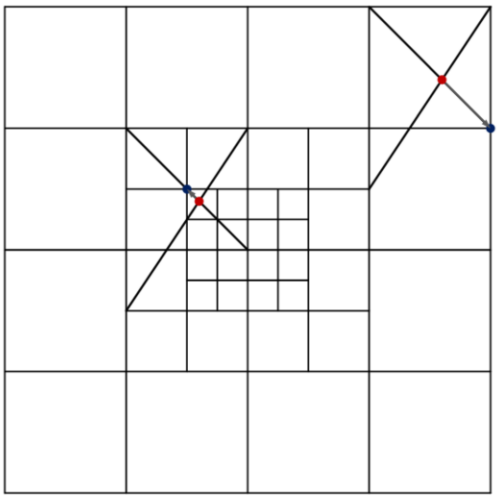


A ray is simple to see that the further away we move the worse results can get. We might still get lucky and have several values line up on the grid even very far away, but it's much less likely

Geometry Issues – Irregular Spacing

The same two lines at different positions can produce greatly different results!

We have more precision near the origin!



The more interesting case to look at is the intersection of two lines. If we look at the left two set of lines we'll see that the result is a little ways off from where it should be, however if we look at the right set of lines we'll see that the result is incredibly far away. In fact, this is a degenerate result and we get back one of our input points!

This brings us to the conclusion that the same set of data can produce a very different result depending on where it lines up on the grid! This is one reason why many geometry tests first translate to one object's local coordinate frames and perform the test there so they can make use of the precision near the origin.

Geometry Examples

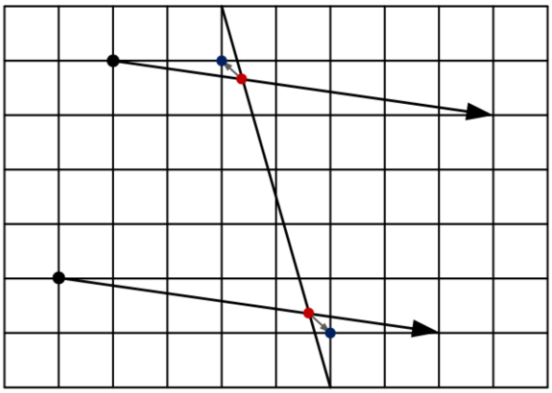
Now for some more detailed examples from this class

I'll mostly use a uniform grid because it's easier to work with

Just remember it's not uniform

Ray Plane

The same ray at different locations can end up behind or in front of the plane



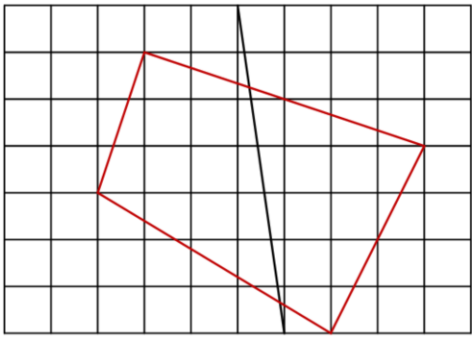
*this is even ignoring that t is snapping to a different grid!

The easiest examples to look at involve planes and lines. In particular if we look at ray plane we'll see that the computed intersection point is often not on the number grid and must be snapped. This is the primary reason why we will never get an actual coplanar result. That being said, the same ray can produce a point that ends up on different sides of the plane depending on how it snaps. This can cause tons of problems in any algorithm that needs to have consistent results.

Technically the snapping is a lot worse than it seems here. First we actually have a non-uniform grid, but even worse we have a second grid for t. Remember t is based upon the distance from the ray's start meaning that it's another grid that will get snapped differently.

Polygon splitting

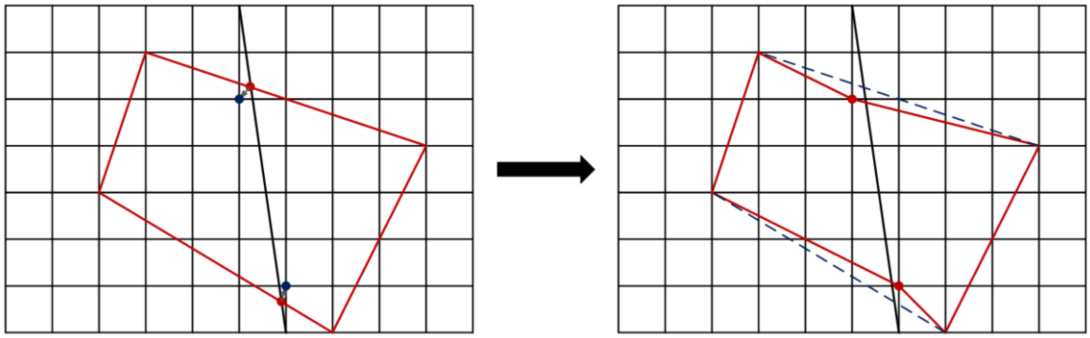
Split this polygon against the input plane



Another common example of numerical robustness issues is polygon splitting.

Polygon splitting

The computed points will snap to the grid

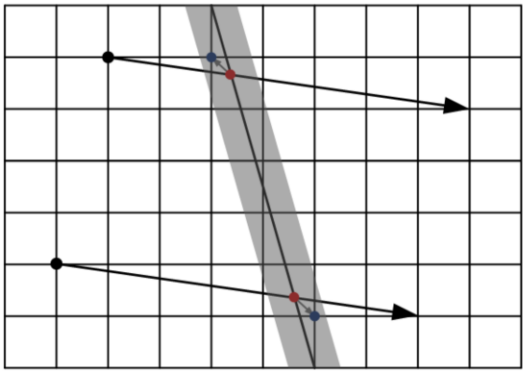


Note that the split points end up on opposite sides of the plane

When we split a polygon we end up doing the same operation as ray vs. plane so we will get the same thing happening. In this case not only does our polygon deform slightly (no real way around this) but we end up with both polygons crossing the plane. This means any test that needs consistent results (bsp tree raycasting) will not test polygons correctly.

Thick Planes

Thick planes produce a consistent classification of points



As we've already seen, the way to deal with this issue is to work with thick planes. Thick planes make it so that any points that happen to be snapped to the floating point grid all get classified as coplanar. With this classification will be consistent and all expected invariants will hold true.

Shared Calculations

If a calculation isn't properly shared errors can occur

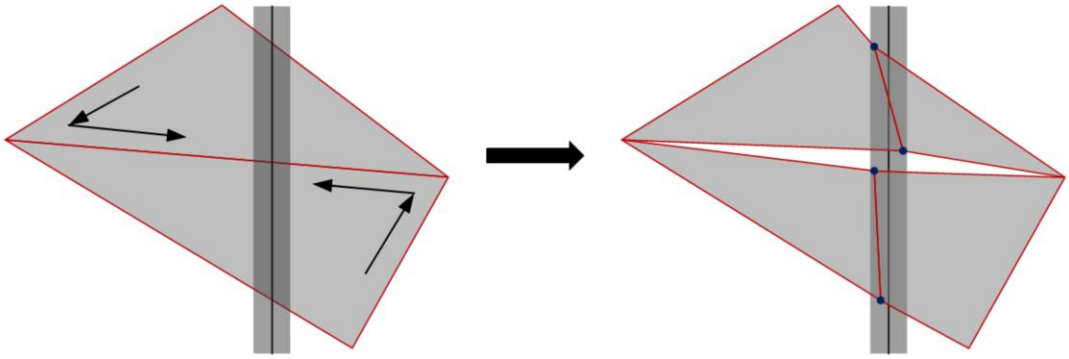
The simplest example is when an edge is shared between two triangles

Unfortunately not all problems can be solved by thick planes (basically epsilons). The other common problem is caused by calculations that should be shared but are not.

Polygon Splitting – Edge Order

The edge order is different when splitting the triangles

Although the plane's invariant holds true a gap shows up



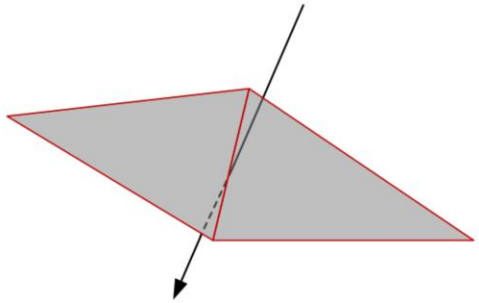
One such problem is when splitting triangles. Commonly two triangles share an edge, but as both triangles are counter-clockwise ordered we'll traverse their shared edge in different orders (from A to B vs. B to A). While this might not seem like much of a problem it can cause minor differences numerically. These differences can cause the same line to produce two value that snap differently which can cause small gaps to open up in the mesh. While using a thick plane guarantees that the plane's invariant holds, it doesn't fix this problem.

The root problem shows up from the ray vs. plane equation: $t = \frac{\vec{n}\cdot(\vec{p}_0 - \vec{s_r})}{\vec{n}\cdot\vec{d}}$. In this equation we have two differences: one is that $\vec{d} = \vec{A} - \vec{B}$ instead of $\vec{d} = \vec{B} - \vec{A}$. The other is that $\vec{s}_r = \vec{A}$ or $\vec{s}_r = \vec{B}$. It's not too hard to craft a small example where different t-values are computed.
The only real solution here is to guarantee a consistent edge ordering for tests.

Ray Triangle

A ray can pass "between" two triangles

Happens because of non-shared edge information



Another fun example we've seen is determining if a ray hits a triangle. By bizarre chance we can have a ray that goes through a triangle's edge that returns false to both triangles. This is also due to the winding order difference of the edges. One solution is to use fat objects, for instance we used a fattened triangle to deal with this (expanding the barycentric range check).

Sphere Expansion

Did you try to validate your spheres from assignment 2?

Did you notice points that weren't contained?

Small float differences can cause an expanded sphere to not contain a point

Another fun one I ran into that I haven't really seen in literature is sphere expansion. When validating my results for assignment 2 I found that occasionally a point wasn't quite within the sphere even though I just expanded it to contain that point. The difference was only somewhere around $1e^{-6}$ but it was still there. This could cause problems if you're trying to create a tight bounding sphere to contain your object (but likely only small visual artifacts).

Epsilons

How do we determine if two numbers are equal?

$a == b$?

No! Bad! Almost never true!

How about we test if they're close enough?

We've now seen how most operations need some kind of an epsilon, but how do we know what epsilon to choose? To determine what's a good epsilon we need to determine how we could even compare two numbers for equality.

It should be obvious by now that directly comparing for equality is bad. As we've seen from all of the arithmetic properties we've assumed, to the approximations in geometry calculations, we'll almost never get an exact comparison of results. This leads us to checking to see if they're close enough somehow.

Epsilons – Absolute Tolerance

$if(Abs(a - b) \leq epsilon)$

What do we use for epsilon?

- 0.0001?
- FLT_EPSILON?

This leads us to write a very simple test to determine if the difference between are two numbers is below some threshold. What threshold should we use though? We could pick some small number that'll deal with most cases like 0.0001. We could also use the already defined machine epsilon of floating point numbers…

Epsilon – Absolute Tolerance

Remember the epsilon table?

Absolute tolerances are bad! (I was lazy in the framework…)

Big numbers need a larger epsilon

This should immediately raise a red flag. Remember in the epsilon table where large numbers (like 1,000,000) can't even represent a difference that small? This should immediately lead us to realize that epsilon is actually dependent somehow on the size of our numbers.

Epsilon – Relative Tolerance

Epsilon needs to be scaled by the input size:

$$bool\ isEqual = \Big(Abs(a-b) \leq \epsilon * Max\big(Abs(a), Abs(b)\big)\Big)$$

The idea is to determine if the difference of the two numbers is smaller than some percentage of the larger of the two numbers.

Does this work for all numbers?

The common method to make a relative tolerance is to scale the machine epsilon (or some other small epsilon) by the largest input number. Unfortunately this doesn't work for everything…

Epsilon – Relative Tolerance

Try comparing 67329.234 and 67329.242 (exactly one ulps away)

Our relative test gives true, great!

What if we subtract and compare against zero?

$$67329.242 - 67329.234 = 0.0078$$

How close is 0.0078 to zero?

Unfortunately the relative tolerance has one major flaw. It's very common to write a test that doesn't check if $a == b$ but instead checks if $(a - b) == 0$. So what if we have two numbers very close together, so close in fact that no number can be represented between them? Our current relative tolerance will return true which is great, but what if we perform the subtraction and compare the result against zero. Well in this case the subtraction gives 0.0078 which is a decently small number, right?

Epsilons – Combined Tolerance

0.0078 is 1,006,632,960 ulps away from zero…not so close after all…

Unfortunately, there is no "answer"

- The most common approach is to mix absolute and combined tolerances
- Sorry…

Unfortunately 0.0078 is insanely far away from zero, like there's a billion numbers in between…

The only real way to do this is to do some kind of absolute tolerance check but the tolerance is hard to know. In fact, this basically has to be a special check depending on what our inputs are…

Infinity Arithmetic (IA)

How does the fpu deal with exceptional cases without throwing exceptions?

There are 2 basic values: $\pm INF$ and NAN

The last thing I want to go over is infinity arithmetic, that is what happens with weird edge cases like 1/0? The fpu could throw an exception just like integer divisions but that's disabled by default. Instead the fpu sets certain special patterns that have very specific rules for how they interact with other numbers. In particular, there's 2 primary sets of numbers, infinity and not-a-number.

Infinity Arithmetic - $\#INF$

$\#INF$ comes from overflows or expression limits:
$\frac{1e^{20}}{1e^{-20}}, \frac{1}{0}, etc \ldots$

Many operations with $\#INF$ result in either $\#INF$ or 0

$\frac{1}{\infty} = 0$
$\infty + x = \infty$
etc…

#INF comes up in 2 main cases. The first is when there's an overflow to a number that can't be represented such as dividing a really large number by a really small one. The second case requires some basic calculus understanding. If you take an expression with zero replace it with x such as $\frac{1}{x}$ and evaluate the limit as $x \to 0$ you'll see that the expression approaches infinity.

The question is how does infinity play with other values? In many cases it produces either infinity again or zero. To determine which one just replace ∞ with x and take the limit as $x \to \infty$.

Infinity Arithmetic - $\#NAN$

If an expression can't be evaluated or the limit doesn't exist then the result is not-a-number:

$$\sqrt{-1}, \frac{\infty}{\infty}, \frac{0}{0}, 0 * \infty, etc \ldots$$

Similarly, #NAN occurs when an expression can't be evaluated or a limit doesn't exist. The simplest example is the square root of a negative number, but there are lots of other examples, mostly involving 0 or ∞.

Infinity Arithmetic - $\#NAN$

$\#NAN$ is like a virus, anything it touches becomes $\#NAN$

$\#NAN$ doesn't compare true with anything:

$\#NAN \neq 0$
$\#NAN \geq 0 \rightarrow False$
$\#NAN < 0 \rightarrow False$
$\#NAN \neq \#NAN$

#NAN is a very terrible thing to have floating around in your calculations; anything it touches becomes #NAN. #NAN is also really weird and doesn't compare like normal. In fact, any comparison involving #NAN will return false.

Infinity Arithmetic - $\#NAN$

Are these equivalent?

```
if(a < b)
    DoA();
  else
    DoB();
```

=?

```
if(a >= b)
    DoB();
  else
    DoA();
```

Not with $\#NAN$!

Because of this property some seemingly simple changes that ***should*** be equivalent aren't!

Infinity Arithmetic

Takeaways:

- IA is complicated
- Don't rely on it if you can avoid it
- There are also small performance ramifications in using them

The one thing I really want you to take away from this section on IA is that it's much harder to deal with than you think. You should not be relying on this in any of your logic, take the (very) small performance hit of checking where possible. In fact, this is why I turn on fpu exceptions because I do not want these values to ever show up!

Questions?

References

Bruce Dawnson's blog:
https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

What Every Computer Scientist Should Know About Floating-Point Arithmetic:

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Christer Ericson's Geomtry Talk:
http://gamedevs.org/uploads/numerical-robustness-geometric-calculations.pdf