Optimizing Sin/Cos

For SwiftShader

Nicolas Capens <capn@google.com>

Introduction

SwiftShader is a conformant implementation of the Vulkan graphics API which runs entirely on the CPU. We've identified trigonometric functions, and in particular *sin* and *cos*, to require optimization.

Objective

Vulkan specifies that the single-precision (32-bit) result of the `sin()` and `cos()` shader operations must have an "Absolute error ≤ 2^{-11} inside the range [−π,π]." That's a tolerance smaller than 0.0005.

SwiftShader historically had a cheap approximation based on a parabolic curve and a weighted average with the square of this curve. Unfortunately this was only precise to about 0.001. It was replaced by an implementation derived from A Fast, Vectorizable Algorithm for Producing Single-Precision Sine-Cosine Pairs by *Marcus H. Mendenhall*. It is highly precise but takes 18 multiplications, a division, and several other operations.

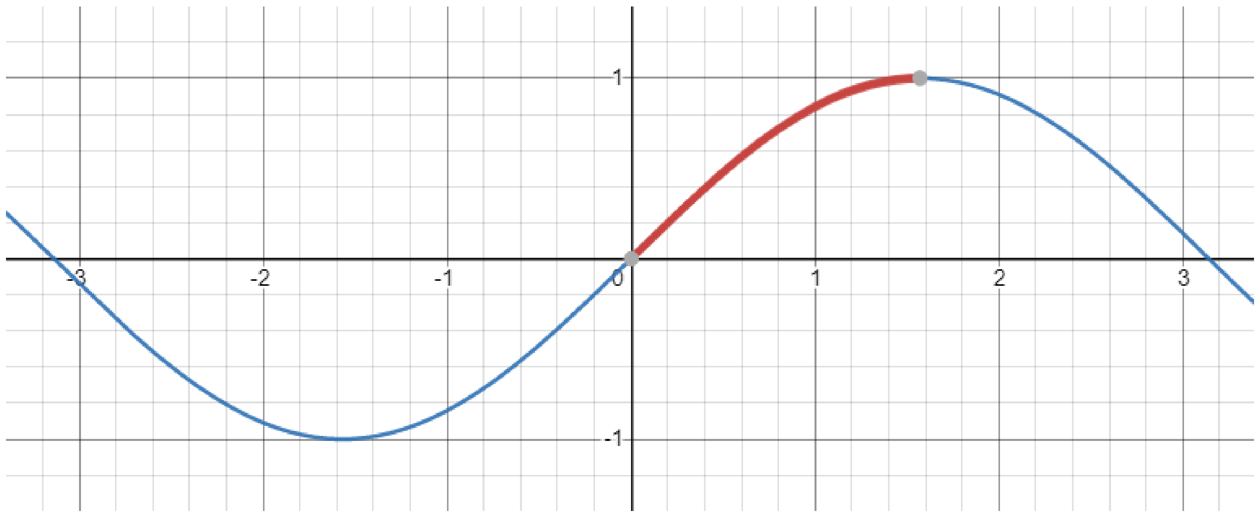
A new implementation with precision in between these two will also have to be vectorizable to take advantage of wide SIMD.

Polynomial Approximation

Mendenhall's algorithm is based around a rational function, using polynomials of 7th and 6th degree. So we could consider using a rational function of smaller degree. Bhaskara I's sine approximation formula uses a rational function which requires only three multiplications total, but has an precision of just 0.0016. So a rational function of slightly higher degree may look promising.

Unfortunately, divisions are very expensive compared to multiplication. Intel's latest CPUs can do 20 fused-multiply-add operations in the same time as one division. This means even Bhaskara I's seemingly simple formula is actually not even two times faster than Mendenhall's.

Instead we can consider ordinary polynomials. Note that due to the symmetry of sine and cosine, we could consider approximating just one 'quadrant' from $x = 0$ to $\pi / 2$:



It’s important to note that taking advantage of this symmetry takes additional instructions as well. If we have a generic polynomial approximation for the segment highlighted above, we still need to produce negative results for two of the other segments. Doing so requires either conditionally multiplying by -1 or copying the sign from another operand. Neither operation can be done in a single instruction on any known x86 CPUs. It may seem premature to worry about that, but bear with me.

Due to this issue it is useful to have an approximation which is symmetrical around the origin, i.e. $f(-x) == -f(x)$. Functions with this property are called odd functions, and polynomials with this property use only odd powers of x. The sine wave is an odd function, but cosine is not. An approximation of the cosine for all x between $-\pi/2$ and $\pi/2$ would still not produce negative numbers. It is therefore better to find a solution for sine and then use the identity $cos(x) = sin(\pi/2 - x)$ rather than try to approximate cosine and implement sine as $cos(\pi/2 - x)$.

The question now becomes whether there’s a cheap but good approximation of $sin(x)$ between $-\pi/2$ and $\pi/2$ using an odd polynomial. Or do we lose precision compared to a generic polynomial of comparable computational complexity if we approximate just one quadrant with it and deal with the oddness issue separately?

A polynomial of n terms has n coefficients, and using a system of linear equations any polynomial with n coefficients can satisfy n conditions. These conditions can be things like $f(a) = sin(a)$, meaning we can make the polynomial exactly match the function we’re trying to approximate, at n points. The conditions can also be other things like precisely matching the derivative (i.e. slope) at certain x. Either way, each condition greatly improves the ‘fit’ of the approximation, and for n conditions we need a generic polynomial of degree $n - 1$, or an odd polynomial of degree $2n - 1$.

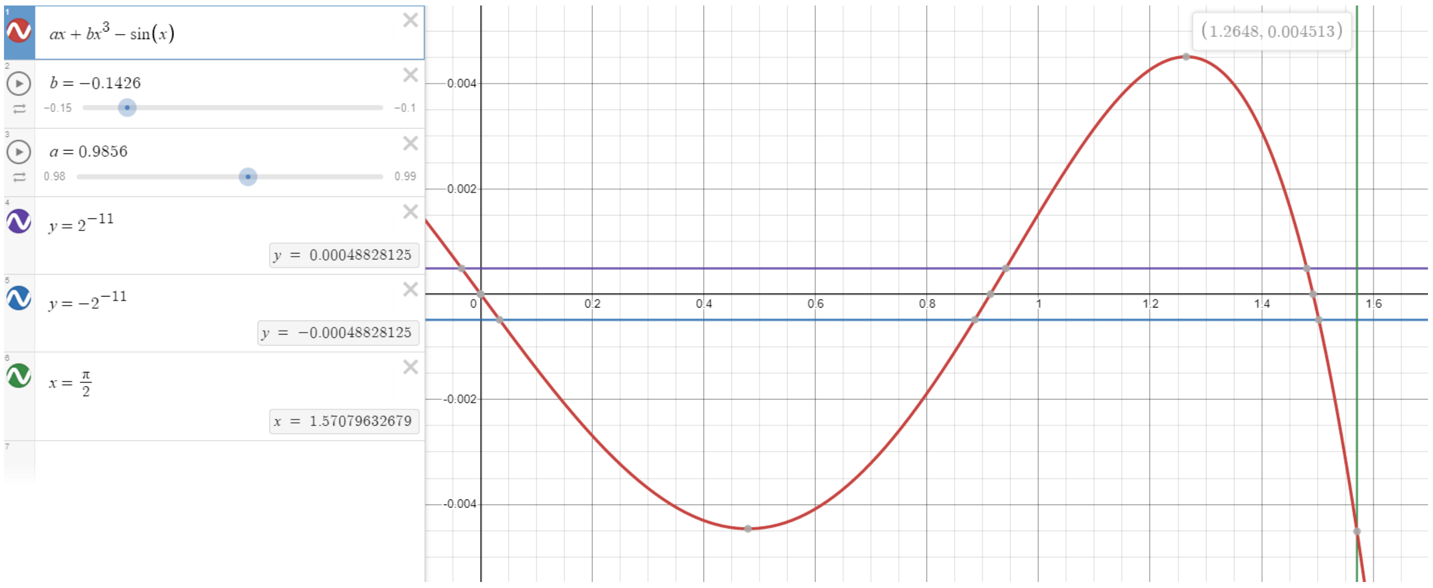
Using Horner’s method, computing the result of a n’th degree polynomial only takes n multiplications. This appears to strongly favor not restricting ourselves to odd polynomials.

Fortunately, Horner’s method can easily be adapted to include x^2, which only has to be computed once: $x(a + x^2(b + x^2(c + x^2(...))))$. Thus the cost for n conditions is only $n + 1$ multiplications.

But wait, there’s more! With odd polynomials we already get one essential condition for free: $f(0) = 0$. So really for the cost of one multiplication we get the symmetry around the origin that we were after, versus the multiple instructions it would have otherwise taken.

Fitting the Curve

Using the [Desmos](https://www.desmos.com) online graphing calculator, I used a 2-term odd polynomial ($ax + bx^3$) where the coefficients are controlled by 2 sliders, to visually prove it’s not possible to achieve Vulkan’s precision requirements:


1 $ax + bx^3 - \sin(x)$

2 $b = -0.1426$ -0.15 -0.1

3 $a = 0.9856$ 0.98 0.99

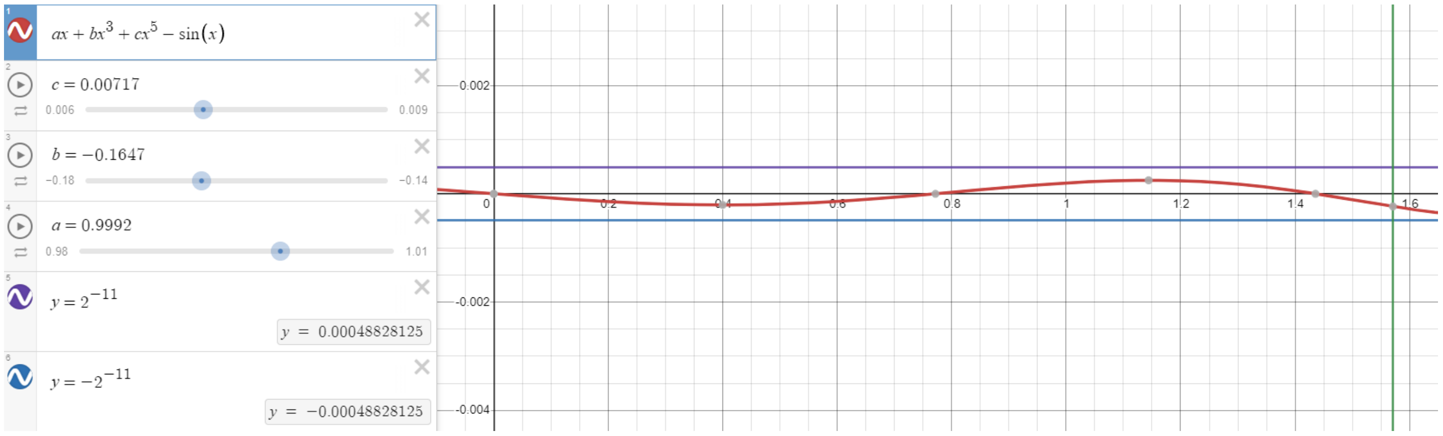
4 $y = 2^{-11}$ — $y = 0.00048828125$

5 $y = -2^{-11}$ — $y = -0.00048828125$

6 $x = \frac{\pi}{2}$ — $x = 1.57079632679$

The red graph is the difference between our approximation and $sin(x)$, while the purple and blue lines are the allowable error limits. Note this optimal result (for the absolute error) even sacrifices the condition that $f(\pi/2) = 1$, which some applications might expect even though Vulkan does not explicitly guarantee it (or demand it, depending on your point of view).

But once we use a 5’th degree odd polynomial…


$ax + bx^3 + cx^5 - \sin(x)$

$c = 0.00717$

$b = -0.1647$

$a = 0.9992$

$y = 2^{-11}$

$y = 0.00048828125$

$y = -2^{-11}$

$y = -0.00048828125$

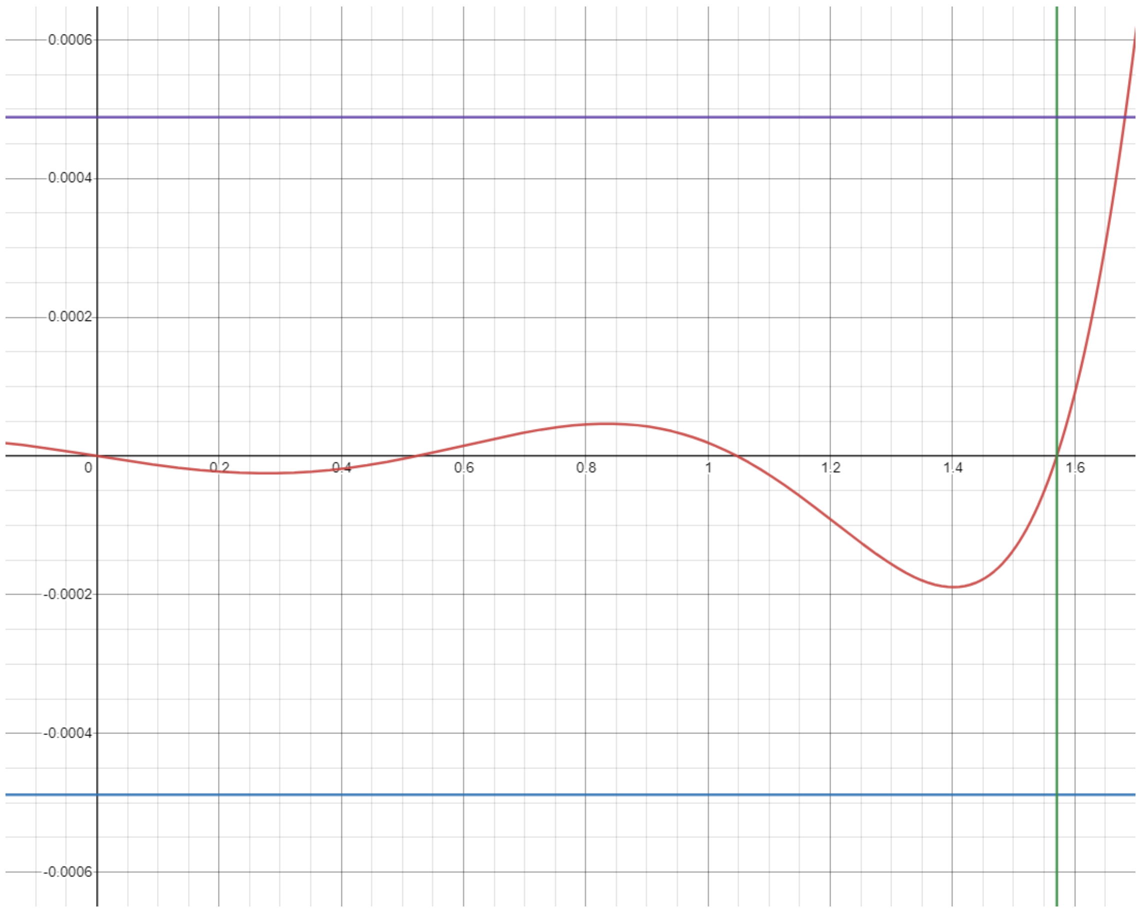
Success! With some *fingerspitzengefühl* we can fit the error between the tolerance boundaries.

Note this example crosses the x-axis twice (besides the 'free' one at $x = 0$). With three coefficients, we can actually satisfy three precise conditions, but note there's no guarantee the error would remain low enough. Also, picking specific conditions and trying to achieve them with an interactive graphing calculator is practically futile.

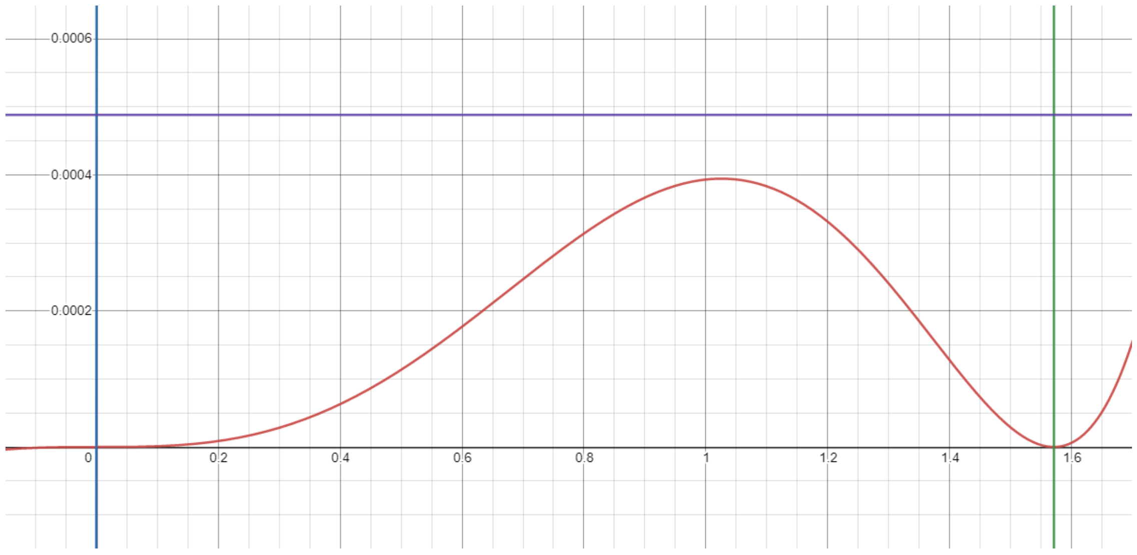
This is where we have to resort to algebra. Symbolab offers a nice online system of equations calculator. Again note that as far as Vulkan is concerned, we've already reached our goal, so whichever conditions we choose that result in an approximation that still fits its criteria, are a bonus. Let's see if we can get at least the aforementioned $f(\pi/2) = 1$. That is, use $a(\pi/2) + b(\pi/2)^3 + c(\pi/2)^5 = sin(\pi/2)$ as our first equation.

For the second and third condition I think either $f(\pi/6) = sin(\pi/6)$ and $f(\pi/3) = sin(\pi/3)$, or $f'(0) = 1$ and $f'(\pi/2) = 0$ are the most interesting. The first pair of conditions equally spaces out where the error graph crosses the x-axis, while the second pair ensures the approximation's slopes match that of $sin(x)$ at the start and end of our quadrant. Enter the three equations into Symbolab and we get, respectively:

1) $a = \frac{47-9\sqrt{3}}{10\pi}$, $b = \frac{72\sqrt{3}-135}{2\pi^3}$, $c = \frac{1134-648\sqrt{3}}{5\pi^5}$



2) $a = 1, b = -\frac{8\pi - 20}{\pi^3}, c = \frac{16\pi - 48}{\pi^5}$



Both still meet Vulkan's criterion! The former even has practically half the maximum error allowed by Vulkan. And that's not the best possible result. Indeed algorithms exist that minimize the maximum error. There's many other criteria that one may want to go after; minimizing the maximum *relative* error, balancing the positive and negative error, keeping the values between $[-1, 1]$ bounds, etc. Solution (1) above has a nice blend of desirable qualities, so I'll use that in the rest of this document.

To recap this result in the form of code, we now have:

```
// Polynomial approximation of degree 5 for sin(x) in the range [-pi/2, pi/2]
float sin5(float x)
{
    // A * x + B * x^3 + C * x^5
    // Exact at x = 0, pi/6, pi/3, pi/2, and their negatives
    constexpr float A = 0.999860466f;
    constexpr float B = -0.165971905f;
    constexpr float C = 0.00760152191f;

    float x2 = x * x;

    return x * (A + x2 * (B + x2 * C));
}
```

Note that functions like `sqrt()` and `pow()` are not `constexpr`, so we have to precompute the coefficients ourselves. The full cost of this function is four multiplications and two additions, and the latter could be fused so it's just four instructions. With two FMA units per core, this takes two clock cycles (reciprocal throughput).

Odd Even Powers

The above result is great but before we move on to turning it into an approximation of $sin(x)$ which can take a wider range of input values, let's see how we could produce higher or lower precision results at minimal cost. For example Vulkan requires a precision of merely 2^{-7} for 'half precision' calculations.

If you look closely at SwiftShader's legacy low-precision $sin(x)$ implementation, you'll notice a little trick. The approximation is based on a parabola, which is a 2nd degree even polynomial, but the function is made odd by using $x \cdot abs(x)$ instead of x^2. Can we do the same for higher degree polynomials? Certainly:

$$ax + bx|x| + cx^3 + dx^3|x| + \dots$$

We can even apply Horner's method by observing that $|x|^2 = x^2$:

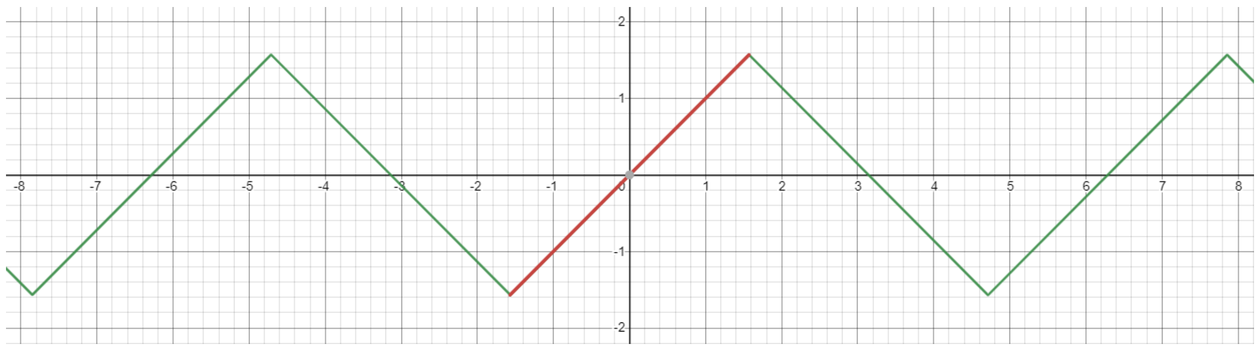
$$x(a + |x|(b + |x|(c + |x|(d + \dots))))$$

The significance of this is that we can now have n coefficients for a cost of n multiplications, instead of $n + 1$. The only added cost is that of an `abs(x)`, which on most CPU can be handled by execution ports other than the ones for multiplication, and at lower latency.

The minimum maximum absolute error that can be achieved for $ax + bx^3$ is 0.0057, while for $ax + bx|x| + cx^3$ it is 0.0019. Fortunately both are below the 0.0078 tolerance for Vulkan's half-precision requirement, but the former formula does not meet this criterion when requiring that $f(\pi/4) = sin(\pi/4)$ and $f(\pi/2) = sin(\pi/2)$, whereas the latter allows for $f(\pi/6) = sin(\pi/6)$, $f(\pi/3) = sin(\pi/3)$, and $f(\pi/2) = sin(\pi/2)$.

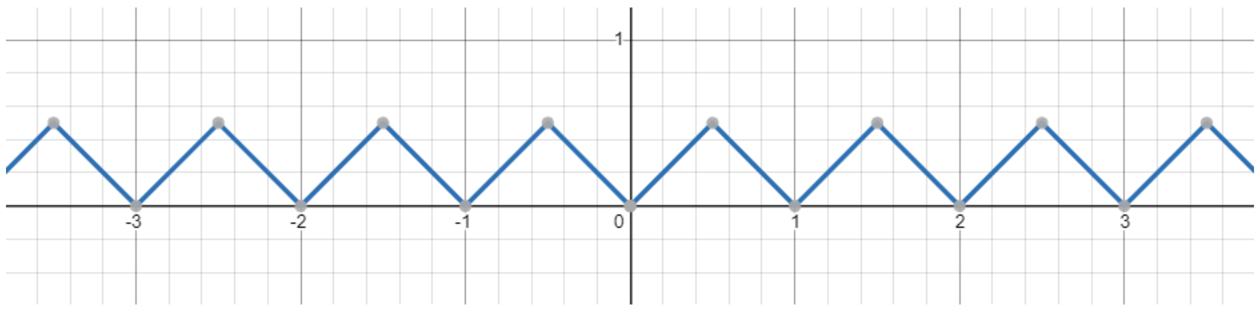
Triangle Wave

We now have approximations of $sin(x)$ for x in the range $[-\pi/2, \pi/2]$. For the quadrant in the range $[\pi/2, \pi]$ we need to mirror the $[0, \pi/2]$ section, and for the $[-\pi, \pi/2]$ quadrant we need to mirror the $[-\pi/2, 0]$ section. Then, the result of this needs to be repeated to achieve the full periodicity of $sin(x)$. This can also be expressed as replacing the input with the result of a triangle wave:



The red section is simply $x \mapsto x$ within the range $[-\pi/2, \pi/2]$.

To achieve periodicity one might be eager to use $x - floor(x)$ as a starting point. Unfortunately it doesn't have any negative portion, so one would have to subtract 0.5 and then adjust the phase with another subtraction or addition. Instead $x - round(x)$ can be used, and then we just need to take the absolute value to achieve *a* triangle wave:



To make π-sized symmetrical waves, we have to bias, scale, and phase shift so we end up with:

$$\frac{\pi}{2} - 2\pi \cdot abs((\frac{1}{4} - \frac{x}{2\pi}) - round(\frac{1}{4} - \frac{x}{2\pi}))$$

Note that $\frac{1}{4} - \frac{x}{2\pi}$ only needs to be computed once, and the division can be replaced by a multiplication by $1/2\pi$. So that's two multiplications, three subtractions, a `round()` and an `abs()`.

That's relatively expensive given how cheaply we can compute the polynomial. Can we do any better? Yes! Note that the y-axis of this triangle wave is 'connected' to the y-axis of the polynomial graph (i.e. the output of the former is the input to the latter). They have to match in scale, but this scale is arbitrary. If we make the triangle wave two times taller, we can 'stretch' the polynomial to become two times wider, and the combination would still be the same approximation of $sin(x)$. But we can also scale down the triangle wave by a factor of 2π to get rid of that first multiplication, and squeeze our polynomial into the range $[-\ 1/4,\ 1/4]$. Putting it all together, we get:

```
// Polynomial approximation of degree 5 for
// sin(x * 2 * pi) in the range [-1/4, 1/4]
static float sin5q(float x)
{
    // A * x + B * x^3 + C * x^5
    // Exact at x = 0, 1/12, 1/6, 1/4, and their negatives,
    // which correspond to x * 2 * pi = 0, pi/6, pi/3, pi/2
    constexpr float A = 6.28230858f;
    constexpr float B = -41.1693687f;
    constexpr float C = 74.4388885f;

    float x2 = x * x;

    return x * (A + x2 * (B + x2 * C));
}

float sin(float x)
{
    constexpr float pi2 = 1 / (2 * 3.1415926535f);

    // Range reduction and mirroring
    float x_2 = 0.25f - x * pi2;
    float z = 0.25f - abs(x_2 - round(x_2));

    return sin5q(z);
}
```

Cosine's Revenge

As noted early on, $cos(x)$ can just be computed as a phase-shifted $sin(x)$. So it's one extra operation, right? Nope:

```
float cos(float x)
{
    constexpr float pi2 = 1 / (2 * 3.1415926535f);

    // Range reduction and mirroring
    float x_2 = x * pi2;
    float z = 0.25f - abs(x_2 - round(x_2));

    return sin5q(z);
}
```

$cos(x)$ actually ends up taking one fewer subtraction operations, because the phase shift cancels out the one that was needed for the sine's triangle wave!

It doesn't seem feasible to produce an odd triangular wave for use by $sin(x)$ without at least three additions/subtractions, but I'd love to be proven wrong…

Results

We've reduced the calculation of $sin(x)$ and $cos(x)$ from 18 multiplications and 1 division, to 5 multiplications and no division. Based on those operations alone, a theoretical speedup of 7.6x could be achieved for AVX-512 code, or 4.8x for AVX-128 on Intel Skylake (division is slower for AVX-512 because the execution units are not full width). In practice, I've measured a 4.1x speedup for AVX-128 using SwiftShader's benchmarks.

Note this is the result compared to the vectorizable high-precision implementation by Mendenhall. Compared to the standard C/C++ `sin()` and `cos()` functions we're looking at well over 20x per scalar.