Another fast fixed-point sine approximation

2009-07-16 22:19

Gad*dammit*!

So here I am, looking forward to a nice quiet weekend; hang back, watch some telly and maybe read a bit – but *NNnnneeeEEEEEUUUuuuuuuuu!! Someone* had to write an interesting article about sine approximation. With a *challenge* at the end. *And* using an inefficient kind of approximation. And so now, instead of just relaxing, I have to spend my entire weekend *and* most of the week figuring out a better way of doing it. I hate it when this happens >_<.

Okay, maybe not.

Sarcasm aside, it is an interesting read. While the standard way of calculating a sine – via a look-up table – works and works well, there's just something unsatisfying about it. The LUT-based approach is just … dull. Uninspired. Cowardly. *Inelegant*. In contrast, finding a suitable algorithm for it requires effort and a modicum of creativity, so something like that always piques my interest.

In this case it's sine approximation. I'd been wondering about that when I did my arctan article, but figured it would require too many terms to really be worth the effort. But looking at Mr Schraut's post (whose site you should be visiting from time to time too; there's good stuff there) it seems you can get a decent version quite rapidly. The article centers around the work found at devmaster thread 5784, which derived the following two equations:

(1)

$$\begin{aligned} S_2(x) &= \frac{4}{\pi}x - \frac{4}{\pi^2}x^2 \\ S_{4d}(x) &= (1-P)S_2(x) + PS_2^2(x) \end{aligned}$$

These approximations work quite well, but I feel that it actually uses the wrong starting point. There are alternative approximations that give more accurate results at nearly no extra cost in complexity. In this post, I'll derive higher-order alternatives for both. In passing, I'll also talk about a few of the tools that can help analyse functions and, of course, provide some source code and do some comparisons.

1 Theory

1.1 Symmetry

The first analytical tool is symmetry. Symmetry is actually one of the most powerful concepts ever conceived. Symmetry of time leads to the conservation of energy; symmetry of space leads to conservation of momentum; in a 3D world, symmetry of direction gives rise to the inverse square law. In many cases, symmetry basically defines the kinds of functions you're looking for.

One kind of symmetry is parity, and functions can have parity as well. Take any function $f(x)$. A function is ***even*** if $f(-x) = f(x)$; it is ***odd*** if $f(-x) = -f(x)$.

This may not sound impressive, but a function's parity can be a great source of information and a way of error checking. For example, the product of two odd or even functions is an even function, and an odd-even product is odd (compare positive/negative number products). If in a calculation you notice this doesn't hold true, then you know there's an error somewhere.

Symmetry can also significantly reduce the amount of work you need to do. Take the next sum, for example.

(2) $$y = \int_{-N}^{N} sin^7(x^3) + \frac{x^5}{x^2+1} - xe^{\frac{x^2}{2\sigma^2}} dx$$

If you find something like this in the wild on on a test, your first thought might be “WTF?!?” (assuming you don't run away screaming). As it happens, *y* = 0, for reasons of symmetry. The function is odd, so the parts left and right of *x* = 0 cancel out. Instead of actually trying to do the whole calculation, you can just write down the answer in one line: “0, cuz of symmetry”.

Another property of symmetrical functions is that, if you break them down into series expansions, odd functions will only have odd terms, and even functions only have even terms. This becomes important in the next subsection.

1.2 Polynomial and Taylor expansions

Every function can be broken down into a sum of more manageable functions. One fairly obvious choice for these sub-functions is increasing powers of *x*: polynomials. The most common of these is Taylor series, which uses a reference point (*a*, *f*(*a*)) and extrapolates to another point some distance *h* away by using the derivatives of *f* at the reference point. In equation form, it looks like this:

$$f(a+h) = f(a) + f'(a)h + \frac{f''(a)}{2}h^2 + \frac{f'''(a)}{6}h^3 + ...$$

(3)

$$= \sum_{n=0} \frac{f^{(n)}(a)}{n!}h^n$$

Chances are you've actually used part of the Taylor series in game programming. On implementing movement with acceleration, you'll often see something like Eq 4. These are the first three terms of the Taylor expansion.

(4) $$x_{new} = x_{old} + v\Delta t + \frac{1}{2}a(\Delta t)^2$$

The step-size (*h* in Eq 3 and Δ*t* in Eq 4) is small, the higher-order terms will have less effect on the end result. This allows you to cut the expansion short at some point. This leaves you with a short equation that you do the calculations with and some sort of error term, composed of the part you have removed. The error term is usually linked to the order you've truncated the series at; the higher the order, the more accurate the approximation.

(5) $$f(a+h) = f(a) + f'(a)h + \frac{f''(a)}{2}h^2 + \frac{f'''(a)}{6}h^3 + O(h^4)$$

If you work out the math for a sine Taylor series, with *a* = 0 as the reference point, you end up with Eq 6.

(6) $$sin(h) = h - \frac{1}{6}h^3 + \frac{1}{5!}h^5 - \frac{1}{7!}h^7 + ...$$

Note that all the even powers are conspicuously absent. This is what I meant by symmetry being useful: a sine function is odd, therefore only odd terms are needed in the expansion. But there's more to it than that. The accuracy is given by the highest order in the approximating polynomial. This shows that there's just no point in even starting with any even-powered polynomial, because you can get one extra order basically for free!

This is why using a quadratic approximation for a sine is somewhat useless; a cubic will have two terms as well, and be more accurate to boot. Just because it's curved doesn't mean a parabola is the most suitable approximation.

1.3 Curve fitting (and a 3rd order example)

Using the Taylor series as a basis for a sine approximation is nice, but it also has a problem. The series is meant to have an infinite number of terms and when you truncate the series, you will lose some accuracy. Of course, this was to be expected, but this isn't the real problem; the *real* problem is that if your function has some crucial points it *must* pass through (which is certainly true for trigonometry functions), the truncation will move the curve away from those points. Example for a sine, it's *really* important that sin(π/2) = 1. With Taylor, that simply isn't the case (see Fig 1).

To fix this, you need to use a polynomial with as-yet unknown coefficients (that is, multipliers to the powers) and a set of conditions that need to be satisfied. These conditions

will determine the exact value of the coefficients. The Taylor expansion can serve as the basic for your initial approximation, and the final terms should be pretty close to the Taylor coefficients.

Let's try this for a third-order (cubic) sine approximation. Technically, a third-order polynomial means four unknowns, *but*, since the sine is odd, all the coefficients for the even powers are zero. That takes care of half the coefficients already. I told you symmetry was useful :). The starting polynomial is reduced to Eq 7, which has two coefficients *a* and *b* that have to be determined. For good measure I've also added the derivative, as that's often useful to have as well.

(7)
$$\begin{array}{rcl} S_3(x) &=& ax - bx^3 \quad = \quad x(a - bx^2) \\ S_3'(x) &=& a - 3bx^2 \end{array}$$

Two unknowns means we need two conditional to solve the system. The most useful conditions are usually the behaviour at the boundaries. In the case of a sine, that means look at $x = 0$ and/or $x = \frac{1}{2}\pi$. The latter happens to be more useful here, so let's look at that. First, $\sin(\frac{1}{2}\pi) = 1$, so that's a good one. Also, we know that at $\frac{1}{2}\pi$ a sine is flat (a derivative of 0). This is the second condition.

The conditions are listed in Eq 8. Solving this system is rather straightforward and will give you values for *a* and *b*, which are also given in Eq 8. Notice that the values are roughly 5% and 30% away from the pure Taylor coefficients.

(8)
$$\begin{array}{rcl} S_3(\frac{\pi}{2}) &=& 1 \quad = \quad \frac{\pi}{2}a - (\frac{\pi}{2})^3 b \\ S_3'(\frac{\pi}{2}) &=& 0 \quad = \quad a - 3(\frac{\pi}{2})^2 b \end{array} \quad \rightarrow$$
$$\begin{array}{rcl} a &=& \frac{3}{\pi} \quad \approx \quad 0.955 \\ b &=& \frac{4}{\pi^3} \quad \approx \quad 0.129 \end{array}$$

The final equation is then:

(9) $$S_3(x) = \frac{3}{\pi}x - \frac{4}{\pi^3}x^3$$

In Fig 1 you can see a number of different approximations to the sine. Note that I've done a little coordinate transformation for the *x*-axis: $z = x/(\frac{1}{2}\pi)$, so $z = 1$ means $x = \frac{1}{2}\pi$. The benefit of this will become clear later.

As you can see, the third order Taylor expansion starts out all-right, but veers off course near the end. In contrast, the third-order fit matches the sine at both end points. There is also the second-order fit from the devmaster site. As you can see, the third-order approximation is closer.

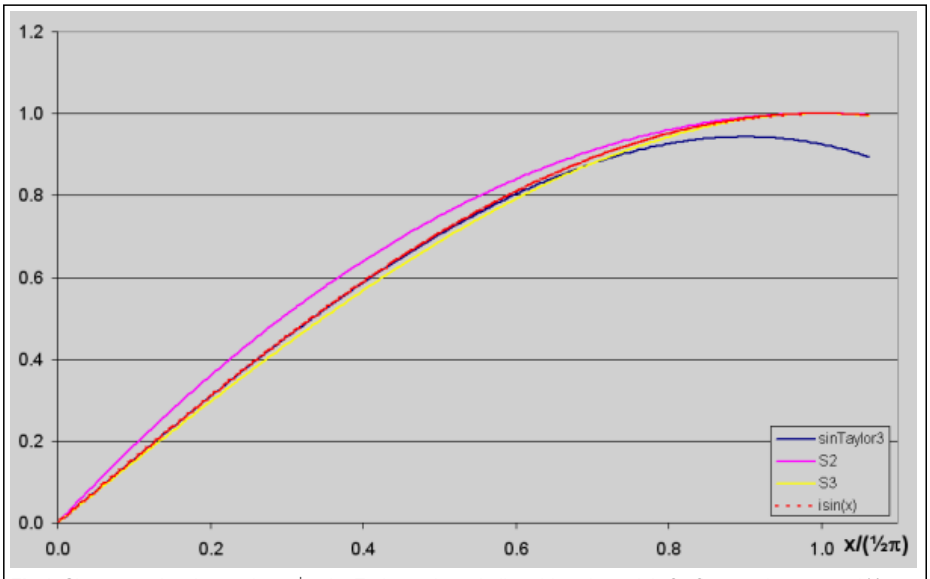

Fig 1. Sine approximations using 3rd order Taylor and parabolic cubic polynomials for first quadrant. $z = x / \frac{1}{2}\pi$

Now, please remember that coefficients from Eq 8 are not the only ones you can use. The conditions define what the values will be: different conditions lead to different values. For example, instead using the derivative at $\frac{1}{2}\pi$, I could have used it at $x = 0$. This forms the set of equations of Eq 10 and, as you can see, the coefficients are now different. This set is

actually more accurate (a 0.6% average error instead of 1.1%), but it also has some rather unsavoury characteristics of having a maximum that's not at ½π and goes over 1.0; this can be *really* unsettling if you intend to use the sine in something like rotation.

(10)
$$\left. \begin{array}{} S_3(\frac{\pi}2) &=& 1 &=& \frac{\pi}2 a - (\frac{\pi}2)^3 b \\ S_3'(0) &=& 1 &=& a \end{array} \; \; \rightarrow \; \; \begin{array}{} a &=& 1 \\ \\ b &=& \frac4{\pi^2}(1-\frac2\pi) \approx 0.147 \end{array}$$

1.4 Dimensionless variables and coordinate transformations

For higher accuracy, a higher-order polynomial should be used. Before doing that, though, I'd like to mention one more trick that can make your mathematical analysis considerably easier: dimensionless variables.

The problem with most quantities and equations is units. Metres, feet, litres, gallons; those kinds of units. Units suck. For one, there are different units for identical quantities which can be a total pain to convert and can sometimes lead to disaster. Literally. Then there's the fact that the unit sizes are basically picked at random and have nothing to do with the physical situation they're used for. So you have weird values for constants like *G* in Newton's law of universal gravitation, the speed of light *c* and the Planck constant, *h*. Keeping track of these things in equations is annoying, especially since they tend to pile up and everybody would rather that they'd just *go away*!

Enter dimensionless variables. The idea here is that instead of using standard units, you express quantities as ratios to some meaningful size. For example, in relativity you often get *v*/*c* : velocity over speed of light. Equations become much simpler if you just denote velocities as fractions of the speed of light: β = *v*/*c*. Using β in the equations simplifies them immensely and has the bonus that you're not tied to any specific speed-unit anymore.

The dimensionless variable is a type of coordinate transformation. In particular, it's a scaling of the original variable into something more useful. Another useful transformation is translation: moving the variable to a more suitable position. We will come accross this later; but first: an example of dimensionless variables.

A sine wave has lots of symmetry lines, all revolving around the quarter-circles. Because of this, the term that keeps showing up everywhere is ½π. This is the characteristic size of the wave. By using *z* = *x*/(½π), all those important points are now at integral *z* values. Having ones in your equations is generally a good thing because they tend to disappear in multiplications. Look at what Eq 9 becomes when expressed in terms of *z*

(11)
$$\begin{aligned} S_3(x) &= \frac{3}{\pi}x - \frac{4}{\pi^3}x^3 \\ &= \frac{3}{2}\frac{2x}{\pi} - \frac{1}{2}\left(\frac{2x}{\pi}\right)^3 \\ &= \frac{3}{2}z - \frac{1}{2}z^3 \\ S_3(z) &= \frac{1}{2}z(3 - z^2) \end{aligned}$$

Doesn't that look a lot nicer? It goes deeper than that though. With dimensionless units, the units your measurements are in simply cease to matter! For angles, this means that whether you're working in radians, degrees or brads, they'll all result in the same circle-fraction, *z*. This makes converting algorithms to fixed-point notation considerably easier.

2 Derivations and implementations

In the section above, I discussed the tools used for analysis and gave an example of a cubic approximation. In this section I'll also derive high-accuracy fourth and fifth order approximations and show some implementations. Before that, though, there's some terminology to go through.

Since multiple different approximations will be covered, there needs to be a way to separate all of them. In principle, the sine approximation will be named S_n, where *n* is the order of the polynomial. So that'll give S_2 to S_5. I will also use S_{4d} for the fourth-order approximation from devmaster. In the derivation of my own fourth-order function, I'll use C_n, because what will actually be derived is a cosine.

Third-order implementation

Let's start with finishing up the story of the third-order approximation. The main equation for this is Eq 11. Because this equation is still rather simple, I'll make this a fixed-point implementation. The main problem with turning a floating-point function into a fixed-point one is keeping track of the fixed-point during the calculations, always making sure there's no overflow, but no underflow either. This is one of the reasons why I wrote Eq 11 like it is: by using nested parentheses you can maximize the accuracy of intermediate calculations and possibly minimize the number of of intermediate calculations and possibly minimize the number of operations to boot.

To correctly account for the fixed-point positions, you need to be aware of the following factors:

- The scale of the outcome (i.e., the amplitude): 2^A
- The scale on the inside the parentheses: 2^p. This is necessary to keep the multiplications from overflowing.
- The angle-scale: 2^n. This is basically the value of ½π in the fixed-point system. Using x for the angle, you have $z = x/2^n$.

Filling this into Eq 11 will give the following:

$$\begin{aligned} S_3(z) &= \tfrac{1}{2} z(3 - z^2)2^A \\ &= z(3 - z^2)2^{A-1} \\ &= x \cdot 2^{-n}(3 - x^2 \cdot 2^{-2n})2^{A-1} \\ &= x(3 - x^2 \cdot 2^{-2n})2^{A-1-n} \\ &= x(3 \cdot 2^p - x^2 2^{p-2n})2^{A-n-1-p} \\ S_3(x) &= x(3 \cdot 2^p - x^2/2^r)/2^s \ , \end{aligned} \tag{12}$$

with $r = 2n-p$ and $s = n+p+1-A$. These represent the fixed-point shifts you need to apply to keep everything on the level. With p as high as multiplication with x will allow and the standard libnds units leads to the following numbers.

A	n	p	r	s
12	13	15	11	17

That's the calculation necessary for the first quadrant, but the domain of a sine is infinite. To get the rest of the domain, you can use the symmetries of the sine: the 2π periodicity and the ½π mirror symmetries. The first is taken care of by doing z % 4. This reduces the domain to the four quadrants of a circle. The next part is somewhat tricky, so pay attention.

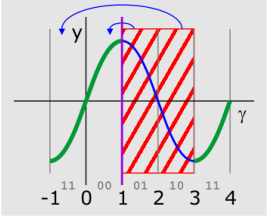

Fig 2. Mirror domain to the right quadrants.

Look at Fig 2. S_3 works for quadrant 0. Because it's antisymmetric, it will also correctly calculate quadrant 3, which is equivalent to quadrant −1. Quadrants 1 and 2 are the problem. As you can see in Fig 2, what needs to happen is for those quadrants to mirror onto quadrants 0 and −1. A reflection of x at D is defined by Eq 13. In this case, that means that $z = 2 - z$

$$x = D - (x - D) = 2D - x \tag{13}$$

Some test need to be done to see when the reflection should take place. The quadrant numbers in binary are 00, 01, 10, 11. If you build a truth-table around that, you'll see that a XOR of the two bits will do the trick. If you really want to show off, you can combine the periodicity modulo and the quadrant test by doing the arithmetic in the top bits. The implementation is now complete.

```
/// A sine approximation via a third-order approx.
/// @param x    Angle (with 2^15 units/circle)
/// @return     Sine value (Q12)
s32 isin_S3(s32 x)
{
    // S(x) = x * ( (3<<p) - (x*x>>r) ) >> s
    // n : Q-pos for quarter circle              13
    // A : Q-pos for output                      12
    // p : Q-pos for parentheses intermediate    15
    // r = 2n-p                                  11
    // s = A-1-p-n                               17

    static const int qN = 13, qA= 12, qP= 15, qR= 2*qN-qP, qS=
qN+qP+1-qA;

    x= x<<(30-qN);          // shift to full s32 range (Q13->Q30)
```

```
    if( (x^(x<<1)) < 0)      // test for quadrant 1 or 2
        x= (1<<31) - x;

    x= x>>(30-qN);

    return x * ( (3<<qP) - (x*x>>qR) ) >> qS;
}
```

And, of course, there's an assembly version as well. It's only ten instructions, which I think is actually shorter than a LUT+lerp implementation.

```
@ ARM assembly version, using n=13, p=15, A=12

@ A sine approximation via a third-order approx.
@ @param r0   Angle (with 2^15 units/circle)
@ @return     Sine value (Q12)
    .arm
    .align
    .global isin_S3a
isin_S3a:
    mov     r0, r0, lsl #(30-13)
    teq     r0, r0, lsl #1
    rsbmi   r0, r0, #1<<31
    mov     r0, r0, asr #(30-13)
    mul     r1, r0, r0
    mov     r1, r1, asr #11
    rsb     r1, r1, #3<<15
    mul     r0, r1, r0
    mov     r0, r0, asr #17
    bx      lr
```

Radians?

Oh wait, the requirement was for the input to be in Q12 radians, right? Weeell, that's no biggy. You just have to do the $x \rightarrow z$ conversion yourself. Take, say, $2^{20}/(2\pi)$. Multiply x by this gives z as a Q30 number; exactly what the first line in the C code resulted in. This means that all you have to do is change the first line to `` `x *= 166886;' ``.

NDS special

The assembly version given above uses standard ARM instructions, but one of the interesting things is that the NDS' ARM9 core has special multiplication instructions. In particular, there is the `SMULWx` instruction, which does a word*halfword multiplication, where the halfword can be either the top or bottom halfword of operand 2.The main result is 32×16→48 bits long, of which only the top 32 bits are put in the destination register. Effectively it's like *a*b*>>16 without overflow problems. As a bonus, it's also slightly faster than the standard `MUL`. By slightly changing the parameters, the down-shift factors *r* and *s* can be made 16, fitting perfectly with this instruction, although the internal accuracy is made slightly worse. Additionally, careful placement of each instruction can avoid the interlock cycle that happens for multiplications.

The alternate `isin_S3a()` becomes:

```
@ Special ARM assembly version, using n=13 and lots of Q14

@ A sine approximation via a third-order sine, using special ARM9
instructions
@ @param r0   Angle (with 2^15 units/circle)
@ @return     Sine value (Q12)
    .arm
    .align
    .global isin_S3a9
isin_S3a9:
    mov     r0, r0, lsl #(30-13)    @ x                 ; Q30
    teq     r0, r0, lsl #1
    rsbmi   r0, r0, #1<<31

    smulwt  r1, r0, r0              @ y=x*x             ; Q30*Q14/Q16
= Q28
    mov     r2, #3<<13              @ B_14=3/2
    sub     r1, r2, r1, asr #15     @ 3/2-y/2           ;
Q14+Q28/Q14/2
    smulwt  r0, r1, r0              @                   ; Q14*Q14/Q16
= Q12

    bx      lr
```

Technically it's only two instructions less, but is quite a bit faster due to the difference in speed between `MUL` and `SMULWx`.

2.1 High-precision, fifth order

The third order approximation actually still has a substantial error, so it may be useful to use an additional term. This would be the fifth-order approximation, S_5. It and its derivative are given in Eq 14.

$$\begin{aligned} S_5(x) &= ax - bx^3 + cx^5 \\ S'_5(x) &= a - 3bx^2 + 5cx^4 \end{aligned} \quad \textbf{(14)}$$

To find the terms, I will again use *z* instead of *x*. The conditions of note are the position and derivative at *z* = 1 and the derivative at 0. With these conditions the approximation should behave amicably at both edges.

$$\begin{aligned} S_5(z=1) &= 1 = a - b + c \\ S'_5(z=1) &= 0 = a - 3b + 5c \\ S'_5(z=0) &= \tfrac{\pi}{2} = a \end{aligned} \quad \textbf{(15)}$$

Notice that these equations are linear with respect to *a*, *b* and *c*, which means that it can be solved via matrices. Technically this system of equations forms a 3×3 matrix, but since *a* is already immediately known it can be reduced to a 2×2 system. I'll spare you the details, but it leads to the coefficients of Eq 16. Note the complete absence of any horrid π^5 terms that would have appeared if you had decided *not* to use dimensionless terms.

$$\begin{aligned} a &= \pi/2 \\ b &= \pi - 5/2 \\ c &= \pi/2 - 3/2 \end{aligned} \quad \textbf{(16)}$$

$$S_5(z) = \tfrac{1}{2}z(\pi - z^2[(2\pi - 5) - z^2(\pi - 3)]) \quad \textbf{(17)}$$

Eq 17 is the final quintic approximation in the form that's most accurate and easiest to implement. The implementation is basically an extension of the S_3 function and left as an exercise for the reader.

2.2 High precision, fourth order

Lastly, a fourth-order approximation. Normally, I wouldn't even consider this for a sine (odd function == odd power series and all that), but since the devmaster post uses them and they even seem to work, there seems to be something to them after all.

The reason those approximations work is simple: they don't actually approximate a sine at all; they approximate a **co**sine. And, because of all the symmetries and parallels with sines and cosines, one can be used to implement the other.

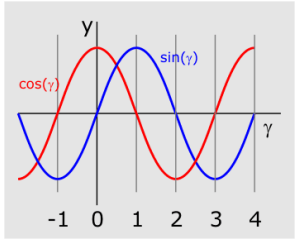

Fig 3. Sine and cosine.

$$\begin{aligned} sin(x) &= cos(x - \pi/2) \\ sin(z) &= cos(z - 1) \end{aligned} \quad \textbf{(18)}$$

Eq 18 is the transformation you need to perform to turn a cosine into a sine wave. This can be easily done in at the start of an algorithm. What's left is to derive a cosine approximation. Because a cosine is even, only even powers will be needed. The base form and its derivative are given in Eq 19.

$$\begin{aligned} C_4(x) &= a - bx^2 + cx^4 \\ C'_4(x) &= -2bx + 4cx^3 \end{aligned} \quad \textbf{(19)}$$

For the conditions, we once again look at *z* = 0 and *z* = 1, which comes down to the eqt of equations in Eq 20. One of the interesting thing about even functions is that the derivative at 0 is zero, so that's a freebie. A very important freebie, as it means that one of the required symmetries happens automatically.

$$
\begin{array}{lrcrcrcr}
 & C_4(z=0) & = & 1 & = & a & & \\
\textbf{(20)} & C_4(z=1) & = & 0 & = & a & - b & + c \\
 & C_4'(z=1) & = & -\frac{\pi}{2} & = & & - 2b & + 4c
\end{array}
$$

The resulting set of coefficients are listed in Eq 21. Note that *b* = *c*+1, which may be of use later. The final equation for the fourth order cosine approximation is Eq 22. Only three MULs and two SUBs; nice.

$$
\begin{array}{lcl}
a & = & 1 \\
\textbf{(21)} \quad b & = & 2 - \pi/4 \\
c & = & 1 - \pi/4
\end{array}
$$

$$\textbf{(22)} \quad C_4(z) = 1 - z^2[(2 - \pi/4) - z^2(1 - \pi/4)]$$

Implementation

The floating-point implementation of Eq 22 is again too easy to mention here, so I'll focus on fixed-point variations. Like with S_3, you can mix and match fixed-point positions until you get something you like. In this case I'll stick to Q14 for almost everything to keep things simple.

The real trick here is to find out what you need to do about all the other quadrants. Cutting down to four quadrants is, again, easy. For the rest, remember that the cosine approximation calculates the top quadrants and you need to flip the sign for the bottom quadrants. If you think in terms of the parameter that a sine gets, you see that only for odd semi-circles the sign needs to change. Tracing this can be done with a single bitwise AND or a clever shift.

```
/// A sine approximation via a fourth-order cosine approx.
/// @param x   angle (with 2^15 units/circle)
/// @return    Sine value (Q12)
s32 isin_S4(s32 x)
{
    int c, x2, y;
    static const int qN= 13, qA= 12, B=19900, C=3516;

    c= x<<(30-qN);              // Semi-circle info into carry.
    x -= 1<<qN;                 // sine -> cosine calc

    x= x<<(31-qN);              // Mask with PI
    x= x>>(31-qN);              // Note: SIGNED shift! (to qN)
    x= x*x>>(2*qN-14);          // x=x^2 To Q14

    y= B - (x*C>>14);           // B - x^2*C
    y= (1<<qA)-(x*y>>16);       // A - x^2*(B-x^2*C)

    return c>=0 ? y : -y;
}
```

And an ARM9 assembly version too. As it happens, it's only two instuctions longer than `isin_S3a9()`.

```
@ ARM assembly version of S4 = C4(gamma-1), using n=13, A=12 and ...
miscellaneous.

@ A sine approximation via a fourth-order cosine
@ @param r0   Angle (with 2^15 units/circle)
@ @return     Sine value (Q12)
    .arm
    .align
    .global isin_S4a9
isin_S4a9:
    movs    r0, r0, lsl #(31-13)    @ r0=x%2 <<31       ; carry=x/2
    sub     r0, r0, #1<<31          @ r0 -= 1.0         ; sin <-> cos
    smulwt  r1, r0, r0              @ r1 = x*x          ;
Q31*Q15/Q16=Q30

    ldr     r2,=14016               @ C = (1-pi/4)<<16
    smulwt  r0, r2, r1              @ C*x^2>>16         ; Q16*Q14/Q16
= Q14
    add     r2, r2, #1<<16          @ B = C+1
    rsb     r0, r0, r2, asr #2      @ B - C*x^2         ; Q14
    smulwb  r0, r1, r0              @ x^2 * (B-C*x^2)   ; Q30*Q14/Q16
= Q28
    mov     r1, #1<<12
    sub     r0, r1, r0, asr #16     @ 1 - x^2 * (B-C*x^2)
    rsbcs   r0, r0, #0              @ Flip sign for odd semi-circles.
```

```
    bx      lr
```

3 Testing

Deriving approximations is nice and all, but there's really no point unless you do some sort of test to see how well they perform. I'll look at two things: accuracy and some speed-tests. For the speed-test, I'll only consider the functions given here along with some traditional ones. The accuracy test is done only for the first quadrant and in floating-point, but the results should carry over well to a fixed-point case. Finally, I'll show how you can optimize the functions for accuracy.

3.1 Third and fourth-order speed

For the speed test I calculated the sine at 256 points for $x \in [0, 2\pi)$. There will be some loop-overhead in the numbers, but it should be small. Tests were performed on the NDS.

Functions under investigation are the three S_3 and two S_4 functions given earlier. I've also tested the standard floating-point `sin()` library function, the libnds `sinLerp()` and my own `isin()` function that you can find in arctan:sine. The cumulative and average times can be found in Table 1.

Function (thumb/ARM)	Total cycles	average cycles
sin (F)	300321	1175.1
sinLerp (T)	10051	39.2
isin (T)	7401	28.9
isin_S3 (T)	5267	20.5
isin_S4 (T)	6456	25.2
isin_S3a (A)	3438	13.4
isin_S3a9 (A)	2591	10.1
isin_S4a9 (A)	3123	12.1

Table 1: sine cycle-times (roughly).

The first thing that should be clear is just why we don't use the floating-point sine. I mean, seriously. There is also a clear difference between the Thumb-compiled and ARM assembly versions, the latter being significantly faster.

Within the compiled versions, I find it interesting to see that the algorithmic calculations are actually faster than the LUT+lerp-based implementations. I guess loading all those numbers from memory really does suck.

And *then* there's the assembly versions. Wow. Compared to the compiled version they're twice as fast, and up to four times as fast as the LUT-based functions.

NDS timers measure half-cycles

The cycle-times from Table 1 do not make sense if you count instruction cycles. For example, for `isin_S3a` the function overhead alone should already be around 10 cycles. The thing here is that the numbers are taken from the hardware timers, which use the bus-frequency (33 MHz) rather than the ARM9 cpu (66 MHz). As such, it measures in half-cycles. For details, see gbatek:nds-timings.

3.2 Accuracy

Fig 4 shows all the approximations in one graph. It only shows one quadrant because the rest can be retrieved by symmetry. I've also scaled the sine and its approximations by 2^{12} because that's the scale that usual fixed-point scale right now. And to be sure, yes, this is a different chart than Fig 1; it's just hard to tell because the fourth and fifth order functions are virtually identical to the real sine line.

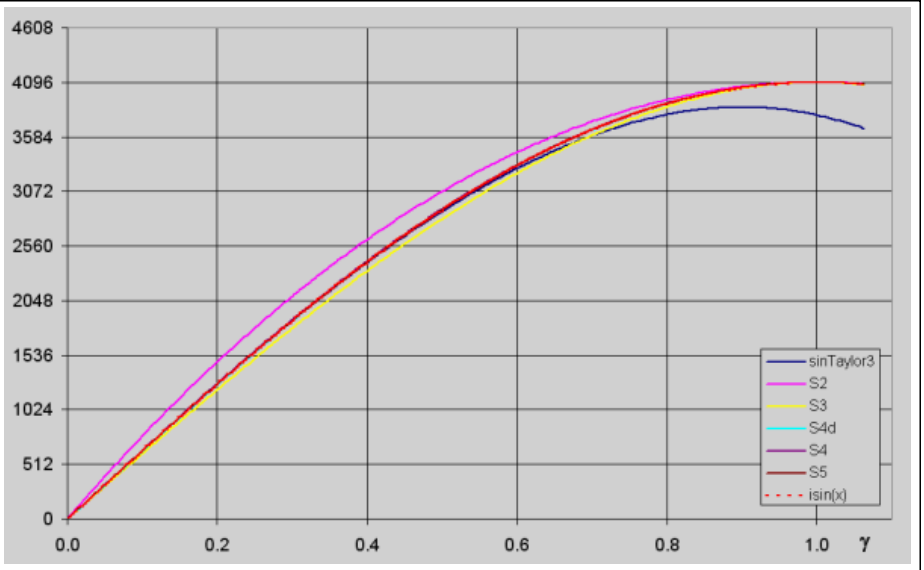

Fig 4. Taylor and second to fifth order approximations to $2^{12}\sin(x)$.

For the high-accuracy approximations, it's better to look at Fig 5, which shows the errors. Here you can clearly see a difference between S_{4d} and S_5, the latter is roughly 3 times better.

There's also a large difference between the devmaster fourth-order sine and my own. The reason behind this is a difference in conditions. In my case, I've fixed the derivatives at both end-points, which always results in an over- or underestimate. The devmaster's S_{4d} let go of those conditions and minimized the error. I'll also do this in the next sub-section.

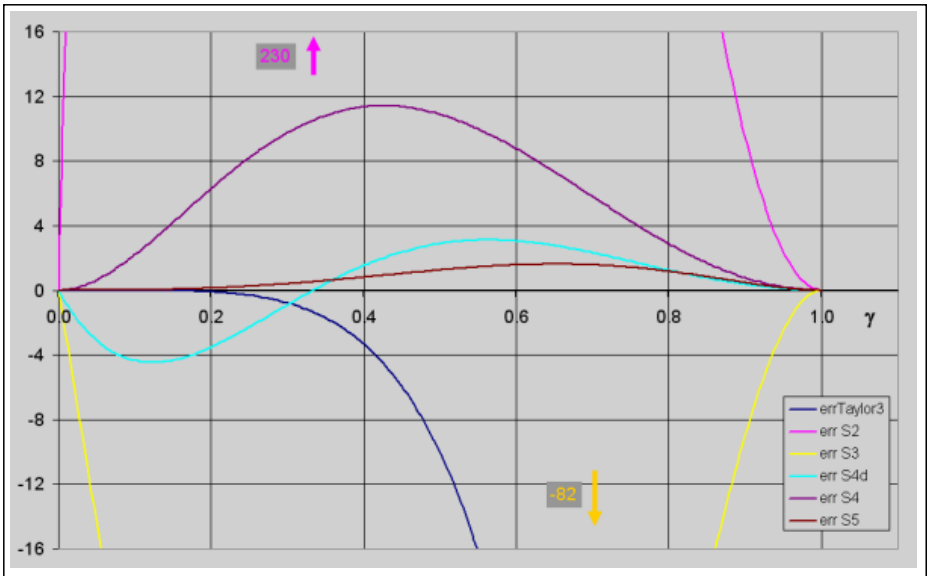

Fig 5. Errors for Fig 4.

Table 2 and Table 3 list some interesting statistics about the various approximations, namely the minimum, average and maximum errors. It also contains a Root Mean Square Deviation (RMSD), which is a special kind of distance. If you consider the data-points as a vector, the RMSD is the average Pythagorean length for each point. Table 2 is normed to 2^{12}, whereas Table 3 is table for the traditional floating-point sine scale.

The RMSD values are probably the most useful to look at. From them you can see that there is a huge gap between the low-accuracy and high-accuracy functions of about a factor 60. And if you do your math right, all it costs is one multiplication and one addition, and maybe some extra shifts in the fixed-point case. That's quite a bargain. Compared to that, the difference between the odd and even functions is somewhat meager: only a factor three or so. Still, it is something.

If you look at the fixed-point table, you can see that the error you make with S_{4d} and S_5 is in the single digits. This means that this is probably accurate enough for practical purposes. Combined with the fact that even fifth order polynomials can be made pretty fast, this makes them worth considering over LUTs.

	min	avg	max	rms
Taylor3	-302.1	-51.5	0	92.7

	min%	avg%	max%	rms%
Taylor3	-7.37	-1.26	0	2.26

S2	0	123.1	229.4	146.8
S3	-82.0	-47.6	0	55.0
S4d	-4.47	0.19	3.11	2.44
S4	0	5.87	11.4	7.11
S5	0	0.74	1.62	0.94

Table 2: error statistics for 2^{12}sin(x) approx.

S2	0	3	5.6	3.58
S3	-2	-1.16	0	1.34
S4d	-0.11	0.0047	0.076	0.06
S4	0	0.143	0.278	0.174
S5	0	0.018	0.039	0.023

Table 3: error statistics in percentages.

3.3 Optimizing higher-order approximations

From the charts, you can see that S_4 and S_5 all err on the same side of the sine line. You can increase the accuracy of the approximation by tweaking the coefficients in such a way that the errors are redistributed in a preferable way. Two methods are possible here: shoot for a zero error average, or minimize the RMSD. Technically minimizing the RMSD is standard (it comes down to least-squares optimization), but because a zero-average allows for an analytical solution, I'll use that. In any case, the differences in outcomes will be small.

First, think of what an average of a function means. The average of a set of numbers is the sum divided by the size of the set. For functions, it's the integral of that function divided by the interval. When you want a zero-average for an approximation, the integral of the function and that of the approximation should be equal. With a polynomial approximation to a sine, we get:

$$\begin{aligned} \int_0^1 \sum_n a_n x^n dx &= \int_0^1 sin(x\pi/2)dx \quad \rightarrow \\ \sum_n \frac{a_n}{n+1} &= 2/\pi, \end{aligned} \tag{23}$$

with a_n reducing to the coefficients of the polynomials we had before. This can be used as an alternate condition to the derivative at 0. For S_4 and S_5, you'll end up with the following coefficients.

$$\begin{aligned} a_4 &= 1 \\ b_4 &= c_4 + 1 \\ c_4 &= 5(1 - \tfrac{3}{\pi}) \approx 0.225351707 \end{aligned} \tag{24}$$

$$\begin{aligned} a_5 &= 4(\tfrac{3}{\pi} - \tfrac{9}{16}) \approx 1.569718634 \\ b_5 &= 2a_5 - 5/2 \\ c_5 &= a_5 - 3/2 \end{aligned} \tag{25}$$

If you're still awake and remember the devmaster S_{4d} coefficients, there should be something familiar about a_4. Yes, they're practically identical. If you optimize S_4 for the RMSD, you actually get the exact same function as S_{4d}.

Table 4 shows the statistics for the original approximations and the new optimized versions, S_{4o} and S_{5o}. The numbers for S_{4o} are basically those from S_{4d} seen earlier. More interesting are the details for S_{5o}. The maximum and minimum errors are now within ±1. That is to say, this approximation gives values that are at most 1 off from the proper Q12 sine. This is about as good as any Q12 approximation is able to get.

	min	avg	max	rmsd
S4	0	5.87	11.4	7.11
S5	0	0.74	1.616	0.94
S4o	-4.72	0	2.89	2.47
S5o	-0.73	0	0.79	0.52

Table 4: Optimized Q12 S_4 and S_5.

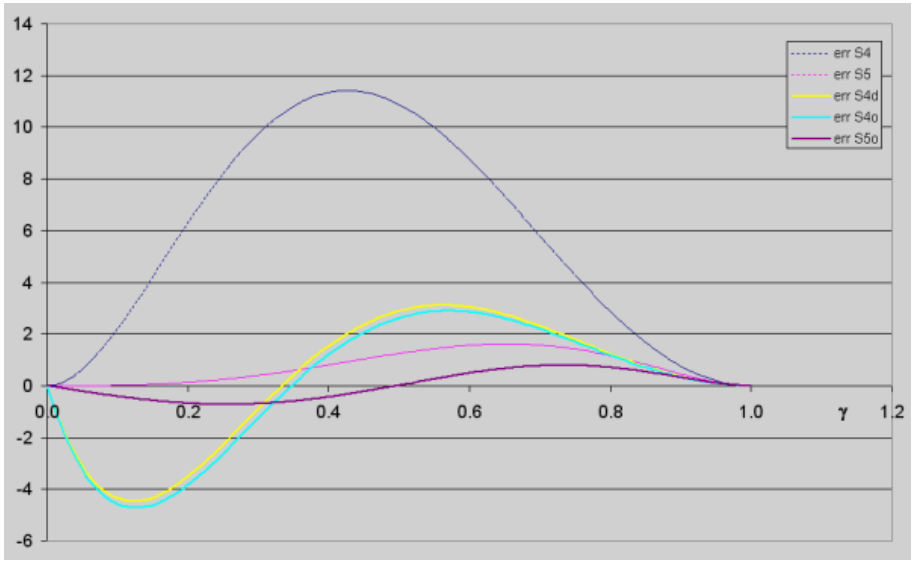

Fig 6. Errors for optimized S_{4o} and S_{5o}.

4 Summary and final thoughts

Here's a few things to take from all this.

- Symmetry is your friend.
- When constructing a polynomial approximation, more terms mean higher accuracy. Symmetry properties of the function approximated allow you to remove terms from consideration, simplifying the equation.
- Coordinate transformations are your friends too. Sometimes it's much easier to work on a scaled or moved version of the original problem. If your situation has a characteristic length (or time, velocity, whatever) consider using dimensionless variables: expressing parameters as ratios of the characteristic length. This makes the initial units pretty much irrelevant. For angles, think circle-fractions.
- Zero and one (0 and 1) are the best values to have in your equations, as they tend to vanish to easily.
- Any approximation formula will have coefficients to be determined. In general, the Taylor series terms are *not* the best set; values slightly offset from these terms will be better as they can correct for the truncation. To determine the values of the coefficients, define some conditions that need to be satisfied. Examples of conditions are values of the function and its derivative at the boundaries, or its integrals. Or you can wuss out and just dump the thing in the Excel Solver.
- When converting to fixed-point, accuracy and overflow comes into the fray. If you know the domain of the function beforehand, you can optimize for accuracy. Also, it helps if you construct the algorithm in a sort of recursive form instead of a pure polynomial: not $ax + bx^2$ but $x(a + xb)$. Ordered like this, each new additional term only requires one multiplication and one addition extra.
- For fixed-point work, `SMULWx` is teh awesome.
- Even a fourth order (and presumably fifth order as well) polynomial implementation in C is faster than the LUT-based sines on the NDS. And specialized assembly versions are considerably faster still.
- The difference in accuracy of S_4 vs S_2 or S_5 vs S_3 is huge: a factor of 60. Going from an even to the next odd approximation only gains you a factor 3. Shame; I'd hoped it'd be more.
- Unlike I initially thought, the even-powered polynomials work out quite well. This is because they're actually modified cosine approximations.

Exercises for the reader

1. Express the parabolic approximation $S_2(x)$ of Eq 1 in terms of z. 's Not hard, I promise.
2. Implement the fixed-point version of the fifth-order sine approximation, $S_5(x)$.
3. For the masochists: derive the coefficients for $S_5(x)$ *without* dimensionless variables. That is to say, with the conditions at $x = ½\pi$ instead of $z = 1$.
4. Solve Eq 24 and Eq 25 for minimal RMDS. Also, try to derive an analytical form for minimal RMDS; I think it's exists, but it may be tricky to come up with the right form.