**Embedded Trig**

Here're some algorithms to make it easier to compute complex trig functions.

Published in Embedded Systems Programming, May 1991

**By Jack Ganssle**

The real world is an ugly place where the laws of physics reign supreme. Whether an input device is a sensor measuring some physical parameter, or our controller causes some event to happen, frequently we model some aspect of the real world in our code.

This means sooner or later we have to dirty our hands with math. Sometimes I wonder how pure computer science types cope with coding Fast Fourier Transforms or regression equations with perhaps little calculus background, but perhaps that is a philosophical subject better suited for a late night party discussion...

Recently I talked at length with an engineer writing code to control a milling machine's X-Y table. He was looking for a quick assembly language Sine routine, since the motion of the mill's table is related to the sine of the angle traversed by the controller's stepper motors.

An awful lot of embedded systems compute trig functions to sequence an output or translate incoming data. Page though any physics book; sines and cosines seem to describe just about every sort of phenomena. Vectors, kinematics, rotation - every sort of two or three dimensional motion is best described by a trig relationship. The X-Y Digital to Analog converters in a vector CRT are synchronized in a trig ballet, as is the complex motion of a robot arm or a flatbed plotter.

Working in C with lots of free memory we never have to worry about taking the sine or cosine of an angle. The compiler's libraries have trig resources; we just invoke the routines without giving them much thought. If only all of the embedded world were so easy! Too many of us fight for every last byte or microsecond. Maybe we can't afford the overhead of a standard math library. Perhaps we're working in assembly where such resources just don't exist.

Packaged libraries are no panacea. The vendor selects some speed versus precision mix to satisfy 90% of the customers, giving a mix that might not be right for your particular application. Worse, (gasp!) the libraries that come with some well-known C compilers are not reentrant, so you can't use these functions in interrupt or multitasking routines. Even in C sometimes ROM space is a problem; if you only need a cosine routine, why link in an entire trig module?

One solution is to roll your own trig algorithms. Very high precision trig routines, in particular, are easy to write.

**Errors**

Trigonometric functions yield irrational results. That is, they have no exact solution except at certain "round" numbers like pi/2. However, given a decent algorithm and enough computer time any desired level of precision is possible.

Just how accurate is a particular algorithm? Most accuracy measurements (or better, precision: accuracy is really a measure of repeatability, but in most computer literature the two words are used interchangeably) use one of two methods to answer this question.

Absolute accuracy tells us how much an approximation is in error at any point. Mathematically it is:

|f(x)-a(x)|

In English, this is the absolute value of the correct answer (f(x)) minus the one given by the approximation (a(x)). In one sense this is just what we're looking for. However, what if the function's value is very close to 0? An absolute accuracy of .00001 sounds great, but it is a 50% error if f(x) is .00002.

Relative accuracy is normalized to f(x) to account for points around any singularity. It is expressed as:

|f(x)-a(x)| / |f(x)|

and closely resembles percentage error. A relative accuracy of .01 is only an absolute error of .001 if f(x) is 10, but is .0002 if the function's value is .02.

Select an algorithm based on its relative accuracy if your code requires extremely precise answers over a wide range of inputs. Be aware that most functions behave oddly near certain points. The sine, for example, changes very slowly in the vicinity of 0, but accelerates rapidly near pi/4.

Don't be lulled into a quest for the best possible accuracy. High precision carries a correspondingly high computational cost. Know what the project needs, and then select the appropriate algorithm.

No algorithm will behave as expected if the floating point package (add, subtract, divide and multiply) loses much precision. Every addition, multiplication, division and subtraction in even the best package will reduce the answer's precision by some small amount. Never make assumptions about your floating point libraries, whether written by your company, supplied by a compiler vendor, or implemented in a coprocessor chip.

**Trig Basics**

Most programmers think in terms of degrees, with 360 forming a circle. The radian is a more natural unit of measurement for angles. Radians are based on pi, the number that relates diameter to circumference and ties all of trigonometry together.

A circle consists of 2 \* pi radians. Thus, pi radians corresponds to 180 degrees, pi/2 to 90 degrees, etc. The formal relationship between degrees and radians is:

2 \* pi radians = 360 degrees

so,

radians = (pi/180) \* degrees

To put this in easier-to-understand terms, one radian is about 57 degrees.

The trig libraries used by different languages seem to almost arbitrarily support radians or degrees. The following discussion is in radians.

The standard trig functions are interrelated as follows:

sin(x + 90)=cos(x) (in degrees)

sin(x + pi/2)=cos(x) (in radians)

tan(x)=sin(x)/cos(x)

There's little reason to develop approximations for both the sine and cosine, since it is so easy to convert between them. Use these relations, write one approximation function, and save coding time and memory.

**Floating Point Approximations**

We really don't have deterministic ways to compute irrational functions. There is no magic formula that figures a sine or cosine. In the computer biz we rely on an amazing fallout of calculus: we can approximate any continuous function, over some range, using nothing more exotic than polynomials.

This is truly profound. Does your instrument process some obscure input whose behavior is a really weird curve? If it is continuous, then it is possible to come up with one or more polynomials that describe it any desired degree of accuracy.

One way to figure a sine is the Taylor series:

sine(x)=x-(x\*\*3)/3! + (x\*\*5)/5! - (x\*\*7)/7! + . . .

We all talk about Taylor series approximations, but in fact no one uses them. The Taylor series converge too slowly to be useful. In practice we use approximations derived from Chebyshev series and other expansions that have been extensively modified for quick convergence. (The bible of approximation is John Hart's "Computer Approximations", published in 1968 by John Wiley & Sons.)

Figure 1 shows a C function that computes the cosine of an angle x in radians. The accuracy is measured in absolute terms and slightly exceeds 9.6 decimal digits.

Variable quad assumes the values 0, 1, 2, or 3, corresponding to the quadrant x lies in. Quadrant 0 is corresponds to angles of 0 to 90 degrees, quadrant 2 to 90 to 180 degrees, three to 180 to 270 degrees, and the fourth quadrant is the last 90 degrees of a circle. The routine decomposes the input angle (x) into quad and frac, where frac lies between 0 and pi/2, the range of the polynomial.

The coefficients p0 to p5 form an equation of the form: cos(x)=p(x\*\*2). In other words, the equation is:

p0 + (p1\*x\*\*2) + (p2\*x\*\*4) + (p3\*x\*\*6) + (p4\*x\*\*8) + (p5\*x\*\*10)

Different systems require a wide range of accuracy, from low precision fast approximations for real time imaging to slow but ultra-precise numbers for orbit corrections. Following are a number of coefficient sets with different speed/accuracy tradeoffs. All coefficients cover the same range as the function shown; they plug directly into the code in the Figure. Of course, where a coefficient is zero, remove that entire term from the polynomial to save two floating point operations.

Absolute accuracy: 3.22 decimal digits

p0= 0.99940307

p1= -0.49558072

p2= 0.03679168

p3= 0.0

p4= 0.0

p5= 0.0

Absolute accuracy: 12.12 decimal digits

p0= 0.99999999999925182

p1= -0.49999999997024012

p2= 0.04166666647338454

p3= -0.00138888841800042

p4= 0.000024801040648456

p5= -0.000000275246963843

p6= 0.000000001990785685

**Integer Approximations**

Embedded systems are peculiar beasts. Some applications demand very fast results with practically no accuracy. Sometimes we have no floating point capability at all, and just need a rough integer result. If the D/A converters on your vector CRT are only 8 bits, a floating point solution will be too accurate and too slow.

Digging through my algorithm collection I found two separate integer approximations based on the set of coefficients (which as used as shown in Figure 1):

p0= 0.99940

p1=-0.49558

p2= 0.03679

The algorithms scale these numbers to integer approximations by multiplying by 255 or 65536. Then they solve for an angle that ranges from 0 to 255 or 65536 as follows:

sine(x)= .99940\*255 - 0.49448\*255\*x\*\*2 + .03679\*255\*x\*\*4

or,

sine(x)= 255 - 126\*x\*\*2 + 9\*x\*\*4

Theoretically this is a nice, fast way to compute a low precision integer sine. It doesn't really work well unless you use 32 bit math, since the x\*\*2 and x\*\*4 terms quickly grow to huge numbers we can't stuff into a byte or word.

A crude alternative is to use a table lookup for the trig values. This might eat up a lot of memory, even if you only need a byte's worth of accuracy. It is, however, the fastest method you'll find.

Where memory is a problem limit the table's size and then interpolate between values. Suppose you need 16 bit accuracy (i.e., one part in 65536, or .002%). A table of 65000 integers will fill several PROMs, and just is not practical.

It might be better to build a table of 256 integers that approximates the sine every 256 points. That is, the table has an entry for the sine at scaled input values of 0, 256, 512, ... 65536.

Now write a routine to decompose the input value x. Compute an N and D such that x=256\*N+D. Then, compute the sine by:

sine(x)=f(N) + (D/256)\*(f(N+1) - f(N))

(where f(a) is the table lookup at point a).

Suppose the input argument is 300. N is 1, D is 44. The formula just sums the sine at point 256 (the second table entry) with the difference between that point and the next from the table, divided by the ratio of the input value to the difference in table entries.

This is an example of a linear interpolation. Whenever an input value corresponds exactly to a table entry we simply return that entry. Where the input falls between two values in the table, we draw a straight line connecting the two points and return an answer from that line. Of course, the trig functions are not lines, so we do loose some precision doing this. You can increase accuracy by making the table bigger, or coming up with a better, non-linear interpolation. As with all approximations, this involves some tradeoff between speed, memory, and accuracy.

**Conclusion**

Computer approximations receive little attention, yet are used in most systems. I find them fun to experiment with; so many sorts of formulas and algorithms yield wildly different accuracies and efficiencies. Over the years I've collected a huge folder of techniques that range from the ridiculous to the truly elegant, but no one method is ideal for all applications.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

double cosine(double x)

{

double p0,p1,p2,p3,p4,p5,y,t,absx,frac,quad,pi2;

p0= 0.999999999781;

p1=-0.499999993585;

p2= 0.041666636258;

p3=-0.0013888361399;

p4= 0.00002476016134;

p5=-0.00000026051495;

pi2=1.570796326794896; /\* pi/2 \*/

absx=x;

if (x<0) absx=-absx; /\* absolute value of input \*/

quad=(int) (absx/pi2); /\* quadrant (0 to 3) \*/

frac= (absx/pi2) - quad; /\* fractional part of input \*/

if(quad==0) t=frac \* pi2;

if(quad==1) t=(1-frac) \* pi2;

if(quad==2) t=frac \* pi2;

if(quad==3) t=(frac-1) \* pi2;

t=t \* t;

y=p0 + (p1\*t) + (p2\*t\*t) + (p3\*t\*t\*t) + (p4\*t\*t\*t\*t) + (p5\*t\*t\*t\*t\*t);

if(quad==2 | quad==1) y=-y; /\* correct sign \*/

return(y);

};

Figure 1: Function to compute cos(x)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*