Optimizing Trig Calculations

by Don Cross - dcross@intersrv.com

Last update to this page: 14 January 1998

1. Introduction

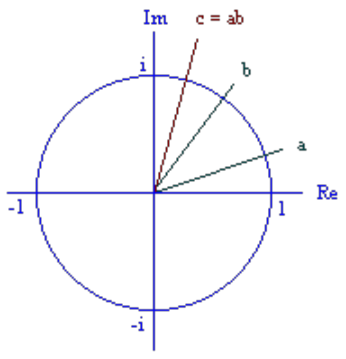
This discussion shows how to efficiently perform a series of sine and/or cosine calculations of an angle which is repeatedly increasing (or decreasing) by a fixed amount. This can be useful for digital audio applications, graphics, or any situation where lots of trig function calls would be undesirably slow.

If you need to calculate both the sine and cosine of a steadily increasing angle, see the next two sections.

If you need only sines or cosines, but not both, you should see Section 4.

2. Sines and Cosines: The Easy Way

This method is based on a geometric interpretation of complex numbers. A complex number can be thought of as a two-dimensional vector possessing a magnitude (length) and an angle measured counterclockwise from the positive real axis. In the diagram below, the complex number *c* = *ab*. The magnitude of *c* is the magnitude of *a* times the magnitude of *b*, and the angle of *c* is the angle of *a* plus the angle of *b*. These magnitude and angle relationships work for any pair of complex numbers being multiplied together.



The geometric interpretation of complex number multiplication allows us to see how to rotate a complex number about the origin by merely multiplying it by another complex number. The payoff comes by calculating the product using the usual algebraic interpretation of complex number multiplication:

```
Re{c} = Re{a}Re{b} - Im{a}Im{b}
Im{c} = Re{a}Im{b} + Im{a}Re{b}
```

If we multiply two unit-length complex numbers together, the result will also have a magnitude of 1, but will have its angle shifted. We can do this repeatedly. We need to call trig functions for the starting angle and the increment angle to get things started. After that, we can just use floating point multiplication, addition, and subtraction. The real component of the complex number is the cosine of its angle, and the imaginary component is the sine. Note that both the sine and cosine are available on each iteration, whether or not you need both of them. Even if you need only one of them, this method is still dramatically faster than the equivalent trig functions calls.

Here's a little C program to illustrate this technique. This program prompts the user for both a starting angle *a0* and an increment angle *da*. It then prints out a table of the sines and cosines of the first 20 angles *a* = *a0* + *k***da*, where *k* = 0, 1, 2, ..., 19.

```
/*   trig1.c  -  Example program by Don Cross <dcross@intersrv.com>    */
/*               http://www.intersrv.com/~dcross/fasttrig.html         */

#include <stdio.h>
```

```
#include <math.h>

#define  PI  (3.14159265358979323846)

int main (void)
{
    double a, da, a0_rad, da_rad;
    double a_sin, a_cos, da_sin, da_cos;
    double temp;
    int k;

    printf ( "Enter starting angle in degrees: " );
    scanf ( "%lf", &a );
    a0_rad = a * PI / 180.0;    /* convert to radians */

    printf ( "Enter increment angle in degrees: " );
    scanf ( "%lf", &da );
    da_rad = da * PI / 180.0;   /* convert to radians */

    a_cos = cos(a0_rad);
    a_sin = sin(a0_rad);

    da_cos = cos(da_rad);
    da_sin = sin(da_rad);

    printf ( "\n%15s  %15s  %15s\n", "angle", "cosine", "sine" );
    printf ( "---------------  ---------------  ---------------\n" );

    for ( k=0; k<20; k++ )
    {
        printf ( "%15.5lf  %15.7lf  %15.7lf\n", a, a_cos, a_sin );

        /* Here's the code that updates the trig values... */

        temp  = a_cos*da_cos - a_sin*da_sin;
        a_sin = a_cos*da_sin + a_sin*da_cos;
        a_cos = temp;

        /* Update the angle for display purposes. */
        /* Note that this step is not needed to update the trig values. */

        a += da;
    }

    return 0;
}

/*---  end of file trig1.c  ---*/
```

3. Sines and Cosines: The High-Precision Way

Here's a refinement of the above technique that I got from the book **Numerical Recipes in Fortran** by Press, Teukolsky, Vetterling, and Flannery, published by Cambridge University Press. It is a bit more complicated, but for some reason (which I don't really understand), it results in much less cumulative floating point error when you are doing a lot of iterations. On my computer, floating point error (of the vector's radius) with the first method is on the order of 1.0e-10 (one part in ten billion) after about 1000 iterations, while this method results in an error of about 1.0e-14 (one part in a hundred trillion). Most of the time, I find that it really doesn't matter, so I use the first method because it is easier for me to remember without having to look anything up. But if precision is absolutely critical, use this approach.

```
/*   trig2.c  -  Example program by Don Cross <dcross@intersrv.com>   */
/*               http://www.intersrv.com/~dcross/fasttrig.html        */

#include <stdio.h>
#include <math.h>
```

```
#define  PI  (3.14159265358979323846)

int main (void)
{
    double a, da, a0_rad, da_rad;
    double a_sin, a_cos, alpha, beta;
    double temp;
    int k;

    printf ( "Enter starting angle in degrees: " );
    scanf ( "%lf", &a );
    a0_rad = a * PI / 180.0;    /* convert to radians */

    printf ( "Enter increment angle in degrees: " );
    scanf ( "%lf", &da );
    da_rad = da * PI / 180.0;   /* convert to radians */

    a_cos = cos(a0_rad);
    a_sin = sin(a0_rad);

    alpha = sin(0.5 * da_rad);
    alpha = 2.0 * alpha * alpha;

    beta = sin(da_rad);

    printf ( "\n%15s  %15s  %15s\n", "angle", "cosine", "sine" );
    printf ( "---------------  ---------------  ---------------\n" );

    for ( k=0; k<20; k++ )
    {
        printf ( "%15.5lf  %15.7lf  %15.7lf\n", a, a_cos, a_sin );

        /* Here's the code that updates the trig values... */

        temp  = a_cos - (alpha*a_cos + beta*a_sin);
        a_sin = a_sin - (alpha*a_sin - beta*a_cos);
        a_cos = temp;

        /* Update the angle for display purposes. */
        /* Note that this step is not needed to update the trig values. */

        a += da;
    }

    return 0;
}

/*---  end of file trig2.c  ---*/
```

4. Simple Harmonic Oscillators

[Thanks to Petr Vicherek for telling me about this method. - *Don*]

The following formula is the fastest and most precise I have seen for calculating a single sinusoid. It requires only one multiply and one subtraction per iteration. To calculate $y_n = \sin(nW + B)$, use the following formula:

$$y_n = \begin{cases} \sin(nW + B), & n = -2, -1 \\ 2\cos(W)\, y_{n-1} - y_{n-2}, & n = 0, 1, 2, \ldots \end{cases}$$

Note that you must calculate two seed values for y_{-2} and y_{-1} to get things started. Here is a little sample program in C++.

```
#include <iomanip.h>
#include <math.h>

const double PI = 3.14159265358979323846;

int main()
{
    cout << "Enter W in degrees: " << flush;
    double w;
    cin >> w;
    w *= PI / 180.0;  // convert to radians

    cout << "Enter B in degrees: " << flush;
    double b;
    cin >> b;
    b *= PI / 180.0;   // convert to radians

    double y[3];
    y[0] = sin(-2*w + b);
    y[1] = sin(-w + b);
    const double p = 2.0 * cos(w);

    for ( int i=0; i<40; i++ )
    {
        y[2] = p*y[1] - y[0];
        y[0] = y[1];
        y[1] = y[2];

        cout << setprecision(10) << y[2] << endl;
    }

    return 0;
}
```

[Back to Don Cross home page]
[Go to Digital Audio page]