Machine Computation of the Sine Function

Chuck Allison
CNS 3320 – Numerical Software Engineering
Utah Valley State College

February 2007

Abstract

This note shows how to determine the number of terms to use for the McLaurin series for $\sin x$ to obtain maximum machine accuracy. It also shows some tweaks that lessen roundoff and increase efficiency.

Argument Reduction

Transcendental functions such as $\sin x$ are calculated by a truncated Taylors series. In the case of trigonometric functions, it is important to leverage their periodicity and reduce arguments to the smallest possible range, since power series tend to behave less effectively for points distant from their center of convergence. It is easy to reduce $\sin x$ to $\pm \sin t$ where $-\frac{\pi}{2} \leq t \leq \frac{\pi}{2}$. If $t = x - n\pi$, then we seek an n that minimizes $|t|$. But such an n will also minimize the expression $\left|\frac{x-n\pi}{\pi}\right| = \left|\frac{x}{\pi} - n\right|$. The value of n that will minimize this expression is the integer that is closest to $\frac{x}{\pi}$, so n can be calculated by rounding $\frac{x}{\pi}$ to the nearest integer. $\sin x$ will then be the same as $\sin t$ if n is even, or $-\sin t$ if n is odd. To prove that $-\frac{\pi}{2} \leq t \leq \frac{\pi}{2}$, note that since n is the nearest integer to $\frac{x}{\pi}$, then $n = \frac{x}{\pi} + \delta$, where $|\delta| \leq \frac{1}{2}$. We then see that

$$|t| = |x - n\pi| = |x - (\frac{x}{\pi} + \delta)\pi| = |\delta|\pi \leq \frac{1}{2}\pi$$

Now that we know that we may assume $-\frac{\pi}{2} \leq t \leq \frac{\pi}{2}$, we can use the error formula for Taylor Series to determine the numbers of terms of the Taylor polynomial will give us the accuracy we desire for $\sin t$.

Estimating Truncation Error

The Taylor error formula is

$$|R_n| = \left| f^{(n+1)}(\xi) \frac{x^{n+1}}{(n+1)!} \right|, 0 \leq \xi \leq x$$

Since our function $\sin x$ and all its derivatives are no more than 1 in magnitude, we can say that

$$|R_n| \leq \left| \frac{x^{n+1}}{(n+1)!} \right| = \left| \frac{x^n}{(n+1)!} \right| |x|$$

Why $|x|$ was factored out will be made evident in the next paragraph.

Note that we want the truncation error, $|R_n|$, to be less than *half* the floating-point spacing near $\sin t$, so that adding R_n to the running Taylor polynomial will make no difference whatsoever—that means we will have obtained all the accuracy we possibly can. Recall that the spacing near a number, z, is either $\frac{\epsilon}{B}z$ or ϵz, where ϵ is machine epsilon and B is the floating-point base (2 in our case). We will be conservative in our expectations and use the larger of the two values and require that $R_n < \frac{1}{2}\epsilon|\sin t|$. Inspecting the quantity $|\frac{t}{\sin t}|$ in our interval of interest (graph it if you need to), we see that it is bounded above by $\frac{\pi}{2}$, which gives $|t| \leq \frac{\pi}{2}|\sin t|$. Using this fact we obtain

$$|R_n| \leq \left|\frac{t^n}{(n+1)!}\right||t| \leq \left|\frac{t^n}{(n+1)!}\right|\frac{\pi}{2}|\sin t| \leq \frac{(\frac{\pi}{2})^n}{(n+1)!}\frac{\pi}{2}|\sin t| = \frac{(\frac{\pi}{2})^{n+1}}{(n+1)!}|\sin t|$$

So we want the final term on the right to be less than half the spacing near $\sin t$:

$$\frac{(\frac{\pi}{2})^{n+1}}{(n+1)!}|\sin t| < \frac{1}{2}\epsilon|\sin t|$$

which finally gives us

$$\left(\frac{\pi}{2}\right)^{n+1} < \frac{\epsilon}{2}(n+1)!$$

The following program computes n for both single and double precision.

```
// findn.cpp: Finds the stopping term in the Taylor's series
// sufficient to maximize the precision of computing sin x.
// It computes both sides of (pi/2)^(n+1) vs. eps/2*(n+1)!
// and stops when the left is less than the right.

#include <cassert>
#include <cmath>
#include <iostream>
#include <limits>
using namespace std;

template<typename FType>
void findn() {
    cout << typeid(FType).name() << " => ";
    FType pi2 = FType(3.14159265358979323846) / 2;
    FType eps2 = numeric_limits<FType>::epsilon()/2;
    FType lhs = pi2;    // pi2^1
    FType rhs = eps2;   // eps2 * 1!
    int n = 0;
    while (lhs >= rhs) {
        lhs *= pi2;
        rhs *= ++n + 1;
    }
    cout << "n: " << n << ", lhs: " << lhs << ", rhs: " << rhs << endl;
}
```

```
int main(int argc, char* argv[]) {
    findn<float>();
    findn<double>();
}

/* Output:
float => n: 12, lhs: 354.453, rhs: 371.159
double => n: 21, lhs: 20636.6, rhs: 124789
*/
```

Since the even-index terms of the Taylor series for $\sin x$ are zero, the version for `float` only needs to go through term 11, which is $-\frac{x^{11}}{11!}$. Likewise, the `double` version goes through $\frac{x^{21}}{21!}$. Here is a program that implements the single-precision version and compares to the standard library results.

```
// fsin.cpp: Calculates sin(x) as a float. x is
// reduced to the interval [-pi/2, pi/2]. Uses
// the first 6 non-zero terms of the Taylor's Series
// (more would add nothing). Improvements still need
// to be made, however!
#include <iostream>
#include "../ieee.h"    // Defines a sign() function
using namespace std;

double pi = 3.14159265358979323846; const int k = 5;

float mysine(float x) {
   int n = int(x/pi + 0.5*sign(x)); // Uses double precision here
   x -= n*pi;
   float num = x;
   float den = 1.0f;
   float sum = x;
   int fact = 1;  // The denominator contains fact!

   for (int i = 0; i < k; ++i) {
      num = -num*x*x;
      den *= ++fact;
      den *= ++fact;
      sum += num/den;
   }
   return (n%2) ? -sum : sum;
}

int main() {
   cout << mysine(0.0f) << " (" << sin(0.0f) << ")\n";
   cout << mysine(pi/2.0f) << " (" << sin(pi/2.0f) << ")\n";
   cout << mysine(pi) << " (" << sin(pi) << ")\n";
```

```
   cout << mysine(3.0f*pi/2.0f) << " (" << sin(3.0f*pi/2.0f) << ")\n";
   cout << mysine(22.0f) << " (" << sin(22.0f) << ")\n";
   cout << mysine(5.0e8f) << " (" << sin(5.0e8f) << ")\n";
}

/* Output:
0 (0)
1 (1)
-8.74228e-008 (1.22461e-016)
-1 (-1)
-0.00885131 (-0.00885131)
-0.284704 (-0.284704)
*/
```

This version appears to be roughly equivalent to the standard library version. The output for double precision is:

```
0 (0)
1 (1)
0 (1.2246063538224e-016)
-1 (-1)
-0.0088513092904047
(-0.0088513092904039)
-0.28470409192472 (-0.28470407323816)
```

Improving Accuracy and Efficiency

There are a number of improvements we can make, some which are peculiar to $\sin x$ and some which are generally applicable.

"One-and-a-half" Precision

It is possible to get even more precision from a constant like π without resorting to higher-precision arithmetic. This technique is useful because you may not have a higher precision available. The idea is to split π into two parts, π_1 and π_2, which add to π. The special property of these two auxiliary constants is that they occupy overlapping decimal positions, so that one of constants contains decimals at a further decimal position than is normal obtained. Here are two possible values:

$$\pi_1 = 3.1416015625, \pi_2 = -8.908910206761537356617e-6$$

Note how π_2 holds a full contingent of digits for a `double`, but the exponent makes their digits several decimal places beyond what is normally stored in a `double` for π. When calculating $t = x - n\pi$, we use the auxiliary constants instead:

$$t = x - n\pi = x - n(\pi_1 + \pi_2) = x - n\pi_1 - n\pi_2$$

so we compute t in 2 steps:

```
t = x - n * pi1;
t -= n * pi2
```

The change in results for single precision shows improvement except for the very large angle (listed last):

```
0 (0)
1 (1)
-8.74228e-008 (1.22461e-016)
-1 (-1)
-0.00885131 (-0.00885131)
-0.284704 (-0.284704)
```

The results for double precision show improvement in each changed result:

```
0 (0)
1 (1)
1.2246402351403e-016 (1.2246063538224e-016)
-1 (-1)
-0.0088513092904039 (-0.0088513092904039)
-0.28470407323764 (-0.28470407323816)
```

The effect of this trick is to use more digits (approximately one-and-a-half times) than normal precision. Since many digits of π are readily available, this is an accessible technique.

Special Handling for Small Angles

It is well known that for small arguments, $\sin x \approx x$, so for sufficiently small x, it may well be better to just return x instead of risking underflow or further rounding error. The question is, how small does x need to be before we can just return x without losing accuracy? To get an answer, we rewrite the Taylor polynomial as follows:

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} \ldots - \frac{x^{19}}{19!} + \frac{x^{21}}{21!} = x(1 - \frac{1}{6}x^2 + ...) = x(1 + r) = x + xr$$

When $x^2 = \epsilon$ (machine epsilon), then $|r| \approx \frac{1}{6}\epsilon$ (because x is very small we can ignore the subsequent terms). If we call our approximation of $\sin x$ by the name s, then $s = x + xr$. The difference between s and x is therefore

$$|s - x| = |xr| \approx \frac{1}{6}\epsilon|x|$$

Note that this is less than the spacing near x by at least a factor of 3, so we conclude that s and x are indistinguishable by our floating-point system. We can consequently tune our algorithm to just return $\pm x$ for $\sin x$ whenever $|x| < \sqrt{\epsilon}$.

Special Handling for Large Angles

If x is large enough, the integer $n = [\frac{x}{\pi}]$ will overflow, yielding undefined results. For a 32-bit platform, this occurs when $\frac{x}{\pi} > 2^{31} - 1$, or $x \approx 6.7$ billion. Probably due to error in representing π

coupled with roundoff in the division, experimental results show that 1 billion is a safer threshold, so for numbers greater than this value, our routine should return a NaN. Some libraries try to return *something* for large angles, but you usually can't trust what you get. For example, for the number 10^{30}, Microsoft C++ 2005 gives 0.738533997569377, GNU C++ gives -0.838299394218616 and Dinkumware gives 0.990769652599209. Many scientific calculators report a domain error, although one gave a value of -0.09103119, another -0.863505811, and Windows Calculator gave -0.090116901912138058030386428952987. The following program using Java's arbitrary-precision `BigDecimal` and `BigInteger` classes with 40 decimals declares Windows Calculator the "winner." (It has a precision of 32 *decimal* digits.)

```
import java.math.*;

class BigSine {
    public static void main(String[] args) {
        BigDecimal t = residue(new BigDecimal("1.0e30"));
        System.out.println("t = " + t.toEngineeringString());
        System.out.println("sin t = " + Math.sin(t.doubleValue()));
    }
    static BigDecimal residue(BigDecimal arg) {
        // Find nearest integer to arg/pi
        String pi1 = "3.14159265358979323846264338327";
        String pi2 = "95028841971693993751058209749 44";
        BigDecimal pi = new BigDecimal(pi1 + pi2);
        BigDecimal quotient = arg.divideToIntegralValue(pi);
        System.out.println("n = " + quotient);
        n = quotient.toBigInteger();
        return arg.subtract(quotient.multiply(pi));
    }
}

/* Output:
n = 318309886183790671537767526745
t = 0.090239323898053028031181587905554138877362184227620505122720
sin t = 0.09011690191213806
*/
```

This doesn't explain what happened to the `float` version of $\sin x$ computed above for $x =$ `5.0e8`. The problem there is that in that range, the spacing between floating-point numbers is greater than 1, so n may be calculated incorrectly even before it is converted to (the wrong!) integer. The spacing is greater than 1 starting at the interval beginning with $2^{24} \approx 1.678 \times 10^7$, so for single precision it is safer to return a NaN for values greater than 10^7. Any sine approximations for arguments above these thresholds (10^9 for double precision and 10^7 for single precision) are likely to be suspect unless arbitrary-length precision is used.

Economization of Polynomials

Since the natural behavior of Taylor polynomials is to be less accurate at the endpoints, techniques exist that more evenly distribute the error throughout the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The theory of these so-called "minimax" polynomials are based on a technique developed by Chebyshev and are beyond the scope of this note. Suffice it to say that transforming a truncated power series by this technique lessens the errors at the endpoints by decreasing the magnitude of the coefficients of the higher power terms and adjusting the other coefficients as needed. Another benefit of using Chebyshev polynomials is that the transformed Taylor polynomial has fewer terms, hence the term *economization* of polynomials. (It is also called "telescoping a series", but this is easily confused with the other common usage of "telescoping", where a finite series loses all but its first and last terms). Cody and Waite [1] show that for a precision of 51 to 60 bits, the following economized polynomial delivers acceptable, evenly-distributed accuracy:

$$x + c_1x^3 + c_2x^5 + c_3x^7 + c_4x^9 + c_5x^{11} + c_6x^{13} + c_7x^{15} + c_8x^{17}$$

where the coefficients, c_i, are:

$$\begin{aligned}
c_1 &= -0.16666666666666665052 \\
c_2 &= 0.83333333333331550314 \times 10^{-2} \\
c_3 &= -0.19841269841201840457 \times 10^{-3} \\
c_4 &= 0.27557319210152756119 \times 10^{-5} \\
c_5 &= -0.25052106798274584544 \times 10^{-7} \\
c_6 &= 0.16059036490371589114 \times 10^{-9} \\
c_7 &= -0.76429178068910467734 \times 10^{-12} \\
c_8 &= 0.27204790957888846175 \times 10^{-14}
\end{aligned}$$

The degree of our polynomial has decreased by 4 and the number of coefficients by 2. We are almost ready to call our algorithm "done."

Horner's Rule

Our final tweak minimizes the number of operations required to evaluate our polynomial by rearranging it. Instead of computing powers of x each time, we can use *Horner's Rule*, which factors a polynomial into a sequence of nested operations. For example, suppose we want to evaluate the polynomial

$$c_3x^3 + c_2x^2 + c_1x + c_0$$

If we naively evaluate this by computing powers for each term, this will take 6 multiplications and 3 additions. Rewriting this polynomial as

$$c_0 + x(c_1 + x(c_2 + x(c_3)))$$

only 3 multiplications are required. In general, Horner's Rule cuts the number of multiplications in half. An algorithm for evaluating a polynomial of degree n by Horner's Rule is described in the following pseudocode:

```
sum = c[n]
for i = n-1 downto 0
    sum = c[i] + x*sum
```

This algorithm assumes that the c_i are available for *every* power of x, so we would need to include the 0-valued coefficients for the even powers of x from our truncated sine series. We can avoid these needless multiplications by 0 if we rewrite our polynomial like the following:

$$x\left(1 + c_1x^2 + c_2x^4 + c_3x^6 + c_4x^8 + c_5x^{10} + c_6x^{12} + c_7x^{14} + c_8x^{16}\right)$$

Substituting $y = x^2$ gives:

$$x\left(1 + c_1y + c_2y^2 + c_3y^3 + c_4y^4 + c_5y^5 + c_6y^6 + c_7y^7 + c_8y^8\right)$$

We can now evaluate the inner polynomial by Horner's Rule on $y = x^2$, and then multiply the result by x to obtain the final magnitude for $\sin t$.

Summary

To summarize this development, our algorithm for double precision should perform the following steps:

1. If our argument, x, is ∞ or a `NaN`, return `NaN`.
2. If $|x| > 10^9$, return a `NaN`.
3. Reduce x by a factor of $n\pi$ to t where $-\frac{\pi}{2} \leq t \leq \frac{\pi}{2}$ (use the two-step procedure with π_1 and π_2 explained earlier). Record n.
4. If $|t| < \sqrt{\epsilon}$, return $\pm t$, depending on whether n is even or odd.
5. Compute $\sin t$ by Horner's Rule as explained above.
6. If n is even, return the result from the previous step. Otherwise, returns its negation.

References

[1] W. J. Cody and W. Waite. *Software Manual for the Elementary Functions.* Prentice-Hall, 1980.

[2] W. Miller. *The Engineering of Numerical Software.* Prentice-Hall, 1984.