Piles of Sand, Redux

by Chuck Allison

If my previous article left you desperately wanting to know why certain compilers miscalculate `sin(x)` for large arguments and why some get it right, your wait is over.

The results I reported for `sin(10`^{30}`)` are shown in figure 1.

And the winner is . . . Windows Calculator! Read on to discover why.

Recall from the previous article that spacing between floating-point numbers changes every time you cross a power of the floating-point base (2 for IEEE numbers) and that the spacing between numbers near `x` is 2^{1-p+e}, where `p` is the precision (24 for `float`, 53 for `double`) and `e` is the exponent of 2 used in the binary IEEE representation of `x`.

Granted, you may never need to compute `sin(10`^{30}`)`—or the sine of anything for that matter—but we all want that warm, fuzzy feeling that we can trust code libraries.

ISTOCKPHOTO

Microsoft Visual C++ 2005	-0.756263
GNU g++ 3.4.4 (under Cygwin)	0.00933147
Java SDK 1.5.0_08	0.009331468931175825
Python 2.5	-0.75626273033357649
HP 11C calculator	-0.863505811
Windows Calculator	-0.090116901912138058030386428952987

Figure 1

So how do they compute `sin(x)`, anyway? Algorithms for `sin(x)` take advantage of the periodicity of trigonometric functions by reducing `x` to an "equivalent" value in a small range about zero. Most implementations subtract the appropriate multiple of π or π /2 to end up in the intervals [-π /2, π /2] or [-π /4, π /4], respectively. And that's where the difficulty lies.

```
double x = sqrt(-1.0);          // One way to beget a NaN
cout << x << endl;
double y = x + 2.0;             // Can't shake it!
cout << y << endl;

// Output:
-nan
-nan
```

Listing 1

Consider what happens when determining the number `n`, such that `t=x-n`π is in the interval [-π /2,π /2] and `sin(t)= ±sin(x)`. It turns out that `n` is the closest integer to `x`/π, so statements such as the following are executed:

```
int n = int(x/pi + 0.5*sign(x));
t = x - n*pi;
```

In the case of `x` = 10^{30}, `n` is approximately 3.18 * 10^{29}. Can you say "integer overflow"? It would take ninety-nine bits to store such a signed integer. Unless you have 128-bit hardware handy, it is not feasible to even attempt the calculation, and even with 128 bits, things break at 5.345 * 10^{38} anyway. Since integers overflow silently, the problem can go undetected.

Even if you had all the bits you needed, another problem arises in computing the nearest integer to `x`/π. Since the expression `x/pi + 0.5*sign(x)` yields a floating-point number, there better not be any integer "holes" in the vicinity. But for `x` = 10^{30}, the exponent in the IEEE representation of `x`/π is 98 (because 10^{30}/π = 1. 00000001… * 2^{98}), making the inter-number spacing there $2^{98-52} \approx 7.04 \times 10^{13}$. Uh, that skips over quite a few integers, so the chances of finding the nearest integer to $10^{30}/\pi$ are slim to nonexistent!

What should library developers do? According to William Kahan, the "Old Man of Floating-point," they should return NaN when they can't guarantee an acceptable answer. NaN, which stands for *Not a Number*, is a special IEEE floating-point value that taints all calculations it touches. Once you get a NaN, you can't get rid of it, as the code snippet in listing 1 illustrates.

Returning a NaN is much better than misleading users.

Infinity is another IEEE value that comes in handy. It is well behaved in that if you divide a number by it, you get 0, as expected. This allows certain formulas to play nice when dividing by zero, such as the one from electronics shown in listing 2.

In the first calculation, 1.0/x evaluates to infinity, so the final result is $1/\infty = 0$. The second invocation returns $1/(0 + \frac{1}{2}) = 2$.

Now, why did Windows Calculator compute the right answer for `sin(10³⁰)`? Because it uses a 128-bit representation for its floating-point numbers—so none of the problems described above apply. To validate its glorious triumph, the program in listing 3 uses Java's arbitrary-precision arithmetic class, `BigDecimal`, to compute the correct `n` that reduces 10^{30} to its corresponding argument in $[-\pi/2, \pi/2]$. From there it just uses the built-in sine function. The result agrees with Windows Calculator.

```
#include <limits>
#include <iostream>
using namespace std;

double resistance(double x, double y) {
    return (1.0 / (1.0/x + 1.0/y));
}

int main() {
    cout << resistance(0.0, 1.0) << endl;
    cout << resistance(2.0, numeric_limits<double>::infinity()) << endl;
}

// Output:
0
2
```

Listing 2

```
import java.math.*;

class BigSine {
    static BigInteger n;
    static BigInteger two = new BigInteger("2");
    public static void main(String[] args) {
        BigDecimal t = residue(new BigDecimal("1.0e30"));
        System.out.println("t = " + t.toEngineeringString());
        System.out.print("sin t = ");
        if (n.mod(two).equals(BigInteger.ONE))
            System.out.print("-");
        System.out.println(Math.sin(t.doubleValue()));
    }
    static BigDecimal residue(BigDecimal arg) {
        // Find nearest integer to arg/pi
        String pi1 = "3.14159265358979323846264338327";
        String pi2 = "9502884197169399375105820974944";
        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
```

Listing 3

Tuning Algorithms

Understanding floating-point spacing is the key to getting the most from numeric computations. Consider the method of "bisection" for finding roots of equations. It starts with an interval `[a,b]` that contains a sign change in the function `f(x)`. It first inspects the interval's midpoint, `x = (a+b)/2`. If `f(x)` ≠ 0, it replaces either `a` or `b` with `x`, depending on whether the interval `(a,x)` or the interval `(x,b)` preserves the sign change. Listing 4 shows how some people implement it.

The variable `tol` is the user's "tolerance"—meaning if `a` and `b` are at most `tol` apart, users will call that "close enough." Can you see the problem? Of course you can! The spacing between floating-point numbers near `a` or `b` might be greater than `tol`, resulting in an infinite loop. When that happens, the expression `(a+b)/2.0` will return either `a` or `b`, and we're off to Spin City!

There are two alternatives. The first is to adjust the tolerance to be no smaller than the inter-number spacing near a and b. While you could use the formula mentioned above to get the exact spacing between floating-point numbers, it is more efficient—and quite sufficient—to compute an approximation for the spacing up front. Recall that:

$$2^e \leq |x|$$

where e is the exponent in the IEEE representation of x. Multiplying both sides by 2^{1-p}, where p is the floating-point precision, we get:

$$2^{1-p+e} \leq 2^{1-p}|x|$$
$$\Rightarrow \epsilon 2^e \leq \epsilon|x|$$

(Recall from last time that 2^{1-p} is *machine epsilon*, denoted by ϵ.) Since the term on the left side of the inequality is the spacing in question, we have a ready upper bound for it: $\epsilon|x|$. With that in mind, we can avoid the possibility of an infinite loop by prefacing the `while` loop in listing 4 with the statements shown in listing 5. This guarantees that `tol` does not exceed any of the floating-point spacings in the interval [`a,b`]. Note that machine epsilon is provided for you in C++ via `numeric_limits::epsilon()`.

If you want to get maximum machine accuracy, you can dispense with the tolerance altogether and just continue bisecting until a and b become adjacent `doubles`, or until you get lucky and stumble on a root. As explained earlier, you'll know that a and b are adjacent if (a+b)/2.0 comes back as a or b. The version shown in listing 6 checks for that and makes other simplifications.

```
double bisect(double tol, double a, double b, double f(double)) {
   while ((b-a) > tol) {
      double c = (a+b)/2.0;
      if (f(a)*f(c) < 0)
         b = c;
      else if (f(b)*f(c) < 0)
         a = c;
      else
         return c;
   }
   return (a + b) / 2.0;
}
```

Listing 4

```
double eps = numeric_limits<double>::epsilon();
tol = max(tol, eps*abs(a));
tol = max(tol, eps*abs(b));
```

Listing 5

```
double bisect(double a, double b, double f(double)) {
    for (;;) {
        double c = (a+b)/2.0;

        // Are a and b adjacent?
        if (c == a || c == b)
            return a;   // Or could return b

        double fc = f(c);
        if (fc == 0.0) {
            return c;   // Stumbled across a zero.
        }
        else if (sign(f(a)) == sign(fc))
            a = c;
        else
            b = c;
    }
}
```

Listing 6

Summary

Even if you're not a scientific programmer, you likely will use floating-point arithmetic from time to time, or you may test code that does. Understanding the architecture of a floating-point number system—and, in particular, the effects of inter-number spacing—can help you attain the highest accuracy possible while avoiding classic blunders. **{end}**

Chuck Allison developed software for twenty years before becoming a professor of computer science at Utah Valley State College. He was senior editor of the C/C++ Users Journal *and is founding editor of* The C++ Source. *Chuck is the author of two C++ books and gives onsite training in C++, Python, and design patterns.*