Piles of Sand

by Chuck Allison

ISTOCKPHOTO

Computers. They're in our cars, our phones, our entertainment systems. They manage our paychecks, our medical records, our credit. In fact, for most of us, our money consists of bits in some computer located who-knows-where. Computers were invented for number crunching, but now they do just about everything. Cool.

Somewhere along the way, however, we seem to have forgotten how floating-point numbers operate. Consider the simple calculation in figure 1.

This code subtracts 0.1 from 1.0 ten times, resulting in zero, right? Well, not quite. The output is `-7.45058e-08`.

Why does this happen?

This number may be close enough to zero—or it may not be—depending on what you're doing. When in doubt, you can use `double` instead of `float`, which yields a result of `1.38778e-16`. Certainly we can't expect to get any closer to zero than that. Or can we? Over time errors like this can accumulate, which can have serious consequences such as causing missile defense systems to miss their targets.

```
float x = 1.0f;               // Set x to the floating point number 1.0
for (int i = 0; i < 10; ++i)  // Run a loop ten times to
    x -= 0.1f;                // Subtract 0.1 from x
cout << x << endl;            // Does not print 0!
```

Figure 1

Even stranger, if we change the numbers just a little, as in figure 2, everything works fine.

All we did was multiply the numbers involved by a factor of five. What gives?

With computers affecting virtually every aspect of our society, perhaps we have made a mistake by focusing on "data processing" while neglecting computing's numeric roots. To prevent future computational difficulties, developers need to understand floating-point arithmetic if they are going to attain maximum accuracy.

```
float x = 5.0f;               // Start with 5.0 floating point instead of 1.0
for (int i = 0; i < 10; ++i)
    x -= 0.5f;                // Subtract 0.5 ten times
cout << x << endl;            // Prints exactly 0
```

Figure 2

Floating-Point Number Systems

A computer's floating-point number system is modeled after scientific notation as taught in school—you remember, 1.234567×10^3. Each such number system is characterized by four parameters:

1. The numeric base of the digits (β; usually 2 or 10)
2. The fixed number of digits in the mantissa (`p`, aka the "coefficient" or "significand")
3. The minimum exponent allowed (`m`; negative, of course, to allow for fractions)
4. The maximum exponent allowed (`M`)

Floating-point number systems usually are *normalized* meaning they allow exactly one non-zero digit before the decimal (radix) point. Such numbers consist of digits in some numeric base in the following form:

$$\pm d_0.d_1d_2 \ldots d_{p-1} \cdot \beta^e, \text{ where } d_0 \neq 0,\ 0 \leq d_i < \beta, \text{ and } m \leq e \leq M$$

In a fixed-point number system (such as the integers), numbers are evenly spaced. Not so with floating-point numbers. Since each mantissa is multiplied by a power of the base, β, the spacing is determined by the value of the unit in the last place multiplied by β^e.

Consider a normalized number system with the parameters $\beta = 2$, $p = 4$, $m = -1$, and $M = 1$. Since the first digit must be 1, there are only eight distinct mantissas possible. In this example,

Bit patterns	x 2^{-1}	x $2^0 = 1$	x 2^1
(1).000	.1	1	10
(1).001	.1001	1.001	10.01
(1).010	.101	1.01	10.1
(1).011	.1011	1.011	10.11
(1).100	.11	1.1	11
(1).101	.1101	1.101	11.01
(1).110	.111	1.11	11.1
(1).111	.1111	1.111	11.11
Spacing:	(.0001)	(.001)	(.01)

Table 1: Calculating the possible mantissas in a normalized number system

there are three possible exponents, so there are $8 \cdot 3 = 24$ distinct magnitudes, as table 1 illustrates. (The mantissas go down each column—each column represents the numbers between powers of the base. All numbers are in base 2.)

Note that the spacing between numbers in the first interval is $.001 \cdot 2^{-1} = .0001$ (1/16), and that it increases by powers of two as you move each power interval to the right. This has some interesting consequences. When the numbers get large enough in magnitude in a typical floating-point system, even some integers are not representable, since the inter-number spacing becomes greater than one.

This spacing between floating-point numbers is easy to calculate. Since the unit in the last place is in the (`p-1`)-th decimal position, it is the number β^{1-p} ($2^{1-4} = .001_2$ in the example above) times β^e, where e is the exponent of the base for the current interval.

The IEEE standard for floating-point arithmetic (IEEE 754) specifies the parameter values in table 2 for single precision (`float`) and double precision (`double`) numbers.

This means that the spacing between floats is $2^{1-24} \cdot 2^e = 2^{e-23}$, where e is the exponent of the interval in question. So, for example, the spacing between numbers between 1.0f and 2.0f is 2^{-23}. We now can discover where we'll start dropping integers by solving for e in $2^{e-23} > 1 = 2^0$, giving $e > 23$. So in the interval starting with 2^{24}, the spacing between adjacent floats is two, so every other integer there isn't even representable. This explains the surprise in figure 3.

	β	p	m	M
Single precision (`float`)	2	24	-126	127
Double precision (`double`)	2	53	-1022	1023

Table 2

The largest 32-bit signed integer is of course 2^{31}-1, but when stored as a `float`, out comes 2^{32} = 2,147,483,648. This isn't just an off-by-one error. We get the same value when we subtract 64, but subtracting 65 gives us 2,147,483,520. Now you know why—the spacing there is 128 ($2^{30-23}=2^7$), and the nearest number in the floating-point system is chosen.

```
// lostints.cpp: Reveals floating-point "holes"
#include <iostream>
#include <limits>
using namespace std;

int main() {
    int m = numeric_limits<int>::max(); // The largest int
    cout << m << endl;
    float x = m;
    cout << fixed << x << endl;
    cout << x - 64.0f << endl;
    cout << x - 65.0f << endl;
}

/* Output:
2147483647
2147483648.000000
2147483648.000000
2147483520.000000
*/
```

Figure 3

Sources of Numerical Error

What we've just seen is called "rounding error" or, more traditionally, "roundoff." Roundoff occurs when a real number is not present in a floating-point number system so the closest candidate takes its place. The *absolute* error due to roundoff can be quite large, since the spacing between floating-point numbers can be large, but the *relative* error due to roundoff is bounded by the quantity β^{1-p}, which we saw earlier. This number comes up often enough that it has a special name, *machine epsilon* (ϵ), and is available in C++ via the function `numeric_limits<>::epsilon()`, defined in the `<limits>` header.

Now we can explain the first two programs in this article. The first used the constant .1 as the loop decrement value. This is a fine decimal number, but in binary it is an infinite repeating fraction (.000110011001100…). Since this number is not representable in a finite binary system, the closest floating-point number is used instead. After a while the roundoff starts to be noticeable. The second program used the number .5, a number exactly representable in a binary system—so no roundoff occurs. Understanding and controlling numerical error is a useful skill known to too few developers, including library developers.

> **"When the numbers get large enough in magnitude in a typical floating-point system, even some integers are not representable, since the inter-number spacing becomes greater than one."**

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 4

To illustrate how difficult it can be to craft quality libraries, figure 4 gives a sample of results for the mathematical function `sin(x)` for x = 10^{30} from various sources:

Hmmm. Microsoft and Python use the same algorithm, as do Java and GNU. All are wrong. Windows Calculator got it right. How do I know? I'd love to tell you, but I've run out of space for this month. Let me conclude by quoting from Kernighan and Plauger's classic, *The Elements of Programming Style*:

"Floating-point numbers are a lot like sandpiles: Every time you move one you lose a little sand and pick up a little dirt."

Come back next time, when all shall be revealed. **{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*. He is also the author of two C++ books and gives onsite training in C++, Python, and design patterns.*

Do you routinely verify the results of your floating-point calculations, or do you just accept what the computer gives you? Do you have any floating-point horror stories to share?

Follow the link on the StickyMinds.com homepage to join the conversation.

Don't forget to log on to

StickyMinds.com to post comments and questions about this or any of the previous Code Craft, Test Connection, or Management Chronicles columns.

Get insight from some great industry minds including Tod Golding, Chuck Allison, Michael Bolton, Esther Derby, Johanna Rothman, Peter Clark, and many others.

Topics covered in these columns include: the importance of code reviews, handling overtime requests from management, understanding Ajax, tracking down bugs with log files, the joy of discovery, why using a product yourself is the best way to test it, catching failures with regression tests, and many more.

Don't miss your chance to chat with an expert. Follow the links on the StickyMinds.com homepage or search by author.