Floating-Point Numbers Aren't Real

Chuck Allison

FLOATING-POINT NUMBERS ARE NOT "REAL NUMBERS" in the mathematical sense, even though they are called real in some programming languages, such as Pascal and Fortran. Real numbers have infinite precision and are therefore continuous and nonlossy; floating-point numbers have limited precision, so they are finite, and they resemble "badly behaved" integers, because they're not evenly spaced throughout their range.

To illustrate, assign 2147483647 (the largest signed 32-bit integer) to a 32-bit `float` variable (`x`, say), and print it. You'll see 2147483648. Now print `x-64`. Still 2147483648. Now print `x-65`, and you'll get 2147483520! Why? Because the spacing between adjacent floats in that range is 128, and floating-point operations round to the nearest floating-point number.

IEEE floating-point numbers are fixed-precision numbers based on base-two scientific notation: $1.d_1d_2...d_{p\ 1} \times 2^e$, where p is the precision (24 for `float`, 53 for `double`). The spacing between two consecutive numbers is 2^{1-p+e}, which can be safely approximated by $\varepsilon|x|$, where ε is the *machine epsilon* (2^{1-p}).

Knowing the spacing in the neighborhood of a floating-point number can help you avoid classic numerical blunders. For example, if you're performing an iterative calculation, such as searching for the root of an equation, there's no sense in asking for greater precision than the number system can give in the neighborhood of the answer. Make sure that the tolerance you request is no smaller than the spacing there, otherwise you'll loop forever.

Since floating-point numbers are approximations of real numbers, there is inevitably a little error present. This error, called *roundoff*, can lead to surprising results.

When you subtract nearly equal numbers, for example, the most significant digits cancel one another out, so what was the least significant digit (where the roundoff error resides) gets promoted to the most significant position in the floating-point result, essentially contaminating any further related computations (a phenomenon known as *smearing*). You need to look closely at your algorithms to prevent such *catastrophic cancellation*. To illustrate, consider solving the equation $x^2 - 100000x + 1 = 0$ with the quadratic formula. Since the operands in the expression $-b + sqrt(b^2 - 4)$ are nearly equal in magnitude, you can instead compute the root $r_1 = -b - sqrt(b^2 - 4)$, and then obtain $r_2 = 1/r_1$, since for any quadratic equation, $ax^2 + bx + c = 0$, the roots satisfy $r_1r_2 = c/a$.

Smearing can occur in even more subtle ways. Suppose a library naïvely computes e^x by the formula $1 + x + x^2/2 + x^3/3! + \ldots$. This works fine for positive x, but consider what happens when x is a large negative number. The even-powered terms result in large positive numbers, and subtracting the odd-powered magnitudes will not even affect the result. The problem here is that the roundoff in the large, positive terms is in a digit position of much greater significance than the true answer. The answer diverges toward positive infinity! The solution here is also simple: for negative x, compute $e^x = 1/e^{|x|}$.

It should go without saying that you shouldn't use floating-point numbers for financial applications—that's what decimal classes in languages like Python and C# are for. Floating-point numbers are intended for efficient scientific computation. But efficiency is worthless without accuracy, so remember the source of rounding errors, and code accordingly!