CSC165H, Mathematical expression and reasoning for computer science
week 12

22nd December 2005

Gary Baumgartner and Danny Heap
heap@cs.toronto.edu
SF4306A
416-978-5899
http://www.cs.toronto.edu/~heap/165/S2005/index.shtml

Rounding

Most numbers are not exactly representable in a floating-point system using a given base β. For example, when $\beta = 10$, you cannot represent $1/3$ exactly (no matter how large t is), so we used 3.33×10^{-1} when $t = 3$. What should we do with something like the base of natural logarithms, $e = 2.718281828\ldots$? Two approaches are used:

- Round to nearest: 2.72×10^0.
- Truncate to zero: 2.71×10^0.

Overflow

There is no way to represent a number larger than the largest floating-point number. In our first example ($\beta = 10, t = 3, e \in [-4, +4]$), there is no way to represent 99901 or greater.

Underflow

There is no way to represent a positive number smaller than the smallest positive floating-point number. In our first example, there is no way to represent 0.00001 (or a smaller positive number).

Absolute rounding error

We can calculate the difference between the true value we're trying to represent and the value of its floating-point representation. For example, the absolute error in our representation of e is $|2.71 - 2.718281828\ldots| = |0.008281828\ldots|$.

Relative error

100 and 100.1 are "closer" than 1 and 1.1, even though the absolute difference is 0.1 in both cases. Look at the size of the error in terms of the size of the value being represented.

Relative error: For $x \neq 0$, the relative error between the approximate value x' and the "real" value x is

$$\frac{|x - x'|}{|x|}$$

For example, $|1.1-1|/|1.1| = 0.0909 \approx 9\%$. However, $|100.1-100|/|100| = 0.000999\cdots \approx 0.1\%$.

Relative error in round-to-nearest

When we round numbers to represent them in a floating-point system, can we bound relative error for positive numbers with no overflow or underflow?[1].

In general, a number of the form

$$d_0.d_1 \cdots d_{t-1}d_t \cdots \times \beta^e$$

...gets rounded either up or down to one of

$$d_0.d_1 \cdots (d_{t-1} + 1) \times \beta^e$$
$$d_0.d_1 \ldots d_{t-1} \times \beta^e$$

The representation of the first number may be different (the $+1$ may cause a number of "carries" in the addition), but the value is the same. The difference between these two numbers is simply a 1 in the position occupied by d_{t-1}, for a difference of $0.00\cdots 1 \times \beta^e = \beta^{e-(t-1)}$. Since we round to nearest, our error is at most half this value:

$$\frac{\beta^{e-(t-1)}}{2}.$$

The relative error can be calculated, since we use the convention that the leading digit, d_0, is non-zero, so the smallest denominator (hence the largest bound) is when $d_0 = 1$, giving a relative error of:

$$\frac{|\beta^{e-(t-1)}/2|}{|1.0\cdots 0 \times \beta^e|} = \frac{\beta^{1-t}}{2}$$

This matches our intuition that by increasing t (the number of digits) we get more precision.

In our example of a floating-point system with $\beta = 2, t = 3$, this gives a bound on the relative error of round-to-nearest of $2^{1-3}/2 = 1/8$. This is also clear from the number line or the 24 values in this representation.

Accumulation of error

Since we can't represent all real numbers exactly in a floating point representation, what happens when we repeat operations. For example, take $\beta = 10$, $t = 3$, $e \in [-2, +2]$, and consider the sum

$$100 + 0.1 + 0.1 + \cdots + 0.1 \qquad [n \text{ times}]$$

The first part of the sum, $100 + 0.1$ is represented by 1.00×10^2. As we add additional terms with value 0.1, the result is still 1.00×10^2, since our representation cannot represent that additional 0.1. The relative error can become arbitrarily large, given large enough n. If you play around with FloatExample.cumulativeError (on the web page), with say t1 = 1.0, $n \geq 10$, and t2 = 10^{-16}, you'll see this demonstrated.

The easy way to work around this particular problem is to add the 0.1 terms first, and then add the 100 — in other words, addition is NOT associative for floating-point numbers: $(a + b) + c \neq a + (b + c)$.

Catastrophic cancellation

Use the same floating-point system as in the previous example to compute $b^2 - 4ac$ for $b = 3.34$, $a = 1.22$, and $c = 2.28$. The exact value is $0.0292 = 2.92 \times 10^{-2}$, and this exact value is representable in our floating-point system. Look at how the value is calculated, though:

$$\begin{aligned}
b^2 &= (3.34)^2 \\
&= 11.1556 \approx 1.12 \times 10^1 \\
4ac &= 4 \times 1.22 \times 2.28 \\
&= 4.88 \times 2.28 \\
&= 11.1264 \approx 1.11 \times 10^1 \\
b^2 - 4ac &\approx 1.12 \times 10^1 - 1.11 \times 10^1 \\
&= 0.01 \times 10^1 = 1.00 \times 10^{-1}
\end{aligned}$$

Compared to our exact answer of 2.92×10^{-2}, this has a relative error of

$$\frac{|0.0292 - 0.1|}{0.0292} = \frac{0.0708}{0.0292} = 2.424 \cdots > 240\%.$$

Subtracting two floating-point numbers that are very close together leaves very few significant digits — a great deal of information is lost. Since the true value is very small, the round-off error becomes much more significant, and sometimes becomes much larger than the value being computed (see above).

The expression $b^2 - 4ac$ crops up in the solution to the quadratic equation $ax^2 + bx + c = 0$. The general form of the solution, for the two roots x_1 and x_2 is

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

We may not have to worry about the large relative error in $b^2 - 4ac$, since it may be small in absolute value compared to $-b$. Here's a case where computing x_1 (using the values that lead to catastrophic cancellation above) gives a fairly acceptable value using floating-point operations:

$$x_1 = \frac{-3.34 + \sqrt{0.1}}{2 \times 1.22} = \frac{-3.34 + 0.316}{2.44} = -1.24$$

Compare this to the result if there were no error in the computation of $b^2 - 4ac$, which is:

$$x_1 = \frac{-3.34 + \sqrt{0.0292}}{2 \times 1.22} = \frac{-3.34 + 0.171}{2.44} = -1.30$$

. . . for a relative error of less than 5%.

Stability

There is a built-in problem with the formulas used above to compute the roots of a quadratic equation: if $b^2 - 4ac$ is close to b^2, then there will be catastrophic cancellation between $-b$ and $+\sqrt{b^2-4ac}$. This is separate from the catastrophic cancellation that may happen if b^2 is close to $4ac$.

Definition: a formula (or algorithm) is called "unstable" iff errors in the input values get magnified during the computation (i.e., iff the relative error in the final answer can be larger than the relative error in the input values).

Dealing with instability

Our first example $(100 + 0.1 + \cdots)$ was unstable, but there was an easy way to use a different algorithm that is stable: perform the operations in a different order.

Our second example $(b^2 - 4ac)$ was also unstable, because of potential catastrophic cancellation, and there is, unfortunately, no easy fix. You could increase the number of significant digits to make the round-off error smaller, but you will still have the potential for catastrophic cancellation when you subtract numbers that are very close, even with your increased precision. The formula is unstable.

Our third example is also unstable (it includes the instability of example 2, plus its own instability), but the possibility of cancellation between $-b$ and $+\sqrt{b^2-4ac}$ can be avoid by changing the formula. Suppose $b > 0$ (otherwise swap the role of x_1 and x_2 below if $b < 0$). Then avoid the subtraction in the numerator of the quadratic formula:

$$x_2 = \frac{-b - \sqrt{b^2-4ac}}{2a}.$$

Since both $-b$ and $-\sqrt{b^2-4ac}$ have the same (negative) sign, there will be no catastrophic cancellation. Now compute x_1 using x_2

$$x_1 = \frac{c}{ax_2} \qquad \text{(multiply the two roots to see this)}$$

This formula involves no subtraction, so there is no catastrophic cancellation.

There are two ways, in general, to deal with unstable formulas or algorithms:

- Increase the precision (the number of significant digits). This does not change the fact that the formula or algorithm is unstable, but can help minimize the magnitude of the errors,for some inputs.
- Use a different, stable, algorithm or formula to compute the result. When possible, this is preferred.

Conditioning

In the previous example we examined the error produced in calculating the roots of $ax^2 + bx + c$. The numbers a, b, and c may come from measurement and already have some error associated with them. By using a stable algorithm we get a relatively correct answer for slightly incorrect inputs.

But, independently of the particular algorithm used to compute roots, what can be said about the effect that errors in the measurement of a, b, and c have on values of roots? That is, if the values of a, b, or c change slightly, could the roots change dramatically?

To simply the discussion, let's look at the special case of a quadratic formula $x^2 - c = 0$ (so we're finding $\sqrt{c}$).

Suppose that $c = 0.25$, but we use a bad approximation $c' = 0.36$ instead of the true value. Then our answer will be 0.6 instead of 0.5. The relative error in the input is $0.11/0.25 = 0.44$, while the relative error in the result is $0.1/0.5 = 0.2$. Taking the square root makes the relative error smaller!

This isn't something special about the particular case we chose. Let's work the algebra to see the general case of computing $\sqrt{c}$ using an approximation c' of c. The ratio of the relative error of the result to the relative error of the input is (assuming c and c' are close enough to have the same sign):

$$\frac{|\sqrt{c} - \sqrt{c'}|/|\sqrt{c}|}{|c - c'|/|c|} \quad = \quad \sqrt{c}\frac{|\sqrt{c} - \sqrt{c'}|}{|c - c'|} \quad [\ |c|/\sqrt{c} = \sqrt{c}] \ = \frac{\sqrt{c}}{\sqrt{c} + \sqrt{c'}} \quad [\ |c - c'| = (\sqrt{c} + \sqrt{c'})(\sqrt{(c)} -$$

This ratio is always less than 1 (so the error improves), and when c' is very close to c, the ratio is close to 1/2. So the error is never increased, and as the measurements improve it is reduced to almost 1/2.

When computing the function $f(x)$ using the approximation x' instead of the true value x, we say that the CONDITION NUMBER is equal to the ratio of the relative error of the result and the relative error of the input, i.e.:

$$\frac{|f(x) - f(x')|/|f(x)|}{|x - x'|/|x|}.$$

If you take the limit as $x' \to x$ (and assume that f is differentiable, and that $f(x) \neq 0$), this is:

$$\lim_{x' \to x} \frac{|f(x) - f(x')|/|f(x)|}{|x - x'|/|x|} = \frac{|x|}{|f(x)|} \lim_{x' \to x} \frac{|f(x) - f(x')|}{|x - x'|} = \frac{|xf'(x)|}{|f(x)|}$$

For $f(x) = \sqrt{x}$, the condition number is (well, do the derivative)![2] Not all functions have good condition numbers. Compute the condition number for $\cos(x)$,[3] and you'll see that you can a huge condition number by choosing an appropriate x.[4]

Notes

[1] In fact, with overflow or underflow, the error can be arbitrarily large

[2] 1/2.

[3] Did you get $|x \tan(x)|$?

[4] Such as x close to $\pi/2$ radians.