2023-02-03

Binary floating point

Why do we study conditioning of problems? One reason is that we may have input data contaminated by noise, resulting in a bad solution even if the intermediate computations are done perfectly accurately. But our concern in this class is less with measurement errors and more with numerics, and so we will study condition numbers with an eye to floating point error analysis.

Binary floating point arithmetic is essentially scientific notation. Where in decimal scientific notation we write

$$\frac{1}{3} = 3.333\ldots \times 10^{-1},$$

in floating point, we write

$$\frac{(1)_2}{(11)_2} = (1.010101\ldots)_2 \times 2^{-2}.$$

Because computers are finite, however, we can only keep a finite number of bits after the binary point.

In general, a *normal floating point number* has the form

$$(-1)^s \times (1.b_1 b_2 \ldots b_p)_2 \times 2^E,$$

where $s \in \{0, 1\}$ is the *sign bit*, E is the *exponent*, and $(1.b_2 \ldots b_p)_2$ is the *significand*. In the 64-bit double precision format, $p = 52$ bits are used to store the significand, 11 bits are used for the exponent, and one bit is used for the sign. The valid exponent range for normal floating point numbers is $-1023 < E < 1024$; this leaves two exponent encodings left over for special purpose. One of these special exponents is used to encode *subnormal numbers* of the form

$$(-1)^s \times (0.b_1 b_2 \ldots b_p)_2 \times 2^{-1022};$$

the other special exponent is used to encode $\pm\infty$ and NaN (Not a Number).

For a general real number x, we will write

$$\mathrm{fl}(x) = \text{ correctly rounded floating point representation of } x.$$

By default, "correctly rounded" means that we find the closest floating point number to x, breaking any ties by rounding to the number with a zero in the last bit[1]. If x exceeds the largest normal floating point number, then $\mathrm{fl}(x) = \infty$.

Basic floating point arithmetic

For basic operations (addition, subtraction, multiplication, division, and square root), the floating point standard specifies that the computer should produce the *true result, correctly rounded*. So the Julia statement

```
# Compute the sum of x and y (assuming they are exact)
z = x + y;
```

actually computes the quantity $\hat{z} = \mathrm{fl}(x+y)$. If $\hat{z}$ is a normal double-precision floating point number, it will agree with the true z to 52 bits after the binary point. That is, the relative error will be smaller in magnitude than the *machine epsilon* $\epsilon_{\mathrm{mach}} = 2^{-53} \approx 1.1 \times 10^{-16}$:

$$\hat{z} = z(1+\delta), \quad |\delta| < \epsilon_{\mathrm{mach}}.$$

More generally, basic operations that produce normalized numbers are correct to within a relative error of ϵ_{mach}. The floating point standard also recommends that common transcendental functions, such as exponential and trig functions, should be correctly rounded, though compliant implementations that do not comply with this recommendation may produce results with a relative error just slightly larger than ϵ_{mach}.

The fact that normal floating point results have a relative error less than ϵ_{mach} gives us a useful *model* for reasoning about floating point error. We will refer to this as the "$1+\delta$" model. For example, suppose x is an exactly-represented input to the Julia statement

```
z = 1-x*x
```

[1]There are other rounding modes beside the default, but we will not discuss them in this class

We can reason about the error in the computed $\hat{z}$ as follows:

$$\begin{aligned}
t_1 &= \mathrm{fl}(x^2) = x^2(1+\delta_1) \\
t_2 &= 1 - t_1 = (1-x^2)\left(1 + \frac{\delta_1 x^2}{1-x^2}\right) \\
\hat{z} &= \mathrm{fl}(1-t_1) = z\left(1 + \frac{\delta_1 x^2}{1-x^2}\right)(1+\delta_2) \\
&\approx z\left(1 + \frac{\delta_1 x^2}{1-x^2} + \delta_2\right),
\end{aligned}$$

where $|\delta_1|, |\delta_2| \leq \epsilon_{\mathrm{mach}}$. As before, we throw away the (tiny) term involving $\delta_1\delta_2$. Note that if z is close to zero (i.e. if there is *cancellation* in the subtraction), then the model shows the result may have a large relative error.

Exceptions

We say there is an *exception* when the floating point result is not an ordinary value that represents the exact result. The most common exception is *inexact* (i.e. some rounding was needed). Other exceptions occur when we fail to produce a normalized floating point number. These exceptions are:

Underflow: An expression is too small to be represented as a normalized floating point value. The default behavior is to return a subnormal.

Overflow: An expression is too large to be represented as a floating point number. The default behavior is to return `inf`.

Invalid: An expression evaluates to Not-a-Number (such as 0/0)

Divide by zero: An expression evaluates "exactly" to an infinite value (such as 1/0 or log(0)).

When exceptions other than inexact occur, the usual "$1+\delta$" model used for most rounding error analysis is not valid.

Finding and fixing floating point problems

Floating point arithmetic is not the same as real arithmetic. Even simple properties like associativity or distributivity of addition and multiplication

only hold approximately. Thus, some computations that look fine in exact arithmetic can produce bad answers in floating point. What follows is a (very incomplete) list of some of the ways in which programmers can go awry with careless floating point programming.

Cancellation

If $\hat{x} = x(1+\delta_1)$ and $\hat{y} = y(1+\delta_2)$ are floating point approximations to x and y that are very close, then $\mathrm{fl}(\hat{x} - \hat{y})$ may be a poor approximation to $x - y$ due to *cancellation*. In some ways, the subtraction is blameless in this tail: if x and y are close, then $\mathrm{fl}(\hat{x} - \hat{y}) = \hat{x} - \hat{y}$, and the subtraction causes no additional rounding error. Rather, the problem is with the approximation error already present in $\hat{x}$ and $\hat{y}$.

The standard example of loss of accuracy revealed through cancellation is in the computation of the smaller root of a quadratic using the quadratic formula, e.g.

$$x = 1 - \sqrt{1-z}$$

for z small. Fortunately, some algebraic manipulation gives an equivalent formula that does not suffer cancellation:

$$x = \left(1 - \sqrt{1-z}\right)\left(\frac{1+\sqrt{1-z}}{1+\sqrt{1-z}}\right) = \frac{z}{1+\sqrt{1-z}}.$$

Sensitive subproblems

We often solve problems by breaking them into simpler subproblems. Unfortunately, it is easy to produce badly-conditioned subproblems as steps to solving a well-conditioned problem. As a simple (if contrived) example, try running the following Julia code:

```
function silly_sqrt()
    x = 2.0
    for k = 1:60 x = sqrt(x) end
    for k = 1:60 x = x^2   end
    println(x)
end
```

In exact arithmetic, this should produce 2, but what does it produce in floating point? In fact, the first loop produces a correctly rounded result, but the second loop represents the function $x^{2^{60}}$, which has a condition number far greater than 10^{16} — and so all accuracy is lost.

Unstable recurrences

We gave an example of this problem in the last lecture notes when we looked at the recurrence

$$E_0 = 1 - 1/e$$
$$E_n = 1 - nE_{n-1}, \quad n \geq 1.$$

No single step of this recurrence causes the error to explode, but each step amplifies the error somewhat, resulting in an exponential growth in error.

Undetected underflow

In Bayesian statistics, one sometimes computes ratios of long products. These products may underflow individually, even when the final ratio is not far from one. In the best case, the products will grow so tiny that they underflow to zero, and the user may notice an infinity or NaN in the final result. In the worst case, the underflowed results will produce nonzero subnormal numbers with unexpectedly poor relative accuracy, and the final result will be wildly inaccurate with no warning except for the (often ignored) underflow flag.

Bad branches

A NaN result is often a blessing in disguise: if you see an unexpected NaN, at least you *know* something has gone wrong! But all comparisons involving NaN are false, and so when a floating point result is used to compute a branch condition and an unexpected NaN appears, the result can wreak havoc. As an example, try out the following code in Julia.

```
function testnan()
    x = 0.0/0.0;
    if x < 0.0     println("x is negative")
    elseif x >= 0.0 println("x is non-negative")
    else           println("Uh...")
    end
end
```

Limits of the "$1 + \delta$" model

Apart from the fact that it fails when there are exceptions other than inexact, the "$1 + \delta$" model of floating point does not reflect the fact that some computations involve no rounding error. For example:

- If x and y are floating point numbers within a factor of two of each other, $\mathrm{fl}(x - y)$ is computed without rounding error.
- Barring overflow or underflow to zero, $\mathrm{fl}(2x) = 2x$ and $\mathrm{fl}(x/2) = x/2$.
- Integers between $\pm(2^{53} - 1)$ are represented exactly.

These properties of floating point allow us to do some clever things, such as using ordinary double precision arithmetic to simulate arithmetic with about twice the number of digits. You should be aware that these tricks exist, even if you never need to implement them – otherwise, I may find myself cursing a compiler you wrote for rearranging the computations in a floating point code that I wrote!

Let's consider an example analysis that illustrates both the usefulness and the limitations of the $1 + \delta$ model. Suppose I have points A, B, and C in the box $[1, 2]^2$, and that the coordinates are given as exact floating point numbers. I want to determine whether C is above, below, or on the oriented line going through A and B. The usual way to do this is by looking at the sign of

$$\det \begin{bmatrix} B_x - A_x & C_x - A_x \\ B_y - A_y & C_y - A_y \end{bmatrix}.$$

We might compute this determinant as something like this:

$$\begin{aligned} t_1 &= B_x - A_x \\ t_2 &= B_y - A_y \\ t_3 &= C_x - A_x \\ t_4 &= C_y - A_y \\ t_5 &= t_1 \times t_4 \\ t_6 &= t_2 \times t_3 \\ t_7 &= t_5 - t_6. \end{aligned}$$

Now, suppose we do this computation in floating point. We will call the floating point results $\hat{t}_j$, to distinguish them from the exact results. According to the $1+\delta$ model, we have $|\delta_i| \leq \epsilon$ so that

$$\begin{aligned}
\hat{t}_1 &= (B_x - A_x)(1+\delta_1) \\
\hat{t}_2 &= (B_y - A_y)(1+\delta_2) \\
\hat{t}_3 &= (C_x - A_x)(1+\delta_3) \\
\hat{t}_4 &= (C_y - A_y)(1+\delta_4) \\
\hat{t}_5 &= (\hat{t}_1 \times \hat{t}_4)(1+\delta_5) = (t_1 \times t_4)(1+\delta_1)(1+\delta_4)(1+\delta_5) = t_5(1+\gamma_5) \\
\hat{t}_6 &= (\hat{t}_2 \times \hat{t}_3)(1+\delta_6) = (t_2 \times t_3)(1+\delta_2)(1+\delta_3)(1+\delta_6) = t_6(1+\gamma_6) \\
\hat{t}_7 &= (\hat{t}_5 - \hat{t}_6)(1+\delta_7) \\
&= (t_5 - t_6)\left(1 + \frac{t_5\gamma_5 - t_6\gamma_6}{t_5 - t_6}\right)(1+\delta_7).
\end{aligned}$$

Here, $1+\gamma_5 = (1+\delta_1)(1+\delta_2)(1+\delta_3) = 1+\delta_1+\delta_2+\delta_3+O(\epsilon^2)$; that is, $|\gamma_5| \lesssim 3\epsilon$. Similarly, $|\gamma_6| \lesssim 3_\epsilon$.

Now, how large can the relative error in $\hat{t}_7$ be? Ignoring the final rounding error δ_7 – which is in this case insignficant – we have that the relative error is bounded by

$$\left|\frac{t_5\gamma_5 - t_6\gamma_6}{t_5 - t_6}\right| \leq 3\epsilon \frac{|t_5| + |t_6|}{|t_5 - t_6|}.$$

If t_5 and t_6 are not small but $t_5 - t_6$ is small, the relative error in t_7 could be quite large – even though the absolute error remains small. This effect of a large relative error due to a small result in a subtraction is called *cancellation*. In this case, if the relative error is one or larger, then we don't even necessarily have the right sign! That's not a good thing for our test.

Is this error analysis pessimistic? Yes and no. Some things in floating point are *exact*, such as multiplication by a power of two or subtraction of two numbers within a factor of two of each other. Thus, there will be no rounding error in the first four steps ($\delta_i = 0$ for $i = 1, 2, 3, 4$), since we have constrained each of the coordinates to lie in the interval $[1, 2]$. Note that this means that if t_5 and t_6 are close to each other, there will be no rounding error in the last step – δ_7 will be zero! But the overall outlook remains grim: the rounding errors in the multiplications are generally really there; and even if the subtraction is done exactly, it is the propogated error

from the multiplication steps that destroys the relative accuracy of the final results.

For this computation, then, the critical rounding error is really in the multiplications. If we could do those multiplications exactly, then we would be able to compute the desired determinant to high relative accuracy (and thus reliably test the determinant's sign). As it turns out, we *can* perform the multiplication exactly if we use a higher precision for intermediate computations than for the input data. For example, suppose the input data are in single precision, but the intermediate computations are all performed in double precision. Then t_1 through t_4 can be represented with significands that are at most 24 bits, and the products t_5 and t_6 require at most 48 bits – and can thus be exactly represented as double precision numbers with 53 bits in the significand.

For this computation, we see a payoff by doing a more detailed analysis than is possible with only the $1+\delta$ model. But the details were tedious, and analysis with the $1+\delta$ model will usually be good enough.

Problems to ponder

1. How do we accurately evaluate $\sqrt{1+x} - \sqrt{1-x}$ when $x \ll 1$?
2. How do we accurately evaluate $\ln\sqrt{x+1} - \ln\sqrt{x}$ when $x \gg 1$?
3. How do we accurately evaluate $(1-\cos(x))/\sin(x)$ when $x \ll 1$?
4. How would we compute $\cos(x) - 1$ accurately when $x \ll 1$?
5. The *Lamb-Oseen vortex* is a solution to the 2D Navier-Stokes equation that plays a key role in some methods for computational fluid dynamics. It has the form

   $$v_\theta(r,t) = \frac{\Gamma}{2\pi r}\left(1 - \exp\left(\frac{-r^2}{4\nu t}\right)\right)$$

   How would one evaluate $v(r,t)$ to high relative accuracy for all values of r and t (barring overflow or underflow)?
6. For $x > 1$, the equation $x = \cosh(y)$ can be solved as

   $$y = -\ln\left(x - \sqrt{x^2-1}\right).$$

What happens when $x = 10^8$? Can we fix it?

7. The difference equation

$$x_{k+1} = 2.25x_k - 0.5x_{k-1}$$

with starting values

$$x_1 = \frac{1}{3}, \qquad x_2 = \frac{1}{12}$$

has solution

$$x_k = \frac{4^{1-k}}{3}.$$

Is this what you actually see if you compute? What goes wrong?

8. Considering the following two Julia fragments:

```
# Version 1
f = (exp(x)-1)/x

# Version 2
y = exp(x)
f = (1-y)/log(y)
```

In exact arithmetic, the two fragments are equivalent. In floating point, the first formulation is inaccurate for $x \ll 1$, while the second formulation remains accurate. Why?

9. Running the recurrence $E_n = 1 - nE_{n-1}$ *forward* is an unstable way to compute $\int_0^1 x^n e^{x-1}\,dx$. However, we can get good results by running the recurrence *backward* from the estimate $E_n \approx 1/(N+1)$ starting at large enough N. Explain why. How large must N be to compute E_{20} to near machine precision?

10. How might you accurately compute this function for $|x| < 1$?

$$f(x) = \sum_{j=0}^{\infty} \left(\cos(x^j) - 1\right)$$