CS 6210: Matrix Computations

Floating point and error analysis

David Bindel

2025-09-08

Binary floating point

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. We can also only keep a finite number of bits for the exponent field. These facts turn out to have interesting implications.

Normalized representations

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*. The normalized representations are called normalized because they start with a one before the binary point. Because this is always the case, we do not need to store that digit explicitly; this gives us a "free" extra digit.

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 double precision floating point numbers is $-1023 < E < 1024$; the number E is encoded as an unsigned binary integer E_{bits} which is implicitly shifted by 1023 ($E = E_{\text{bits}} - 1023$). This

leaves two exponent encodings left over for special purpose, one associated with $E_{\text{bits}} = 0$ (all bits zero), and one associated with all bits set; we return to these in a moment.

In the 32-bit single-percision format, $p = 23$ bits are used to store the significand, 8 bits are used for the exponent, and one bit is used for the sign. The valid exponent range for normal is $-127 < E < 128$; as in the double precision format, the representation is based on an unsigned integer and an implicit shift, and two bit patterns are left free for other uses.

We will call the distance between 1.0 and the next largest floating point number one either an *ulp* (unit in the last place) or, more frequently, machine epsilon (denoted ϵ_{mach}). This is $2^{-52} \approx 2 \times 10^{-16}$ for double precision and $2^{-23} \approx 10^{-7}$ for single precision. This is the definition used in most numerical analysis texts, and in MATLAB and Octave, but it is worth noting that in a few places (e.g. in the C standard), call machine epsilon the quantity that is half what we call machine epsilon.

Subnormal representations

When the exponent field consists of all zero bits, we have a *subnormal* representation. In general, a *subnormal floating point number* has the form

$$(-1)^s \times (0.b_1 b_2 \ldots b_p)_2 \times 2^{-E_{\text{bias}}},$$

where E_{bias} is 1023 for double precision and 127 for single. Unlike the normal numbers, the subnormal numbers are evenly spaced, and so the *relative* differences between successive subnormals can be much larger than the relative differences between successive normals.

Historically, there have been some floating point systems that lack subnormal representations; and even today, some vendors encourage "flush to zero" mode arithmetic in which all subnormal results are automatically rounded to zero. But there are some distinct advantage to these numbers. For example, the subnormals allow us to keep the equivalence between $x - y = 0$ and $x = y$; without subnormals, this identity can fail to hold in floating point. Apart from helping us ensure standard identities, subnormals let us represent numbers close to zero with reduced accuracy rather than going from full precision to zero abruptly. This property is sometimes known as *gradual underflow*.

The most important of the subnormal numbers is zero. In fact, we consider zero so important that we have two representations: $+0$ and -0! These representations behave the same in most regards, but the sign does play a subtle role; for example, $1/+0$ gives a representation for $+\infty$, while $1/-0$ gives a representation for $-\infty$. The default value of zero is $+0$; this is what is returned, for example, by expressions such as $1.0 - 1.0$.

Infinities and NaNs

A floating point representation in which the exponent bits are all set to one and the signficand bits are all zero represents an *infinity* (positive or negative).

When the exponent bits are all one and the significand bits are not all zero, we have a *NaN* (Not a Number). A NaN is quiet or signaling depending on the first bit of the significand; this distinguishes between the NaNs that simply propagate through arithmetic and those that cause exceptions when operated upon. The remaining significand bits can, in principle, encode information about the details of how and where a NaN was generated. In practice, these extra bits are typically ignored. Unlike infinities (which can be thought of as a computer representation of part of the extended reals[1]), NaN "lives outside" the extended real numbers.

Infinity and NaN values represent entities that are not part of the standard real number system. They should not be interpreted automatically as "error values," but they should be treated with respect. When an infinity or NaN arises in a code in which nobody has analyzed the code correctness in the presence of infinity or NaN values, there is likely to be a problem. But when they are accounted for in the design and analysis of a floating point routine, these representations have significant value. For example, while an expression like 0/0 cannot be interpreted without context (and therefore yields a NaN in floating point), given context — eg., a computation involving a removable singularity — we may be able to interpret a NaN, and potentially replace it with some ordinary floating point value.

Basic floating point arithmetic

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[2]. If x exceeds the largest normal floating point number, then $\mathrm{fl}(x) = \infty$; similarly, if x is a negative number with magnitude greater than the most negative normalized floating point value, then $\mathrm{fl}(x) = -\infty$.

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

[1] The extended reals in this case means $\mathbb{R}$ together with $\pm\infty$. This is sometimes called the *two-point compactification* of $\mathbb{R}$. In some areas of analysis (e.g. complex variables), the *one-point compactification* involving a single, unsigned infinity is also useful. This was explicitly supported in early proposals for the IEEE floating point standard, but did not make it in. The fact that we have signed infinities in floating point is one reason why it makes sense to have signed zeros — otherwise, for example, we would have $1/(1/-\infty)$ yield $+\infty$.

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

```
# 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[3], though compliant implementations that do not follow with this recommendation may produce results with a relative error just slightly larger than ϵ_{mach}. Correct rounding of transcendentals is useful in large part because it implies other properties: for example, if a computer function to evaluate a monotone function returns a correctly rounded result, then the computed function is also monotone.

Operations in which NaN appears as an input conventionally (but not always) produce a NaN output. Comparisons in which NaN appears conventionally produce false. But sometimes there is some subtlety in accomplishing these semantics. For example, the following code for finding the maximum element of a vector returns a NaN if one appears in the first element, but otherwise results in the largest non-NaN element of the array:

```
# Find the maximum element of a vector -- naive about NaN
function mymax1(v)
  vmax = v[1];
  for k = 2:length(v)
    if v[k] > vmax
      vmax = v[k]
    end
  end
  vmax
end
```

In contrast, the following code always propagates a NaN to the output if one appears in the input

[3]For algebraic functions, it is possible to determine in advance how many additional bits of precision are needed to correctly round the result for a function of one input. In contrast, transcendental functions can produce outputs that fall arbitrarily close to the halfway point between two floating point numbers.

```
# Find the maximum element of a vector -- more careful about NaNs
function mymax2(v)
  vmax = v[1];
  for k = 2:length(v)
    if isnan(v[k]) | (v[k] > vmax)
      vmax = v[k]
    end
  end
  vmax
end
```

You are encouraged to play with different vectors involving some NaN or all NaN values to see what the semantics for the built-in vector max are in MATLAB, Octave, or your language of choice. You may be surprised by the results!

Apart from NaN, floating point numbers do correspond to real numbers, and comparisons between floating point numbers have the usual semantics associated with comparisons between floating point numbers. The only point that deserves some further comment is that plus zero and minus zero are considered equal as floating point numbers, despite the fact that they are not bitwise identical (and do not produce identical results in all input expressions)[4].

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)).

[4]This property of signed zeros is just a little bit horrible. But to misquote Winston Churchill, it is the worst definition of equality except all the others that have been tried.

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

An important feature of the floating point standard is that an exception should *not* stop the computation by default. This is part of why we have representations for infinities and NaNs: the floating point system is *closed* in the sense that every floating point operation will return some result in the floating point system. Instead, by default, an exception is *flagged* as having occurred[5]. An actual exception (in the sense of hardware or programming language exceptions) occurs only if requested.

Modeling floating point

The fact that normal floating point results have a relative error bounded by 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
```

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

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

where $|\delta_1|, |\delta_2| \leq \epsilon_{\text{mach}}$. 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.

First-order error analysis

Analysis in the $1+\delta$ model quickly gets to be a sprawling mess of Greek letters unless one is careful. A standard trick to get around this is to use *first-order* error analysis in which

[5] There is literally a register inside the computer with a set of flags to denote whether an exception has occurred in a given chunk of code. This register is highly problematic, as it represents a single, centralized piece of global state. The treatment of the exception flags — and of exceptions generally — played a significant role in the debates leading up to the last revision of the IEEE 754 floating point standard, and I would be surprised if they are not playing a role again in the current revision of the standard.

we linearize all expressions involving roundoff errors. In particular, we frequently use the approximations

$$(1+\delta_1)(1+\delta_2) \approx 1+\delta_1+\delta_2$$
$$1/(1+\delta) \approx 1-\delta.$$

In general, we will resort to first-order analysis without comment. Those students who think this is a sneaky trick to get around our lack of facility with algebra[6] may take comfort in the fact that if $|\delta_i| < \epsilon_{\text{mach}}$, then in double precision

$$\left| \prod_{i=1}^{n} (1+\delta_i) \prod_{i=n+1}^{N} (1+\delta_i)^{-1} \right| < (1+1.03N\epsilon_{\text{mach}})$$

for $N < 10^{14}$ (and a little further).

Shortcomings of the model

The $1+\delta$ model has two shortcomings. First, it is only valid for expressions that involve normalized numbers — most notably, gradual underflow breaks the model. Second, the model is sometimes pessimistic. Certain operations, such as taking a difference between two numbers within a factor of 2 of each other, multiplying or dividing by a factor of two[7], or multiplying two single-precision numbers into a double-precision result, are *exact* in floating point. There are useful operations such as simulating extended precision using ordinary floating point that rely on these more detailed properties of the floating point system, and cannot be analyzed using just the $1+\delta$ model.

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 $\text{fl}(\hat{x}-\hat{y})$ 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 $\text{fl}(\hat{x}-\hat{y}) = \hat{x}-\hat{y}$, and the

[6] Which it is.

[7] Assuming that the result does not overflow or produce a subnormal.

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(n=100)
  x = 2.0
  for k = 1:n
    x = sqrt(x)
  end
  for k = 1:n
    x = x^2
  end
  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

One of my favorite examples of this problem is the recurrence relation for computing the integrals

$$E_n = \int_0^1 x^n e^{x-1} \, dx.$$

Integration by parts yields the recurrence

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

This looks benign enough at first glance: 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[8].

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 with '0.0/0.0' as input.

```
function test_negative(x)
  if x < 0.0
    "$(x) is negative"
  elseif x >= 0.0
    "$(x) is non-negative"
  else
    "$(x) is ... uh..."
  end
end
```

Problems to ponder

1. How do we accurately evaluate $\sqrt{1+x} - \sqrt{1-x}$ when $x \ll 1$?

[8] Part of the reason that I like this example is that one can run the recurrence *backward* to get very good results, based on the estimate $E_n \approx 1/(n+1)$ for n large.

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 $|x| < 1$?

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