2019-09-13

1 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.

1.1 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.

1.2 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

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

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 $1.0 - 1.0$.

1.3 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.

2 Basic floating point arithmetic

For a general real number x, we will write

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

[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 $+\infty$.

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 MATLAB 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[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:

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

[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.

```
function [vmax] = mymax1(v)
  % Find the maximum element of a vector -- naive about NaN

  vmax = v(1);
  for k = 2:length(v)
    if v(k) > vmax, vmax = v(k); end
  end
```

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

```
function [vmax] = mymax2(v)
  % Find the max a vector -- or NaN if any element is NaN

  vmax = v(1);
  for k = 2:length(v)
    if isnan(v(k))
      vmax = v(k);
    elseif v(k) > vmax
      vmax = v(k);
    end
  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].

3 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*

[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.

(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.

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.

4 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 MATLAB statement

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

[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 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.

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

$$\begin{aligned}
(1+\delta_1)(1+\delta_2) &\approx 1+\delta_1+\delta_2 \\
1/(1+\delta) &\approx 1-\delta.
\end{aligned}$$

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 $|\delta_i| < \epsilon_{\mathrm{mach}}$, then in double precision

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

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

4.2 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

[6]Which it is.

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.

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