Floating-Point Arithmetic

Raymond J. Spiteri

Lecture Notes for CMPT 898:
Numerical Software

University of Saskatchewan

January 9, 2013

Objectives

- Floating-point numbers
- Floating-point arithmetic
- Analysis of floating-point arithmetic

Floating-Point Arithmetic

Floating-point arithmetic is the hardware analogue of scientific notation.

A basic understanding of floating-point arithmetic is essential when solving problems numerically because certain things happen in a floating-point environment that might surprise you otherwise.

We study floating-point arithmetic because of how computers store and operate on real numbers.

Digital computers must represent real numbers with a *finite number of bits*; i.e., only a finite set of finite-digit numbers can be represented.

Representing numbers on computers

There are various ways to represent number systems with a fixed amount of information, e.g., floating-point, fixed-point, or “level index” number systems.

The advantage of the floating-point representation is that it can encompass a large range of numbers.

For example, suppose we can store 6-digit numbers with 5 digits after the decimal place.

Then the largest number we can store is $9.99999 \approx 10$ and the smallest is $0.00001 = 10^{-5}$.

If we allocate 2 digits to represent an exponent (to base 10), then we can represent a larger range of numbers.

The largest representable number is now 9.999×10^{99}.

The smallest representable number is 1.000×10^{-99}.

The price we have paid to get the increase in range is that we only have 4 significant digits instead of 6.

However, this tradeoff is favourable.

IEEE standard

We are able to study floating-point arithmetic in a systematic way because in 1985 it was agreed that all computers adopt the IEEE convention for floating-point arithmetic.

Before then, every computer could (and often did!) have its own brand of floating-point arithmetic!

This allows us to study floating-point arithmetic without referring to a specific computer.

Note 1. *This does not mean every computer will give the same answer to the same sets of floating-point operations!*

There is still some room for interpretation, even in the IEEE standard!

Floating-point numbers

A floating-point number system $\mathcal{F} \subset \mathbb{R}$ is a subset of real numbers with elements of the form

$$x = \pm \left(\frac{m}{\beta^t}\right) \times \beta^e.$$

The system $\mathcal{F}$ is characterized by four integers:

1. the base β,

2. the precision t,

3. the minimum exponent $e_{\min}$, and

4. the maximum exponent $e_{\max}$.

The exponent range is defined as $e_{\min} \leq e \leq e_{\max}$.

Floating-point numbers

The mantissa m is an integer satisfying $0 \le m \le \beta^t - 1$.

To ensure uniqueness of representation, we assume a normalization $m \ge \beta^{t-1}$ for all $x \ne 0$.

The range of non-zero floating-point numbers of $\mathcal{F}$ is

$$\beta^{e_{\min}-1} \le |x| \le \beta^{e_{\max}}(1 - \beta^{-t}).$$

A more expressive way to describe x is

$$\begin{aligned} x &= \pm\,\beta^e \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) \\ &= \pm\,\beta^e \times 0.d_1 d_2 \ldots d_t, \end{aligned}$$

where each digit d_i satisfies $0 \le d_i \le \beta - 1$ and $d_1 \ne 0$ for normalized numbers.

The digits d_1 and d_t are respectively called the most and least significant digits.

Floating-Point Numbers

The finiteness of e is a limitation on what range of numbers can be represented.

The finiteness of t is a limitation on how many significant digits can be stored.

These limitations imply:

1. Floating-point numbers cannot be arbitrarily large or arbitrarily small.

2. Floating-point numbers cannot be arbitrarily close together (i.e., there must be gaps between them!)

Any numbers that don't meet these limitations must be approximated by ones that do.

This leads to the concept of *round-off errors*.

Floating-point numbers

The limitation on range is often not of concern; e.g., the IEEE double precision standard floating-point number system supports a largest number of 1.79×10^{308} and a smallest number of 2.23×10^{-308}; i.e., *overflow* and *underflow* are not common problems.

However, the gaps between numbers can pose serious problems if you're not careful!

The little errors made in representing real numbers by their floating-point counterparts can accumulate to the extent that the errors dominate the number that is to be represented.

This is what happens when using an unstable algorithm (or when trying to solve an unstable problem).

Floating-point numbers

It is important to realize that floating-point numbers are not equally spaced (in an absolute sense).

Imagine a floating-point number system with

$$\beta = 2,\ t = 3,\ e_{\mathsf{min}} = -2,\ \text{and}\ e_{\mathsf{max}} = 1.$$

Then all the positive numbers in this floating-point number system are all the possible combinations of the binary numbers

$$0.100, 0.101, 0.110, 0.111$$

and the exponents

$$-2, -1, 0, 1.$$

This leads to a total of 16 positive floating-point numbers in our number system. See `floatgui.m`

Floating-point numbers

Some things to notice:

- In each interval $\beta^{e-1} \le x \le \beta^e$, the floating-point numbers are equally spaced with increment β^{e-t}.

  For example, if $\beta = 2,\ e = 1$, and $t = 3$, the spacing of the floating-point numbers in $[1, 2]$ is $1/4$.

- As e increases, the (absolute) spacing between floating-point numbers increases.

- However, the relative spacing is constant across all intervals $\beta^{e-1} \le x \le \beta^e$.

- The distance between 1 and the next larger floating-point number is $\epsilon_{\text{machine}} = \beta^{1-t}$; $\epsilon_{\text{machine}}$ is known as *machine epsilon*.

- The smallest number that can be added to 1 and results in a number greater than 1 is $u = \beta^{-t}$; u is known as *unit round-off*.

On errors . . .

Let $\hat{x}$ be an approximation to a real number x.

Then the *absolute error* in $\hat{x}$ is defined to be

$$E_{\text{abs}} = |x - \hat{x}|,$$

and the *relative error* is defined to be

$$E_{\text{rel}} = \frac{|x - \hat{x}|}{|x|}.$$

Notes:

1. E_{rel} is not defined for $x = 0$.

2. The absolute value ($| \cdot |$) can be removed in the definition of E_{abs} if the sign of the error is important.

3. In practice, often a mix of absolute and relative errors are used to assess accuracy (or set tolerances).

Unit round-off

Unit round-off represents the worst case of the relative amount by which a given real number is rounded off when represented as a machine number; i.e.,

$$\text{for all } x \in \mathbb{R},\ \text{there is an } \hat{x} \in \mathbb{F} \text{ such that}$$

$$|x - \hat{x}| \le u\,|x|.$$

We can define a function

$$\mathrm{fl} : \mathbb{R} \to \mathbb{F}$$

that takes a real number and *rounds it off* to the nearest floating-point number; so,

for all $x \in \mathbb{R}$, there is an ϵ satisfying $|\epsilon| \le u$
such that fl$(x) = x(1 + \epsilon)$.

That is, the relative difference between a number and its closest floating-point number is always less than u.

Wobbles

It is interesting to note that

$$x = \left(\beta^{t-1} + \frac{1}{2}\right) \times \beta^e \implies \left|\frac{x - \mathsf{fl}(x)}{x}\right| \approx \frac{1}{2}\beta^{1-t}$$
$$x = \left(\beta^{t} - \frac{1}{2}\right) \times \beta^e \implies \left|\frac{x - \mathsf{fl}(x)}{x}\right| \approx \frac{1}{2}\beta^{-t}$$

So the relative error in x is bounded by $\frac{1}{2}\beta^{1-t}$ (as it should be!), but it can vary by as much as β.

This phenomenon has been called wobbling precision.

It is one of the reasons why small bases (e.g., $\beta = 2$) are preferred.

In IEEE single precision, these relative distances vary from about $2^{-24} \approx 6 \times 10^{-8}$ just to the left of numbers of the form 2^e to $2^{-23} \approx 1 \times 10^{-7}$ just to the right.

ulp

When describing the accuracy of a floating-point number, the concept of unit in the last place (or *ulp*) is sometimes used.

If $\hat{x} = \pm\beta^e \times 0.d_1d_2\ldots d_t$, then

$$\text{ulp}(\hat{x}) = \beta^e \times 0.00\ldots01 = \beta^{e-t}.$$

Accordingly, we say that any real number x and $\hat{x}$ agree to $|x - \hat{x}|/\text{ulp}(\hat{x})$ ulps in $\hat{x}$.

This measure of accuracy wobbles when $\hat{x}$ changes from β^e to the next smaller floating-point number because ulp$\hat{x}$ decreases by a factor of β.

Rounding

It is possible that a given x be precisely halfway between its two closest floating-point numbers.

There are several ways to break such ties, including taking fl(x) to be the number with

- the larger magnitude (*rounding away from 0*)
- the lesser magnitude (*rounding toward 0; also known as chopping or truncating*)
- an even last digit d_t (*round to even*)
- an odd last digit d_t (*round to odd*)

Although still existent on systems such as Cray, rounding toward or away from 0 are not recommended because round-off errors can be twice as large as with other rounding schemes and repeated addition and subtraction of the same number can yield *drift*.

Rounding

For bases 2 and 10, rounding to even is preferred over rounding to odd.

After rounding to even, a subsequent rounding to one less place will not involve a tie.

For example, starting with 2.445, round to even yields the sequence

$$2.445,\ 2.44,\ 2.4, \qquad \text{(only the initial tie)}$$

whereas round to odd yields the sequence

$$2.445,\ 2.45,\ 2.5 \qquad \text{(all ties)}$$

For base 2, rounding to even causes produces a zero least significant bit, thus increasing the production of integer results.

The statistical distribution of rounding errors

Most analyses of round-off errors provide worst-case bounds on the total (accumulated) error, often of the form $C(n)u$, where $C(m)$ is a dimension-dependent constant and u is unit round-off.

These analyses ignore the signs of round-off errors and simply accumulate them.

Thus, although they may offer some insight, they are often so pessimistic as to have no practical value.

One usually has to couple some kind of probabilistic model of rounding with statistical analysis to say something useful.

For example, one might assume round-off errors to be independent random variables and apply the central limit theorem.

This leads to a well-known rule of thumb that a more realistic error bound for $C(n)u$ is actually $\sqrt{C(n)}u$.

The statistical distribution of rounding errors

However, rounding errors are generally not random!

For example, when computing a $\mathbf{QR}$ factorization of a matrix $\mathbf{A}$ by various (but not all!) standard methods, it can be shown that the computed quantities $\tilde{\mathbf{Q}}$ and $\tilde{\mathbf{R}}$ are not close to the true quantities $\mathbf{Q}$ and $\mathbf{R}$ in the sense that the norms of their differences are not a small multiples of u relative to the size of $\mathbf{A}$.

However, it turns out that the norm of difference between the product $\tilde{\mathbf{Q}}\tilde{\mathbf{R}}$ and $\mathbf{A}$ is a small multiple of u relative to the size of $\mathbf{A}$!

Wilkinson has famously described the errors as being "diabolically correlated".

IEEE Double Precision

IEEE double-precision floating-point numbers are stored in a 64-bit word, with 52 bits for the mantissa, 11 bits for e, and 1 bit for the sign of the number.

The precision for normalized numbers is $t = 52+1 = 53$ because the leading bit is 1 by convention and hence need not be stored explicitly.

The exponent e is an integer in the interval

$$-1022 \leq e \leq 1023.$$

The sign of e is accommodated by actually storing $e + 1023$, which is a number between 1 and $2^{11} - 2$.

The two extreme values for the exponent field (0 and $2^{11} - 1$) are reserved for exceptional floating-point numbers that we will describe momentarily.

Machine Epsilon

For IEEE double precision,

$$\epsilon_{\text{machine}} = 2^{-52} \approx 2.2204 \times 10^{-16}$$

(i.e., such numbers are stored to about 16 digits).

In MATLAB, $\epsilon_{\text{machine}}$ is called eps.

Many modern programming languages have built-in constants to denote $\epsilon_{\text{machine}}$ or u.

In the past, routines would estimate such quantities directly, as in the following MATLAB code segment:

```
a = 4/3
b = a - 1
c = 3*b
e = 1 - c
```

This code segment can used as a quick (and dirty) way to estimate $\epsilon_{\text{machine}}$ on a given computer.

$\epsilon_{\text{machine}}$ *is not "floating-point zero"*

There are of course many floating-point numbers between $\epsilon_{\text{machine}}$ and 0.

The smallest positive normalized floating-point number has $m = 1$ and $e = e_{\min} = -1022$.

MATLAB calls this number `realmin`.

The largest floating-point number has $m = 2^{53} - 1$ and $e = e_{\max} = 1023$.

MATLAB calls this number `realmax`.

- If any computation tries to produce a value larger than `realmax`, it is said to *overflow*.

  The result is an exceptional floating-point value called *infinity* or `Inf` in MATLAB.

  `Inf` is represented by taking $e = 1024$ and $m = 0$.

  `Inf` can make sense in calculations;
  e.g., 1/`Inf` = 0 and `Inf`+`Inf` = `Inf`.

- If any computation tries to produce a value that is **undefined** in the real number system, the result is an exceptional value called *Not-a-Number*, or `NaN`.

  Examples include 0/0 and `Inf−Inf`.

  `NaN` is represented by $e = 1024$ and any $m \neq 0$.

- If any computation tries to produce a value smaller than `realmin`, it is said to *underflow*.

  This involves *subnormal* (or denormal) floating-point numbers in the interval between `realmin` and $\epsilon_{\text{machine}}$*`realmin`.

  The smallest positive subnormal number is $2^{-52-1022} \approx 4.94 \times 10^{-324}$.

  Any results smaller than this are set to 0.

  On machines without subnormal numbers, any results less than `realmin` are set to 0.

  Subnormal elegantly handle underflow, but their importance for computation is rare.

  Subnormal numbers are represented by $e = -1023$, so the biased exponent $(e + 1023)$ is 0.

An Earthly analogy

How much sleep should we lose over the gaps in the IEEE floating-point number system?

In one sense, not much. Here is a way to see it.

According to molecular physics, there are approximately 3×10^8 molecules per meter in a gas at atmospheric conditions — essentially the cube root of Avogadro's number.

The circumference of the Earth is 4×10^7 meters; thus in a circle around the Earth, there would be approximately 10^{16} molecules.

In IEEE double-precision arithmetic, there are 2^{52} numbers between 1 and 2, and $2^{52} \approx 10^{16}$.

So if we put a giant circle around the Earth to represent the range from 1 to 2, the spacing of the double-precision floating-point numbers along the circle is about the same as the spacing of the air molecules.

Analysis of floating-point arithmetic

The classic arithmetic operations are $+, -, \times, \text{ and } /$.

These are of course operations on elements of $\mathbb{R}$.

On a computer, they have analogous operations on $\mathbb{F}$.
We denote these by $\oplus, \ominus, \otimes, \text{ and } \oslash$.

These operations are most naturally defined as follows:

Let $x, y \in \mathbb{F}$.

Let $*$ stand for one of $+, -, \times, \text{ and } /$,
and let $\circledast$ be its floating-point analogue.

Then

$$x \circledast y = \mathrm{fl}(x * y).$$

This leads us to the "fundamental axiom of the floating-point arithmetic":

$$\text{For all } x, y \in \mathbb{F}, \text{ there is an } \epsilon \text{ satisfying } |\epsilon| \leq u_e$$

$$\text{such that } x \circledast y = (x * y)(1 + \epsilon).$$

That is, *every operation of floating-point arithmetic is exact up to a relative error of size at most u_e.*

Note 2. *We are interested in $x, y \in \mathbb{R}$ not $x, y \in \mathbb{F}$!*

In this case,

$$\begin{aligned} x \circledast y &= \mathrm{fl}(\mathrm{fl}(x) * \mathrm{fl}(y)) \\ &= \mathrm{fl}(x(1+\epsilon_1) * y(1+\epsilon_2)) \\ &= (x * y)(1 + \epsilon_1 + \epsilon_2)(1 + \epsilon) \\ &= (x * y)(1 + \epsilon_1 + \epsilon_2 + \epsilon), \end{aligned}$$

where $|\epsilon_1|$, $|\epsilon_2|$, $|\epsilon| \leq u_e$.

Summary

- Floating-point arithmetic is the hardware analogue of scientific notation.
- The effects of using a floating-point number system are usually innocuous, but not always!
- The fundamental axiom of floating-point arithmetic allows us to analyze the propagation of round-off errors.