Theme 1: Roundoff and population modeling

Martin Berggren

November 4, 2009

Content

- Computer arithmetic, floating-point numbers
- "The" standard: IEEE 754 double precision floating point format
- Rounding error analysis, machine epsilon
- Warnings, consequences, rules of thumb for practical computations

The lab will clarify the relation to population modeling!

Error Concepts

- Approximate solutions of mathematical problems using computers introduce various errors
- Distinguish between **discretization error** and **roundoff error**

Ex: Computer representation of a black-and-white picture

- **Discretization error:** a spatially continuous image is *rasterized* to pixels (say 1024×768)
- **Rounding error:** only a fixed number (say 256) of gray tones at each pixel

Error Concepts

- Discretization errors typically dominate the total error
- Rounding errors can in many practical cases be neglected!

Although rounding errors typically are small, they are noticeably annoying in practical computations with real numbers:

Expression	Value	in Matlab
$\cos \pi/2$	0	`6.1232e-017`
$0.08 + 0.42 - 0.5$	0	`0`
$0.42 - 0.5 + 0.08$	0	`-1.3878e-017`

Binary numbers

- Computers usually stores numbers in binary form:

$$(\overbrace{1101}^{\text{4 bit}})_2 = 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = (13)_{10}$$

- Integers are stored *exactly* in binary form up to 2^n (n bit)
- Fractional binary numbers:

$$\begin{aligned}(.1101)_2 &= 1\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4}\\ &= \frac{1}{2} + \frac{1}{4} + 0 + \frac{1}{16} = \frac{13}{16} = (0.8125)_{10}\end{aligned}$$

- *Note:* The decimal fractions 0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8, 0.9 cannot be exactly represented as a fractional binary number! (But 0.5 can.)

Floating point numbers

- Real numbers cannot be stored exactly; they need to be *rounded* and *bounded*
- Almost all computer hardware and software support the **IEEE Standard for Floating-Point Arithmetic** IEEE 754
- IEEE 754 adopted in 1985. Latest version IEEE 754-2008 (from year 2008)
- Yields a machine-independent model of how floating point arithmetic behaves
- Matlab supports **IEEE binary double precision format**, the most common format for floating point numbers

IEEE 754 double precision floating point format

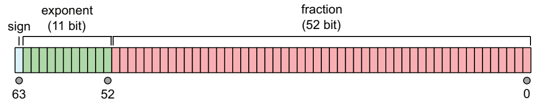
- The format stores the numbers in **normalized** form, that is, it can be expressed as

$$x = \pm(1 + f)\cdot 2^e,$$

  where
  - $0 \le f < 1$ (the **mantissa**, or **fraction**) is represented in binary form using 52 bits
  - e (the **exponent**) is an integer satisfying $-1022 \le e \le 1023$ (using 11 bits)
  - 1 bit is used for the sign (0 positive, 1 negative)
- Finiteness of f is a limitation on *precision*
- Finiteness of e is a limitation on *range*
- Only f, e, and sign is stored; not the initial 1 ("hidden bit")
- Number 0 is handled separately ($e = -1023$ and $f = 0$ indicates zero)

IEEE 754 double precision floating point format

- Thus, 64 bits, or 8 bytes (1 byte = 8 bits), is used for each floating-point number



Picture: Wikipedia

- *Ex:* A 1000×1000 real matrix. Requires 10^6 8-byte floating point numbers, thus 8 Mb storage

Machine epsilon

- The number of digits in f (the mantissa) limits the precision of the floating point system
- f is represented by 52 binary digits in IEEE 754 double precision
- For any floating point system, the distance between the number 1 and the next representable number is called the **machine epsilon** ϵ_M
- For IEEE 754 double precision, $\epsilon_M = 2^{-52} \approx 2.2204 \times 10^{-16}$:

$$(1.\underbrace{000000000000000000000000000000000000000000000000000}_{1...51}1)_2$$

- ϵ_M quantifies the precision of the floating point system

Spacing between floating point numbers

$$x = \pm(1 + f) \cdot 2^e,$$

- For $e = 0$, the spacing between each consecutive numbers is ϵ_M. *Ex:*

$$\begin{aligned}
&(1.0110000000000000000000000000000000000000000000001000)_2\\
-&(1.0110000000000000000000000000000000000000000000000111)_2\\
=&(0.0000000000000000000000000000000000000000000000000001)_2
\end{aligned}$$

- For $e = 1$, the spacing between consecutive numbers is $2\epsilon_M$
- In general, the spacing between consecutive numbers is $\epsilon_M \cdot 2^e$
- Thus, there is a constant spacing between numbers for a fixed exponent, but the spacing grows with the exponent

Overflow and underflow

- Recall: $x = \pm(1 + f) \cdot 2^e$ with $-1022 \le e \le 1023$
- Smallest (in magnitude) normalized number $x_{\text{min}} = 2^{-1022}$

Note: **much** smaller than ϵ_M!

- Largest (in magnitude) representable number: $x_{\text{max}} = (2 - \epsilon_M) \cdot 2^{1023}$
- Attempt to store numbers with $|x| > x_{\text{max}}$ yields **overflow** (many programs terminate with error when this happens)
- Attempt to store numbers with $|x| < x_{\text{min}}$ yields **underflow** (many programs set $x = 0$ and continue)

The above is a slight lie: IEEE 754 actually supports "subnormal numbers" or "gradual underflow". When $e = -1023$, $f = 0$ indicates zero, but any nonzero f indicates the number $0.f \cdot 2^{-1023}$, which allows storage of numbers down to 2^{-1074} with reduced accuracy.

Specials

The standard also defines the following quantities:

- The (extended real) numbers $+\infty$ and $-\infty$ (stored using the sign flag and $e = 1024$ and $f = 0$)
- The symbol **not-a-number**, or NaN (stored in $e = 1024$ when $f \neq 0$). NaN is typically used as the result of an operation using invalid inputs, such as $0/0$.

Absolute and relative error

- x: exact (real) number
- $\hat{x}$: number with error (due to measurement error, roundoff, ...)
- **Absolute error:** $|x - \hat{x}|$
- **Relative error:** $\dfrac{|x - \hat{x}|}{|x|}$

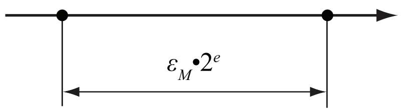
If x is a vector, use *vector norm* to express errors:

- **Absolute error:** $\|x - \hat{x}\|$
- **Relative error:** $\dfrac{\|x - \hat{x}\|}{\|x\|}$

$\|x\| = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}$ (e. g.; we will introduce other vector norms later!)

Rounding errors

- Assume that a given real number x is approximated by a floating point number $fl(x)$ (using IEEE 754 double precision)
- How big is the error $|x - fl(x)|$, the **rounding error**?
- $fl(x) = m \cdot 2^e$ with $m = 1.f$ or $m = 0$ (when $x = 0$)
- Also, we may write $x = \hat{m} \cdot 2^e$, with same exponent as for $fl(x)$, and $1 \leq \hat{m} < 2$, with infinite precision, or $\hat{m} = 0$
- Recall that the distance between each floating point number is $\epsilon_M \cdot 2^e$



- Thus, for any sensible rounding $|x - fl(x)| \leq \epsilon_M \cdot 2^e$
- When rounding to nearest floating point number $|x - fl(x)| \leq \frac{1}{2}\epsilon_M \cdot 2^e$ (the default rounding and the one Matlab uses)

Rounding errors

- Note that $|x| = |\hat{m} \cdot 2^e| \geq 2^e$ whenever $x \neq 0$
- Thus, for $x \neq 0$, and when rounding to nearest floating-point number, the **relative error** is

$$\frac{|x - fl(x)|}{|x|} \leq \frac{\frac{1}{2}\epsilon_M \cdot 2^e}{2^e} = \frac{1}{2}\epsilon_M \tag{1}$$

- Thus, when rounding to nearest floating point number:

> The relative error in the floating point approximation of any nonzero number is bounded by $\frac{1}{2}\epsilon_M$

- In particular: the *relative* error is independent of the size of the number

Note: Some authors attach the name "machine epsilon" or "unit roundoff" to the quantity $\mu = \frac{1}{2}\epsilon_M$ (in Eldén, Wittmeyer–Koch *avrundningsenheten*). However, we follow Matlab's definition.

Rounding errors in practical computations

- Machine epsilon is a measure of the relative accuracy of a stored real number
- IEEE 754 double precision format provides a precision of about 16 decimal digits
- During practical computations, many floating point operations are performed on numbers that has been rounded. Nevertheless, the accumulated relative error in the final result is usually not more than a few orders of magnitude greater than ϵ_M
- Rounding errors are in the majority of cases **much** smaller than other errors (discretization errors, measurement errors)!
- However, there are a few "dangerous" cases to watch out for!

Cancellation of significant digits

- Watch out when subtracting almost-equal numbers:

$$1.23456789 - 1.23456700 = 0.00000089$$

- If both numbers to the left have 9 correct digits, the resulting number to the right only has 2 correct digits!
- The phenomenon is called **cancellation** of significant digits
- Cancellation can sometimes be avoided by rewriting:

$$\sqrt{1+x} - \sqrt{1-x} = \frac{(\sqrt{1+x} - \sqrt{1-x})(\sqrt{1+x} + \sqrt{1-x})}{\sqrt{1+x} + \sqrt{1-x}} = \frac{2x}{\sqrt{1+x} + \sqrt{1-x}}$$

Consequences, rules of thumb

- `if x==y then...` a dangerous statement when x and y are general floating point numbers
- Better to use `if abs(x-y) <= tolerance then...` where `tolerance` is a small number
- Avoid, if possible, subtraction of almost-equal numbers
- The associative and distributive laws of arithmetic does not hold exactly for floating point numbers (often not so important)
- For $\sum_{n=1}^{N} s_n$, try to add up the terms starting with the smallest in magnitude

When are rounding errors noticeable?

- Recall example with computer representation of a black-and-white picture
  - **Discretization error:** a spatially continuous image is *rasterized* to pixels (say 1024×768)
  - **Rounding error:** only a fixed number (say 256) of gray tones at each pixel
- Using e. g. double precision floating point numbers for the gray tones, the rounding error can be completely neglected, it will only be the discretization error that matter!
- Similarly, in most cases when using numerical software, we can forget about rounding errors
- Two important exceptions!

When are rounding errors noticeable?

1. Sensitive problems. The solution to a mathematical problems can sometimes be very sensitive to changes in the input data: **small** changes in the data creates **large** changes in the solution. The small errors induced by rounding the input can therefore cause noticeable changes in the solution. Such problems are called *ill-conditioned* or in extreme cases *ill-posed*.
2. Numerically unstable algorithms. Some numerical algorithms are sensitive to roundoff even when applied to a well-conditioned problem. Avoid such algorithms if possible!