Finite Precision Arithmetic

Based on: **Lessons in Scientific Computing: Numerical Mathematics, Computer Technology, and Scientific Discovery** ByNorbert Schorghofer
Ch. 3 - *Roundoff & Number Representation*

E-book available from UW Libraries. Ch. 3 on Canvas (Week 2).

For *much* more detail on IEEE754 Standard Floating-Point Arithmetic, see "What Every Computer Scientist Should Know About Floating-Point Arithmetic," by David Goldberg, ACM Computing Surveys. Available in the Canvas "readings" folder.

A bit of context

What is this class all about? **Practical** numerical methods for obtaining good approximate solutions to engineering problems (for which analytic solutions may not be readily available)

What are **numerical methods** about?
Using digital computers to obtain those good approximate solutions…

Computers are both **fast/capable** and **dumb/literal**.

The computer is **not** smarter than you, so you have to tell it **very explicitly** what you want it to do. (*Exception?*)

The specification of what to do is an **algorithm**:
Unambiguous finite rule that, after a finite number of steps, provides a solution to a class of problems.

We will look at **algorithms** and **implementations** as codes/programs.

Algorithms have inputs (often in $\mathbb{Z}^n$ or $\mathbb{R}^n$) that are converted into outputs by a sequence of **elementary operations** including `+,-,*,/`.

Classes of problems

Ill-conditioned problem:
Small change in input can cause LARGE change in output

Well-conditioned problem:
Small change in input $\iff$ small change in output

Try to identify well-conditioned problems that we can solve with **numerically stable algorithms**:
Actual computational error comparable to **unavoidable error**.

Why are there unavoidable errors?

- Exact representation of $\mathbb{R}$ requires infinitely many digits.
- Infinite data storage requires infinite computing resources.
- Memory chip prices have come down, but infinite memory is impossible (physically and economically).

Typical compromise: use finite precision numeric representations.

§ 3.1 - Number Representation

Real numbers in an algorithm get approximated for digital computing purposes as **floating-point** numbers of the form:

$$(-1)^s \; (d_0 . \, d_1 d_2 \ldots d_{p-1}) \; \beta^e$$

- $s \in 0, 1$ indicates the sign
- p is the number of digits or **precision**
- each digit $d_i \in [0, 1, \ldots, \beta - 1]$
- β is the base and e is the exponent.

Humans may prefer base $\beta = 10$, but computers almost universally use a **binary representation with** $\beta = 2$.

A floating-point representation corresponds to a unique real number:
$\bar{x} = (-1)^s \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{p-1}}{\beta^{p-1}}\right) \beta^e$

Some real numbers have an exact (but not unique) floating-point representation; e.g.
$x = 0.5$ has the following representations:
$\bar{x} = (-1)^0 \ (1.0) \ 2^{-1} = (-1)^0 \ (1 + \frac{0}{2}) \ 2^{-1} = \frac{1}{2}$
and
$\bar{x} = (-1)^0 \ (0.1) \ 2^0 = (-1)^0 \ (0 + \frac{1}{2}) \ 2^0 = \frac{1}{2}$

For uniqueness, choose the **normalized** (first) version with $d_0 \neq 0$.

A floating point number system is defined by base β, precision (number of digits) p, and range of exponents $[e_{min}, e_{max}]$.

Let's take a look at an example: $\beta = 2, p = 2, e \in [-2, 3]$.
Positive normalized numbers:
$\bar{x} = (-1)^0\ (1.0)\ 2^{-2} = \frac{1}{4}$, $\bar{x} = (-1)^0\ (1.1)\ 2^{-2} = \frac{3}{8}$
$\bar{x} = (-1)^0\ (1.0)\ 2^{-1} = \frac{1}{2}$, $\bar{x} = (-1)^0\ (1.1)\ 2^{-1} = \frac{3}{4}$
$\bar{x} = (-1)^0\ (1.0)\ 2^{0} = 1$, $\bar{x} = (-1)^0\ (1.1)\ 2^{0} = \frac{3}{2}$
$\bar{x} = (-1)^0\ (1.0)\ 2^{1} = 2$, $\bar{x} = (-1)^0\ (1.1)\ 2^{1} = 3$
$\bar{x} = (-1)^0\ (1.0)\ 2^{2} = 4$, $\bar{x} = (-1)^0\ (1.1)\ 2^{2} = 6$
$\bar{x} = (-1)^0\ (1.0)\ 2^{3} = 8$, $\bar{x} = (-1)^0\ (1.1)\ 2^{3} = 12$

Gamut and observations

$\bar{x} \in \mathbf{X} = \pm\{1/4, 3/8, 1/2, 3/4, 1, 3/2, 2, 3, 4, 6, 8, 12\}$

- Not all reals can be represented
- No representation for zero (although one can be designated)
- Uneven spacing: **gapsize roughly proportional to magnitude**

Typical usage: round real number x to nearest element of $\mathbf{X}$.

- Rounding incurs an **absolute error**: $|x - \bar{x}| = E(x)$
- More often consider **relative error**: $\frac{E(x)}{|x|} = R(x)$

Floating-point representation and error

Floating point systems aim to even out relative error: larger gaps occur between larger elements of $\mathbf{X} \implies \mathbf{R(x)} \overset{\sim}{\propto} |\mathbf{x}|$.

Floating point systems have arithmetic limitations:
For $\bar{x} \in \mathbf{X} = \pm\{1/4, 3/8, 1/2, 3/4, 1, 3/2, 2, 3, 4, 6, 8, 12\}$
$12 * 2 = 24 \notin \mathbf{X} \implies$ overflow ($\pm$ `Inf`)
$\frac{1}{4}/2 = \frac{1}{8} \notin \mathbf{X} \implies$ underflow (± `0`, `NaN`)

IEEE754 **de-normalizes** for gradual underflow: $(-1)^0\ (0.1)\ 2^{-2} = \frac{1}{8}$

Section 3.2 - IEEE Standardization

Most common floating point systems:

- **Single precision**: 24 bits (binary digits), 1 sign bit, 7 bits of exponent
- **Double precision**: 52 bits, 1 sign bit, 11 bits of exponent

Note that bit counts are multiples of 8 because hardware organizes the bits into **bytes** (1 byte = 8 bits, single = 4 bytes, double = 8 bytes)

Single precision: 4 bytes, about 6-9 significant decimal digits
Double precision: 8 bytes, about 15-17 significant decimal digits
See Table 3.1 for details of representation range.

Consider error for $x \in \mathbb{R}$ that rounds to $\bar{x}$ so they agree to p digits:

$$\bar{x} = x\,(1+\epsilon)$$

$R(x) = |\epsilon| \leq (1/2)\beta^{1-p} = u$ (**unit roundoff**: upper bound on $R(x)$)
Spacing between normalized floats is $2u|x|$
For $x = 1$, the gap is $2u$, so the next f.p. number available is $1+2u$.

Alternate precision spec: machine epsilon, ϵ_M:
Smallest number you can add to 1 without producing the result 1:

$$\overline{1+x} = 1\, \forall x\ s.t.\ |x| < \epsilon_M$$

Some authors define u to differ by a factor of 2, but either u or ϵ_M characterize f.p. precision.

Section 3.3 - Roundoff Sensitivity

Consider how roundoff error in inputs propagates when performing floating point arithmetic operations:
Let the real inputs be x_1 and x_2 with rounded versions $\bar{x}_1 = x_1(1+\epsilon_1)$ and $\bar{x}_2 = x_2(1+\epsilon_2)$

ϵ indicates signed relative error bounded by $u < 10^{-6} << 1$ (for typical python precision)

Determine bounds on relative error for result of basic arithmetic operations.

Multiplication:

$$\bar{x}_1 * \bar{x}_2 = x_1(1+\epsilon_1) * x_2(1+\epsilon_2)$$

$$\bar{x}_1 * \bar{x}_2 = x_1 * x_2 \left(1+\epsilon_1+\epsilon_2+\epsilon_1 * \epsilon_2\right)$$

Product of relative errors is VERY small $(\epsilon_1 * \epsilon_2 << 1)$, so ignore...

$$\bar{x}_1 * \bar{x}_2 = x_1 * x_2 \left(1+\epsilon_1+\epsilon_2+\ldots\right)$$

$\implies$ **when multiplying, relative errors add**.

Division:

$$\bar{x}_1/\bar{x}_2 = x_1(1+\epsilon_1)/(x_2(1+\epsilon_2))$$

$$\bar{x}_1/\bar{x}_2 = x_1/x_2(1+\epsilon_1) * \frac{1}{1+\epsilon_2}$$

$$\approx x_1/x_2 * (1+\epsilon_1) * (1-\epsilon_2+\epsilon^2+\ldots)$$

$$\approx x_1/x_2 * (1+\epsilon_1-\epsilon_2+\ldots)$$

Again ignore product of relative errors, so output error is bounded by sum of input relative errors.

$\implies$ **Mutiply or divide: relative errors add.**

(Output relative error bounded by sum of input relative errors.)

Addition:

$$\bar{x}_1 + \bar{x}_2 = x_1(1+\epsilon_1) + x_2(1+\epsilon_2)$$

$$= x_1 + x_2 + (x_1 * \epsilon_1 + x_2 * \epsilon_2) = (x_1 + x_2) * (1 + \epsilon_+)$$

$$\implies |\epsilon_+| = \frac{|x_1 * \epsilon_1 + x_2 * \epsilon_2|}{\underbrace{|x_1 + x_2|}_{\text{very small}}} \lessapprox u$$

Addition: Relative error near unit roundoff

if x_1 and x_2 have the same sign!

Addition/Subtraction

$$|\epsilon_{\pm}| = \frac{|x_1 * \epsilon_1 + x_2 * \epsilon_2|}{|x_1 + x_2|}$$

When adding $x_1 \approx -x_2$ (or subtracting $x_1 - x_2$ with $x_1 \approx x_2$)

- denominator becomes arbitrarily small
- error is not well bounded.

***Catastrophic cancellation*: Subtracting nearly equal numbers wipes out the significant digits and leaves noise...**

Dealing with limitations of fixed precision (example 1)

Compute $x^2 - y^2$ in example number system with $x = 2$ and $y = 1$

$\bar{x} \in \mathbf{X} = \pm\{1/2, 3/4, 1, 2, 3\}$

Blindly computing $x * x = 2 * 2$ causes overflow.

Instead, rewrite the expression: $x^2 - y^2 = (x - y) * (x + y)$

Computation becomes $(2 - 1) * (2 + 1) = 1 * 3 = 3$

Avoids overflow and produces a result (accidentally exact)

More usual case: $x = 2, y = \frac{3}{4}$

Exact (and rounded) result:

$(2 - \frac{3}{4}) * (2 + \frac{3}{4}) \to 1 * 3 = 3$ (differs from exact result $\frac{7}{4}$)

More general approach to overflow:

Normalize with units that make your numerical values close to 1

Dealing with finite precision (Example 2: Quadratic formula)

Evaluate roots of $ax^2 + bx + c = 0 = a(x - x_+)(x - x_-)$ using quadratic formula:

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What about this could be problematic?

4ac > b^2

Quadratic formula (cont'd)

When $4ac \ll b^2$ the square root is nearly equal to b, so if $b > 0$ the root x_+ is subject to catastrophic cancellation. However the root x_- should be OK and we know that $a * x_+ * x_- = c$.

$\implies$ Effective plan for $b > 0$:

1. Compute $q = b + \sqrt{b^2 - 4ac}$
2. Compute $x_1 = -\frac{q}{2a}$, $x_2 = \frac{c}{ax_1} = -2\frac{c}{q}$

   Possible improvement: Compute $q = (-1/2)(b + \mathrm{sgn(b)}\sqrt{b^2 - 4ac}) \implies$ $x_1 = \frac{q}{a}$, $x_2 = \frac{c}{ax_1} = \frac{c}{q}$

   Note that basic python does not include a "sign" function. Define your own or import from python's symbolics package `sympy`.

Other "tricks"

Sometimes you can rearrange to avoid cancellation.

Consider a sum where cancellation would be of concern:

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

Collect pairs of terms over common denominator:

$$S = \frac{1}{1 \cdot 2} + \frac{1}{3 \cdot 4} + \dots$$

Eliminates subtraction and avoids cancellation trouble.

Interval Arithmetic (IA): Alternative to dealing with roundoff errors (aside)

- Instead of trying to represent a particular real number (which we cannot do with finite precision), keep track of the lower and upper bounds of an interval that is guaranteed to contain the exact number
- Operation output must guarantee inclusion of exact answer
- Useful property for root-finding/isolation: finite convergence

IA example due to Rump

E. Loh, G. W. Walster, “Rump’s example revisited”, Reliable Computing, vol. 8 (2002), n. 2, pp. 245–248.

$$f(x,y) = (\frac{1335}{4} - x^2)y^6 + x^2(11x^2y^2 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$$

With rational inputs, this function with rational coefficients can be evaluated exactly (e.g. with Mathematica):

$$f(77617, 33096) = -\frac{54767}{66192} \approx -0.827396\ldots$$

Now let's try floating point evaluation:

Floating point evaluation of Rump's example

```
def f(x,y):
    return ((333.75 - x**2)* y**6 + x**2 * (11* x**2 * y**2 - 121 * y**4 - 2) + 5.5 * y**8 + x/(2*y))

f(77617.0, 33096.0)
```

$\rightarrow 1.1726039400531787 \neq -0.827396\ldots$

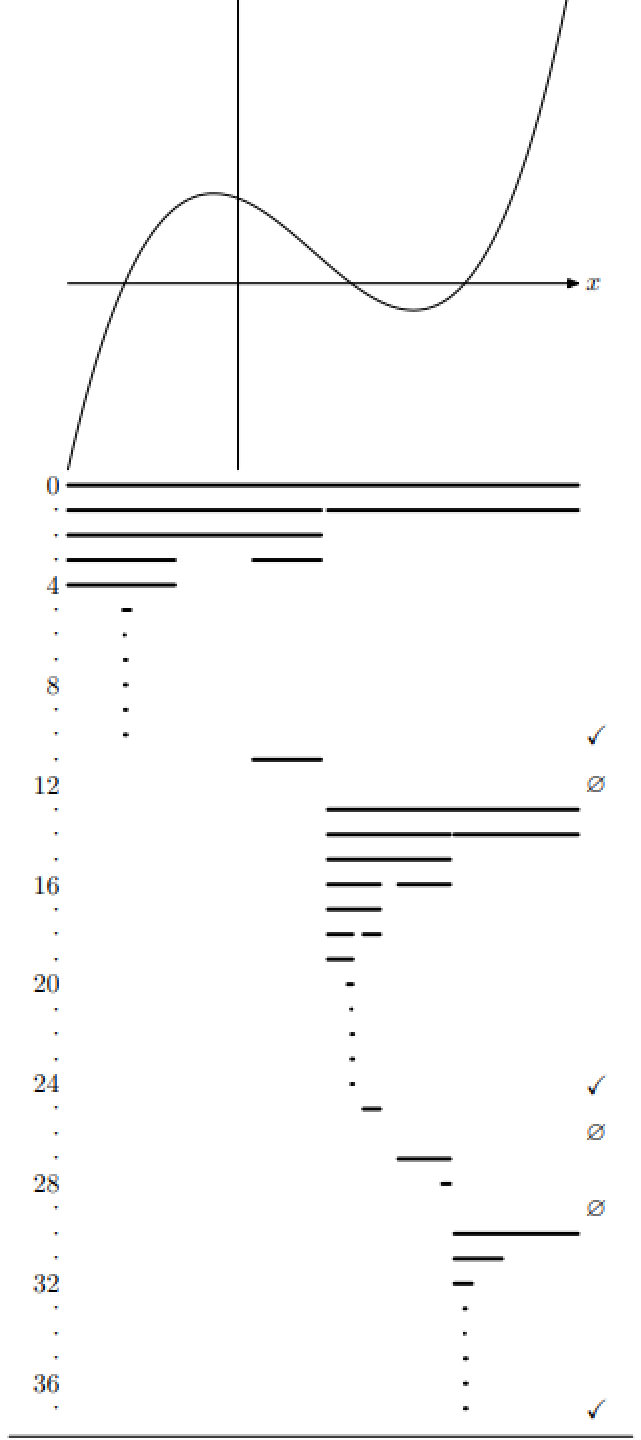
Can't even evaluate, so would there be any chance of locating roots?

Answer turns out to be YES (but stick to simpler case with 1 variable)

IA rootfinding/exclusion

Basic plan:

- Define **interval extensions** that reliably contain correct results
- Evaluate interval extension of the function over some input interval to obtain an output interval
- If output interval includes 0, then input interval can contain a root
- Subdivide and evaluate on subinterval until the output interval excludes 0 $\implies$ NO ROOTS in the subinterval
- Eventually (in finite steps) obtain a set of narrow intervals that are candidate root locations. See full example in notebook...


x
0
4
8
12
16
20
24
28
32
36
✓
∅
✓
∅
∅
✓