STAT 309: MATHEMATICAL COMPUTATIONS I
FALL 2023
LECTURE 7

- we will always have errors in our computations and when we store our inputs or intermediate quantities
- to ensure accuracy of our 'final answer', we need to be able to bound errors
- one great thing about numerical analysis is the following:
  (1) it allows you to compute approximately
  (2) but it also tells you how far away your approximation is from your true solution
- e.g. we can often say something like: this computed number agrees with the true solution up to 5 decimal digits
- the ability to do this is due primarily to two things:
  (1) a standard for performing floating point arithmetic that respect certain rules
  (2) backward error analysis
- failing to understand such issues can lead to serious problems:

`http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html`

1. ERRORS

- errors are of great importance in numerical computations because they allow us to quantify how far our computed solution is from the true solution that we are seeking
- this is important because we do everything in numerical computations in the presence of rounding errors but we still want to guarantee that the solution we found is accurate to some degree
- we usually use norms to measure the size of errors in multivariate quantities like vectors or matrices
- three commonly used measures of the error in an approximation $\widehat{\mathbf{x}}$ to a vector $\mathbf{x}$ are
  - the *absolute error*
  $$\varepsilon_{\text{abs}} = \|\mathbf{x} - \widehat{\mathbf{x}}\|$$
  - the *relative error*
  $$\varepsilon_{\text{rel}} = \frac{\|\mathbf{x} - \widehat{\mathbf{x}}\|}{\|\mathbf{x}\|}$$
  - the *point-wise error*
  $$\varepsilon_{\text{elem}} = \|\mathbf{y}\|, \quad y_i = \frac{\widehat{x}_i - x_i}{x_i}$$
  where we set $y_i = 0$ if the denominator is 0
- by the equivalence of norms, errors under different norms differ at most by constant multiples
- ditto if we have matrices in place of vectors
- we will often write $\Delta\mathbf{x} \coloneqq \mathbf{x} - \widehat{\mathbf{x}}$, $\Delta A \coloneqq A - \widehat{A}$, ..., and refer to these as the error

Date: October 18, 2023, version 1.0. Comments, bug reports: `lekheng@uchicago.edu`.

2. FLOATING POINT NUMBERS

- if a computer word has n bits, then we can represent 2^n numbers but which 2^n numbers should we choose?
- we will describe W. Kahan's choice of these 2^n numbers, this is called the IEEE 754 floating point standard and is universally used for decades (but there's now a competing choice: `https://posithub.org`)
- F = floating numbers, a finite subset of $\mathbb{Q}$, essentially numbers representable as $\pm a_0.a_2a_3\cdots a_k \times 2^{e_1e_2\cdots e_l}$ where $a_i, e_j \in \{0, 1\}$
  - $\pm$ is called the *sign*
  - $a_1.a_2a_3\cdots a_k$, called the *mantissa*, is a positive number $a \in \mathbb{Q}$, expressed in *base* 2, i.e.,
  $$a = a_1 \times 2^0 + a_2 \times 2^{-1} + a_3 \times 2^{-2} + \cdots + a_k \times 2^{-k+1}$$
  - $e_1e_2\cdots e_l$ is called the *exponent*, is an integer $e \in \mathbb{N}$ expressed in *two's complement* (which allows representation of both positive and negative integers), i.e.,
  $$e = -e_1 \times 2^l + e_2 \times 2^{l-1} + e_3 \times 2^{l-2} + \cdots + e_{l-1} \times 2^1 + e_l \times 2^0$$
  - a floating number where $a_1 \neq 0$ is called *normal*
  - a floating number where $a_1 = 0$ is called *subnormal* or *denormal*
  - note that $0 \leq a < 2$ and $-2^l \leq e < 2^l$

FIGURE 1. Subnormal numbers in blue, normal numbers in red.

- note that the mantissa is in unsigned binary (left) and the exponent is in two's complement (right) — the difference is as illustrated below for $k = l = 3$ bits:

decimal	unsigned binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

decimal	two's complement
0	000
1	001
2	010
3	011
−4	100
−3	101
−2	110
−1	111

- floating point representation:
$$\mathbb{R} \to F, \quad x \mapsto \mathrm{fl}(x)$$
- e.g. $\pi \mapsto \mathrm{fl}(\pi) = 3.1415926$
- special numbers like 0, $-\infty$, $+\infty$, `NaN` (not a number) requires special representation defined in the standard
- IEEE floating point standard defines a set F satisfying following properties: for every $x \in [-2^M, -2^m] \cup [2^m, 2^M]$, there exists $x' \in F$ such that
$$|x - x'| \leq \varepsilon_{\text{machine}}|x|$$
or equivalently,
$$\mathrm{fl}(x) = x(1 + \varepsilon_0), \qquad |\varepsilon_0| \leq \varepsilon_{\text{machine}}$$

and for any $x, y \in F$,

$$\begin{aligned} \mathrm{fl}(x \pm y) &= (x \pm y)(1+\varepsilon_1), & |\varepsilon_1| \le \varepsilon_{\text{machine}} \\ \mathrm{fl}(xy) &= (xy)(1+\varepsilon_2), & |\varepsilon_2| \le \varepsilon_{\text{machine}} \\ \mathrm{fl}(x/y) &= (x/y)(1+\varepsilon_3), & |\varepsilon_3| \le \varepsilon_{\text{machine}} \end{aligned}$$

in the last case $y \neq 0$ of course

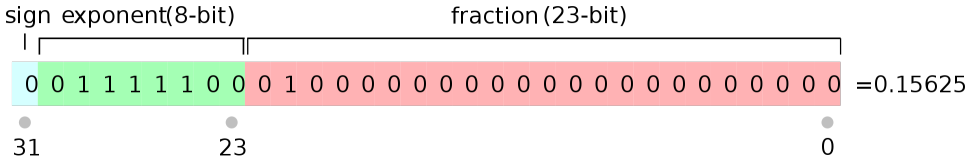
- here m and M are the smallest and largest integers that can be represented as an exponent in F and the set $[-2^M, -2^m] \cup [2^m, 2^M] \subseteq \mathbb{R}$ is called the *range* of F
- *machine epsilon*, a.k.a. unit roundoff, $\varepsilon_{\text{machine}} > 0$ is a constant depending on computing machine used, usually defined as

$$\varepsilon_{\text{machine}} := \inf\{x \in \mathbb{R} : x > 0 \text{ and } \mathrm{fl}(1+x) \neq 1\}$$

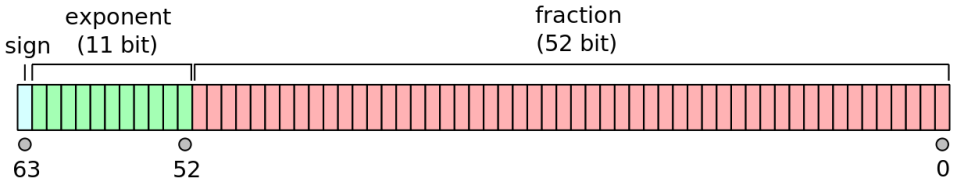
- caution: some people would define unit roundoff u as $\varepsilon_{\text{machine}}/2$ instead
- $\varepsilon_{\text{machine}}$ also gives an upper bound on the relative error due to rounding in floating point arithmetic
- in the first IEEE floating point standard (IEEE 754-1985)
  - floating point numbers are stored in the form

$\pm$	$e_1 e_2 \cdots e_l$	$a_1 a_2 \cdots a_k$

    requiring $1 + l + k$ bits
  - single precision is 32 bits
    - 1 bit for sign, 8 bits for exponent, 23 bits for mantissa
    - allows storage of positive/negative floating numbers of 23 binary (around 7 decimal) digits with magnitude from $2^{-128} \approx 10^{-38}$ to $2^{128} \approx 10^{38}$
    - $\varepsilon_{\text{machine}} = 2^{-23} \approx 1.2 \times 10^{-7}$



  - double precision is 64 bits
    - 1 bit for sign, 11 bits for exponent, 52 bits for mantissa
    - allows for storage of positive/negative floating numbers of 52 binary (around 16 decimal) digits with magnitude from $2^{-1024} \approx 10^{-308}$ to $2^{1024} \approx 10^{308}$
    - $\varepsilon_{\text{machine}} = 2^{-52} \approx 2.2 \times 10^{-16}$



  - the more recent standards (IEEE 754-2008 and IEEE 754-2019) also defined extended precision (80 bits) and quad precision (128 bits)
- let's look at a toy example to get an idea of issues involved when dealing with floating point numbers

- for simplicity, let us assume a floating point system in base 10 (i.e., usual decimal numbers) with a 4-decimal-digit mantissa and a 2-decimal-digit exponent and that has no subnormal numbers, i.e., numbers of the form

$$\pm a_1.a_2a_3a_4 \times 10^e$$

where $a_1 \in \{1, \ldots, 9\}$, $a_2, a_3, a_4 \in \{0, 1, \ldots, 9\}$, and $-99 \leq e \leq 99$

- several different types of errors result from the use of floating-point numbers:

(1) *roundoff errors*

– storage: we can't store 1.1112×10^5 since it has a 5-digit mantissa, so

$$\text{fl}(1.1112 \times 10^5) = 1.111 \times 10^5$$

– arithmetic: we can't store the result of the product of 1.111×10^1 and 1.111×10^2 since that requres a 7-digit mantissa

$$(1.111 \times 10^1) \times (1.111 \times 10^2) = 1.234321 \times 10^3$$

and so

$$\text{fl}((1.111 \times 10^1) \times (1.111 \times 10^2)) = 1.234 \times 10^3$$

– for a real example, evaluate (in radians)

$$\cos(452175521116192774)$$

get:

MATHEMATICA	$-0.5229034783961185\cdots$
MATLAB	-0.263904875163271
R	-0.2639049
Python 3.9 (with `math`)	-0.2639048751632709
Python 3.9 (with `mpmath`)	-0.5229034783961185

– why? in binary:

$$n = 1100100011001110011001111011010000001000010000000000000000110_2$$
$$\text{fl}(n) = 1100100011001110011001111011010000001000010000000000000000000_2$$

– IEEE double precision stores 52 digits after leading 1, red part rounded to 0's

– in decimal

$$n = 452175521116192774$$
$$\text{fl}(n) = 452175521116192768$$

– MATLAB, R, and Python 3.9 (`math`) computed $\cos(\text{fl}(n))$

– MATHEMATICA and Python 3.9 (`mpmath`) computed $\cos(n)$

– the results given by MATLAB, R, and Python 3.9 (`math`) are completely off!

(2) *overflows* and *underflows*

– overflow: exponent too big

$$\text{fl}((1.000 \times 10^{55}) \times (1.000 \times 10^{50})) \rightarrow \texttt{overflow}$$

– underflow: exponent too small

$$\text{fl}((1.000 \times 10^{-55}) \times (1.000 \times 10^{-50})) \rightarrow \texttt{underflow}$$

(3) *cancellation errors*

– if you try to compute

$$844487^5 + 1288439^5 - 1318202^5$$

directly in MATLAB, a *numerical computing* software, you get zero as your answer — does that mean you have found a counterexample to Fermat's last theorem?

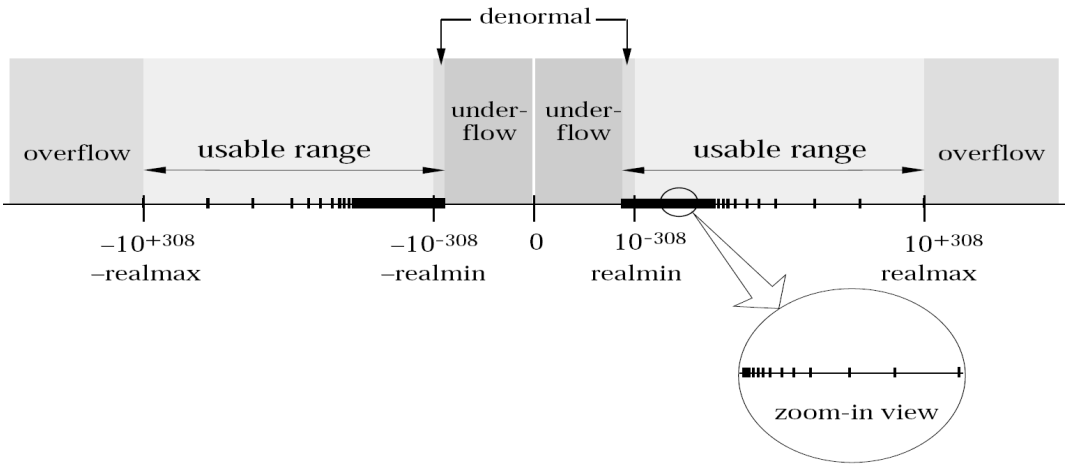


FIGURE 2. The set of floating point numbers F. Note in particular that F is not regularly spaced.

  - now try it on MATHEMATICA, a *symbolic computing* software, you get
$$844487^5 + 1288439^5 - 1318202^5 = -235305158626,$$
which is far from zero
  - the reason is cancellation error (here we need a 16-digit decimal mantissa to match what you get in MATLAB):
$$\mathrm{fl}(844487^5 + 1288439^5) = 3.980245235185639 \times 10^{30},$$
$$\mathrm{fl}(1318202^5) = 3.980245235185639 \times 10^{30},$$
i.e., the numbers have the same significant[1] digits in floating point representation

- issues like these require that we exercise care in designing numerical algorithms — avoid forming numbers that are too large or too small to minimize overflows; avoid subtracting two numbers that are almost equal to avoid cancellation errors
- a simple example is the evaluation of the vector 2-norm, the usual formula
$$\|\mathbf{x}\|_2 = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}$$
gives a poor way of computing the value in the presence of rounding error
  - assuming our toy model for F, take
$$\mathbf{x} = \begin{bmatrix} 10^{-49} \\ 10^{-50} \\ \vdots \\ 10^{-50} \end{bmatrix} \in \mathbb{R}^{101}$$
  - this can be stored exactly in our toy floating point system and so there is no rounding error here
  - but since $x_i^2 = 10^{-100}$ for $i = 2, \ldots, 101$,
$$\mathrm{fl}(x_2^2) = \cdots = \mathrm{fl}(x_{101}^2) = 0$$
and applying the usual formula in floating point arithmetic gives $\|\mathbf{x}\|_2 \approx 10^{-49}$ although $\|\mathbf{x}\|_2 = \sqrt{2 \times 10^{98}} \approx 1.1414 \times 10^{-49}$ — a 40% error

[1] `https://en.wikipedia.org/wiki/Loss_of_significance`

– a better algorithm would be a 2-step process

$$\widehat{\mathbf{x}} = \begin{cases} \mathbf{x}/\|\mathbf{x}\|_\infty & \text{if } \|\mathbf{x}\|_\infty \neq 0 \\ \mathbf{0} & \text{if } \|\mathbf{x}\|_\infty = 0 \end{cases}$$
$$\|\mathbf{x}\|_2 = \|\mathbf{x}\|_\infty \|\widehat{\mathbf{x}}\|_2$$

note that $|\widehat{x}_i| \leq 1$ for every $i = 1, \ldots, n$ and so there's no overflow; there's no underflow as long as none of the $|\widehat{x}_i|$ are much smaller than 1

- there are other less obvious examples due to all kinds of intricate errors in floating point computations, we will state two of them but won't go into the details

sample variance: if we want to compute the sample variance of n numbers $x_1, \ldots, x_n$, we could do it in either of the following ways:

(1) first compute sample mean

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

and then compute sample variance via

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2$$

(2) compute sample variance directly via

$$s^2 = \frac{1}{n-1} \left[\sum_{i=1}^{n} x_i^2 - \frac{1}{n} \left(\sum_{i=1}^{n} x_i \right)^2 \right]$$

- mathematically the are equivalent and in exact arithmetic they should give identical values
- computationally both require the same number of operations but the first way requires two passes over the data while the second requires only one pass over the data, so it would appear that the second way is better
- in fact many statistics textbooks recommend the second way but it is actually a *very poor* way to compute sample variance in the presence of rounding error — the answer can even be negative
- the first way is much more accurate (and always nonengative) in floating point arithmetic

difference of squares: the same phenomenon is exhibited in

$$x^2 - y^2 = (x - y)(x + y)$$

- the cancellation error involved in computing $x^2 - y^2$ is bigger (and can be much bigger) than that involved in computing $x - y$
- as the arithmetic costs of computing $x^2 - y^2$ and $(x - y)(x + y)$ are roughly the same, the latter is always preferred

area of triangle: Heron's formula for the area of a triangle with sides of lengths a, b, c is

$$\sqrt{s(s-a)(s-b)(s-c)}, \qquad s \coloneqq (a + b + c)/2$$

- when the triangle is very flat, i.e., $a \approx b + c$, then $s \approx a$ and $s - a$ subtracts two nearly equal number, resulting in large cancellation error
- a better way is to first sort the lengths so that $a \geq b \geq c$ and then use

$$\frac{1}{4} \sqrt{\big(a + (b + c)\big)\big(c - (a - b)\big)\big(c + (a - b)\big)\big(a + (b - c)\big)}$$

quadratic formula: the usual quadratic formula

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

is a poor way for computing the roots x_1, x_2 in floating point arithmetic

- a better way to compute them is via

$$x_1 = \frac{-\big[b + \text{sign}(b)\sqrt{b^2 - 4ac}\big]}{2a}, \qquad x_2 = \frac{c}{ax_1}$$

- reason again is cancellation error, see `https://en.wikipedia.org/wiki/Loss_of_significance` for an example with actual numerical values for a, b, c

3. Takeaway

- working with finite precision arithmetic is an art
- workarounds are often on a case-by-case basis
- for example to get around the rounding error in computing $\cos(452175521116192774)$, we might use

$$\cos(n) = \cos(\text{fl}(n) + 6) = \cos(\text{fl}(n))\cos(6) - \sin(\text{fl}(n))\sin(6)$$

with $n = 452175521116192774$ and $\text{fl}(n) = 452175521116192768$

- or more generally expressing n as a sum of numbers with exact floating point representations
- but if the difference on the right involves numbers that are too close, we might get cancellation error and the result might be worse
- alternatively we might exploit periodicity $\cos(n) = \cos(n \bmod 2\pi)$ but unless we approximate π to sufficiently high accuracy, the result might be worse too
- it is never clear-cut and which method to use depends on the value of n
- some of cleverest real-life algorithms actually depend on exploiting the design of IEEE floating arithmetic: a famous one is the algorithm for $1/\sqrt{x}$ in the old computer game Quake III Arena[2]
- we will not worry too much about these low-level issues in our course but you should be aware of their existence

[2] `https://en.wikipedia.org/wiki/Fast_inverse_square_root`