

# # 3

Errors in computation; where do they come from?

---

L. Olson

September 1, 2015

Department of Computer Science  
University of Illinois at Urbana-Champaign

# objectives

- look at floating point representation in its basic form
- expose errors of a different form: rounding error
- highlight IEEE-754 standard

# why this is important:

- Errors come in two forms: truncation error and rounding error
  - we always have them ...
  - case study: Intel
  - our jobs as developers: reduce impact

example: calculating  $x = x + 0.1$

## next: floating point numbers

- We're familiar with base 10 representation of numbers:

$$1234 = 4 \times 10^0 + 3 \times 10^1 + 2 \times 10^2 + 1 \times 10^3$$

and

$$.1234 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3} + 4 \times 10^{-4}$$

- we write 1234.1234 as an integer part and a fractional part:

$$a_3 a_2 a_1 a_0 . b_1 b_2 b_3 b_4$$

- For some (even simple) numbers, there may be an *infinite* number of digits to the right:

$$\pi = 3.14159 \dots$$

$$1/9 = 0.11111 \dots$$

$$\sqrt{2} = 1.41421 \dots$$

# other bases

- So far, we have just base 10. What about base  $\beta$ ?
- binary ( $\beta = 2$ ), octal ( $\beta = 8$ ), hexadecimal ( $\beta = 16$ ), etc
- In the  $\beta$ -system we have

$$(a_n \dots a_2 a_1 a_0 . b_1 b_2 b_3 b_4 \dots)_\beta = \sum_{k=0}^n a_k \beta^k + \sum_{k=0}^{\infty} b_k \beta^{-k}$$

# integer conversion

An algorithm to compute the base 2 representation of a base 10 integer

$$\begin{aligned}(N)_{10} &= (a_j a_{j-1} \dots a_2 a_1 a_0)_2 \\ &= a_j \cdot 2^j + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0\end{aligned}$$

Compute  $(N)_{10}/2 = Q + R/2$ :

$$\frac{N}{2} = \underbrace{a_j \cdot 2^{j-1} + \dots + a_1 \cdot 2^0}_{=Q} + \underbrace{\frac{a_0}{2}}_{=R/2}$$

## Example

Example: compute  $(11)_{10}$  base 2

$$11/2 = 5R1 \Rightarrow a_0 = 1$$

$$5/2 = 2R1 \Rightarrow a_1 = 1$$

$$2/2 = 1R0 \Rightarrow a_2 = 0$$

$$1/2 = 0R1 \Rightarrow a_3 = 1$$

So  $(11)_{10} = (1011)_2$

## the other way...

Convert a base-2 number to base-10:

$$(11\ 000\ 101)_2$$

$$= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 1 + 2(0 + 2(1 + 2(0 + 2(0 + 2(0 + 2(1 + 2(1))))))))$$

$$= 197$$



# converting fractions

- straight forward way is not easy
- goal: for  $x \in [0, 1]$  write

$$x = 0.b_1b_2b_3b_4 \dots = \sum_{k=1}^{\infty} c_k \beta^{-k} = (0.c_1c_2c_3 \dots)_{\beta}$$

- $\beta(x) = (c_1.c_2c_3c_4 \dots)_{\beta}$
- multiplication by  $\beta$  in base- $\beta$  only shifts the radix

# fraction algorithm

An algorithm to compute the binary representation of a fraction  $x$ :

$$\begin{aligned}x &= 0.b_1b_2b_3b_4\dots \\ &= b_1 \cdot 2^{-1} + \dots\end{aligned}$$

Multiply  $x$  by 2. The integer part of  $2x$  is  $b_1$

$$2x = b_1 \cdot 2^0 + b_2 \cdot 2^{-1} + b_3 \cdot 2^{-2} + \dots$$

## Example

Example: Compute the binary representation of 0.625

$$2 \cdot 0.625 = 1.25 \quad \Rightarrow \quad b_{-1} = 1$$

$$2 \cdot 0.25 = 0.5 \quad \Rightarrow \quad b_{-2} = 0$$

$$2 \cdot 0.5 = 1.0 \quad \Rightarrow \quad b_{-3} = 1$$

So  $(0.625)_{10} = (0.101)_2$

# a problem with precision

```
1  $r_0 = x$ 
2 for  $k = 1, 2, \dots, m$ 
3   if  $r_{k-1} \geq 2^{-k}$ 
4      $b_k = 1$ 
5      $r_k = r_{k-1} - 2^{-k}$ 
6   else
7      $b_k = 0$ 
8   end
9 end
```

$k$	$2^{-k}$	$b_k$	$r_k = r_{k-1} - b_k 2^{-k}$
0			0.8125
1	0.5	1	0.3125
2	0.25	1	0.0625
3	0.125	0	0.0625
4	0.0625	1	0.0000

binary fraction example

## a problem with precision

For other numbers, such as  $\frac{1}{5} = 0.2$ , an infinite length is needed.

0.2  $\rightarrow$  .0011 0011 0011 ...

So 0.2 is stored just fine in base-10, but needs infinite number of digits in base-2

!!!

This is *roundoff* error in its basic form...

## Roundoff

*Roundoff* occurs when digits in a decimal point (0.3333...) are lost (0.3333) due to a limit on the memory available for storing one numerical value.

## Truncation

*Truncation* error occurs when discrete values are used to approximate a mathematical expression.

# uncertainty: well- or ill-conditioned?

Errors in input data can cause *uncertain* results

- input data can be experimental or rounded. leads to a certain variation in the results
- *well-conditioned*: numerical results are insensitive to small variations in the input
- *ill-conditioned*: small variations lead to drastically different numerical calculations (a.k.a. poorly conditioned)

# our job

As numerical analysts, we need to

1. solve a problem so that the calculation is not susceptible to large roundoff error
2. solve a problem so that the approximation has a *tolerable* truncation error

How?

- incorporate roundoff-truncation knowledge into
  - the mathematical model
  - the method
  - the algorithm
  - the software design
- awareness → correct interpretation of results



## Normalized Floating-Point Representation

Real numbers are stored as

$$x = \pm(0.d_1d_2d_3 \dots d_m)_\beta \times \beta^e$$

- $d_1d_2d_3 \dots d_m$  is the mantissa,  $e$  is the exponent
- $e$  is negative, positive or zero
- the general normalized form requires  $d_1 \neq 0$

## Example

In base 10

- 1000.12345 can be written as

$$(0.100012345)_{10} \times 10^4$$

- 0.000812345 can be written as

$$(0.812345)_{10} \times 10^{-3}$$

# floating point

Suppose we have only 3 bits for a mantissa and a 1 bit exponent stored like



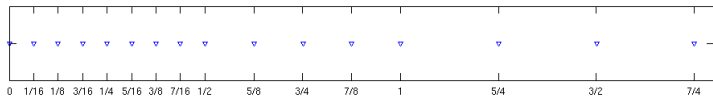
All possible combinations give:

$$000_2 = 0$$

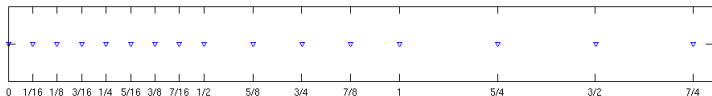
$$\dots \times 2^{-1,0,1}$$

$$111_2 = 7$$

So we get  $0, \frac{1}{16}, \frac{2}{16}, \dots, \frac{7}{16}, 0, \frac{1}{4}, \frac{2}{4}, \dots, \frac{7}{4}$ , and  $0, \frac{1}{8}, \frac{2}{8}, \dots, \frac{7}{8}$ . On the real line:



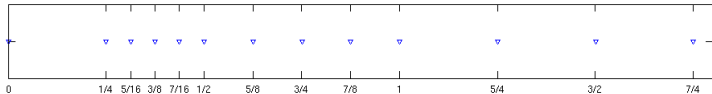
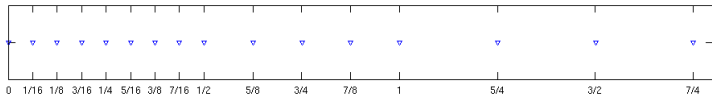
# overflow, underflow



- computations too close to zero may result in *underflow*
- computations too large may result in *overflow*
- overflow error is considered more severe
- underflow can just fall back to 0

# normalizing

If we use the normalized form in our 4-bit case, we lose  $0.001_2 \times 2^{-1,0,1}$  along with other. So we cannot represent  $\frac{1}{16}$ ,  $\frac{1}{8}$ , and  $\frac{3}{16}$ .

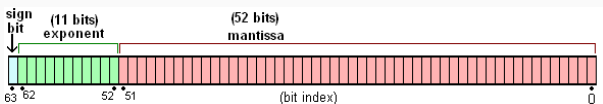


# ieee-754 why this is important:

- IEEE-754 is a widely used standard accepted by hardware/software makers
  - defines the floating point distribution for our computation
  - offer several rounding modes which effect accuracy
- Floating point arithmetic emerges in nearly every piece of code
  - even modest mathematical operation yield loss of significant bits
  - several pitfalls in common mathematical expressions

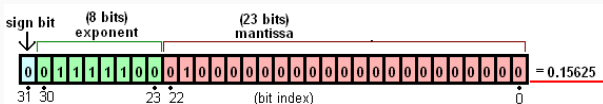
# ieee floating point (v. 754)

- How much storage do we actually use in practice?
- 32-bit word lengths are typical
- IEEE Single-precision floating-point numbers use 32 bits
- IEEE Double-precision floating-point numbers use 64 bits
- Goal: use the 32-bits to best represent the normalized floating point number



# ieee single precision (marc-32)

$$x = \pm q \times 2^m$$



## Notes:

- 1-bit sign
- 8-bit exponent  $|m|$
- 23-bit mantissa  $q$
- The leading one in the mantissa  $q$  does not need to be represented:  
 $b_1 = 1$  is hidden bit
- IEEE 754: put  $x$  in  $1.f$  normalized form
- $0 < m + 127 = c < 255$
- Largest exponent = 127, Smallest exponent = -126
- Special cases:  $c = 0, 255$



$$x = \pm q \times 2^m$$

Process for  $x = -52.125$ :

1. Convert both integer and fractional to binary:

$$x = -(110100.0010000000)_2$$

2. Convert to 1.f form:  $x = - \underbrace{(1)}_1 \underbrace{(101\ 000\ 010\ 000\ \dots\ 0)}_{23}_2 \times 2^5$

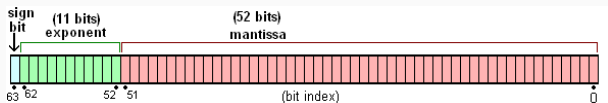
3. Convert exponent  $5 = c - 127 \Rightarrow c = 132 \Rightarrow c = \underbrace{(10\ 000\ 100)}_8_2$

$$\underbrace{1}_1 \underbrace{10\ 000\ 100}_8 \underbrace{101\ 000\ 010\ 000\ \dots\ 0}_{23}$$

## Special Cases:

- denormalized/subnormal numbers: use 1 extra bit in the significant: exponent is now  $-126$  (less precision, more range), indicated by  $00000000_2$  in the exponent field
- two zeros:  $+0$  and  $-0$  (0 mantissa, 0 exponent)
- two  $\infty$ 's:  $+\infty$  and  $-\infty$
- $\infty$  (0 mantissa,  $11111111_2$  exponent)
- NaN (any mantissa,  $11111111_2$  exponent)
- see appendix C.1 in NMC 6th ed.

# ieee double precision



- 1-bit sign
- 11-bit exponent
- 52-bit mantissa
- single-precision: about 6 decimal digits of precision
- double-precision: about 15 decimal digits of precision
- $m = c - 1023$

# precision vs. range

type	range	approx range
single	$-3.40 \times 10^{38} \leq x \leq -1.18 \times 10^{-38}$	$2^{-126} \rightarrow 2^{128}$
	$1.18 \times 10^{-38} \leq x \leq 3.40 \times 10^{38}$	
double	$-1.80 \times 10^{318} \leq x \leq -2.23 \times 10^{-308}$	$2^{-1022} \rightarrow 2^{1024}$
	$2.23 \times 10^{-308} \leq x \leq 1.80 \times 10^{308}$	

small numbers example

plus one example



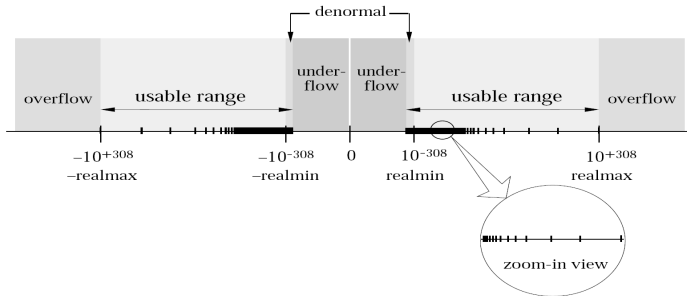
## $\epsilon_m$ : machine epsilon

Machine epsilon  $\epsilon_m$  is the smallest number such that

$$fl(1 + \epsilon_m) \neq 1$$

- The double precision machine epsilon is about  $2^{-52}$ .
- The single precision machine epsilon is about  $2^{-23}$ .

## Floating Point Number Line





# floating point errors

- Not all reals can be exactly represented as a machine floating point number. Then what?
- Round-off error
- IEEE options:
  - Round to next nearest FP (preferred), Round to 0, Round up, and Round down

Let  $x_+$  and  $x_-$  be the two floating point machine numbers closest to  $x$

- round to nearest:  $round(x) = x_-$  or  $x_+$ , whichever is closest
- round toward 0:  $round(x) = x_-$  or  $x_+$ , whichever is between 0 and  $x$
- round toward  $-\infty$  (down):  $round(x) = x_-$
- round toward  $+\infty$  (up):  $round(x) = x_+$

# floating point errors

How big is this error? Suppose ( $x$  is closer to  $x_-$ )

$$x = (0.1b_2b_3 \dots b_{24}b_{25}b_{26})_2 \times 2^m$$

$$x_- = (0.1b_2b_3 \dots b_{24})_2 \times 2^m$$

$$x_+ = ((0.1b_2b_3 \dots b_{24})_2 + 2^{-24}) \times 2^m$$

$$|x - x_-| \leq \frac{|x_+ - x_-|}{2} = 2^{m-25}$$

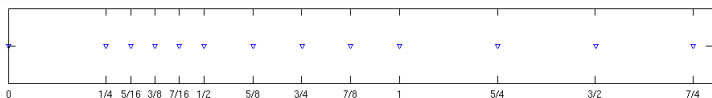
$$\left| \frac{x - x_-}{x} \right| \leq \frac{2^{m-25}}{1/2 \times 2^m} \leq 2^{-24} = \epsilon_m/2$$

# floating point arithmetic

- Problem: The set of representable machine numbers is FINITE.
- So not all math operations are well defined!
- Basic algebra breaks down in floating point arithmetic

## Example

$$a + (b + c) \neq (a + b) + c$$



# floating point arithmetic

Rule 1.

$$fl(x) = x(1 + \epsilon), \quad \text{where } |\epsilon| \leq \epsilon_m$$

Rule 2.

For all operations  $\odot$  (one of  $+$ ,  $-$ ,  $*$ ,  $/$ )

$$fl(x \odot y) = (x \odot y)(1 + \epsilon_{\odot}), \quad \text{where } |\epsilon_{\odot}| \leq \epsilon_m$$

Rule 3.

For  $+$ ,  $*$  operations

$$fl(a \odot b) = fl(b \odot a)$$

There were many discussions on what conditions/rules should be satisfied by floating point arithmetic. The IEEE standard is a set of standards adopted by many CPU manufacturers.

# floating point arithmetic

Consider the sum of 3 numbers:  $y = a + b + c$ .

Done as  $fl(fl(a + b) + c)$

$$\begin{aligned}\eta &= fl(a + b) = (a + b)(1 + \epsilon_1) \\ y_1 &= fl(\eta + c) = (\eta + c)(1 + \epsilon_2) \\ &= [(a + b)(1 + \epsilon_1) + c](1 + \epsilon_2) \\ &= [(a + b + c) + (a + b)\epsilon_1](1 + \epsilon_2) \\ &= (a + b + c) \left[ 1 + \frac{a + b}{a + b + c} \epsilon_1 (1 + \epsilon_2) + \epsilon_2 \right]\end{aligned}$$

So disregarding the high order term  $\epsilon_1\epsilon_2$

$$fl(fl(a + b) + c) = (a + b + c)(1 + \epsilon_3) \quad \text{with} \quad \epsilon_3 \approx \frac{a + b}{a + b + c} \epsilon_1 + \epsilon_2$$

# floating point arithmetic

If we redid the computation as  $y_2 = fl(a + fl(b + c))$  we would find

$$fl(a + fl(b + c)) = (a + b + c)(1 + \epsilon_4) \quad \text{with} \quad \epsilon_4 \approx \frac{b + c}{a + b + c} \epsilon_1 + \epsilon_2$$

Main conclusion:

The first error is amplified by the factor  $(a + b)/y$  in the first case and  $(b + c)/y$  in the second case.

In order to sum  $n$  numbers more accurately, it is better to start with the small numbers first. [However, sorting before adding is usually not worth the cost!]

# floating point arithmetic

One of the most serious problems in floating point arithmetic is that of cancellation. If two large and close-by numbers are subtracted the result (a small number) carries very few accurate digits (why?). This is fine if the result is not reused. If the result is part of another calculation, then there may be a serious problem

## Example

Roots of the equation

$$x^2 + 2px - q = 0$$

Assume we want the root with smallest absolute value:

$$y = -p + \sqrt{p^2 + q} = \frac{q}{p + \sqrt{p^2 + q}}$$

# catastrophic cancellation

Adding  $c = a + b$  will result in a large error if

- $a \gg b$
- $a \ll b$

Let

$$a = x.xxx \dots \times 10^0$$

$$b = y.yyy \dots \times 10^{-8}$$

Then

$$\begin{array}{r} \text{finite precision} \\ \overbrace{x.xxx \quad xxxx \quad xxxx \quad xxxx} \\ + \quad 0.000 \ 0000 \ yyy \ yyy \quad yyy \ yyy \\ \hline = \quad x.xxx \quad xxxx \quad zzzz \quad zzzz \quad \underbrace{???? \quad ????}_{\text{lost precision}} \end{array}$$



# catastrophic cancellation

Subtracting  $c = a - b$  will result in large error if  $a \approx b$ . For example

$$a = x.xxxx\ xxxx\ xxx1 \overbrace{ssss\dots}^{\text{lost}}$$

$$b = x.xxxx\ xxxx\ xxx0 \overbrace{tttt\dots}^{\text{lost}}$$

Then

$$\begin{array}{r} \overbrace{x.xxx\ xxxx\ xxx1}^{\text{finite precision}} \\ + \quad x.xxx\ xxxx\ xxx0 \\ \hline = \quad 0.000\ 0000\ 0001 \quad \underbrace{????\ ????}_{\text{lost precision}} \end{array}$$

# summary

- addition:  $c = a + b$  if  $a \gg b$  or  $a \ll b$
- subtraction:  $c = a - b$  if  $a \approx b$
- catastrophic: caused by a single operation, not by an accumulation of errors
- can often be fixed by mathematical rearrangement

## Example

$x = 0.3721448693$  and  $y = 0.3720214371$ . What is the relative error in  $x - y$  in a computer with 5 decimal digits of accuracy?

$$\frac{|x - y - (\bar{x} - \bar{y})|}{|x - y|} = \frac{|0.3721448693 - 0.3720214371 - 0.37214 + 0.37202|}{|0.3721448693 - 0.3720214371|}$$
$$\approx 3 \times 10^{-2}$$

## Loss of Precision Theorem

Let  $x$  and  $y$  be (normalized) floating point machine numbers with  $x > y > 0$ .

If  $2^{-p} \leq 1 - \frac{y}{x} \leq 2^{-q}$  for positive integers  $p$  and  $q$ , the significant binary digits lost in calculating  $x - y$  is between  $q$  and  $p$ .

# loss of significance

## Example

Consider  $x = 37.593621$  and  $y = 37.584216$ .

$$2^{-11} < 1 - \frac{y}{x} = 0.0002501754 < 2^{-12}$$

So we lose 11 or 12 bits in the computation of  $x - y$ . yikes!

## Example

Back to the other example (5 digits):  $x = 0.37214$  and  $y = 0.37202$ .

$$10^{-4} < 1 - \frac{y}{x} = 0.00032 < 10^{-5}$$

So we lose 4 or 5 bits in the computation of  $x - y$ . Here,  $x - y = 0.00012$

which has only 1 significant digit that we can be sure about

# loss of significance

So what to do? Mainly rearrangement.

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

# loss of significance

So what to do? Mainly rearrangement.

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

Problem at  $x \approx 0$ .

# loss of significance

So what to do? Mainly rearrangement.

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

Problem at  $x \approx 0$ .

One type of fix:

$$\begin{aligned} f(x) &= \left( \sqrt{x^2 + 1} - 1 \right) \left( \frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1} \right) \\ &= \frac{x^2}{\sqrt{x^2 + 1} + 1} \end{aligned}$$

no subtraction!