# Introduction to floating point arithmetic

Eric J. Kostelich

ASU
Arizona State University
School of Mathematical & Statistical Sciences

September 5, 2018

# Main topics today

- Absolute and relative error
- Why base 2?
- What do NaN and Inf mean?
- What is a ulp?
- Principal reference: David Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys* **28** (1991)

# Notions of error

- Given an approximation $\hat{x}$ to a “true” value $x$
- The absolute error is $|\hat{x} - x|$
- The relative error is defined for $x \neq 0$:

$$\text{relative error} = \frac{\text{absolute error}}{|x|} = \frac{|\hat{x} - x|}{|x|}$$

- The relative error puts the absolute error in context
- If we don’t know $x$, then both measures are approximate

# Base-$b$ representations ($b$ an integer greater than $1$)

- If $x \neq 0$, then its normalized base-$b$ representation is

$$x = \pm a_0.a_1a_2a_3 \cdots \times b^e \quad \text{where } 1 \leq a_0 < b$$

- The floating-point format represents

$$x = \left(a_0 + \frac{a_1}{b^1} + \frac{a_2}{b^2} + \frac{a_3}{b^3} + \cdots\right) \times b^e$$

- Example: The speed of light is 299,792,458 m/s, whose normalized base-10 representation is

$$\underbrace{2.99\,792\,458}_{\text{significand or mantissa}} \times 10^8$$

# Example

- What is the normalized base-$3$ representation of $23/27$?
- Find $a_1, a_2, a_3, \ldots$ such that

$$\frac{23}{27} = \frac{a_1}{3^1} + \frac{a_2}{3^2} + \frac{a^3}{3^3} + \cdots$$

# Example

- What is the normalized base-$3$ representation of $23/27$?
- Find $a_1, a_2, a_3, \ldots$ such that

$$\frac{23}{27} = \frac{a_1}{3^1} + \frac{a_2}{3^2} + \frac{a^3}{3^3} + \cdots$$

- Since $1 > \frac{23}{27} - \frac{2}{3} = \frac{5}{27} > 0$, we have $a_1 = 2$

# Example

- What is the normalized base-$3$ representation of $23/27$?
- Find $a_1, a_2, a_3, \ldots$ such that

$$\frac{23}{27} = \frac{a_1}{3^1} + \frac{a_2}{3^2} + \frac{a^3}{3^3} + \cdots$$

- Since $1 > \frac{23}{27} - \frac{2}{3} = \frac{5}{27} > 0$, we have $a_1 = 2$
- Since $\frac{2}{9} > \frac{5}{27} - \frac{1}{9} = \frac{2}{27}$, we have $a_2 = 1$ and $a_3 = 2$
- Hence $\frac{23}{27} = 2.12_3 \times 3^{-1}$, which terminates

# Geometric series

- The infinite series is $S = a + ar + ar^2 + ar^3 + \cdots$
- The finite series is $S_n = a + ar + \cdots + ar^n$
- Now

$$\begin{array}{rcl} S_n & = & a + ar + ar^2 + \cdots ar^n \\ rS_n & = & \quad ar + ar^2 + \cdots ar^n + ar^{n+1} \\ \hline (1-r)S_n & = & a - ar^{n+1} \end{array}$$

Thus

$$S_n = \frac{a\left(1 - r^{n+1}\right)}{1-r} \longrightarrow S = \frac{a}{1-r} \quad \text{if } |r| < 1$$

# Basic results

- Base-$b$ representations are not necessarily unique:

$$2 = 1.99\overline{9} = 1 + \left( \frac{\frac{9}{10}}{1 - \frac{1}{10}} \right)$$

- Theorem: $x$ is rational if and only if $x$ has an eventually repeating base-$b$ representation
- Theorem: $x = p/q$, written in lowest terms, has a terminating base-$b$ expansion if and only if $q$ divides some power of $b$

# Example

$$
\begin{aligned}
0.\overline{142857} &= 0.142857 \times \left(1 + 10^{-6} + 10^{-12} + \cdots\right) \\
&= 0.142857 \times \left(\frac{1}{1 - 10^{-6}}\right) \\
&= \frac{0.142857}{0.999999} \\
&= \frac{1}{7}
\end{aligned}
$$

## The base-2 representation of $\frac{1}{10}$

- $10$ does not divide any power of $2$, so the representation cannot terminate
- Thus $1/10$ cannot be represented exactly in binary floating-point arithmetic
- A calculation shows that

$$\frac{1}{10} - \frac{1}{16} - \frac{1}{32} = \frac{1}{160} > \frac{1}{256}$$

so the normalized base-$2$ representation starts with $1.100\cdots \times 2^{-4}$

- Lemma: $\frac{1}{10} = 1.100\overline{1100}_2 \times 2^{-4}$

# Why base 2? Two reasons

- Reason #1: Base 2 minimizes wobble
- Consider $x = 0.07\,111\,112\,773$

# Why base 2? Two reasons

- Reason #1: Base 2 minimizes wobble
- Consider $x = 0.07\,111\,112\,773$
- The closest normalized base-16 representation with a 24-bit significand is

$$\begin{aligned}\hat{x}_{16} &= 1.23456_{16} \times 16^{-1} \\ &= \underbrace{0001}_{a_1=1}.\underbrace{0010}_{a_2=2}\ \underbrace{0011}_{a_3=3}\ \underbrace{0100}_{a_4=4}\ \underbrace{0101}_{a_5=5}\ \underbrace{0110}_{a_6=6} \times 16^{-1}\end{aligned}$$

## Why base 2? Two reasons

- Reason #1: Base 2 minimizes wobble
- Consider $x = 0.07\,111\,112\,773$
- The closest normalized base-16 representation with a 24-bit significand is

$$
\begin{aligned}
\hat{x}_{16} &= 1.23456_{16} \times 16^{-1} \\
&= \underbrace{0001}_{a_1=1}.\ \underbrace{0010}_{a_2=2}\ \underbrace{0011}_{a_3=3}\ \underbrace{0100}_{a_4=4}\ \underbrace{0101}_{a_5=5}\ \underbrace{0110}_{a_6=6} \times 16^{-1}
\end{aligned}
$$

- The leading digit wastes 3 significant bits. The effective precision is 21 bits!

## $x = 0.07\,111\,112\,773$, continued

- The normalized 24-bit base-2 representation has 3 more significant bits:

$$\hat{x}_2 = \underbrace{1}_{a_1=1} . \underbrace{0010}_{a_2=2} \; \underbrace{0011}_{a_3=3} \; \underbrace{0100}_{a_4=4} \; \underbrace{0101}_{a_5=5} \; \underbrace{0110}_{a_6=6} \; 011 \times 2^{-4}$$

- The absolute errors due to rounding are

$$\begin{aligned} |x - \hat{x}_{16}| &\approx 4.47 \times 10^{-8} \\ |x - \hat{x}_2| &\approx 2.23 \times 10^{-8} \end{aligned}$$

- The base-2 representation does not wobble and is twice as accurate

# Why base 2, continued

- Reason #2: Base 2 minimizes the absolute error in rounding
- Suppose we have $d$ base-$b$ digits
- The maximum truncation error due to rounding in base $b$ is $(b-1) \times b^{-d-1}$
- For base 10, this is $9 \times 10^{-d-1}$
- For base 2, it's $1 \times 2^{-d-1}$
- For a given number of digits, truncation error is minimized in base 2

# The four rounding modes

In 5-digit decimal arithmetic, given $e = 2.718281\cdots$ and $\pi = 3.14159265\cdots$:

- Round toward 0: $e \models 2.7182$, $-\pi \models -3.1415$ (chopped rounding)
- Round toward $+\infty$: $e \models 2.7183$, $-\pi \models -3.1415$
- Round toward $-\infty$: $e \models 2.7182$, $-\pi \models -3.1416$
- Round to nearest: $e \models 2.7183$, $-\pi \models -3.1416$

# Unit in the last place (ulp)

- Refers to the change in value when the least significant digit in the significand of a floating-point number is changed by one unit
- Examples: In 5-digit decimal arithmetic, $\pi$ can be rounded to either $3.1415$ or to $3.1416$
- The difference between the two rounded values is $1 \text{ ulp} = 0.0001 = 10^{-4}$
- Likewise, $c = 299{,}792{,}458 \models 2.9979 \times 10^8$ or $2.9980 \times 10^8$

# Ulp measures the absolute error due to rounding

- If $c = 299{,}792{,}458$ is truncated to a 5-digit base-10 representation, then the absolute error due to rounding is
$$|c - 2.9979 \times 10^8| = 0.2458 \text{ ulp}$$

- If $\pi \models 3.1416$, then
$$|\pi - 3.1416| = |3.14\,159\,265 \cdots - 3.1416| \approx 0.0735 \text{ ulp}$$

# The machine epsilon measures relative error in rounding

- Suppose we represent $x \neq 0$ as the normalized $d + 1$ digit base-$b$ floating-point number

$$\hat{x} = \pm a_0.a_1 a_2 \cdots a_d \times b^e$$

- The absolute error due to rounding is at most 1 ulp
- The relative error due to rounding is at most

$$\frac{|\hat{x} - x|}{|x|} \leq \frac{b^{e-d}}{|x|} \leq \frac{b^{e-d}}{b^e} = b^{-d}$$

  which is 1 ulp in the floating-point representation of 1.0
- $b^{-d}$ is called the machine epsilon

## Example

- Consider 6-digit decimal arithmetic:
  $a_0.a_1a_2a_3a_4a_5 \times 10^e$
- The machine epsilon is $10^{-5}$
- Let $x = 123.4567 \models 1.23457 \times 10^2 = \hat{x}$
- The absolute error is $|x - \hat{x}| = 0.3 \text{ ulp}$
- The relative error is

$$\frac{|x - \hat{x}|}{|x|} = \frac{0.0003}{123.4567} \approx 2.43 \times 10^{-6} = 0.243\epsilon$$

# IEEE floating-point representations

- First standardized in 1985 and updated in 2008
- Single precision numbers are represented with 24-bit precision as

$$1.a_1a_2\cdots a_{23}\times 2^e$$

  where $-126 \le e \le 127$
- Occupies 32 bits (1 sign bit, 23 significand bits, and 8 exponent bits)
- The smallest normalized representable number is $2^{-126}\approx 1.2\times 10^{-38}$
- The largest is $(2-2^{-23})\times 2^{127}\approx 3.4\times 10^{38}$
- The machine epsilon is $2^{-23}\approx 1.2\times 10^{-7}$

## IEEE floating-point representations, 2

- Double precision numbers are represented with 52-bit precision as

$$1.a_1a_2\cdots a_{52}\times 2^e$$

  where $-1022 \le e \le 1023$
- Occupies 64 bits (1 sign bit, 52 significand bits, and 11 exponent bits)
- The smallest normalized representable number is $2^{-1022} \approx 2.225 \times 10^{-308}$
- The largest is $(2 - 2^{-52}) \times 2^{1023} \approx 1.798 \times 10^{308}$
- The machine epsilon is $2^{-52} \approx 2.22 \times 10^{-16}$

# IEEE floating-point representations, 3

- Quadruple precision numbers (1 sign bit, 112 significand bits, 16 exponent bits)
- Three decimal types (32, 64, 128 bits) containing 7, 16, and 34 decimal digits, respectively
- Fortran has various kinds for each binary type (and can support decimal types, but I know of no current implementation)
- C/C++ define float, double, and long double (which is allowed to be the same as double) and some compilers support decimal types

# Example

- Suppose the math library guarantees that the error in $\sin x$ is no more than 1 ulp if $|x| \leq \pi/4$
- What does this imply about the maximum absolute error for double precision (52-bit) IEEE arithmetic?

# Example

- Suppose the math library guarantees that the error in $\sin x$ is no more than 1 ulp if $|x| \leq \pi/4$
- What does this imply about the maximum absolute error for double precision (52-bit) IEEE arithmetic?
- The maximum value of $\sin x$ in this interval is $1/\sqrt{2}$
- A 1 ulp error in $1/\sqrt{2}$ corresponds to an absolute error of

$$\frac{1}{\sqrt{2}} \times 2^{-52} \approx 1.57 \times 10^{-16}$$

# The double extended format

- x86 architectures have an 8-register stack that is 80 bits wide
- Arithmetic in these registers is done in extended format
- The significand has 64 bits: $1.a_1 a_2 \cdots a_{63} \times 2^e$
- $-16{,}382 \leq e \leq 16{,}383$
- The extra precision mitigates roundoff error in exponentiation and similar operations
- Major difficulty: You don't have control over the compiler's choice of 64- or 80-bit registers
- Fortran "real(10)" and C/C++ long double types are not portable

# Correctly rounded arithmetic

- The arithmetic operations $+$, $-$, $\times$, $\div$, and $\sqrt{\ }$ are correctly or exactly rounded if each is performed as if infinite precision were available, followed by rounding
- IEEE arithmetic is correctly rounded
- Suppose $\cdot$ represents the "exact" value of $+$, $-$, $\times$, or $\div$ and $\odot$ represents the correctly rounded floating-point operation
- Theorem: (Goldberg) If $x \odot y$ is correctly rounded, then there is a real number $\delta$ such that

$$x \odot y = (x \cdot y)(1 + \delta),$$

where $|\delta| \leq \epsilon$ (when rounding to nearest, $|\delta| \leq \epsilon/2$)

# What can happen when arithmetic is not correctly rounded?

- The first generation of IBM System/360 computers used 6-digit chopped hexadecimal arithmetic for single precision: $x \models .a_1a_2a_3a_4a_5a_6 \times 16^e$, where $1 \le a_1 \le 15$ for nonzero $x$
- $x \otimes 1.0 =$?

## What can happen when arithmetic is not correctly rounded?

- The first generation of IBM System/360 computers used 6-digit chopped hexadecimal arithmetic for single precision: $x \models .a_1a_2a_3a_4a_5a_6 \times 16^e$, where $1 \le a_1 \le 15$ for nonzero $x$
- $x \otimes 1.0 = ?$

$$\begin{aligned} & (.a_1a_2a_3a_4a_5a_6 \times 16^e) \otimes (.100000 \times 16^1) \\ = \ & .0a_1a_2a_3a_4a_5a_6 \times 16^{e+1} \\ \models \ & .0a_1a_2a_3a_4a_5 \times 16^{e+1} \qquad \textit{there are only 6 digits!} \\ = \ & .a_1a_2a_3a_4a_50 \times 16^e \qquad \textit{after normalization} \end{aligned}$$

- $x \otimes 1.0 \neq x$ in general!

# What can happen when arithmetic is not correctly rounded, 2

- This kind of behavior makes it difficult to establish the correctness and reliability of floating-point programs
- IBM replaced the arithmetic unit of every installed computer with one that had an extra guard digit for intermediate results
- Without guard digits (or correct rounding), especially large errors are possible in floating-point addition and subtraction

# Example (6-digit decimal arithmetic)

- Consider the floating-point result of $1 - 0.999999$
- The correctly rounded result is $10^{-6}$
- Here's what happens in 6-digit chopped decimal arithmetic:

$$\begin{array}{rrl} & 1.00000 & \textit{The larger operand has 6 significant digits} \\ \ominus & .99999 & \textit{the sixth digit is shifted off} \\ \hline & 0.00001 & = 1.00000 \times 10^{-5} \end{array}$$

- The relative error in the result is 900 percent

## Caveat

- Goldberg's theorem applies only to individual floating-point operations
- A sequence of floating-point operations can expose the effects of previous roundings
- Example: Suppose $x = 123.456$ and $y = 0.000\,789$.
- Correct rounding gives $z = x \oplus y \models 123.457$, with a relative error of about $1.71 \times 10^{-6}$
- The subsequent operation $z \ominus x = 0.001$
- The relative error in approximating $(x + y) - x$ by $(x \oplus y) \ominus x$ is $27$ percent or $27{,}000\epsilon$

## Binary/decimal conversions

- How many digits should you print so that you get exactly the same binary floating-point value that you started with?
- Print IEEE single-precision values to 9 significant digits
- Print IEEE double-precision values to 17 significant digits
- Example: $\pi = 3.14159265$ (single) and $\pi = 3.1415926535897932$ (double)

# Range and scaling

- Consider decimal arithmetic
- The exponent range is limited: say $-99 \le e \le 99$
- Consider $\sqrt{x^2 + y^2}$ when $x = 10^{50}$ and $y = 10^{51}$
- Naive evaluation overflows: $x^2$ and $y^2 > 10^{99}$
- Instead evaluate as $10^{51}\sqrt{0.1^2 + 1^2}$
- Careless squaring can waste half the exponent range!
- Similar issues in binary arithmetic (but the double-extended format helps avoid them)

# The exact laws of correctly rounded arithmetic

- Identity axioms: $1 \cdot x = x$, $0 + x = x$
- Commutative axioms: $x \cdot y = y \cdot x$, $x + y = y + x$
- Inverse axioms: $x/x = 1$, $x + (-x) = 0$
- You can count on these identities in correctly rounded arithmetic (but not necessarily otherwise!)

## The approximate laws of correctly rounded arithmetic

- Associative axioms: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$, $x + (y + z) = (x + y) + z$
- Distributive axioms: $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
- Note: $x \cdot x^{-1}$ need not equal $1$

# Underflow

- Single precision IEEE arithmetic:
  $x = \pm 1.a_1 a_2 \cdots a_{23} \times 2^e$, where $-126 \leq e \leq 127$
- If $|x| > 2^{126}$, then $x^{-1}$ underflows
- IEEE provides for denormalized numbers of the form $0.a_1 a_2 \cdots a_{23} \times 2^{-126}$ where some of the $a_i$'s are $0$
- However, many processors simply flush to zero on underflow because that is faster and easier to implement in hardware
- The premise that underflowed values are good approximations to $0$ is not always a good one!

# Example: Scaling for the Euclidean norm

- $\|\mathbf{x}\|_\infty = \left( \sum_{i=1}^{n} x_i^2 \right)^{1/2}$
- In Fortran, we can scale the calculation as:

```
c = maxval(abs(x))
x = x/c
xnorm = c*sqrt(sum(x**2))
```

$\max_{1 \le i \le n} |x_i|$ (for `c = maxval(abs(x))`)

*elementwise division* (for `x = x/c`)

- What can go wrong?

# Example: Scaling for the Euclidean norm, 2

- Optimizing compilers replace x/c with $x * c^{-1}$
- If c is sufficiently large, then $c^{-1}$ can underflow
- Flush-to-zero replaces $c^{-1}$ with 0
- So a sufficiently large vector x is replaced with the zero vector after normalization!
- Advice: Avoid –fast and similar compiler flags without good reason, as they usually enable flush-to-zero and multiplication by reciprocal

## Catastrophic cancelation

- Example: Consider the solutions of $0.202x^2 + 10x + 0.101 = 0$
- In 6-digit decimal arithmetic, $\hat{a} = 2.02\,000 \times 10^{-1}$, $\hat{b} = 1.00\,000 \times 10^{1}$, and $\hat{c} = 1.01\,000 \times 10^{-1}$.
- Assuming correct rounding,

$$\begin{aligned} \hat{D} = \hat{b} \otimes \hat{b} \ominus (4 \otimes \hat{a} \otimes \hat{c}) \quad & \models \quad 9.99\,184 \times 10^{1} \\ \sqrt[\circ]{9.99\,184 \times 10^{1}} \quad & \models \quad 9.99\,592 \times 10^{0} \\ -\hat{b} \oplus \hat{D} \quad & \models \quad -4.08\,000 \times 10^{-3} \end{aligned}$$

# Catastrophic cancelation, 2

- The "true" value of $-b + D$ is approximately $-4.08\,123 \times 10^{-3}$
- The relative error in $-\hat{b} \oplus \hat{D}$ is about $60\epsilon$
- Usual quadratic formula:

$$r_\pm = \frac{-b \pm D}{2a}, \qquad D = \sqrt{b^2 - 4ac}$$

- A better formula when $|b| \approx D$:

$$r_\mp = \frac{2c}{-b \pm D}$$

# Catastrophic cancelation, 3

- The computed value of $r_+$ from the usual quadratic formula is is $-1.00\,990 \times 10^{-1}$
- The "true" root is $-1.01\,020\,6 \times 10^{-1}$
- The modified quadratic formula gives $r_+ = -1.01\,020 \times 10^{-2}$
- So the relative error is about $1.2\epsilon$—or 50 times smaller than before!

# Computational challenges

- Finite-difference methods for PDEs routinely subtract nearly equal quantities:

$$\frac{\partial u}{\partial x} \approx \frac{u_{k+1} - u_k}{\Delta x}$$

- The derivatives are numerically noisy
- Although $\Delta x \to 0$ may approach the correct solution asymptotically in real arithmetic, such is not the case in floating-point arithmetic!

# Floating-point exceptions in IEEE arithmetic

- Invalid operation: Yields NaN
- Division by 0: Yields Inf or NaN
- Overflow: Yields ±Inf
- Underflow: Yields denormalized numbers or 0, depending on hardware
- Inexact: Rounding has occurred

# Examples

- $\infty/\infty$, $\infty \times 0$, and $\infty - \infty$ yield NaN
- $\pm 1/\infty$ yields $\pm 0$
- $-0 = 0$ (by definition)
- Comparisons like x <NaN and x==NaN are always false (even if x is NaN!)
- $\mathsf{NaN} \pm \mathsf{x} = \mathsf{NaN}$, $\mathsf{NaN} \times \mathsf{x} = \mathsf{NaN}$, $\mathsf{NaN}/\mathsf{x} = \mathsf{NaN}$, $\sqrt{\mathsf{NaN}} = \mathsf{NaN}$, where x is any finite floating-point number

# Selected resources

- IEEE 754-2008 standard document (through lib.asu.edu)
- David Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys* **28** (1991)
- Michael L. Overton, *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, 2001
- Nelson H. F. Beebe, *The Mathematical-Function Computation Handbook*, Springer, 2017