CS370 – Floating Point Error

Announcement: (Optional) MATLAB Tutorial

The tutorial will be offered by Ke Nian, one of your friendly neighbourhood TA’s.

Date: Tuesday, May 10

Time: 6-7pm

Location: MC 1056

Why the name “floating point”?

Alternative systems exist where the number of digits after the decimal (or radix) point is fixed – “fixed point” numbers.

e.g., 3 digits after the decimal: $10.234, 171.001, 0.010$.

Basically an integer representation, scaled by a *fixed* factor.

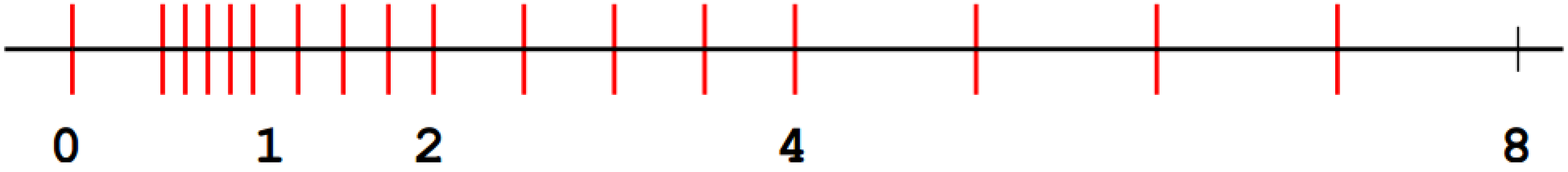
e.g., $10234 \times 10^{-3}, 171001 \times 10^{-3}, 10 \times 10^{-3}$, where 10^{-3} is the scale factor.

Floating point number systems let the radix point “float”, in order to represent a wider range.

Floating Point Density

Unlike fixed point, floating point numbers are *not evenly spaced*!

E.g., For $F = \{\beta = 2, t = 3, L = -1, U = 2\}$ the positive values are spaced like:



Measuring Error

Our algorithms will compute *approximate* solutions to problems.

Difference between…

x_{exact} = true solution, and

x_{approx} = approximate solution,

…gives the *error* of an algorithm.

We distinguish between *absolute error* and *relative error.*

Absolute *v.s.* Relative Error

Sometimes we also consider the signed error, i.e., without abs. values here.

Absolute error: $E_{abs} = \left| x_{exact} - x_{approx} \right|$

Relative error: $E_{rel} = \frac{\left| x_{exact} - x_{approx} \right|}{\left| x_{exact} \right|}$

e.g., Let $x_{exact} = 220, x_{approx} = 198$.

Compute E_{abs} and E_{rel}. What does each tell us?

Relative Error and Significant Digits

Relative error is usually more useful. It…

- is independent of the magnitudes of the numbers involved.
- relates to the number of significant digits in the result.

We say a result is correct to *approximately* s digits if $E_{rel} \approx 10^{-s}$ or…

$$0.5 \times 10^{-s} \leq E_{rel} < 5 \times 10^{-s}$$

e.g., If $E_{rel} \approx 0.8 \times 10^{-3}$, it describes a result correct to approximately how many significant digits?

What about $E_{rel} \approx 4 \times 10^{2}$?

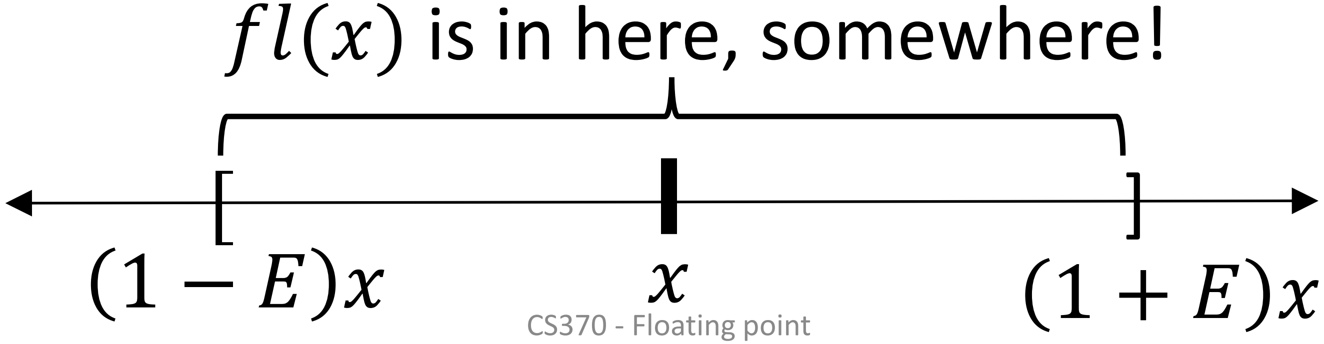
Floating point, F v.s. Reals, $\mathbb{R}$.

We saw that they can behave differently!

But how much can they differ?

For a given FP system F, rel. err. between any $x \in \mathbb{R}$, and its nearest FP approximation, $fl(x)$, has an upper bound, E, such that:

$$(1-E)|x| \leq |fl(x)| \leq (1+E)|x|$$



Machine Epsilon

This maximum relative error, E, for a FP system is called *machine epsilon* or *unit round-off error.*

It is defined as the smallest value such that $fl(1 + E) > 1$ under the given floating point system.

Machine Epsilon

Note! δ may be positive *or* negative.

Hence we have the rule $fl(x) = x(1 + \delta)$ for some $|\delta| \leq E$.

For an FP system $F = \{\beta, t, L, U\}$ with rounding to nearest: $E = \frac{1}{2}\beta^{1-t}$.

For other rounding modes (e.g., chopping/truncation): $E = \beta^{1-t}$.

Example: What is *E* for $F = \{\beta = 10, t = 3, L = -5, U = 5\}$?

We will often care only about the *order of magnitude* of E.

Arithmetic with Floating Point

What guarantees exist on the result of a FP *arithmetic operation*?

IEEE standard *requires* that for $w, z \in F$,

$$w \oplus z = fl(w+z) = (w+z)(1+\delta).$$

$\oplus$ denotes floating point addition.

i.e., the result is the closest **in $\boldsymbol{F}$** to the exact *real* result (unless over/underflows occurs.)

FP implementations typically use extra "guard" digits (behind the scenes) to ensure this.

Floating Point Arithmetic

Warning: This rule only applies to individual FP operations!

So: It is ***not*** generally true that

$$(a \oplus b) \oplus c = a \oplus (b \oplus c) = fl(a + b + c).$$

Result is *order-dependent; associativity is broken*!

Similarly, be careful testing for exact equality.

Nonetheless, FP can still be (very!) useful *if* we understand its behaviour.

Round-Off Error Analysis

What *can* we say about $(a \oplus b) \oplus c$?

Let's find its relative error, to see how FP error propagates…

Round-Off Error Analysis

After two additions, error bounds are already complicated.

And many *useful* computations take $O(10^m)$ arithmetic operations, for $m = 2, 3, 4, 5 \dots$

This analysis describes only the worst case error magnification, as a function of the input data.

Actual error in result *could* be (much) less.

Error Bounds and Condition Number

$$E_{rel} \le \frac{|a+b|}{|a+b+c|}(\mathrm{E}+\mathrm{E}^2)+\mathrm{E}$$

Weakening slightly to get a symmetric expression:

$$E_{rel} \le \frac{|a|+|b|+|c|}{|a+b+c|}(2E+E^2)$$

The term $\frac{|a|+|b|+|c|}{|a+b+c|}$ …

- describes how the error *E* is magnified along the way.
- may be called a *condition number* for this calculation.

When is it large? When is it small?

Cancellation Errors

Worst case magnification is when $|a + b + c| \ll |a| + |b| + |c|$.

i.e., quantities have differing signs, and ***cancellation*** occurs.

e.g. $a = 2000, b = -3.234, c = -2000$ for $F \ = \{10, 4, -10, 10\}$.

Why is this case a problem? Let's consider:

1. the computed and true results for $(a \oplus b) \oplus c$.
2. the condition number.

Catastrophic Cancellation Error

Catastrophic cancellation error arises when subtracting numbers of the same magnitude, *and* (at least one of) those numbers contains error.

e.g.,

$$\underbrace{173.00}_{\substack{\text{Correct}\\\text{digits}}}\underbrace{063}_{\substack{\text{Erroneous}\\\text{digits}}} - \underbrace{173.00}_{\substack{\text{Correct}\\\text{digits}}}\underbrace{017}_{\substack{\text{Erroneous}\\\text{digits}}} = 0.00\underbrace{046}_{\substack{\text{Erroneous}\\\text{digits}}}$$

All of the *significant* digits cancelled; the result might have no correct digits whatsoever!

Benign Cancellation

However, if the input quantities are known to be *exact*, we have our usual guarantee on floating point ops (addition, subtraction, etc.):

$$w \oplus z = fl(w + z) = (w + z)(1 + \delta).$$

Round-off Errors We've Seen

Adding large and small numbers (very different magnitudes).

- Smaller digits get lost or "swamped"!
- Rule of thumb: Try to sum smaller numbers first to reduce swamping.

Subtracting nearby numbers *that contain error.*

- Known as catastrophic cancellation.
- Rule of thumb: Try to reformulate computations to avoid cancellation.

Taylor series example, revisited

So what's the main reason we observed that

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

performs so much worse than

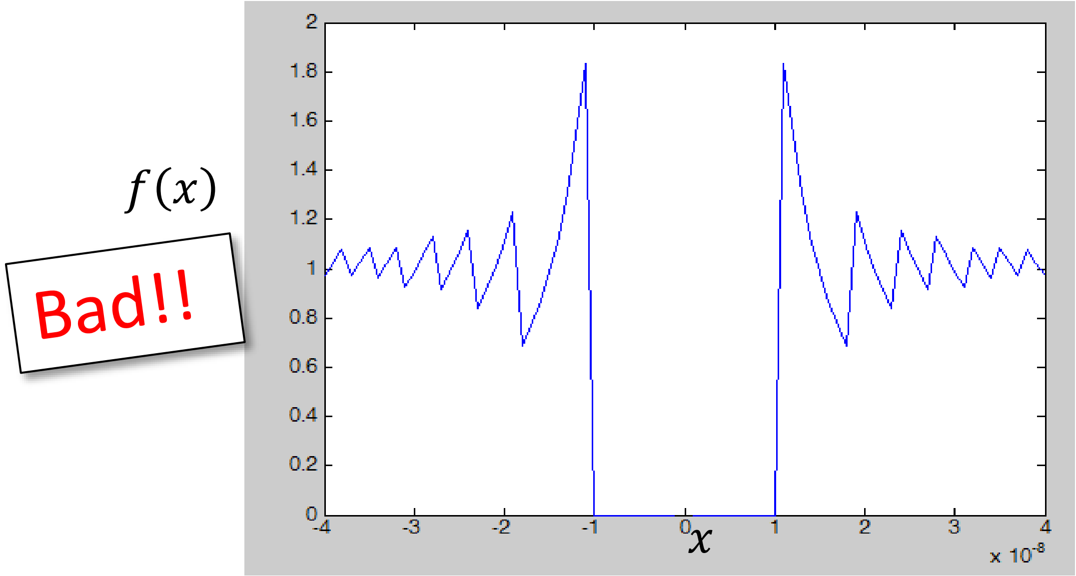
$$e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots}$$

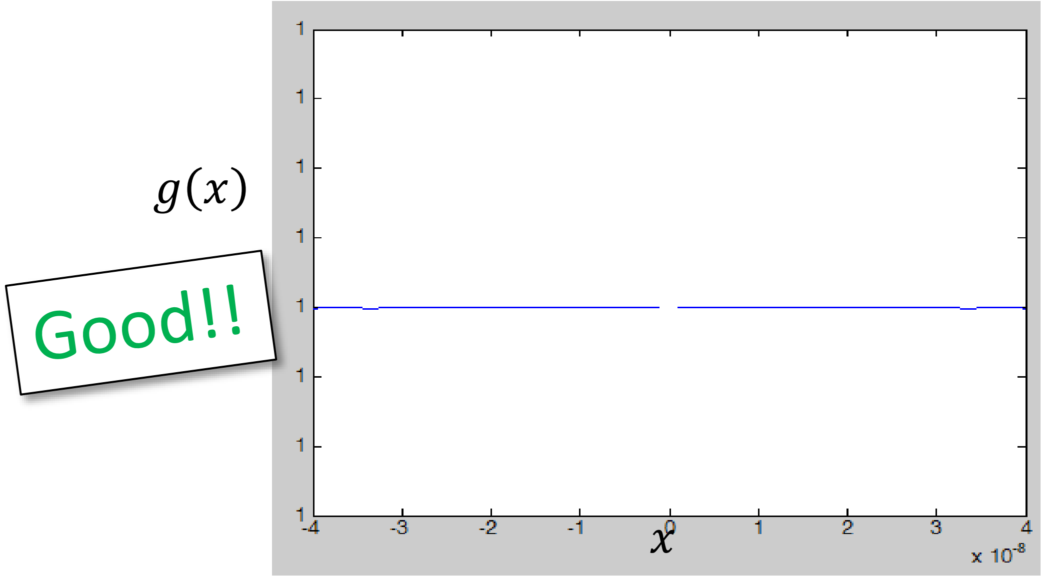
as an algorithm to evaluate $e^{-5.5}$?

Latter *avoids alternating signs* in the sums which lead to cancellation.

A Visualized Example of Cancellation Error

Compare: $f(x) = \frac{1-\cos^2 x}{x^2}$ and $g(x) = \frac{\sin^2 x}{x^2}$ near $x = 0$. True solution approaches 1 as $x \to 0$.





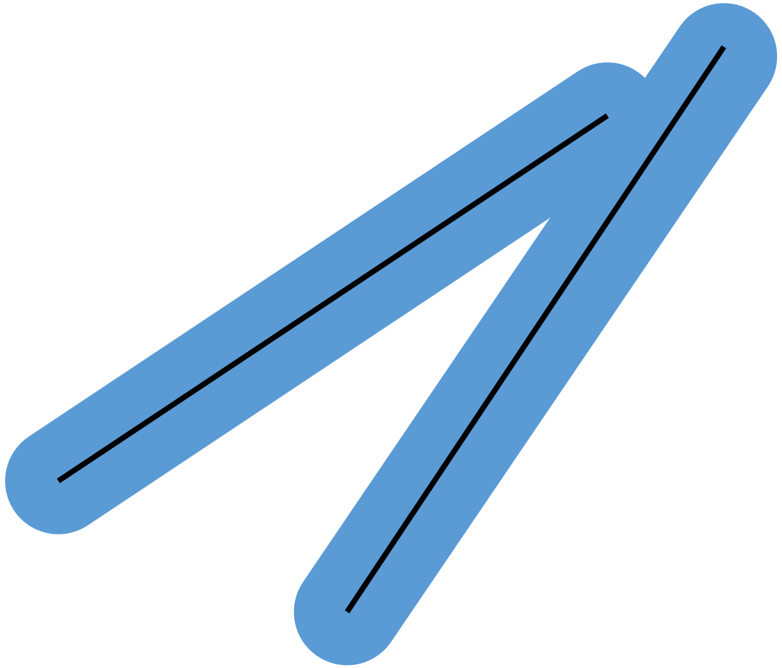
MATLAB plot with IEEE double precision (i.e., $\approx 15 - 17$ decimal digits).

FP Error: A Geometric Example

Do two line segments intersect?

A useful ***robust*** test for this must consider round-off error!

Both false positives or false negatives can occur.



"Epsilon geometry"

Sources of Error

When solving a problem computationally, there are many possible error sources:

- Round-off error – due to FP representation and arithmetic.
- Truncation error – eg. truncating a Taylor series after n terms.
- Uncertainty/error in the input itself (e.g. from measurements).
- Error/approximation in our mathematical model of the “real” problem.

We will (continue to) focus largely on the first two.

Conditioning of Problems

A problem may be called *ill-conditioned* or *well-conditioned.*

For problem P, with input I and output O, if a change to the input, ΔI, gives a "small" change in the output, it is *well-conditioned*.

Otherwise, P is *ill-conditioned*.

This is a property of the problem, *independent* of any specific implementation (algorithm or hardware).

Conditioning of *Problems*

So for a problem or function f, where

$$y = f(x)$$

f is …

- well-conditioned if $\mathrm{f}(x + \Delta x)$ for "small" Δx is "close" to y.
- Ill-conditioned if $\mathrm{f}(x + \Delta x)$ for small Δx is far from y.

So if some error perturbs your input, how badly does it affect the output?

Conditioning is relative; there's a "sliding scale".

Condition number for a function

Condition number (in general) for a function f is a number such that

$$E_{rel_{out}} \approx C_f E_{rel_{in}}.$$

For differentiable f at point x_0 , it is $\frac{x_0 f'}{f}$. Let's see why…

$$C_f = \frac{E_{rel_{out}}}{E_{rel_{in}}} = \frac{\dfrac{f(x_0+\delta) - f(x_0)}{f(x_0)}}{\dfrac{(x_0+\delta) - x_0}{x_0}} = \frac{x_0}{f(x_0)} \cdot \frac{f(x_0+\delta) - f(x_0)}{\delta} \approx \frac{x_0 f'(x_0)}{f(x_0)}$$

Stability of an *Algorithm*

If any initial error in the data is magnified by the algorithm, the algorithm is considered numerically *unstable*.

Can lead to meaningless results, for *seemingly* reasonable methods.

Conditioning v.s. Stability

Conditioning of a problem:

- How sensitive is the *problem* itself to errors/changes in input?

Stability of an algorithm or numerical process:

- How sensitive is the *algorithm* to errors/changes in input?

Observations:

1. An *algorithm* can be unstable even for a well-conditioned problem!
2. An ill-conditioned problem limits how well we can expect an algorithm to perform.

Stability Analysis of an *Algorithm*

Consider the integration problem

$$I_n = \int_0^1 \frac{x^n}{x+\alpha}\,dx$$

for a given n where α is some fixed parameter.

The course notes gives us a recursive algorithm to solve it, for $n \geq 0$:

$$I_0 = \log\frac{1+\alpha}{\alpha}, \qquad I_n = \frac{1}{n} - \alpha I_{n-1}.$$

Stability Analysis of an *Algorithm*

A stability analysis considers how some *initial* error in I_0, say ϵ_0, propagates and magnifies through our algorithm.

(…ignoring other numerical error added along the way.)

Let $(I_n)_E$ indicate exact solution, $(I_n)_A$ the computed numerical solution.

Assume initial error is $\epsilon_0 = (I_0)_A - (I_0)_E$

Stability Analysis of an *Algorithm*

Exact solution $(I_n)_E$ follows $(I_n)_E = \frac{1}{n} - \alpha(I_{n-1})_E$.

Approximate solution $(I_n)_A$ follows $(I_n)_A = \frac{1}{n} - \alpha(I_{n-1})_A$.

What is $\epsilon_n = (I_n)_A - (I_n)_E$, error after n steps?

Stability Analysis of an *Algorithm*

Plugging in…

$$\epsilon_n = (I_n)_A - (I_n)_E$$

$$= \left(\frac{1}{n} - \frac{1}{n}\right) - \alpha\big((I_{n-1})_A - (I_{n-1})_E\big)$$

$$= (-\alpha)\epsilon_{n-1} = (-\alpha)(-\alpha)\epsilon_{n-2} = (-\alpha)(-\alpha)(-\alpha)\epsilon_{n-3} \ldots$$

$$= (-\alpha)^n \epsilon_0$$

Stability Analysis of an *Algorithm*

So the initial error ϵ_0 is scaled to become $\epsilon_n = (-\alpha)^n \epsilon_0$.

What does this tell us about error magnification for different α?

Two possibilities:

1. $|\alpha| < 1$: Initial error is scaled down over time. Stable!
2. $|\alpha| > 1$: Initial error is continually magnified! Unstable!

Stability Analysis of an *Algorithm*

Hence that recurrence algorithm is *unstable* for solving $I_n = \int_0^1 \frac{x^n}{x+\alpha} dx$ for values of $|\alpha| > 1$.

This is borne out if you code up this recurrence in C / Matlab / etc.:

$$I_{100} = 6.64 \times 10^{-3} \text{ for } \alpha = 0.5$$ Correct!

$$I_{100} = 2.1 \times 10^{22} \text{ for } \alpha = 2.0$$ Wrong!

Summary of FP

- F is not $\mathbb{R}$!
- We can analyze the error propagation of FP arithmetic to bound error growth.
- We can analyze whether some initial error will grow or shrink to determine the stability of algorithms.

Further reading on FP [Optional]

"What Every Computer Scientist Should Know About Floating-Point Arithmetic", by David Goldberg, 1991.

Appendix D

What Every Computer Scientist Should Know About Floating-Point Arithmetic

Note – This appendix is an edited reprint of the paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys. Copyright 1991, Association for Computing Machinery, Inc., reprinted by permission.

Abstract

Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. This paper presents a tutorial on those aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with numerous examples of how computer builders can better support floating-point.

Categories and Subject Descriptors: (Primary) C.0 [Computer Systems Organization]: General -- *instruction set design*; D.3.4 [Programming Languages]: Processors -- *compilers, optimization*; G.1.0 [Numerical Analysis]: General -- *computer arithmetic, error analysis, numerical algorithms* (Secondary)

D.2.1 [Software Engineering]: Requirements/Specifications -- *languages*; D.3.4 Programming Languages]: Formal Definitions and Theory -- *semantics*; D.4.1 Operating Systems]: Process Management -- *synchronization*.

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Denormalized number, exception, floating-point, floating-point standard, gradual underflow, guard digit, NaN, overflow, relative error, rounding error, rounding mode, ulp, underflow.

An Aside on Adding Many Numbers [Optional]

How can we add a sequence of many numbers with least error?

One can imagine several possible strategies:

- Sum in order of increasing magnitude?
- Sum +/- pairs to maximize cancellation, stay close to zero?
- Sum all positive and all negative separately, and combine at the end to minimize cancellation?

Adding Many Numbers [Optional]

The answer to the *best* ordering is fairly subtle. See…

"The accuracy of floating point summation" – Higham, 1993.

http://epubs.siam.org/doi/abs/10.1137/0914050

Gives the extension of our 3-number analysis to n numbers, and deeper analytical and experimental comparisons of several schemes.

1989 Turing award winner!

Better algorithms exist (e.g., pairwise summation, Kahan summation) beyond simply re-ordering the sequence.

http://www.phys.uconn.edu/~rozman/Courses/P2200_11F/downloads/sum-howto.pdf

Historical Side Note: William Kahan [Optional]

Turing Award Citation: *For his fundamental contributions to numerical analysis. One of the foremost experts on floating-point computations. Kahan has dedicated himself to "making the world safe for numerical computations"!*

- Known as the “Father of Floating Point”.
- Major contributor to the IEEE-754 floating point standards.
- Born in Canada, grew up around Toronto.

Condition Number Example #1

$a = 2000$
$b = -3.234$
$c = -2000$
$F \ = \{10, 4, -10, 10\}$

True: -3.234 Computed: -3.000

Only 1 correct digit!

E^2 term is small, so we drop it.

$$E_{rel} \leq \frac{|a|+|b|+|c|}{|a+b+c|}(2\mathrm{E} + \mathrm{E}^2) \approx \frac{4003.234}{3.234} \cdot 2\left(\frac{1}{2}10^{-3}\right) \approx 1.238.$$

This is quite large.

Condition number

Condition Number Example #2

$a = 2000$
$b = -3.234$
$c = -2000$
$F = \{10, 4, -10, 10\}$

What is $(a \oplus c) \oplus b$?

-3.234. This ordering gives a much better result!

Observation: We can sometimes improve our algorithms by re-ordering operations.

What is the condition number?

It is the *same as before.*

Observation: The condition number only gives a *worst-case* bound.

Who are you?

1. What area(s) of CS are you most interested in?
2. What aspect/topic of the course are you most interested in?
3. What are your career plans/hopes for after graduation?

Condition Number Example #3

$F = \{10, 4, -10, 10\}$

What about $(a \oplus b) \oplus c$ for $a = -2000, b = -3.234, c = -2000$?
(i.e. all the signs are the same).

True: -4003 Computed: -4003

Condition number: $\frac{4003.234}{4003.234} = 1$

Relative error: $E_{rel} \leq (2E + E^2) \approx 2E = 10^{-3}$

When no cancellation occurs, the bound is much stronger.