Floating-Point Math and Accuracy

Dr. Axel Kohlmeyer

Senior Scientific Computing Expert
International Centre for Theoretical Physics
Trieste, Italy

Associate Dean for Scientific Computing,
College of Science and Technology
Temple University, Philadelphia, USA

http://sites.google.com/site/akohlmey/

ICTP
The Abdus Salam
International Centre
for Theoretical Physics

Errors in Scientific Computing

- Before computations:
  - Modeling: neglecting certain properties
  - Empirical data: not every input is known perfectly
  - Previous computations: data may be taken from other (error-prone) numerical methods
  - Sloppy programming (e.g. inconsistent conversions)
- During computations:
  - Truncation: a numerical method approximates a continuous solution
  - Rounding: computers offer only finite precision in representing real numbers

Example

- Computing the surface of the earth using

$$A = 4\pi r^2$$

- This involves several approximations:
  - Modeling: the earth is not exactly a sphere
  - Measurement: earth's radius is an empirical number
  - Truncation: the value of π is truncated
  - Rounding: all numbers used are rounded due to arithmetic operations in the computer
- Total error is the sum of all errors, but one of them is often the dominant error

Representing Numbers (1)

- Real numbers have unlimited accuracy
- Yet computers “think” digital, i.e. in integer math
  => only a fixed **range** of numbers can be
  represented by a fixed number of bits
  => **distance** between two integers is 1
- We can reduce the distance through fractions
  (= fixed point), but that also reduces the range

	16-bit	32-bit	64-bit	28-bit / 4-bit	22-bit / 10-bit
Min.	-32768	-2147483648	~ $-9.2233 * 10^{-18}$	-16777216.0000	-2048.000000
Max.	32767	2147483647	~ $9.2233 * 10^{-18}$	16777215.9375	~ 2047.999023
Dist.	1	1	1	0.0635	0.0009765625

Representing Numbers (2)

- Need a way to represent a wider range of numbers with a same number of bits
- Need a way to represent numbers with a reasonable amount of precision (distance)
- Same relative precision often sufficient:

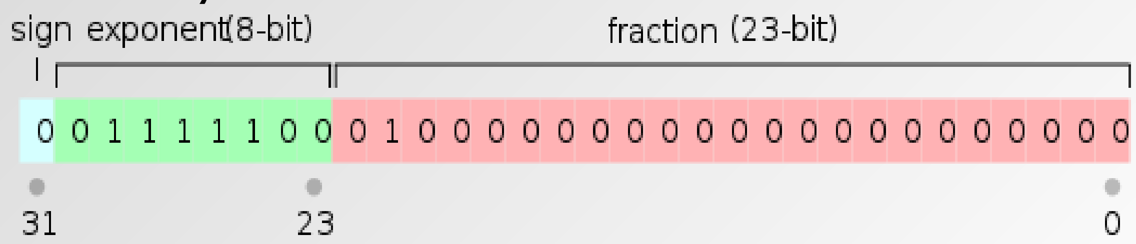
  => Scientific notation:

  $+/-(\text{mantissa}) * (\text{base})^{+/-(\text{exponent})}$

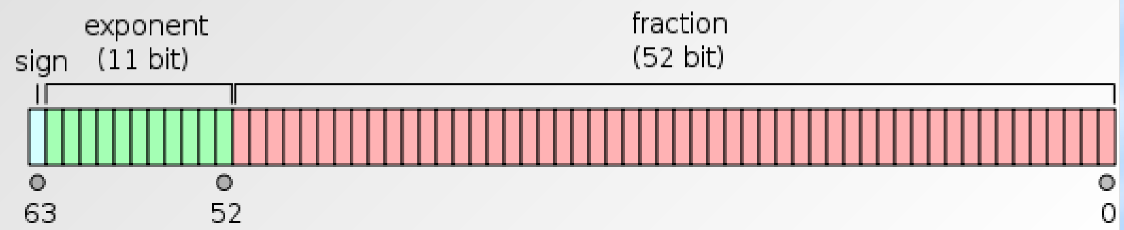
  Mantissa -> integer fraction
  Base -> 2
  Exponent -> a small integer

IEEE 754 Floating-point Numbers

- The IEEE 754 standard defines: storage format, result of operations, special values (infinity, overflow, invalid number), error handling
=> portability of compute kernels ensured
- Numbers are defined as bit patterns with a sign bit, an exponential field, and a fraction field
  - Single precision:
    8-bit exponent
    23-bit fraction



  - Double precision:
    11-bit exponent
    52-bit fraction



Values of Floating-Point Numbers

- Value: $(1 – (\text{mantissa})/(2^{(\text{fraction bits})}) * 2^{(\text{exponent-bias})}$
  $1.0 \leq (\text{mantissa}) < 2.0$, $(\text{exponent}) \geq 0$
- Special case: 0.0 is all bits set to zero
  Special case: -0.0 is like 0.0 but sign bit is set
  More special cases: Inf, -Inf, NaN, -NaN
- Single precision: $\sim\pm 1.2*10^{-38} < x < \sim\pm 3.4*10^{38}$
  actual precision: ~7 decimal digits
- Double precision: $\sim\pm 2.2*10^{-308} < x < \sim\pm 1.8*10^{308}$
  actual precision: ~15 decimal digits

Density of Floating-point Numbers

- How can we represent so many more numbers in floating point than in integer? ***We don't!***
- The number of unique bit patterns has to be the same as with integers of the same bitness
- There are 8,388,607 single precision numbers in $1.0 < x < 2.0$, but only 8191 in $1023.0 < x < 1024.0$
- => absolute precision depends on the magnitude
- => some numbers are not represented exactly
=> approximated using rounding mode (nearest)

Math with Floating Point Numbers

Addition:

- Right bitshift mantissa and increment exponent of smaller number until both exponents are the same
- Add mantissa of both numbers and bitshift until mantissa is between 1.0 and 2.0 again
- Only if both numbers have the same sign and the same exponent precision is preserved

Multiplication:

- Add exponents and multiply mantissa of both numbers
- Bitshift mantissa until its value is between 1.0 and 2.0
- No loss of precision; error is larger error of either number

Floating-Point Math Pitfalls

- Floating point math is commutative,
  but <u>not associative</u>! Example (single precision):
  $1.0 + (1.5*10^{38} + (-1.5*10^{38})) = 1.0$
  $(1.0 + 1.5*10^{38}) + (-1.5*10^{38}) = 0.0$
- => the result of a summation depends on the order of how the numbers are summed up
- => results may change significantly, if a compiler changes the order of operations for optimization
- => prefer adding numbers of same magnitude
  => avoid subtracting very similar numbers

The Abdus Salam International Centre for Theoretical Physics

How To Reduce Errors

- Use double precision unless you can be sure of error cancellation or using an imprecise model => collides with vectorization and GPU/MIC
- When summing numbers of different magnitude
  - Sort first and sum in ascending order
  - Sum in blocks (pairs) and then sum the sums
  - Use integer fraction, if range and precision allow it
- NOTE: summing numbers in parallel may give different results depending on parallelization

Floating Point Comparison

- Floating-point results are usually **inexact**
  => comparing for equality is dangerous
  Example: don't use a floating point number for controlling a loop count. Integers are made for it
- It is OK to use exact comparison:
  - When results have to be bitwise identical
  - To prevent division by zero errors
- => compare against expected absolute error
- => don't expect higher accuracy than possible

The Abdus Salam
International Centre
for Theoretical Physics

Floating Point vs. Math Library

- libm is part of standard C, thus it is ubiquitous
- Provides a large variety of mathematical functions / operations on floating-point numbers but not many alternatives for x86/x86_64 exist
- Focus is typically put on standard compliance
- The x86 floating point unit contains most of the functionality internally, but most as firmware; SSE and AVX do not provide these
- The x86 FPU log() is slower than GNU libm

Test Examples (1)

- **`inverse`**: computes $y=1/x$ and $z=x*y$ and checks if the result is exactly 1.0.
  Compare compilation using `gfortran -O2` and `gfortran -O2 -ffast-math`
- **`loop`**: advance x from 0.0 to 1.0 in increments of 0.01. Compare looping over integer and real
- **`epsilon`**: determine the floating-point precision through searching for the largest epsilon for which 1.0 + ε == 1.0. Start with ε = 1.0 and repeatedly dividing by 2.0

The Abdus Salam International Centre for Theoretical Physics

Test Examples (2)

- **`sum_number`**: compare summing accuracy depending on ascending or descending order. Find the smallest N where the sums differ
- **`paranoia`**: IEEE-754 compliance test
=> use `make` to compile with different compiler flags for optimization and math accuracy
- **`mathopt`**: compute windowed average with a two and three numbers wide window.
=> speed of division by 2 vs division by 3
=> impact of compiler flags vs. code rewrite