

**Advanced School in
High Performance
and GRID Computing:
concepts and applications**



Floating Point Numbers
Stefano Cozzini

Democrito and SISSA/eLAB - Trieste

Outline

- Errors in scientific computing
- How to represent numbers on a computers.
 - IEEE standard floating point formats
 - Floating point arithmetic
- Floating Point Issues
 - Summing FP numbers
 - Comparing FP numbers
 - Subtracting FP numbers
- Role of the compilers in this game
- Some final tips to avoid (known) problems

Scientific Computing

- Traditionally called *numerical analysis*
- Concerned with design and analysis of algorithms for solving mathematical problems that arise in computational sciences
- Distinguish features:
 - concerned with variables that are continuous rather than discrete
 - concerned with *approximations* and their effects
- Approximations are used not just by choice: they are inevitable in most problems

Source of approximations

- Before computation begins:
 - **Modeling:** neglecting certain physical features
 - **empirical measurements:** can't always measure input data to the desired precision
 - **previous computations:** input data may be produced from error-prone numerical methods
- During computation:
 - **truncation:** numerical method approximate a continuous entity
 - **rounding:** computers offer only finite precision in representing real numbers

Example: Approximations

- Computing surface area of Earth using formula

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

- This involves several approximations:
 - **Modeling:** Earth is considered as a sphere...
 - **Measurements:** value of radius is based on empirical methods
 - **Truncation:** value for π is truncated at a finite number..
 - **Rounding:** values for input data and results of arithmetic operations are rounded in computer.

Computational errors

- Truncation Error:
 - errors arising from simplifying mathematics to solve problems on computers:
- Rounding Error:
 - Errors we discuss later..
- NOTE: computational error is sum of truncation error and rounding error, but one of these usually dominates (see later)

Rounding error

- Difference between result produced by a given algorithm using exact arithmetic and result produced by the same algorithm using rounded arithmetic Numbers are stored using a finite number of bits.
- **Due to the inexact representation** of real numbers and arithmetic operations upon them
- To understand where and how they turn out we need to know **how computers deals with numbers..**

Reality = real numbers

- Real number = unlimited accuracy
- How can we store a number on a computer ?
 - Binary coded decimal (BCD)
 - Rational Numbers
 - Fixed Point
 - **Floating point**

Floating-point representation

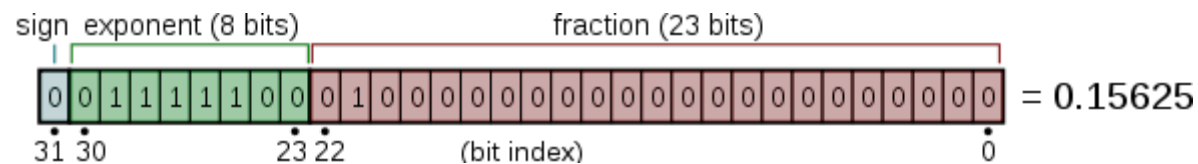
- floating numbers are stored using a kind of scientific notation.

$$\pm \text{mantissa} * 2^{\text{exponent}}$$

- We can represent floating-point numbers with three binary fields: a sign bit *s*, an exponent field *e*, and a fraction field *f*.



- The IEEE 754 standard defines several different precisions.
 - Single precision numbers include an 8-bit exponent field and a 23-bit fraction, for a total of 32 bits.



- Double precision numbers have an 11-bit exponent field and a 52-bit fraction, for a total of 64 bits.

IEEE 754

- During 80's Institute for electrical and electronics Engineers produced a standard for the FP format.
- Now the “ *IEEE754 -1985 Standard for binary Floating Point Arithmetic* “ is adopted by almost all the vendors..
- IEEE Standard specified all the details about FP system:
 - storage format
 - precise specification of the result of operations
 - special values
 - specified runtime behavior on illegal operations
- Life is easier now: portability of kernel computational code is ensured...

Range of single-precision numbers (32 bit)

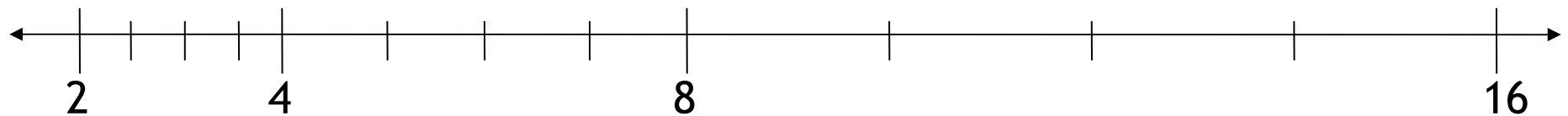
BIASED exponent

$$(1 - 2s) * (1 + f) * 2^{e-127}.$$

- The smallest e is 00000001 (1).
 - The smallest f is 000000000000000000000000 (0).
 - The largest possible e is 11111110 (254).
 - The largest possible f is 111111111111111111111111 (1 - 2⁻²³).
-
- The largest possible “normal” number is $(2 - 2^{-23}) * 2^{127} \approx 3.4 \times 10^{38}$.
 - the smallest positive non-zero number is $1 * 2^{-126} = \approx 1.8 \times 10^{-38}$.
 - In comparison, the range of possible 32-bit integers in two’s complement are only -2^{31} and $(2^{31} - 1)$
 - How can we represent so many more values in the IEEE 754 format, even though we use the same number of bits as regular integers?

FP Finiteness

- There *aren't* more IEEE numbers.
- With 32 bits, there are $2^{32}-1$, or about **4 billion**, different bit patterns.
 - These can represent 4 billion integers or 4 billion reals.
 - But there are an infinite number of reals, and the IEEE format can only represent some of the ones from about -2^{128} to $+2^{128}$.
 - Represent same number of values between 2^n and 2^{n+1} as 2^{n+1} and 2^{n+2}



- Thus, floating-point arithmetic has “issues”
 - Small roundoff errors can accumulate with multiplications or exponentiations, resulting in big errors.
 - Rounding errors can invalidate many basic arithmetic principles such as the associative law, $(x + y) + z = x + (y + z)$.
- The IEEE 754 standard guarantees that all machines will produce the same results—but those results may not be mathematically correct!

density of FP numbers..

- Because the same number of bits are used to represent all normalized numbers, the smaller the exponent, the greater the density of representable numbers.
- For example, there are approximately 8,388,607 single-precision numbers between 1.0 and 2.0, while there are only about 8191 between 1023.0 and 1024.0.
- this means that for large numbers (both positive and negative) there are very few FP numbers to play with and many real numbers map to the same floating-point number.

Floating Point Arithmetic

- Representable numbers:
 - The way the numbers are stored
- Operations:
 - The way the number are handled:
 - arithmetic: +, -, x, /, ... How is result **rounded** to fit in format?
 - comparison (<, =, >)
 - conversion between different formats - short to long FP numbers, FP to integer, etc.
 - **exception handling** - what to do for 0/0, 2*largest_number, etc.
 - binary/decimal conversion - for I/O, when radix is not 10.
- Language/library support is required for all these operations.

IEEE 754 exception handling

- What happens when the “exact value” is not a real number, or too small or too large to represent accurately?
 - Five exceptions:
 - Overflow - exact result $> OV$, too large to represent.
 - Underflow - exact result nonzero and $< UN$, too small to represent.
 - Divide-by-zero – nonzero/0.
 - Invalid - 0/0, $\sqrt{-1}$, ...
 - **Inexact - you made a rounding error (very common!).**
 - Possible responses
 - Stop with error message (unfriendly, not default).
 - Keep computing (default, but how?).

Rounding problem

- Many operations among floating points does not end in a floating point.
- IEEE 754 defines the way to handle this:
 - Take the exact value, and round it to the nearest floating point number (correct rounding).
 - Break ties by rounding to nearest floating point number whose bottom bit is zero (rounding to nearest even).
 - Other rounding options also available (up, down, towards 0).

Even some rather simple numbers can have FP rounding problem...

- $1/3$ is NOT exactly representable
- 0.01 is NOT exactly representable
- Question to be answered during lab this afternoon
 - how many inverse are not exactly representable in the range $[1, 100]$?

How to add FP numbers ?

- Associative rule does not work !

$$x = -1.5 \times 10^{38}$$

$$y = 1.5 \times 10^{38}$$

$$z = 1.0$$

$$\begin{aligned}x + (y + z) &= -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) \\ &= -1.5 \times 10^{38} + 1.5 \times 10^{38} \\ &= 0\end{aligned}$$

$$\begin{aligned}(x + y) + z &= (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 \\ &= 0 + 1.0 \\ &= 1.0\end{aligned}$$

How to avoid FP problems in summing numbers ?

- Using special tricks
 - The first is sorting the numbers and adding them in ascending order.
 - the Kahan Summation Formula.
 - Using integers instead of floats
- See exercise sum numbers in the lab this afternoon..

Please note that such kind of problem impact parallel computing: Summing numbers in parallel could give you different results !

Kahan summation formula

- Errors are reduced by keeping a separate running compensation (a variable to accumulate small errors).

```
function kahanSum(input)
  var sum = input[1]
  var c = 0.0          //A running compensation for lost low-order
bits.
  for i = 2 to input.length
    y = input[i] - c   //So far, so good: c is zero.
    t = sum + y        //Alas, sum is big, y small,
                      // so low-order digits of y are lost.
    c = (t - sum) - y  //(t - sum) recovers the high-order part of y;
                      // subtracting y recovers -(low part of y)
    sum = t           //Algebraically, c should always be zero.
                      // Beware eagerly optimising compilers!
  next i              //Next time around, the lost low part will be
                      // added to y in a fresh attempt.
return sum
```

Working with Integers (1)

- Integers are sometimes called **fixed-point numbers** because they can be viewed as floats with radix point after the least significant digit, and zero fractional digits
 - 1 1.000...
 - 10 10.000...
 - 100 100.000...
- Instead of a float X with p digits decimal mantissa, we can use an integer N :
 - $N = \text{INT}(X * (10^{**p}))$ (Float to integer transformation)
 - $X = N / (10^{**p})$ (Recovering the float from the integer)

Working with Integers (1)

- We can now sum correctly integer numbers
 - $X_1 + X_2 = N_1 / (10^{**p}) + N_2 / (10^{**p}) = (N_1 + N_2) / (10^{**p})$
- Note:
 - No roundoff error in both addition and subtraction
 - Precision of the operation is set in advanced: how many decimal places do I need ?

FP Subtraction : the cancellation problem(1)

- Subtraction between two t-digit number having same sign and similar magnitude yields result with fewer than t digits, hence it is always representable.
- Reason is that the leading digits of two numbers cancel (i.e their difference is zero)
- Example:

$$\begin{aligned} & - \quad 1.92403 \times 10^2 - 1.92275 \times 10^2 = 0.00128 \times 10^2 \quad \text{---->} \\ & \quad \quad \quad 1.28000 \times 10^{-1} \end{aligned}$$

- This is correct, exact representable but has only 3 digits ..

Cancellation (2)

- Despite exactness of results, cancellation often implies **serious loss of information**.
- Operands are often uncertain, due to rounding or other previous errors, in which case relative uncertainty in difference may be large.
- Subtraction itself is not at fault: it signals loss of information that has already occurred.

Checking Floating-Point Equality

- Sometimes okay to compare for equality
 - When calculations are known to be exact
 - To synthesize a comparison
 - Compare against 0.0 to avoid division by zero
- But floating-point results are usually inexact
 - Comparing floating-point numbers for equality may have undesirable results

```
do while (tot=1.0)
    tot=tot+0.1
end do
```

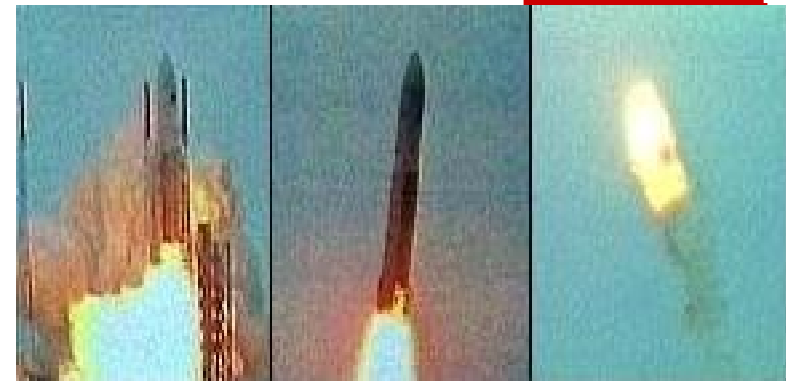
Using float/real*4 for computations

- Storing low-precision data as float is fine, but
- Generally not recommended to use float for computations
 - Float has less than half the precision of double
 - Using double intermediates greatly reduces the risk of roundoff problems polluting the answer
 - Round double value back to float to give a float result
- Extra internal precision is ablative armor against roundoff problems

be careful on float -> integer conversion

- FP numbers converted in integer number can lead to overflow/underflow ...

Data conversion



- On 4 June 1996, the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, exploded.
- The failure of the Ariane 501 was caused by the complete loss of guidance and attitude. This loss of information was due to specification and design errors in the software of the inertial reference system.
- The internal SRI* software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value.
- The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.
<http://www.ima.umn.edu/~arnold/disasters/ariane.html>

What about compilers ?

- IEEE754 standard defines how FP are to be performed
- compilers which translates our high level language (fortran/C) to assembler is responsible to generate IEEE-compliant code
- there are generally specific flags to force the compiler to adhere to the standard
- By default some compiler do not adhere to it: you have to force them to adhere..

Compiler IEEE flags

- Gnu suite:
 - **-ffloat-store** .. a few programs rely on the precise definition of IEEE floating point. Use -ffloat-store for such programs, after modifying them to store all pertinent intermediate computations into variables.
- PGI:
 - **-Kieee -Knoieee** (default) Perform floating-point operations in strict conformance with the IEEE 754 standard. Some optimizations are disabled with -Kieee, and a more accurate math library is used. The default -Knoieee uses faster but very slightly less accurate methods.
- Intel:
 - **-mp** Maintains floating-point precision (while disabling some optimizations). The -mp option restricts optimization to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI* and IEEE standards. This is the same as specifying -fltconsistency or -mieee-fp

A few final warnings..

- Only about 7 decimal digits are representable in single-precision IEEE format, and about 16 in double-precision IEEE format.
 - Use double/real*8 almost everywhere
- Remember: Every time numbers are transferred from external decimal to internal binary or vice-versa, precision can be lost.
- Always use safe comparisons.
- be careful on Conversions between data types
- Don't expect identical results from two different floating-point implementations.

A final citation..

Excerpt from The Art of Computer Programming by Donald E. Knuth:

"Floating-point computation is by nature **inexact**, and it is not difficult to misuse it so that the computed answers consist almost entirely of 'noise'.

One of the principal problems of numerical analysis is to determine **how accurate the results of certain numerical methods will be**; a 'credibility gap' problem is involved here: we don't know how much of the computer's answers to believe.

Novice computer users solve this problem by implicitly trusting in the computer as an infallible authority; they tend to believe all digits of a printed answer are significant.

Disillusioned computer users have just the opposite approach, they are constantly afraid their answers are almost meaningless."

Yet another citation..

It makes me nervous to fly on airplanes since I know they are designed using floating-point arithmetic.”

A. Householder

Further References on Floating Point Arithmetic

- Prof. Kahan's "Lecture Notes on IEEE 754"
 - www.cs.berkeley.edu/~wkahan/ieeestatus/ieee754.ps
- Prof. Kahan's "The Baleful Effects of Computer Benchmarks on Applied Math, Physics and Chemistry"
 - www.cs.berkeley.edu/~wkahan/ieee754status/baleful.ps
- *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys. Copyright 1991
- *The pitfall of verifying floating point computations* by D. Monniaux
 - hal.archivesouvertes.fr/docs/00/28/14/29/PDF/floating-point-article.pdf³⁴