# Computer number representation and errors and uncertainties in computations

M. Peressi - UniTS - Laurea Magistrale in Physics
Laboratory of Computational Physics - Unit I - part II

# Number representation in a given basis

A real number in basis 10:

$$741.36 = 7\cdot 10^2 + 4\cdot 10^1 + 1\cdot 10^0 + 3\cdot 10^{-1} + 6\cdot 10^{-2}$$

If b is the basis, the string: $a_k a_{k-1} a_{k-2} \ldots a_0 a_{-1} a_{-2} \ldots a_{-k}$

represents: $$\sum_{i=-k}^{k} a_i b^i = a_k b^k + a_{k-1} b^{k-1} + \ldots a_0 b^0 + a_{-1} b^{-1} + \ldots a_{-k} b^{-k}$$

Another example: integer number, basis 2:

$$(1001)_2 = 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = (9)_{10}$$

# Number representation in a computer

- the microscopic unit of memory is the BIT=(0,1)

1 BYTE = 1B = 8 BITS
1K = 1KB=$2^{10}$BYTES=1024 BYTES

- BIT=(0,1) => binary form for number representation

- the representation of a number in a computer is characterized by the numbers of bits used to store it

- fixed point (for integers) or floating point (for reals) representation

# Fixed point representation for integers

$$(1001)_2 = 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = (9)_{10}$$

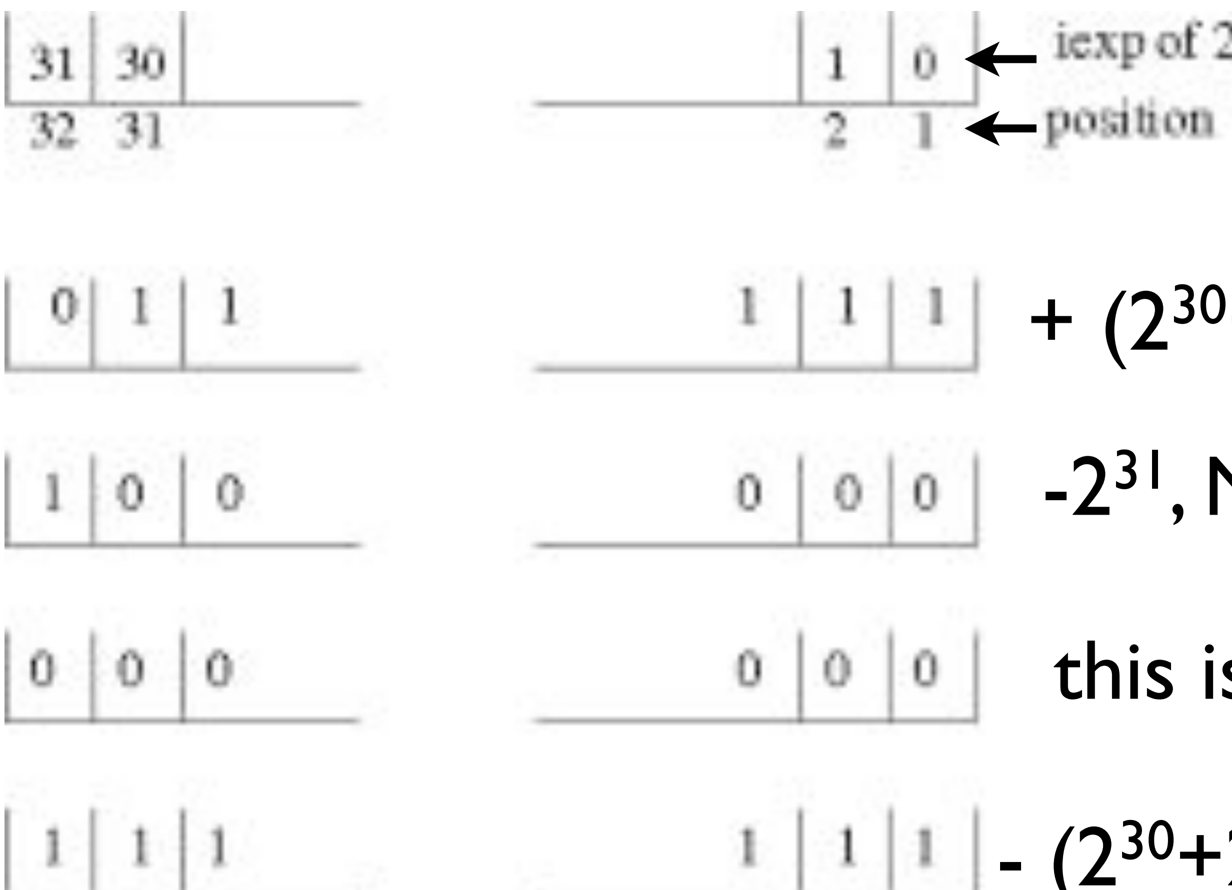
- With N bits, typically the first one is reserved to the sign: N-1 bits available =>

it is possible to represent numbers with absolute value in $[0, 2^{N-1}-1]$

If you try to go beyond: OVERFLOW
(i_min_max.f90)

# Fixed point representation for integers

On INFIS: the result is $[-2^{31}, 2^{31}-1]$; why?




$+ (2^{30}+2^{29}+..+2^{0}) = 2^{31}-1$

$-2^{31}$, NOT $+2^{31}$, NOT 0

this is 0

$- (2^{30}+2^{29}+..+2^{0}) = -(2^{31}-1)$

$(2^{31} \sim 2 \times 10^{9})$

# Floating point representation for real numbers

$$x_{float} = (-1)^s \bullet mantissa \bullet b^{\exp fld - bias}$$

| sign | significant figures of the number | exponent of the number; basis b=2 |
|---|---|---|

- Typically: expfld = 8-bit integer (goes from [0,255])

bias = 128 (or 127) => expfld-bias goes from -128 to +127 (or from -127 to +128) ;

23 bits reserved for the mantissa => tot 32 bits

$$mantissa = m_1 \cdot 2^{-1} + m_2 \cdot 2^{-2} + \ldots m_{23} \cdot 2^{-23}$$ ($m_1$ NOT 0!)

- precision: $2^{-23}$ ~= 6-7 decimal figures
- range : ~ $-10^{-39}$ - $10^{+38}$

# examples of floating point representation for real numbers

$\frac{1}{2} = 0 \quad \underbrace{0111 \quad 1111}_{} \quad 1000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 000$

↓ segno — Expfld=127, Expldf-bias =0 (here: bias=127)

$6 = 0 \quad \underbrace{1000 \quad 0010}_{} \quad 1100 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 000$

↓ segno — Expfld=130, Expldf-bias =3

the smallest: (if mantissa is in the normalized form, i.e., first number =/= 0)

$0 \quad 0000 \quad 0000 \quad 1000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 000 = 2^{-128} = 2.9\times10^{-39}$

the largest:

$0 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 111 = 2^{128} = 3.4\times10^{38}$

# single and double precision

For **double precision**:

- Typically: expfld = 11-bit integer (goes from [0,2047])

bias = 1023 => expfld-bias goes from -1023 to +1024 ;

52 bits reserved for the mantissa => tot 64 bits

- precision: $2^{-52}$ ~= 15-16 decimal figures
- range : ~ $-10^{-322}$ - $10^{+308}$

If you try to go beyond these limits (see rs(d)_under_over.f90):
UNDERFLOW (too small) and OVERFLOW (too large)

# Roundoff errors

$$7+1.0\times10^{-9} = ???$$

Single precision representation:

$7 = 0 \quad 1000 \quad 0010 \quad 1110 \;\; 0000 \;\; 0000 \;\; 0000 \;\; 0000 \;\; 000$

$10^{-9} = 0 \quad 0110 \quad 0000 \quad 1101 \;\; 0110 \;\; 1011 \;\; 1111 \;\; 1001 \;\; 010$

(here: bias=127)

Exponents are different! Make them equal before operating on the mantissas: increase the smallest exponent, but at the same time reduce the corresponding mantissa:

x2 (i.e., +1 in expfield) :2 (right hand-side shift of the bits) (bits lost)

$10^{-9} = 0 \quad 0110\;0001 \quad 0110 \;\; 1011 \;\; 0101 \;\; 1111 \;\; 1100 \;\; 101 \quad (0)$

$= 0 \quad 0110\;0010 \quad 0011 \;\; 0101 \;\; 1010 \;\; 1111 \;\; 1110 \;\; 010 \quad (10)$

$\ldots\ldots,$

$= 0 \quad 1000\;0010 \quad 0000 \;\; 0000 \;\; 0000 \;\; 0000 \;\; 0000 \;\; 000 \quad (0001101\ldots)$

$\Rightarrow \quad 7+1.0\times10^{-9} = 7$ !!!

# Machine precision

The smallest number that, added to 1 represented in the machine, does not change it:

$$\varepsilon_m + 1_c = 1_c$$

$\varepsilon \cong 10^{-7}$ single precision

$\varepsilon \cong 10^{-16}$ double precision

Note: IT IS NOT the smallest representable number!

See also: intrinsic function epsilon(x)

# Source of errors in numerical computing

- Human
- Random (e.g. electrical fluctuations)
- Roundoff $2\left(\frac{1}{3}\right)-\frac{2}{3}=2\times 0.3333333\ -0.6666667\ =-0.0000001\ \neq 0.$
- Truncation $e^{x}=\sum_{n=0}^{\infty}\frac{x^{n}}{n!}\cong\sum_{n=0}^{N}\frac{x^{n}}{n!}=e^{x}+E(x,N)$

Mainly ROUNDOFF,
due to the finite representation of numbers
in a computer

# An example of roundoff+truncation
# Numerical derivatives

Calculate the derivative of:

$$f(x) = sin(x) \text{ in } x = 1$$

We call: $f_0 = f(x),\ f_1 = f(x+h), \text{ e } f_{-1} = f(x-h).$

We can use several algorithms:

$$f'(x) \sim \frac{f_1 - f_{-1}}{2h} \qquad (=f'_{simm})$$

$$f'(x) \sim \frac{f_1 - f_0}{h} \qquad (=f'_{ds})$$

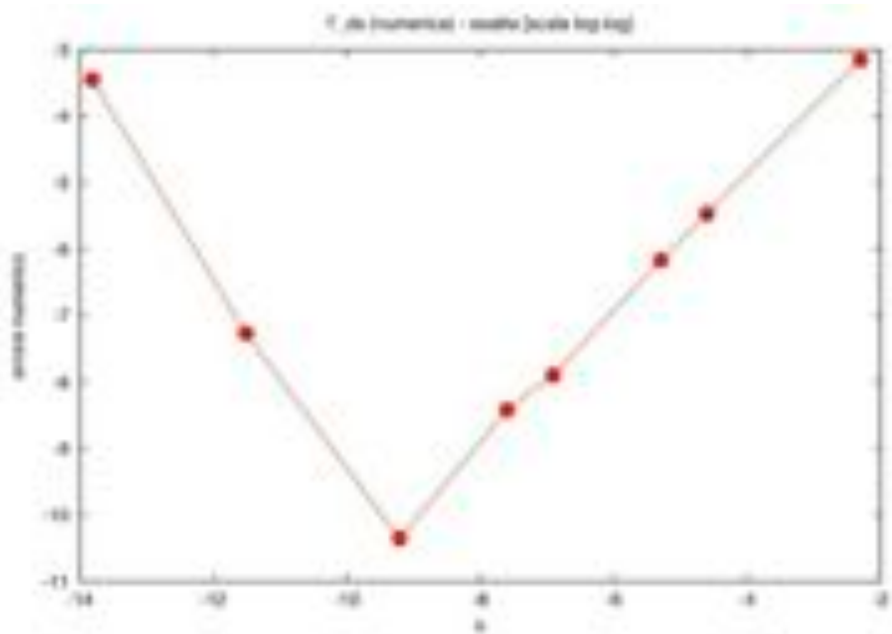
$$f'(x) \sim \frac{f_0 - f_{-1}}{h} \qquad (=f'_{sin})$$

Make numerical experiments with *h* and progressively reduce *it* ...

# An example of roundoff+truncation

# Numerical derivatives

| h | $f'_{ds}$ | $f'_{ds}$-exact | $f'_{sin}$ | $f'_{sin}$-exact | $f'_{simm}$ | $f'_{simm}$ - exact |
|---|---|---|---|---|---|---|
| 0.1 | 0.497364 | -0.042938 | 0.581441 | 0.041138 | 0.539402 | -0.000900 |
| 0.01 | 0.536087 | -0.004215 | 0.544497 | 0.004195 | 0.540294 | -0.000009 |
| 0.005 | 0.538200 | -0.002102 | 0.542398 | 0.002095 | 0.540302 | 0.000000 |
| 0.001 | 0.539930 | -0.000372 | 0.540688 | 0.000386 | 0.540323 | 0.000021 |
| 0.0005 | 0.540081 | -0.000221 | 0.540482 | 0.000180 | 0.540310 | 0.000007 |
| 0.0001 | 0.540334 | 0.000032 | 0.540154 | -0.000148 | 0.540384 | 0.000082 |
| 1E-05 | 0.539602 | -0.000701 | 0.538240 | -0.002063 | 0.540321 | 0.000019 |
| 1E-06 | 0.508436 | -0.031866 | 0.519472 | -0.020830 | 0.527957 | -0.012345 |



- The symmetric algorithm is the best
- By reducing *h* down to $\sim 10^{-4}$ the numerical error decreases, but further reduction of *h* does not improve the result, or better, the result is even worse!

Why? **roundoff error**

# Other possible sources of errors due to roundoff

- subtraction between very large numbers ($\infty$-$\infty$)
(see examples: exp-bad.f90)

# expressions analytically equivalent can be NOT numerically equivalent!

$$\sum_{n=1}^{2N} (-1)^n \frac{n}{n+1}$$

**BAD!**

$$-\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1}$$

<0 >0

**OK!**

$$\sum_{n=1}^{N} \frac{1}{2n(2n+1)}$$

## How does your computer make a calculation?

(remember... this is a common source
of HUMAN error in coding)

1/2 = ???? 0 !!! WRONG

since this operation is done within the INTEGERS

1./2 = ??? 0.5 !!! CORRECT

since 2 is promoted to REAL and the operation is done within the REALS

# Some programs:

in $/home/peressi/comp-phys/I-basics/f90:

deriv.f90;  d_strano.f90
exp-bad.dp.f90  ;  exp-bad.f90
exp-good.dp.f90  ; exp-good.f90
i_min_max.f90
rd_under_over.f90 ; rs_under_over.f90
rs_limit.f90 ;  rd_limit.f90
strano.f90
test1-subr-funct.f90 ; test2-subr-funct.f90
test_factorial.f90