Floating Point Arithmetic

Alessandro Corbetta

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** -- Approximations from numerical method
  - **Rounding -> computers offer only finite precision in representing real numbers** (THIS LECTURE)

Example 1: earth surface calculation

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

- **Modeling** -- Earth is NOT a perfect sphere
- **Empirical data** -- rr contains measurement errors
- **Truncation** -- our calculation employs a finite amount of digits for ππ
- **Rounding -> the multiplications are rounded**

Table of contents

Rounding

- Efficiency/Memory reasons =>
  - **amount of information numeric variables carry:**
    LIMITED to N bits (usually 8/16/32/64/80/128)
  - trade-off between speed and accuracy

- A variable of N bits can represent exactly 2^N states.
  - `E.g. int` -> 32bit numbers between −2.147.483.648 and +2.147.483.647
  - 2^32 ~ 4B states
  - 1 is the smallest non-zero number representable
  - Integer: OK with “digital thinking”
- **ISSUE:** we often need to deal with **real numbers:**
  - **0.03, 445.67, e,** Gas constant R in SI units, pi, Universal gravitational constant in SI units
  - Our height compared to the earth radius…

Rounding

- We can fabricate many encodings carrying real numbers in 32bit
- **Key observation:**

  regardless the encoding

  we **cannot bypass the limit of 32bit worth of information!**

- Simple example - *Fixed point* arithmetic:

  we scale the integers by a given factor

  e.g. Scale: 1/10.000, 32bit

  −214.748,3648 and +214.748,3647

- **Still inconvenient:** need to **agree** on the scale

  cannot represent numbers smaller than 1/10.000 or too large.

IEEE 754 Floating-point number representation
(single precision case)

- Standard representation since the 90's. Now at 2008 revision
- Idea:
  - $x = \underbrace{22.841796875}_{\text{Usual decimal notation}} = \underbrace{+0.\mathbf{2}2841796875 \cdot 10^{-2}}_{\text{Normalized scientific notation}}$

(0.<non_zero>....)

32 bit of information

We conveniently use base 2

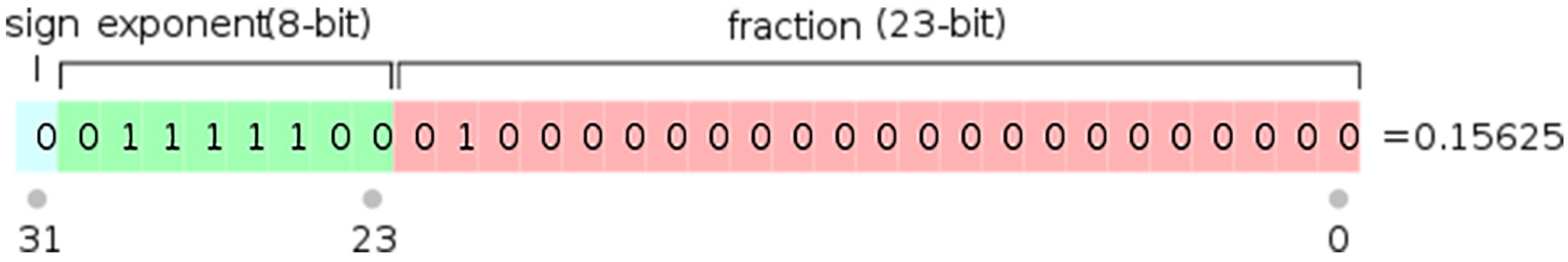
$$x = +0.10110110101111000000000 \cdot 2^{00000101}$$

Sign: 1bit

Mantissa: 23bit

exponent: 8bit
(with bias)

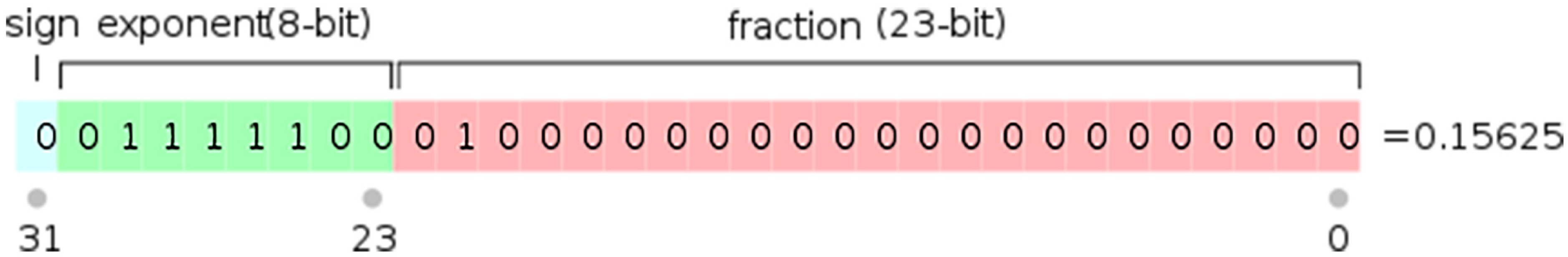
IEEE 754 Floating-point number representation
(single precision case)



$$x = \pm 0.101101 \ldots \cdot 2^{01111-bias}$$

- Different chunks of the 32bits serve different purposes
  - Sign -> 1bit
  - Scale (exponent) -> 8bit
  - Fraction/Significand/Mantissa -> 23bit
- We use normalized representation: the first “fractional” digit is always 1! (one bit gain!)
- The exponent is stored normalized: $real \exp = storedexp - bias(127)$
- The exponent “$storedexp = 0$” is kept for special cases

IEEE 754 Floating-point number representation
(single precision case)



$$x = \pm 0.101101 \ldots \cdot 2^{01111+bias}$$

- Different chunks of the 32bits serve different purposes
  - Sign -> 1bit
  - Scale (exponent) -> 8bit
  - Fraction/Significand/Mantissa -> 23bit
- Heads up: Floating-point arithmetic comes with different rules!
- <u>UNDERSTANDING THE RULES: ESSENTIAL!</u>
  - Past: several devastating accidents (related to guidance problem)

Example 2: Constant speed walking: how far do I get?
(A journey with rounding errors)

- Walking forward with constant velocity

$$\dot{x} = v$$

- This can be discretized as

$$x_{n+1} = x_n + \Delta t\, v$$

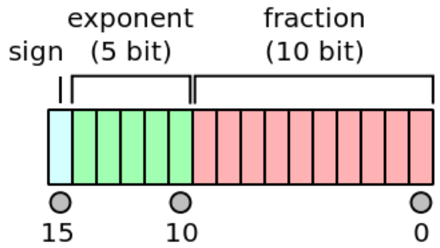
- For simplicity let the displacement be $\Delta t\, v = 1$.

- **ISSUE**: <u>naïve implementations rapidly yield wrong results due to rounding errors</u>

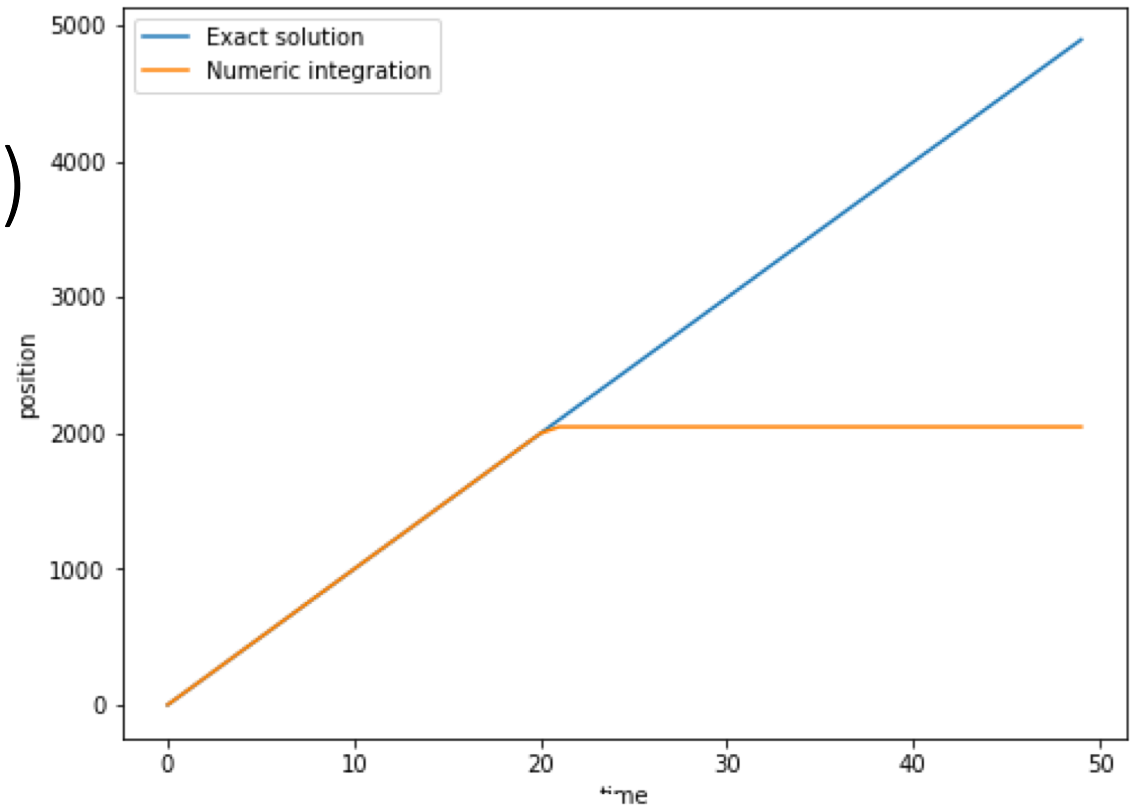
Example 2: Constant speed walking: how far do I get?

(A journey with rounding errors)

```
import numpy as np
%pylab inline

history = []

x = np.float16(0.)  ## initial position at 0. using half-precision to cut it short
u = np.float16(1.)  ## displacement at every time step


time = range(int(5e3))
saving_step = 100
for t in time:
    x += u
    if t % saving_step == 0:
        history.append(x)

plt.figure(figsize = (8,6))
plt.plot(np.array(time[::saving_step]), label = 'Exact solution')
plt.plot(history,label='Numeric integration')
plt.xlabel('time')
plt.ylabel('position')

plt.legend()
```



Example 2
(A journey with rounding errors)



"Walking" with 3 significant digits (as in base for simplicity 10)

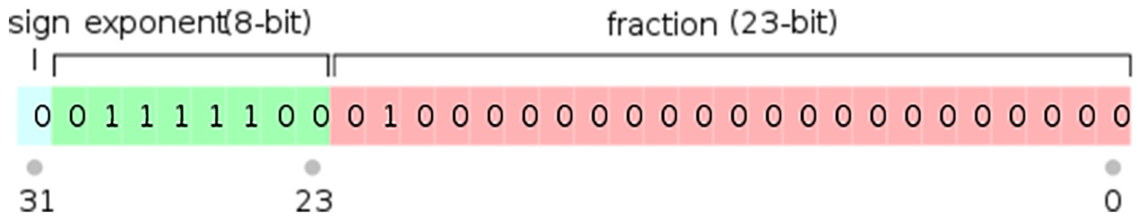
- initial position: $0.000 \cdot 10^0$
- next step: $0.100 \cdot 10^1$
- ....
- position: $999 = 0.999 \cdot 10^3$
- next step: $999 + 1 = 0.100 \cdot 10^5$
- next step:

$$(1000 + 1) = 0.100 \cdot 10^5 + 0.1 \cdot 10^1 = 0.1001 \cdot 10^5 \rightarrow 0.100 \cdot 10^5$$

where the last arrow performed the truncation operation as we retain 3 digits only.

Digging in the IEEE 754 standard

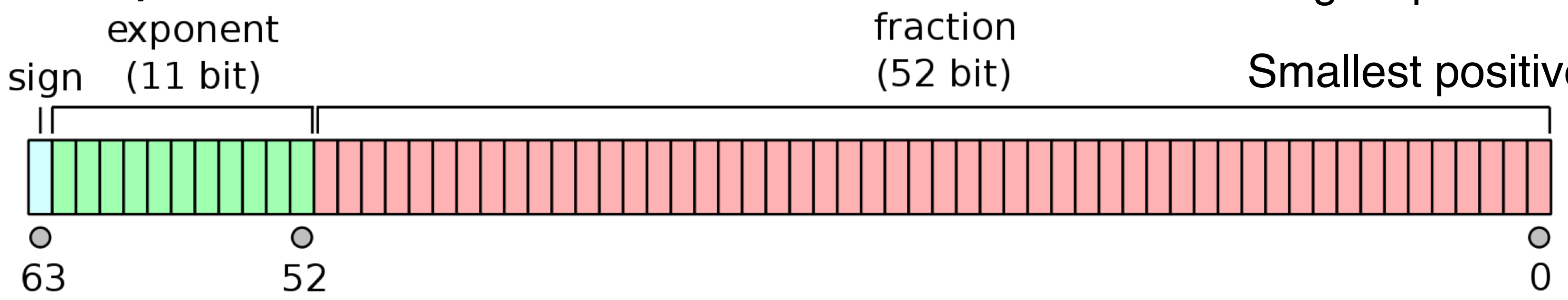
Single precision: 32 bit



Largest possible number is ≈ $3.4 * 10^{38}$ (decimal repr.)

Smallest positive number is ≈$1.8 * 10^{-38}$

Double precision: 64 bit



Largest possible number is ≈ $1.798 \cdot 10^{308}$ (decimal repr

Smallest positive number is ≈ $2.22 \cdot 10^{-308}$

(we can go subnormal -> a little lower)

More recently: quad precision 128 bit, half-precision 16 bit (deep learning)

What the IEEE 754 Standard defines

- Arithmetic operations
  - (add, subtract, multiply, divide, square root, fused multiply-add, remainder)
- Conversions between formats
- Encodings of special values

- This ensures portability of compute kernels

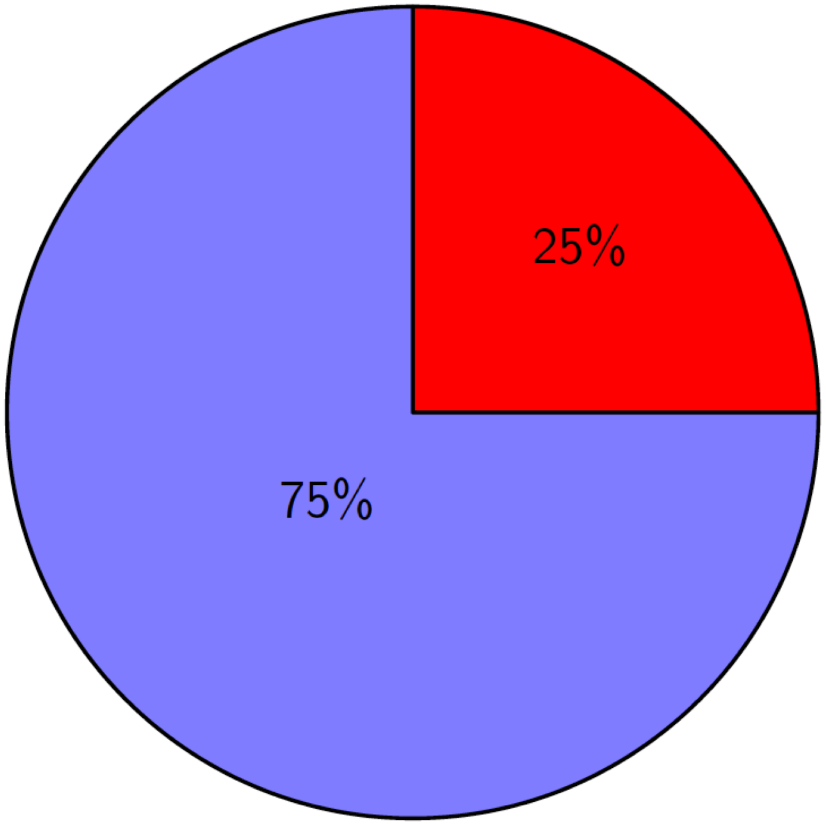
IEEE 754 implications: number density

- number of significant digits FIXED =>number density decreases
  - the exponent determines the number density

- <u>Example:</u> 32 bit float
  - 8 bits exponents
  - 0 is represented with exponent -127
  - 126 negative exponents, each has 2^23 unique numbers
    - Total in (0,1) = 1,056,964,608
  - Boundaries: {0}, {1} (2 contributions)
    - **Total in [0,1] = 1,056,964,610**

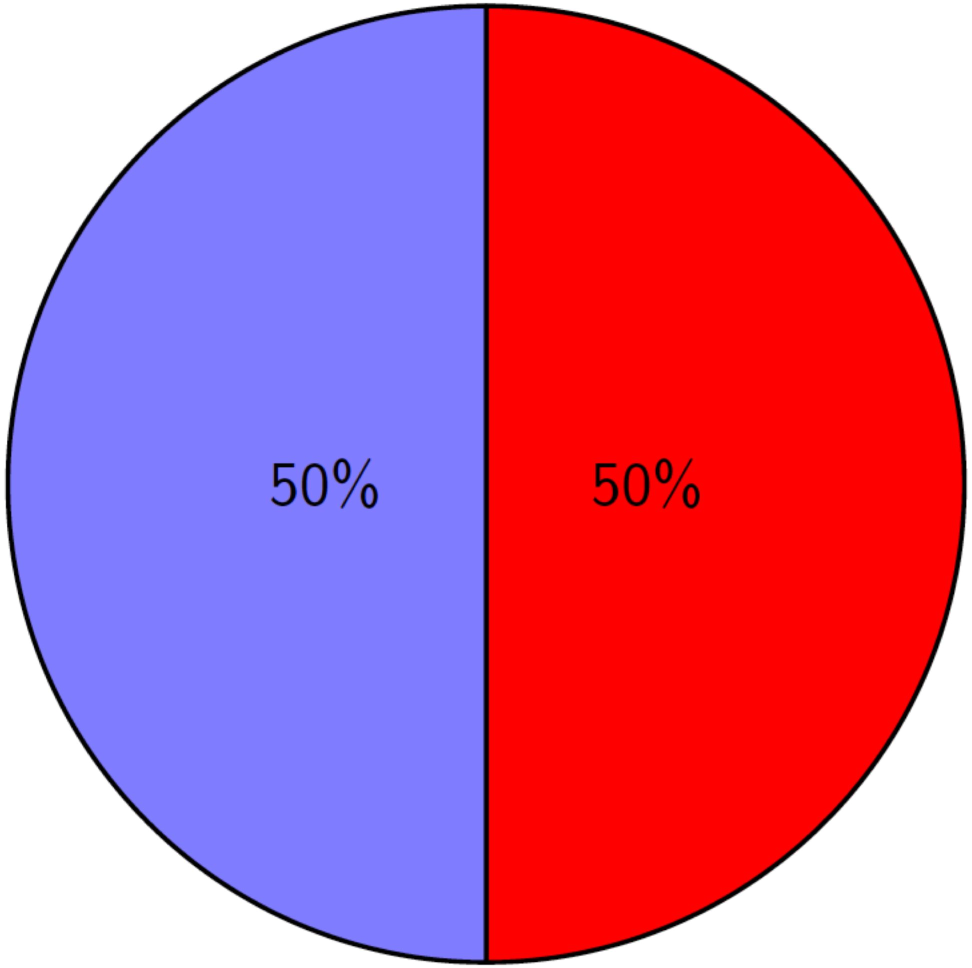
  - REM: Total available numbers in 32bit: 2^32 = **4,294,967,296**

IEEE 754 number density – numbers in [0,1]



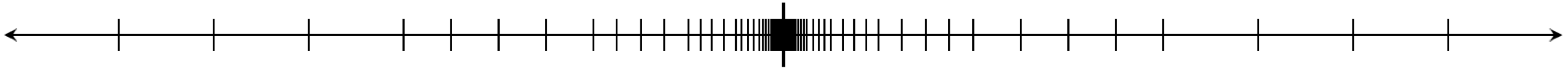
About **25%** of all numbers available in FP are between 0.0 and 1.0

IEEE 754 number density - numbers in [-1,1]



Hence about **50%** of all numbers available in FP are between -1.0 and 1.0

IEEE 754 number density



- the same number of bits is used for the significand =>
  exponent determines representable number density
  - e.g. in a single-precision floating-point number there are 8,388,606 numbers
  - between 1.0 and 2.0, but only 16,382 between 1023.0 and 1024.0
- Accuracy depends on the magnitude

- For instance: all numbers beyond a threshold are even
  -> We lose the “unit bit” O(1)
  - single-precision: all numbers beyond 224 are even
  - double-precision: all numbers beyond 253 are even

IEEE 754 – Unit of Last Position (ULP) & rounding error

- **ULP**: spacing between two **neighboring** floating-point numbers.
  - $x = 0.110\underbrace{1010\mathbf{0}} \cdot 2^{exp}$

    <u>How large is the increment if the last zero is shifted to one?</u>

- ULP ~ 2 x relative error that we make as we truncate

- Machine ϵ: ULP for $x = 1$

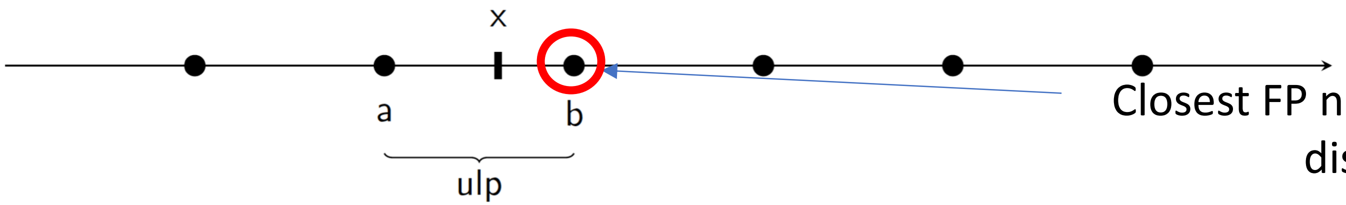
Example:

$$e^{100} \approx 2.6881171418161356 \; 10^{43}$$

- If we take the approximation literarily:
  - **26,881,171,418,161,356**,000,000,000,000,000,000,000,000,000
- **actual value stored** (after float64 binary representation)
  - **26,881,171,418,161,356**,094,253,400,435,962,903,554,686,976

- **correct value**
  - **26,881,171,418,161,354**,484,126,255,515,800,135,873,611,118
- **ISSUE**: the correct value is NOT a float64 number and needs to be approximated by rounded

a=26,881,171,418,161,351,142,493,243,294,441,803,958,190,080
b=26,881,171,418,161,356,094,253,400,435,962,903,554,686,976



Closest FP number:
distance < 0.5ULP => relative error

IEEE 754 Operations: sum

Sum $a \oplus b$:

Algorithm:

1. determine operand with smaller exponent (e.g. *a*)
2. transform the representation of *a* to have the same exponent of *b*
   i.e. shift the mantissa
3. *sum the mantissa (integer operation, temporarily on more bits)*
4. *normalize*
5. *round*
6. *truncate*

ISSUE: the sum $\oplus$ <u>has different rules than</u> on (usual "infinite precision") real numbers

IEEE 754 Operations: sum properties & issues

	Commutative property	Associative property
Real-numbers op	$a + b = b + a$	$(a + b) + c = \mathrm{a} + (\mathrm{b} + \mathrm{c})$
Floating-point op	$a \oplus b = b \oplus a$	$(a \oplus b) \oplus c \neq \mathrm{a} \oplus (\mathrm{b} \oplus \mathrm{c})$

ISSUE 1: Floating point sum is not associative!

Example

```
d = 1.0 + (1.5e38 + (-1.5e38));
printf("%f", d); // prints 1.0

d = (1.0 + 1.5e38) + (-1.5e38);
printf("%f", d); // prints 0.0
```

IEEE 754 Operations: sum properties & issues

ISSUE 2: subtractions of similar numbers (after rounding) <u>yields loss of precision</u>

- After rounding, represented numbers come with error |e|<0.5ULP
- Catastrophic cancellation happens when similar operands subjected to rounding errors are subtracted.

EXAMPLE

$$a = \mathbf{0.1234567}89 \rightarrow a_t = 0.123456\mathbf{8}$$
$$b = \mathbf{0.1234555}55 \rightarrow b_t = 0.123455\mathbf{6}$$
$$\boldsymbol{a - b = 0.000001234 = 0.123400000 \cdot 10^{-7} = (a - b)_t}$$

$$\boldsymbol{a_t \oplus -b_t = 0.1200000 \cdot 10^{-7}}$$

<u>We lost precision with no possibility of gaining it back.</u>

- REM: Extra care when dealing with subtractions of similar numbers
- REM: Extra-extra-extra care when using result in multiplication, since result is tainted by low accuracy

IEEE 754 Operations: multiplication

multiplication $a \otimes b$:

Algorithm:

1. multiply mantissa (yields a valid mantissa by construction)
2. sum exponents
3. normalize
4. round
5. truncate

Again the FP multiplication $\otimes$ is commutative but not associative although "not plagued by cancellation"

ISSUE: we can overflow underflow exponents

IEEE 754 Operations: order relations and comparison

As numbers are known with accuracy 0.5ULP (or lower):

- equality is not well defined
- two numbers are “equal” (considering rounding) if their relative difference is within few machine epsilon
- unsafe to use FP numbers e.g. as loop indices

- **Rem when testing**

IEEE 754 standard – special numbers

The standard prescribes a special set of values to treat exceptions

Type	Exp	Fraction	Sign
Positive Zero	0	0	0
Negative Zero	0	0	1
Denormalised numbers	0	non zero	any
Normalised numbers	$1..2^e - 2$	any	any
Infinities	$2^e - 1$	0	any
NaN	$2^e - 1$	non zero	any

IEEE 754 Operations: best practices

- Avoid summation of numbers with different order of magnitude as “small” terms are discarded
  - We can play smart algorithmic tricks:
    - sum after sort to allow gradual growth of order of magnitude
    - sum in blocks keeping order of magnitude commensurable
    - use Kahan summation
    - use higher precision variables

- Algorithmic tricks cannot help cancellation.
  - Use your math
  - Check for suitable library help (exp(x) -1, for x ~ 0)

Exercises

- Go online https://gitlab.com/acorbe/SMR3199_FP_ex
- `git clone git@gitlab.com:acorbe/SMR3199_FP_ex.git`