© EPITA | MCIT | ITI

Arab Republic of Egypt
Ministry of Communications
and Information Technology

Numerical Algorithms for Machine Learning

Session 1

Computer arithmetic: the IEEE 754 Standard

Dr. Ahmed Abdelreheem

Content

Why we study Floating-Point Representation and Arithmetic?

Assuming that

$$x = 0.4, \qquad y = 0.8 \ , \qquad z = 1.2$$

Mathematically we have

$$x + y = z$$

$$z - y = x$$

$$z - x = y$$

$$z - x - y = 0$$

"the computer calculated it, so it must be right".

Is it "numerically" true with Computer numbers, i.e. with Floating Point Numbers ?

Patriot Missile Interceptor

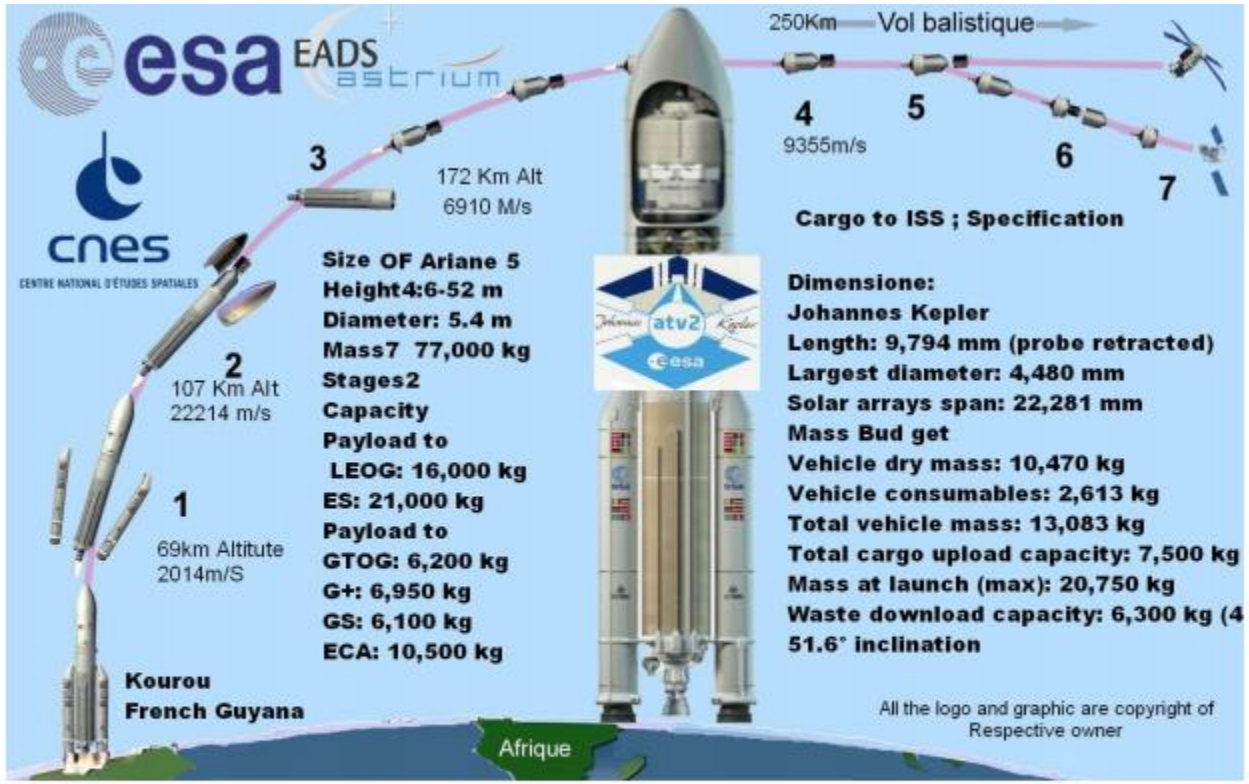
Patriot missile interceptor fails to intercept due to 0.2 second being the 'recurring decimal' $0.00\,1100\,1100\ldots_2$ in binary (1991).

- Clock cycle of 1/10 seconds was represented in 24-bit fixed point register created an error of 9.5 x 10^{-8} seconds.

.1 (decimal) = 00111101110011001100110011001101 (binary)

- The battery was on for 100 consecutive hours, thus causing an inaccuracy of

$$= 9.5\times10^{-8}\,\frac{\text{s}}{0.1\text{s}}\times100\text{hr}\times\frac{3600\text{s}}{1\text{hr}}$$
$$= 0.342s$$



ESP Ariane 5

In 1996, Ariane 5 $500M explosion caused by overflow in converting 64-bit floating-point number to 16-bit integer.

- **Ariane 5 rocket**
  - June 1996 exploded when a 64 bit fl pt number relating to the horizontal velocity of the rocket was converted to a 16 bit signed integer. The number was larger than 32,767, and thus the conversion failed.
  - $500M rocket and cargo

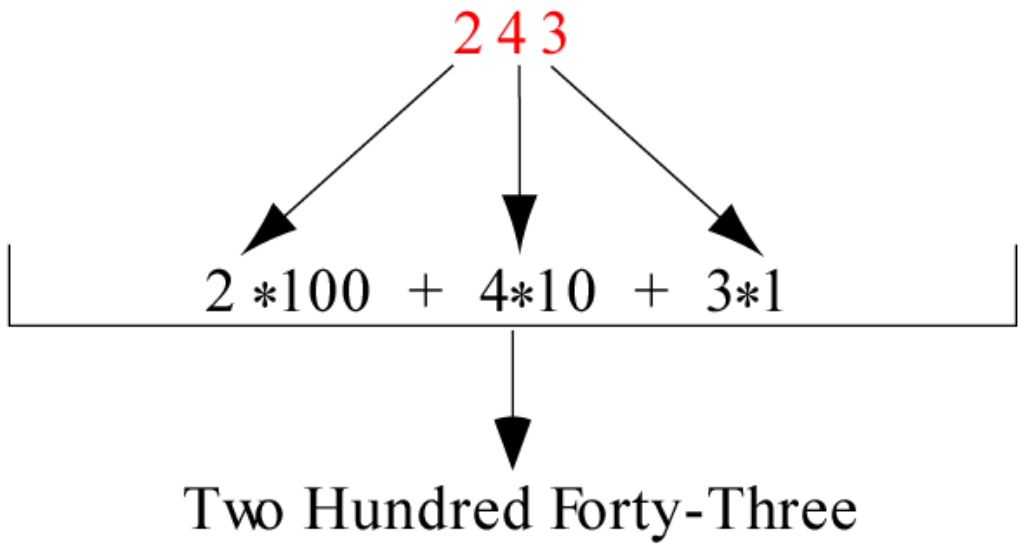
We need to understand how these numbers are represented inside our computers !

Number Representation

Decimal system

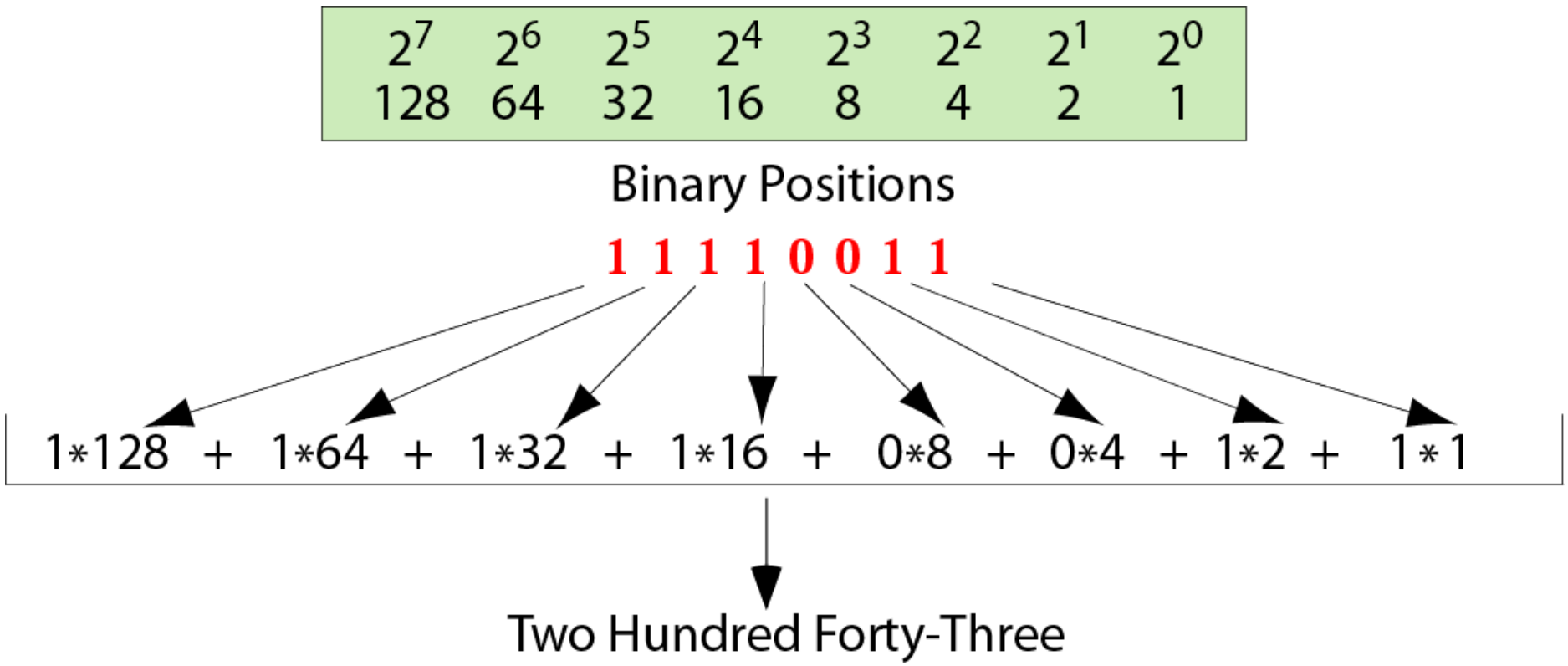
10^4	10^3	10^2	10^1	10^0
10,000	1000	100	10	1

Decimal Positions



Number Representation

Binary system



Number Representation

Converting Base-2 to Base-10

$$(1\ \ 0\ \ 0\ \ 1\ \ 1)$$

Exponent: $2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$

Calculation: $16 + 0 + 0 + 2 + 1 =$

$$(19)_{10}$$

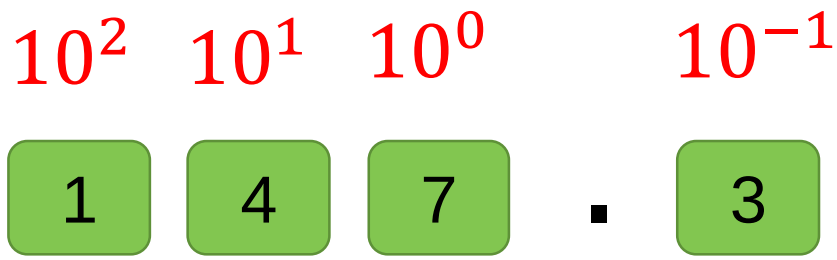
Number Representation

Example of storing unsigned integers in two different computers

Decimal	8-bit allocation	16-bit allocation
7	00000111	0000000000000111
234	11101010	0000000011101010
258	overflow	0000000100000010
24,760	overflow	0110000010111000
1,245,678	overflow	overflow

Base-10 Numbers Representation

Show the decimal expansion for 147.3



$$147.3 = 100 + 40 + 7 + 0.3$$
$$= 1 \times 100 + 4 \times 10 + 7 \times 1 + 3 \times (1/10)$$
$$= 1 \times 10^2 + 4 \times 10^1 + 7 \times 10^0 + 3 \times 10^{-1}$$

❖ If we have a decimal number

$$x = (D_N D_{N-1} D_{N-2} \ldots D_1 D_0 . D_{-1} D_{-2} \ldots D_{-M})_{10}$$

where D_N to D_0 are its digits before the decimal point, D_{-1} to D_{-M} are its digits after the decimal, and $0 \le D_i < 10$

The decimal expansion of this number is

$$D_N \times 10^N + D_{N-1} \times 10^{N-1} + \cdots + D_1 \times 10^1 + D_0 \times 10^0$$
$$+ D_{-1} \times 10^{-1} + D_{-2} \times 10^{-2} + \cdots + D_{-M} \times 10^{-M}$$

Base-2 Numbers Representation

Show the binary representation for $(11.5)_{10}$

2^3	2^2	2^1	2^0		2^{-1}
1	0	1	1	.	1

$$11.5 = 8 + 2 + 1 + (1/2)$$
$$= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 + 1 \times (1/2)$$
$$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1}$$

❖ If we have a binary number

$$x = (b_N b_{N-1} b_{N-2} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-M})_2$$

where b_N to b_0 are its digits before the decimal point, b_{-1} to b_{-M} are its digits after the decimal, and $0 \le b_i < 2$

The decimal value of this number is

$$b_N \times 2^N + b_{N-1} \times 2^{N-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$
$$+ b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-M} \times 2^{-M}$$

Exercise

- **What are the decimal value of the following binary numbers:**

1. $(1010.01)_2$
2. $(111.111)_2$

- **What are the binary representation of the following base-10 numbers:**

1. $(8.75)_{10}$
2. $(0.1)_{10}$

Exercise

- **What are the decimal value of the following binary numbers:**

1. $(1010.01)_2$:

2^3	2^2	2^1	2^0		2^{-1}	2^{-2}
1	0	1	0	.	0	1

$8 + 0 + 2 + 0 . 0 + 1/4 = 10.25$

2. $(111.111)_2$

- **What are the binary representation of the following base-10 numbers:**

1. $(8.75)_{10}$

2. $(0.1)_{10}$

Exercise

- **What are the decimal value of the following binary numbers:**

1. $(1010.01)_2$:

2. $(111.111)_2$

2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
1	1	1	1	.	1	1	1

$$8 + 4 + 2 + 1 \; . \; \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 15.875$$

- **What are the binary representation of the following base-10 numbers:**

1. $(8.75)_{10}$

2. $(0.1)_{10}$

Exercise

Decimal → Binary

8.75

2	8	0
2	4	0
2	2	0
2	1	1
	0	

$0.75 \times 2 = 0.5 \quad | \quad 1$

$0.5 \times 2 = 0.0 \quad | \quad 1$

$0_{(stop\ at\ zero)}$

(when answer > 1, subtract 1)

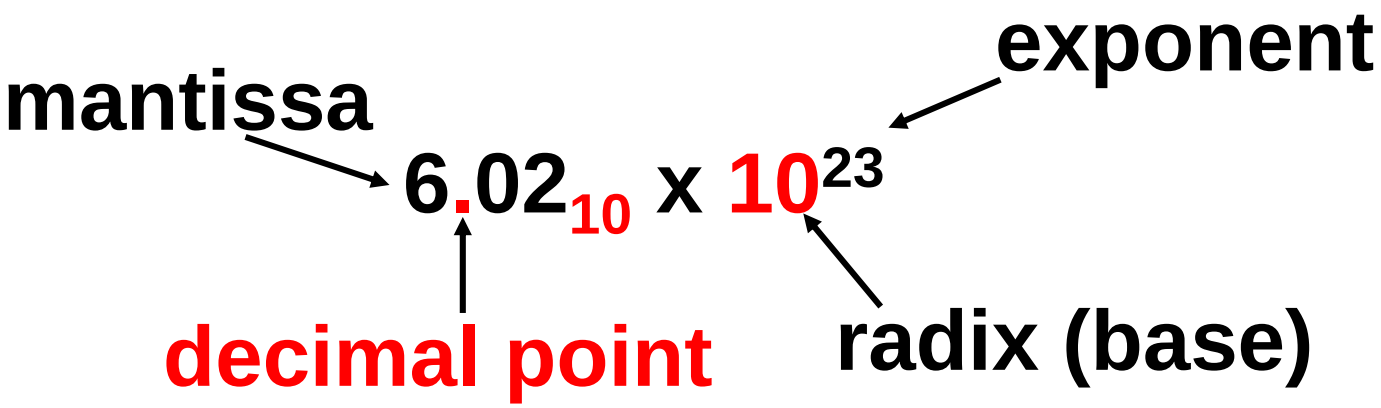
1000.11

Scientific Notation

$$123456. \times 10^{-1} = 12345.6 \times 10^{0}$$
$$= 1234.56 \times 10^{1}$$
$$= 123.456 \times 10^{2}$$
$$= 12.3456 \times 10^{3}$$
$$= 1.23456 \times 10^{4} (normalised)$$
$$\approx 0.12345 \times 10^{5}$$
$$\approx 0.01234 \times 10^{6}$$

Scientific notation is a way to make these numbers easier to work with. In scientific notation, you move the decimal place until you have a number between 1 and 10

Scientific Notation (in Decimal)

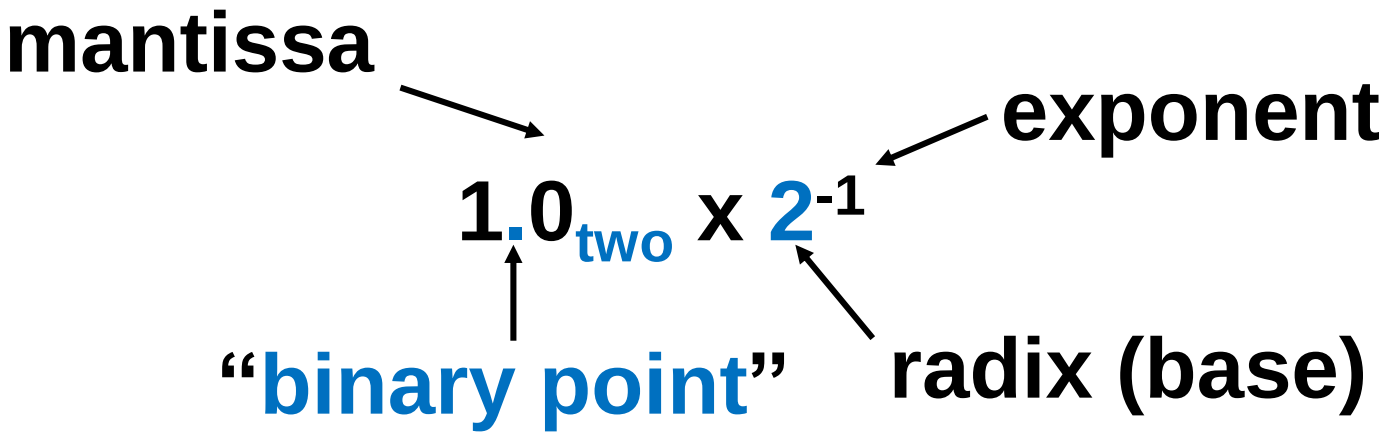


- Normalized form: no leadings 0s

  (**exactly one digit to left of decimal point**)

- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation (in Binary)



- **Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers**

$$(-1)^s \times 1.m \times 2^{e-127}$$

IEEE Floating Point Arithmetic

- Numerical computing is a vital part of the modern scientific infrastructure. Almost all numerical computing uses floating point arithmetic, and almost every modern computer implements the IEEE binary floating-point standard

- Floating point computation was in standard use by the mid 1950s. During the subsequent two decades, each computer manufacturer developed its own floating point system, leading to much inconsistency in how one program might behave on different machines. For example, the IBM 360/370 series, which dominated computing during the 1960s and 1970s, used a hexadecimal system (base 16)

- Consequently, it was very difficult to write portable software that would work properly on all machines. Programmers needed to be aware of various difficulties that might arise on different machines and attempt to forestall them.

- In the past every manufacturer produced their own floating point hardware and floating point programs gave different answers. IEEE standardization fixed this.
- In 1985, the IEEE (Institute for Electrical and Electronic Engineers) published a report called Binary Floating Point Arithmetic Standard 754

Floating Point Arithmetic IEEE Standard 754

- The number stored is:

$$(-1)^{s} \times (1 + 0.Mantissa) \times 2^{\{Exponent-127\}}$$

- The -127 is a bias used to be able to represent negative exponents.

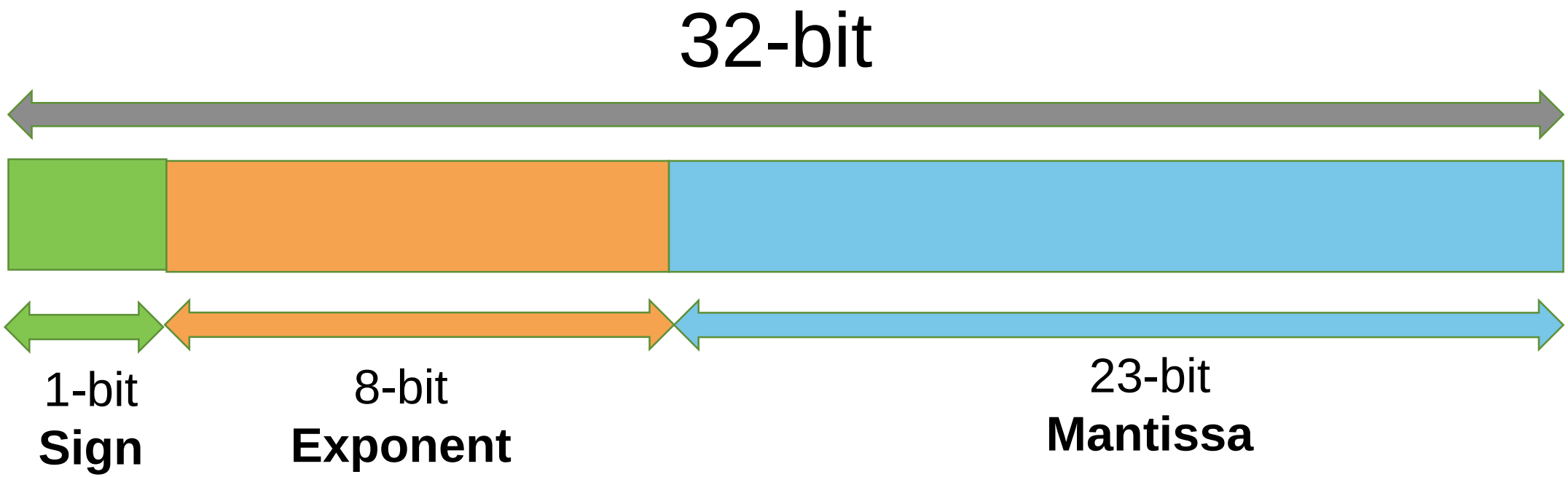
➢ For example, if E = 0, then the number will have the exponent -127 and the number will be

$$(-1)^{s} \times (1 + 0.Mantissa) \times 2^{\{-127\}}.$$

➢ If the E = 127, then the exponent of the number will be 0 and the number will be

$$(-1)^{s} \times (1 + 0.Mantissa) \times 2^{\{0\}}$$

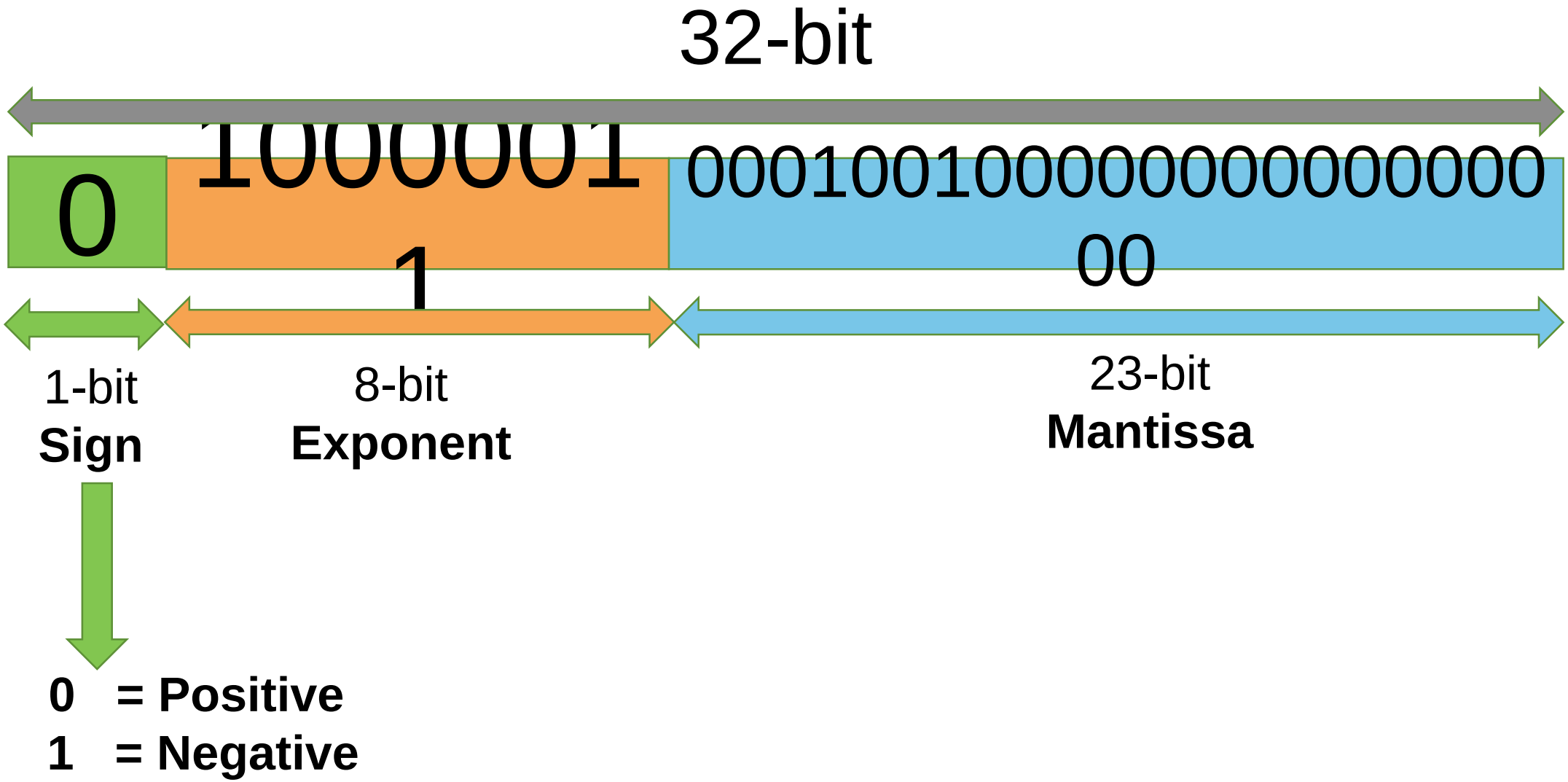
Floating Point Arithmetic IEEE Standard 754 (Single Precision)



$$\text{Sign} \times (1 + \text{Mantissa}) \times 2^{(\text{Exponent} - 127)}$$

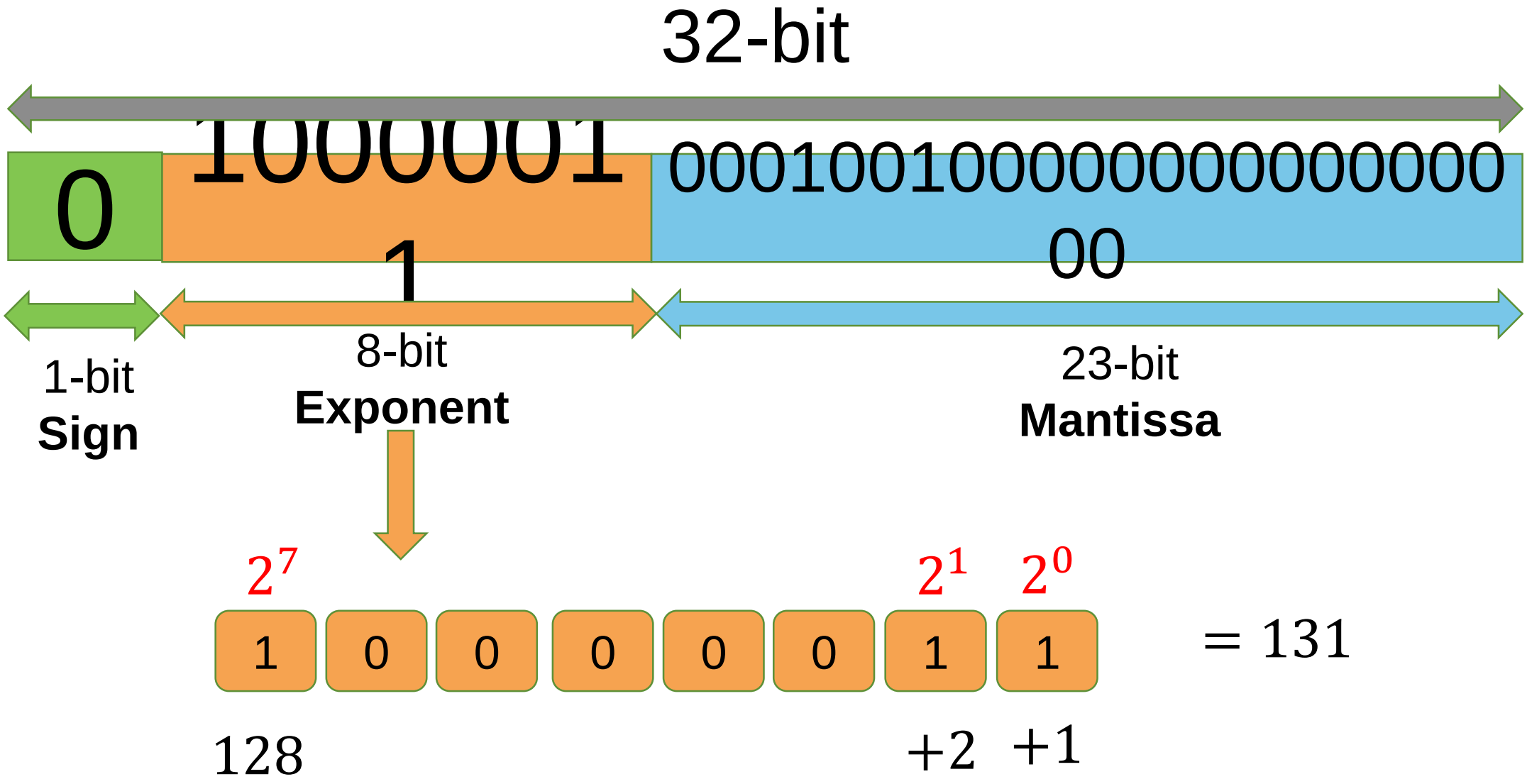
Floating Point Arithmetic IEEE Standard 754

(Single Precision)

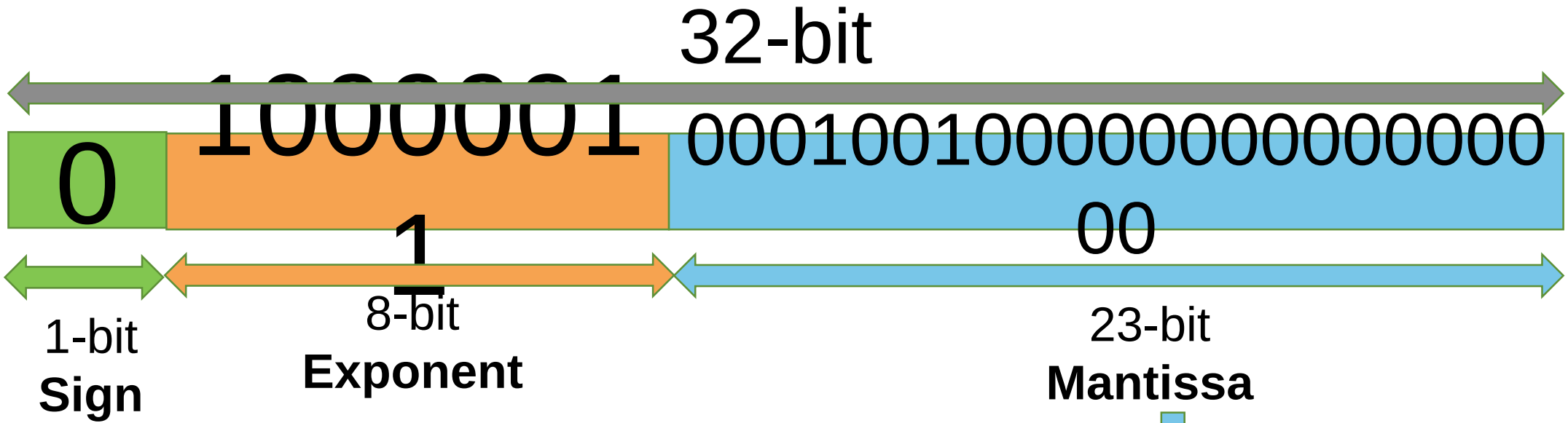


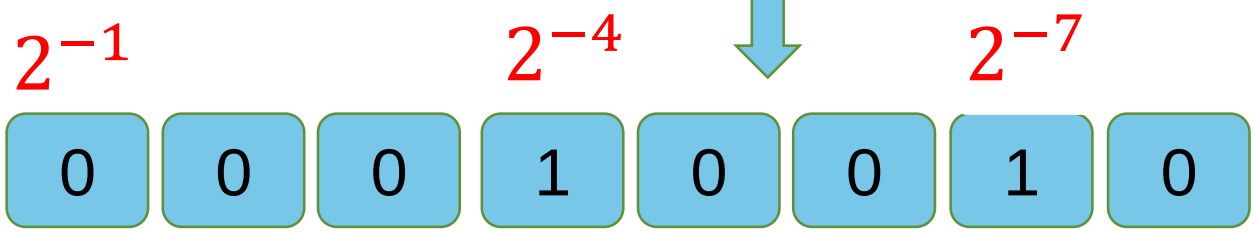
Floating Point Arithmetic IEEE Standard 754

(Single Precision)



Floating Point Arithmetic IEEE Standard 754 (Single Precision)

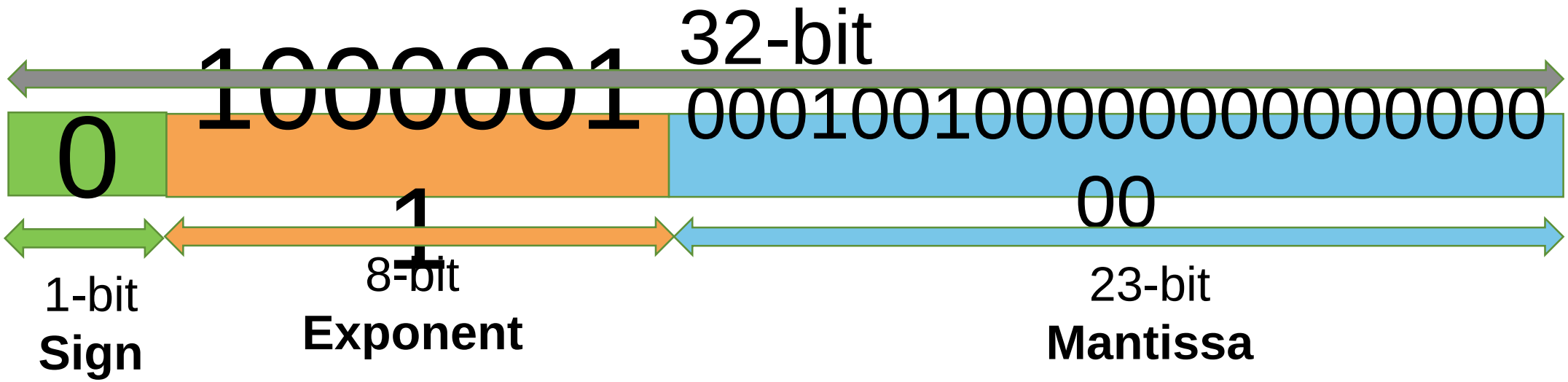




$$\frac{1}{16} + \frac{1}{128} = \frac{9}{128}$$

Floating Point Arithmetic IEEE Standard 754

(Single Precision)



$$\text{Sign}\ (1+\ \text{Mantissa}\ )\ x\ 2^{(\ \text{Exponent}\ -127)}$$

$$+\ \frac{137}{128}\ x\ 2^{4} = \frac{137}{128} x16 = \frac{2192}{128} = 17.125$$

Floating Point Arithmetic IEEE Standard 754 (Single Precision)

Decimal → Binary

17.125

$2 \mid 17$		1
$2 \mid 8$		0
$2 \mid 4$		0
$2 \mid 2$		0
$2 \mid 1$		1
0		

0.125	$\times 2 = 0.25$	0
0.25	$\times 2 = 0.5$	0
0.5	$\times 2 = 1$	1
0		

$(stop\ at\ zero)$

$(when\ answer > 1,\ subtract\ 1)$

Floating Point Arithmetic IEEE Standard 754 (Single Precision)

Decimal → Binary

2	17	1
2	8	0
2	4	0
2	2	0
2	1	1
	0	

← 17.125 →

0.125	× 2 = 0.25	0
0.25	× 2 = 0.5	0
0.5	× 2 = 1	1
0		

10001.001

Floating Point Arithmetic IEEE Standard 754 (Single Precision)

$17.125 =$ **10001.001**

1 0 0 0 1 . 0 0 1 $\times 2^0$

shift the decimal

1 0 0 0 . 1 0 0 1 $\times 2^1$

1 . 0 0 0 1 0 0 1 $\times 2^4$

Floating Point Arithmetic IEEE Standard 754 (Single Precision)

$\pm$	$a_1 a_2 a_3 \ldots a_8$	$b_1 b_2 b_3 \ldots b_{23}$

If exponent bitstring $a_1 \ldots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-124}$
$\downarrow$	$\downarrow$
$(01111111)_2 = (127)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = \cdots = b_{23} = 0$, NaN otherwise

Floating Point Arithmetic IEEE Standard 754 (Single Precision)

- **The smallest positive normalized number that can be stored is represented by**

0	00000001	00000000000000000000000

and we denote this by

$$N_{\min} = (1.000\ldots0)_2 \times 2^{-126} = 2^{-126} \approx 1.2 \times 10^{-38}.$$

- **The largest normalized number (equivalently, the largest finite number) is represented by**

0	11111110	11111111111111111111111

and we denote this by

$$N_{\max} = (1.111\ldots1)_2 \times 2^{127} = (2 - 2^{-23}) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}.$$

Floating Point Arithmetic IEEE Standard 754

(Single Precision)

$17.125 = 10001.001$

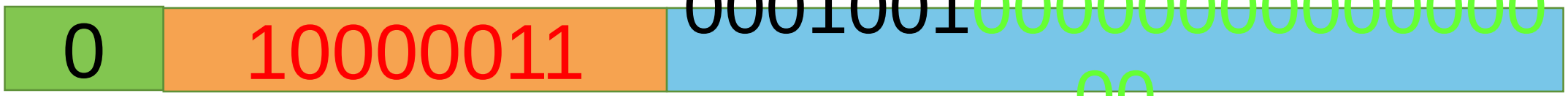
Exponent

$\times 2^{4}$

1 . 0 0 0 1 0 0 1

Mantissa

Convert the exponent to binary : $4 + 127 = 131 \longrightarrow 10000011$



Sign **Exponent** **Mantissa**

Exercise

- Convert to the equivalent Floating Point Arithmetic IEEE Standard 754 (Single Format)

- $+105.625$
- -105.625
- 0.09375
- $+2.7$
- *-123.3 (Homework)*

Exercise

Decimal → Binary

2	105	1
2	52	0
2	26	0
2	13	1
2	6	0
2	3	1
2	1	1
	0	

105.625

$0.625 \times 2 = 0.25 \mid 1$

$0.25 \times 2 = 0.5 \mid 0$

$0.5 \times 2 = 0.0 \mid 1$

1101001.101

Exercise

$105.625 = 1101001.101$

1 1 0 1 0 0 1 . 1 0 1 $\times 2^0$

shift the decimal

1 . 1 0 1 0 0 1 0 1 0 0 1 $\times 2^6$

$1.101001101 \times 2^6 = 1.101001101 \times 2^{133-127}$

$+1.101001101 \times 2^{10000101-01111111}$

In IEEE 754 format:

0 1000 0101 101 0011 0100 0000 0000 0000

Exercise

Consider the decimal number: $+2.7$. The equivalent binary representation is

$$
\begin{aligned}
& +10.10\ 1100\ 1100\ 1100\cdots \\
= \ & +1.010\ 1100\ 1100\cdots \times 2^{1} \\
= \ & +1.010\ 1100\ 1100\cdots \times 2^{128-127} \\
= \ & +1.010\ 1100\ 1100\cdots \times 2^{10000000-01111111}
\end{aligned}
$$

In IEEE 754 format (approximate):

$$\boxed{0\ 1000\ 0000\ 010\ 1100\ 1100\ 1100\ 1100\ 1101}$$

Exercise

Consider the following 32-bit pattern

1 1011 0110 011 0000 0000 0000 0000 0000

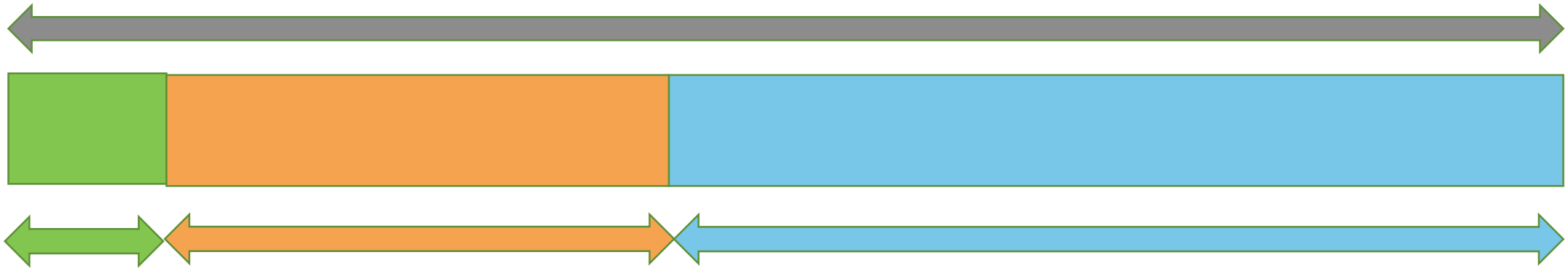
The value is

$$
\begin{aligned}
& (-1)^{1} \times 2^{10110110-01111111} \times 1.011 \\
= \quad & -1.375 \times 2^{55} \\
= \quad & -49539595901075456.0 \\
= \quad & -4.9539595901075456 \times 10^{16}
\end{aligned}
$$

Floating Point Arithmetic IEEE Standard 754

(Double Precision)

64-bit



1-bit
Sign

11-bit
Exponent

52-bit
Mantissa

$$\text{Sign} \times (1 + \text{Mantissa}) \times 2^{(\text{Exponent} - 1023)}$$

Floating Point Arithmetic IEEE Standard 754

(Double Precision)

$\pm$	$a_1 a_2 a_3 \ldots a_{11}$	$b_1 b_2 b_3 \ldots b_{52}$

If exponent bitstring is $a_1 \ldots a_{11}$	Then numerical value represented is
$(00000000000)_2 = (0)_{10}$	$\pm(0.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{-1022}$
$(00000000001)_2 = (1)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{-1022}$
$(00000000010)_2 = (2)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{-1021}$
$(00000000011)_2 = (3)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{-1020}$
$\downarrow$	$\downarrow$
$(01111111111)_2 = (1023)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{0}$
$(10000000000)_2 = (1024)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111111100)_2 = (2044)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{1021}$
$(11111111101)_2 = (2045)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{1022}$
$(11111111110)_2 = (2046)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{1023}$
$(11111111111)_2 = (2047)_{10}$	$\pm\infty$ if $b_1 = \cdots = b_{52} = 0$, NaN otherwise

Floating Point Arithmetic IEEE Standard 754 (Double Precision)

- **The smallest positive normalized number that can be stored is represented by**

$$N_{\min} = 2^{-1022} \approx 2.2 \times 10^{-308}$$

- **The largest normalized number (equivalently, the largest finite number) is represented by**

$$N_{\max} = (2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}.$$

Range of IEEE Floating Point Formats

Format	E_{min}	E_{max}	N_{min}	N_{max}
Single	-126	127	$2^{-126} \approx 1.2 \times 10^{-38}$	$\approx 2^{128} \approx 3.4 \times 10^{38}$
Double	-1022	1023	$2^{-1022} \approx 2.2 \times 10^{-308}$	$\approx 2^{1024} \approx 1.8 \times 10^{308}$

The Extended Format (**Homework**)

Program 1: Convert a real value to its floating point representation

Program 2: Convert a floating point representation to its real value

Precision, Machine Epsilon

- we use the notation p $(precision)$ to denote the number of bits in the significand and e (**machine epsilon**) to mean **the gap between 1 and the next larger floating point number**. The precision of the **IEEE single format is $p = 24$** (including the hidden bit); for **the double format it is $p = 53$** (again, including the hidden bit). When we speak of single precision, we mean the precision of the IEEE single format $(p = 24)$; likewise double precision means the precision of the IEEE double format (p = 53)

- The first single format number larger than 1 is $1 + 2^{-23}$

- The first double format number larger than 1 is $1 + 2^{-52}$

Significant Digits

- The single precision $p = 24$ corresponds to approximately 7 significant decimal digits, since

$$2^{-24} \approx 10^{-7}$$
$$\log_{10}(2^{24}) \approx 7$$

- The double precision $p = 53$ corresponds to approximately 16 significant decimal digits

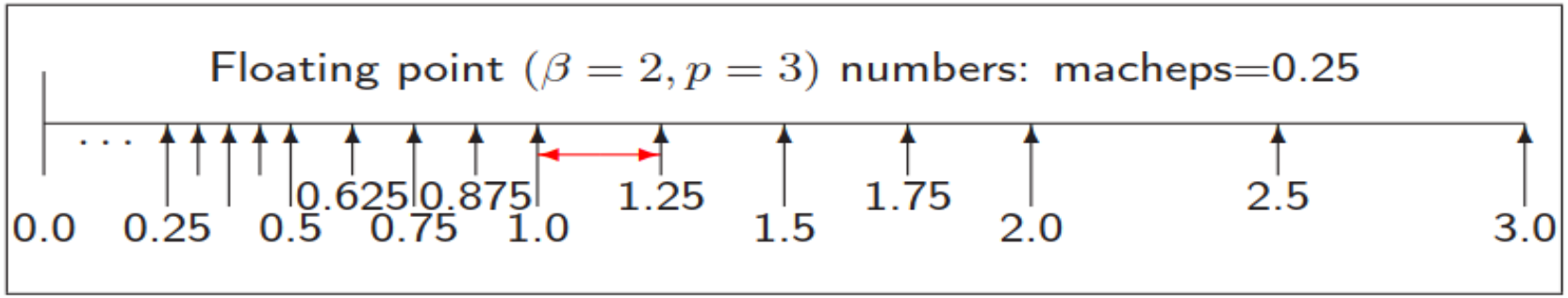
Machine Epsilon

Machine epsilon is defined as the difference between 1.0 and the smallest *representable* number which is greater than one, i.e. 2^{-23} in single precision and 2^{-52} in double (in both cases $\beta^{-(p-1)}$).

> "the difference between 1 and the least value greater than 1 that is representable in the given floating point type"

I.e. machine epsilon is 1 ulp for the representation of 1.0.

Machine epsilon is useful as it gives an upper bound on the relative error caused by getting a floating point number wrong by 1 ulp, and is therefore useful for expressing errors independent of floating point size.



Precision, Machine Epsilon

- **Precision** (p): the number of bits in the mantissa + 1

Returning again to the IEEE formats, if x is in the normalized range, with

$$x = \pm\left(1.\, b_1 b_2 \ \ldots b_{p-1} b_p\, b_{p+1} \ \ldots\right)_2 \times 2^E$$

the absolute rounding error associated with x is less than the gap between x_- and x_+ , regardless of the rounding mode, The relative rounding error associated with a nonzero number x is defined by

$$relerr(x) = |\delta|$$

$$\delta = \frac{round(x)}{x} - 1 = \frac{round(x) - x}{x}$$

- Machine Epsilon

$$relerr(x) = |\delta| = \frac{|round(x) - x|}{|x|} < \frac{2^{-(p-1)} \times 2^E}{2^E} = 2^{-(p-1)} = \varepsilon$$

Precision, Machine Epsilon

$$\text{round}(x) = x(1+\delta), \quad \text{where} \quad |\delta| < 2^{-(p-1)},$$

with $|\delta| < 2^{-p}$ when the rounding mode is *round to nearest*. The quantity $|\delta|$ is called the relative rounding error and its size depends only on the precision p of the floating point system and not on the size of x. The absolute rounding error $|x - \text{round}(x)|$ does depend on the size of x, since the gap between floating point numbers is larger for larger numbers. These results apply to normalized numbers. Subnormal numbers are less accurate. Because of its greater precision, the double format is preferred for most scientific computing applications.

Machine Epsilon **(in Python)**

Machine epsilon: ϵ_m

- Let us have a FPA with p bits of mantissa,
- We define $\epsilon_m = 2^{-p+1}$, the *Machine epsilon* by:
- ϵ_m is the smallest floating point number such that:

$$fl(1.0) \neq fl(1.0 + \epsilon_m)$$

Format	Precision	Machine Epsilon
Single	$p = 24$	$\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$
Double	$p = 53$	$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$
Extended (Intel)	$p = 64$	$\epsilon = 2^{-63} \approx 1.1 \times 10^{-19}$

(IEEE Rounding)

$$x = \pm(b_0.b_1b_2 \ \ldots b_{p-1})_2 \times 2^E$$

where p is the precision of the floating − point system with, for normalized numbers

$$b_0 = 1, \qquad E_{min} \leq E \leq E_{max}$$

We say that a real number x is in the normalized range of the floating point system if

$$N_{min} \leq |x| \leq N_{max}$$

Let us define x_- to be the floating point number closest to x that is less than or equal to x, and define x_+ to be the floating point number closest to x that is greater than or equal to x.

For example, consider the toy floating point number system again. If $x = 1.7, then x_- = 1.5$ and $x_+ = 1.75$

(IEEE Rounding)

- let x be a positive number in the normalized range, and write x in the normalized form

$$x = (1.b_1 b_2 \ \ldots b_{p-1} b_p \, b_{p+1} \ \ldots)_2 \times 2^E$$

It follows that the closest floating point number less than or equal to x is

$$x_- = (b_0.b_1 b_2 \ \ldots b_{p-1})_2 \times 2^E$$

$$x_+ = (b_0.b_1 b_2 \ \ldots b_{p-1})_2 + (0.00 \ \ldots 01)_2 \times 2^E$$

$therefore\ also\ the\ next\ one\ that\ is\ bigger\ than\ \boldsymbol{x}\ (\boldsymbol{which\ must\ lie\ between\ x\_and\ x_+})$.

$Here\ the\ 1in\ the\ increment\ is\ in\ the\ (p - l)th\ place\ after$
$the\ binary\ point, so\ the\ gap\ between\ x\_and\ x_+\ \ is$

$2^{-(p-1)} \times 2^E$

Note that this quantity is the same as ulp(x_-)

Floating point operations

(IEEE Rounding)

(IEEE Rounding)

1. Round down (sometimes called round towards $-\infty$) : $round(x) = x_-$
2. Round up (sometimes called round towards ∞) : $round(x) = x_+$
3. Round towards zero: $round(x) = x_- \ \ if \ x > 0; round(x) = x_+ \ \ if \ x < 0$
4. Round to nearest: $round(x) = x_- \ \ or \ \ x_+$ whichever is nearer to x (unless $|x| > N_{max}$).

If x is positive, then x_- is between zero and x, so **round down and round towards zero have the same effect**. If x is negative, then x_+ is between zero and x;, so **round up and round towards zero have the same effect**. In both cases, round towards zero simply requires truncating the binary expansion, unless x is outside the normalized range. The rounding mode that is almost always used in practice is round to nearest.

Cancellation

Consider the two numbers $x = 3.141592653589793$

$y = 3.141592653585682$

The first number, x , is a 16-digit approximation to π, while the second number agrees with π to only 12 digits. Their difference is

$$z = x - y = 0.000000000004111 = 4.111 \times 10^{-12}$$

This difference is in the normalized range of the IEEE single format. However, if we compute the difference $z = x - y$ in, (C, Python,…) program, using the single format to store the variables x and y before doing the subtraction, and display the result to single precision,

we find that the result displayed for z is `0.000000e+00`

Cancellation

- The reason is simple enough. The input numbers x and y are first converted from decimal to the single binary format; they are not exact floating point numbers, **so the decimal to binary conversion requires some rounding. Because they agree to 12 digits, both x and y round to exactly the same single format number**. Thus, all bits in their binary representation cancel when the subtraction is done; we say that we have a complete loss of accuracy in the computation $z = x - y$.
- If we use the double format to store x and y and their difference z, and if we display the result to double precision, we find that z has the value

```
4.110933815582030e-12
```

This agrees with the exact answer to about four digits, but what is the meaning of the other digits? The answer is that the result displayed is the correctly rounded difference of the double format representations of x and y

It is important to realize that in this case, we may ignore all but the first four or five digits of. The rest may be viewed as garbage, in the sense that they do not reflect the original data x and y. We say that we have a partial loss of accuracy in the computation z = x — y. Regardless of whether the loss of accuracy is complete or partial, the phenomenon is called **cancellation**. It occurs when one number is subtracted from another number that is nearly equal to it. Equivalently, it occurs if two numbers with opposite sign but nearly equal magnitude are added together.

Cancellation in Python

```
1 x = 0.12345678912345678 91
2 print(x, "Diff:  ", x - 0.12345678912345678)
```

Associativity & Absorption error

Associativity ? Lost !!!!!!!

Associativity of $+$ (remember):

$$\forall x, y, z : \qquad x + (y + z)) = (x + y) + z.$$

This can be false with FPA!

Absorption error

$$\exists x, y \neq 0 : fl(x + fl(y)) = fl(x)$$

. We will say that x absorbs y.
Or more exactly $fl(x)$ absorbs $fl(y)$.

Summing a Finite Series

Summing a Finite Series

We've already seen $(a + b) + c \neq a + (b + c)$ in general. So what's the best way to do (say)

$$\sum_{i=0}^{n} \frac{1}{i + \pi} \; ?$$

Of course, while this formula mathematically sums to infinity for $n = \infty$, if we calculate, using `float`

$$\frac{1}{0 + \pi} + \frac{1}{1 + \pi} + \cdots + \frac{1}{i + \pi} + \cdots$$

until the sum stops growing we get 13.8492260 after 2097150 terms.

But is this correct? Is it the only answer?

Summing a Finite Series

Summing a Finite Series (2)

Previous slide said: using `float` with $N = 2097150$ we got

$$\frac{1}{0+\pi} + \frac{1}{1+\pi} + \cdots + \frac{1}{N+\pi} = 13.8492260$$

But, by contrast

$$\frac{1}{N+\pi} + \cdots + \frac{1}{1+\pi} + \frac{1}{0+\pi} = 13.5784464$$

Using double precision (64-bit floating point) we get:

- forward: 13.5788777897524611
- backward: 13.5788777897519921

So the backwards one *seems* better. But why?

When adding $a+b+c$ it is generally more accurate to sum the smaller two values and then add the third.

Compensated summation algorithms

Compensated summation algorithms

1. In numerical analysis, the Kahan summation algorithm (also known as *compensated summation*) significantly reduces the numerical error in the total obtained by adding a sequence of finite precision floating point numbers, compared to the obvious approach.
2. This is done by keeping a separate running compensation (a variable to accumulate small errors).
3. In particular, simply summing n numbers in sequence has a worst-case error that grows proportional to n, and a root mean square error that grows as n $\sqrt{n}$ for random inputs (the roundoff errors form a random walk).
4. With compensated summation, the worst-case error bound is independent of n, so a large number of values can be summed with an error that only depends on the floating-point precision.

Kahan summation

Kahan summation algorithm, also known as compensated summation and summation with the carry algorithm, is used to **minimize the loss of significance** in the total result obtained by adding a sequence of finite-precision floating-point numbers. This is done by keeping a separate running compensation (a variable to accumulate small errors).

```python
# Python3 program to illustrate the
# floating-point error

def floatError(no):
    sum = 0.0
    for i in range(10):
        sum = sum + no
    return sum

if __name__ == '__main__':
    print(floatError(0.1))
```

Output:

```
0.9999999999999999
```

Note: The expected value for the above implementation is 1.0 but the value returned is 0.9999999999999999. Therefore, a method to reduce this error by using Kahan's Summation algorithm is discussed.

Kahan summation

Kahan Summation Algorithm: The idea of the Kahan summation algorithm is to compensate for the floating-point error by keeping a separate variable to store the running time errors as the arithmetic operations are being performed. This can be visualized by the following pseudocode:

```
function KahanSum(input)
    var sum = 0.0
    var c = 0.0
    for i = 1 to input.length do

        var y = input[i] - c
        var t = sum + y
        c = (t - sum) - y

        sum = t

    next i

    return sum
```

This uses a second variable which approximates the error in the previous step which can then be used to compensate in the next step.

For the example in the notes, even using single precision it gives:

- Kahan forward: 13.5788774
- Kahan backward: 13.5788774

which is within one ulp of the double sum 13.57887778975...

The ”Gentleman-Marovich Code”

1. *Mathematically*: the two loops are theoretically infinite loops so they are looping forever

The ”Gentleman-Marovich Code”

```
Algorithm:
Input: A=1.0 ; B=1.0;
Output: A,B
Begin:
# A=1.0
while((A+1.0)-A)-1.0==0:
    A=2*A;
# B=1.0;
while((A+B)-A)-B==0:
    B=B+1.0;
# Results
Return[A,B];
End.
```

The ”Gentleman-Marovich Code”

The Gentleman-Marovich Code

```
# The Gentleman-Marovich Code
A=1.0

while ((A+1.0)-A)-1.0==0:
    A=2*A

# A is "always" a power of two!
# We try to compute  the SMALLEST  value of n such that fl(1+2^n)==fl(2^n)
print(A)
print(A,"  ",log(A)/log(2))
print(A,"  ",log(A,2))
```

```
9007199254740992.0
9007199254740992.0    53.0
9007199254740992.0    53.0
```

The Pichat's algorithm

Session 1: The Pichat's algorithm

Algorithm 1 : "Pichat Algorithm"

Input: A finite sequence of real $\{a_1, \cdots a_n\}$;

Ouput: An "good" approximation of $fl(S) = \sum_{i=1}^{n} fl(a_i)$;

Begin:

$S_1 = fl(a_1)$;

For $k = 1$ **to** $n - 1$ **Do**

$S_{k+1} = S_k + a_{k+1}$;

If $S_k <= a_{k+1}$ **Then** $e_k = fl((S_{k+1} - S_k) - a_{k+1})$;

Else $e_k = fl((S_{k+1} - a_{k+1}) - S_k)$;

EndOfFor ;

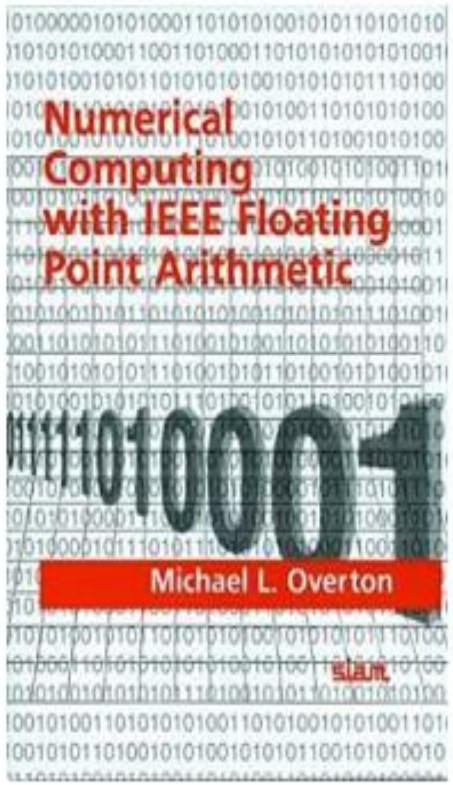
Correction ;

$S_n := fl(S_n + \sum_{i=1}^{n-1} e_i)$;

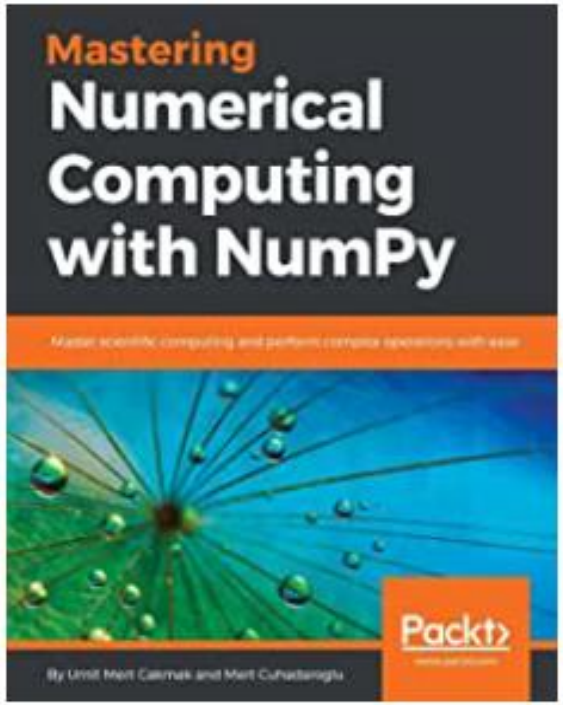
End.

References

- Very well written
- "The" book on the IEEE Floating Point Arithmetic p754 Standard



- A very interesting book about Numerical Algorithms with Numpy
- With a lot of interesting problems



- A very good book on Accuracy
- with a lot of Numerical Algorithms
- With a lot of interesting problems