Floating-point Arithmetic

is not Real

Ianna Osborne, 19th of July 2023

Outline

Goals

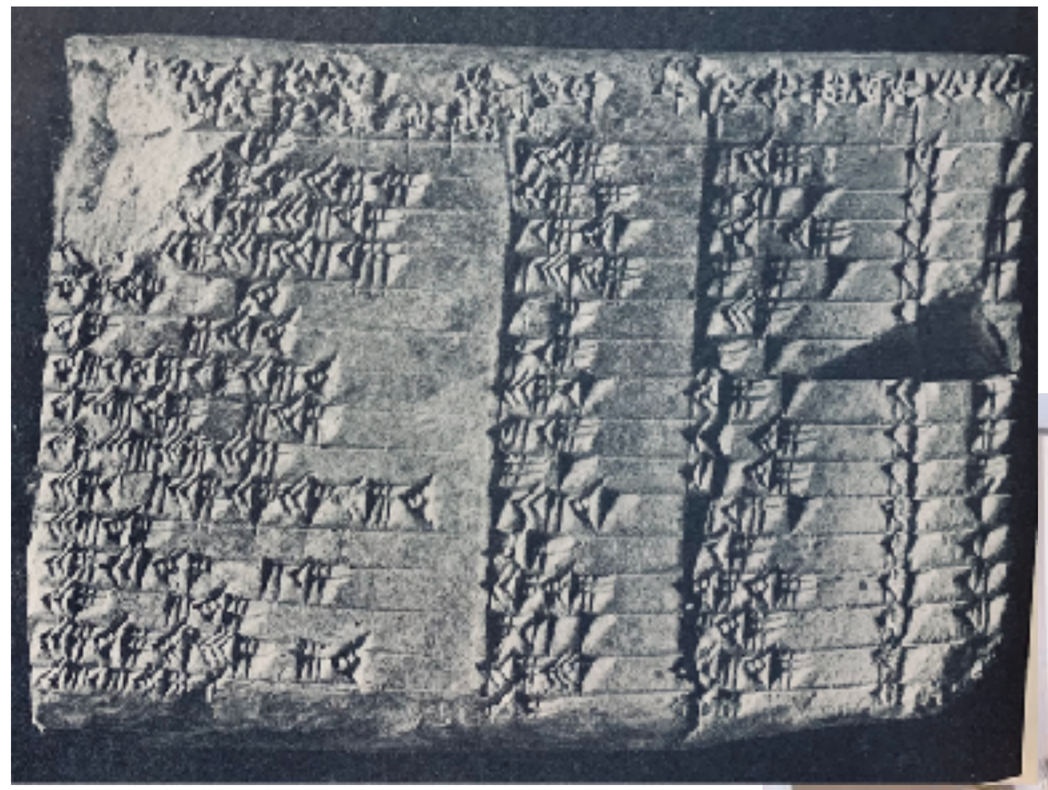
- Basic understanding of computer representation of numbers
- Basic understanding of floating-point arithmetic
- Consequences of floating-point arithmetic for scientific computing
- Basic understanding about fast math

https://github.com/ianna/CoDaS-HEP2023-FloatingPoint

exercises/Floating-point-arithmetics.ipynb
or
exercises/evaluated/Floating-point-arithmetics-evaluated.ipynb

Floating-point Arithmetic Timeline

"…the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!"
https://grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/ieee-computer.pdf



Babylonians worked with floating-point sexagesimal numbers

By Venusianer, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3632073

- Average calculation speed: addition – 0.8 seconds, multiplication – 3 seconds[1]
- Arithmetic unit: Binary floating-point, 22-bit, add, subtract, multiply, divide, square root[1]

1750 B.C.

1941 Zeus Z3

1960's a mainframe era

1964 **IBM 360**
CDC 6600
DEC 10

DEC PDP-11
1970's boom of minicomputers

1976

1977 **IEEE p745**

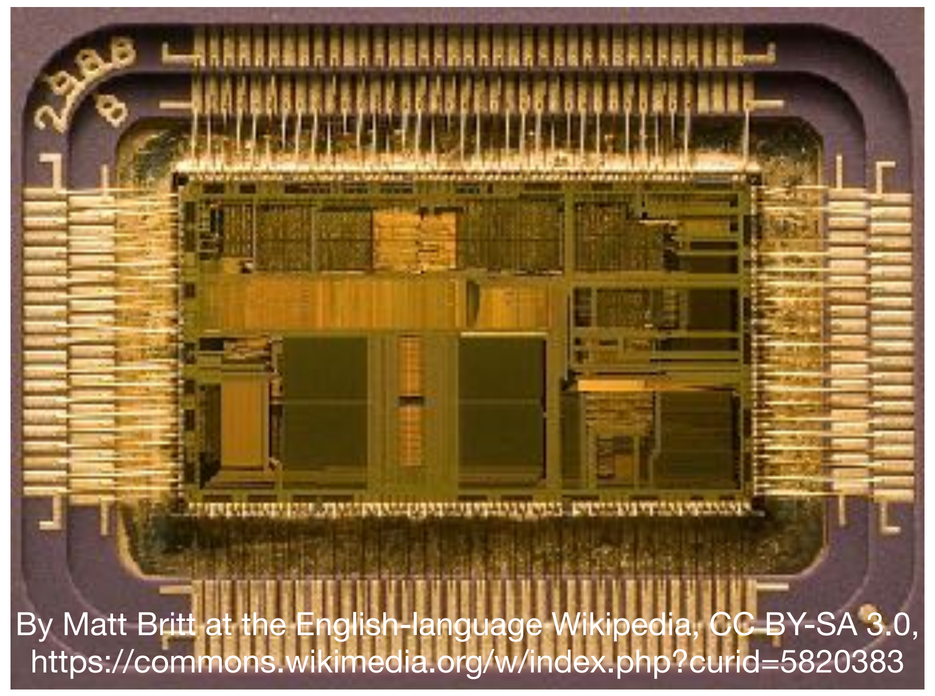
1985 IEEE 754-1985

IEEE 754-1987

Intel began to design a floating-point co-processor for its i8086/8 and i432 microprocessors

Intel 80486

the first tightly-pipelined[c] x86 design as well as the first x86 chip to include more than one million transistors. It offered a large on-chip cache and an integrated floating-point unit.


By Matt Britt at the English-language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5820383

1989 The floating point unit fully conforms to the ANSI/IEEE standard 754-1985 for floating point arithmetic.

IEEE 754-2008

IEEE 754-2019

binary and decimal floating-point arithmetic

subjected to review at least every 10 years

you are here

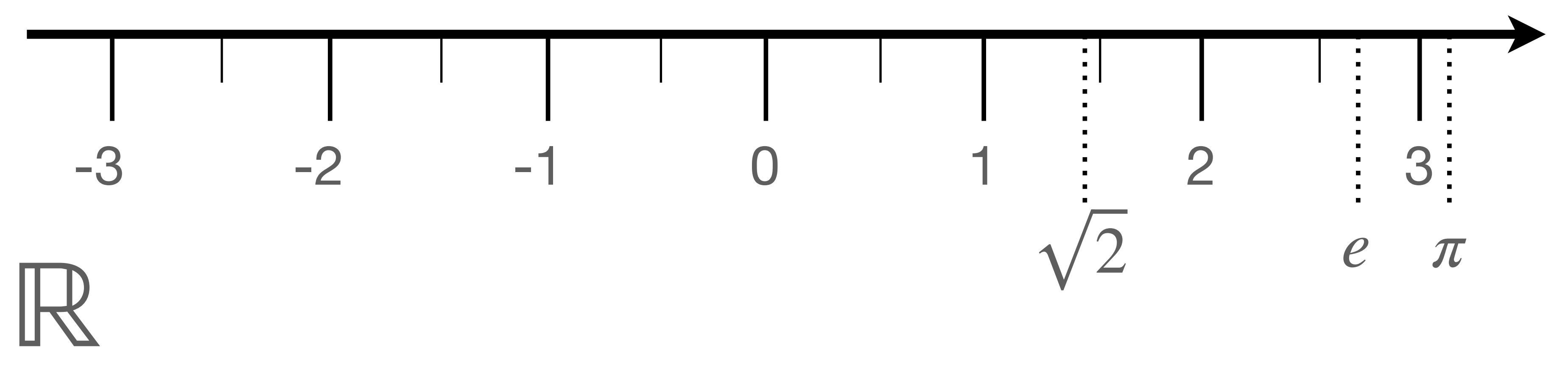
IEEE 754-2029

"new kinds of computational demands might eventually encompass new kinds of standards, particularly for fields like artificial intelligence, machine vision and speech recognition, and machine learning. Some of these fields obtain greater accuracy by processing more data faster rather than by computing with more precision – rather different constraints from those for traditional scientific computing."

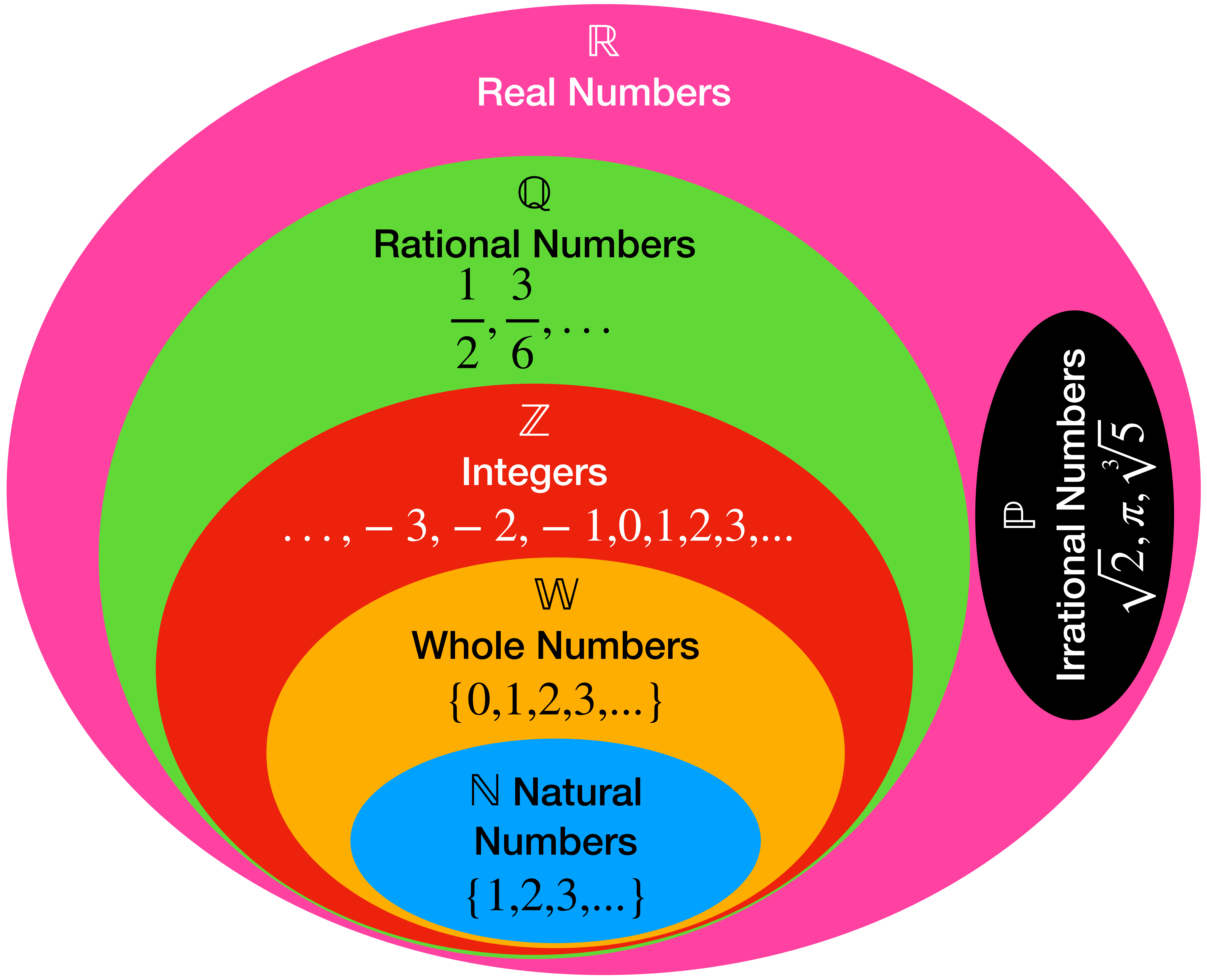
- each hardware manufacturer had its own type of floating point
- different machines from the same manufacturer might have different types of floating point
- when floating point was not supported in the hardware, the different compilers emulated different floating point types

Real Numbers

in mathematics



Real numbers can be thought of as all points on a line called the number line or real line, where the points corresponding to integers are equally spaced



Scientific Notation

real number representation

significand exponent

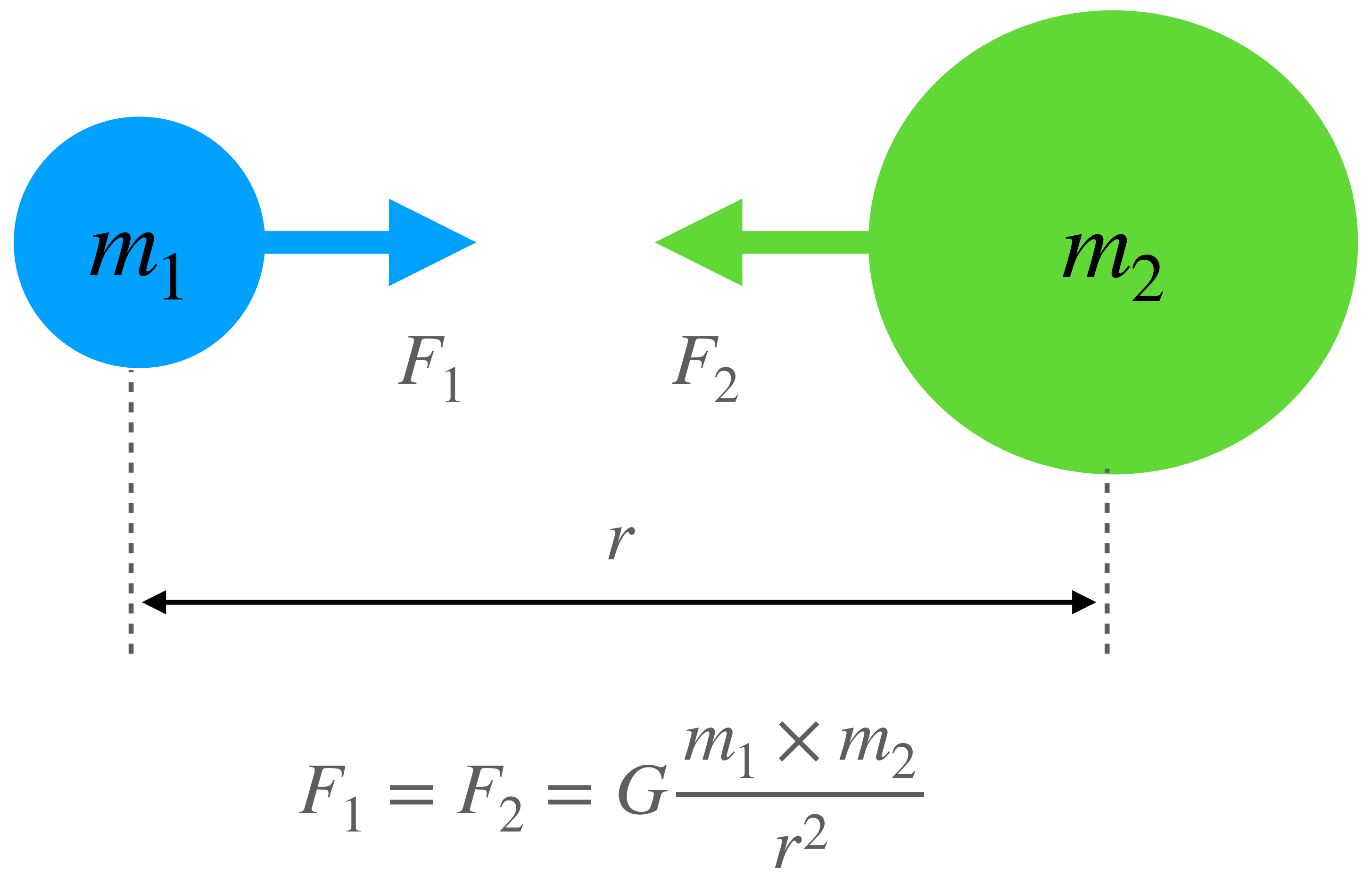
$$G \approx \underbrace{6.674}_{\text{significand}} \times \underbrace{10}_{\text{radix}}^{\overbrace{-11}^{\text{exponent}}} m^3 \cdot kg^{-1} \cdot s^{-2}$$

radix

The above expression means

$$0.00000000006674 \text{ or } 6.674e - 11$$

the exponent shows by how far to "float" the decimal point



Scientific Notation

real number representation

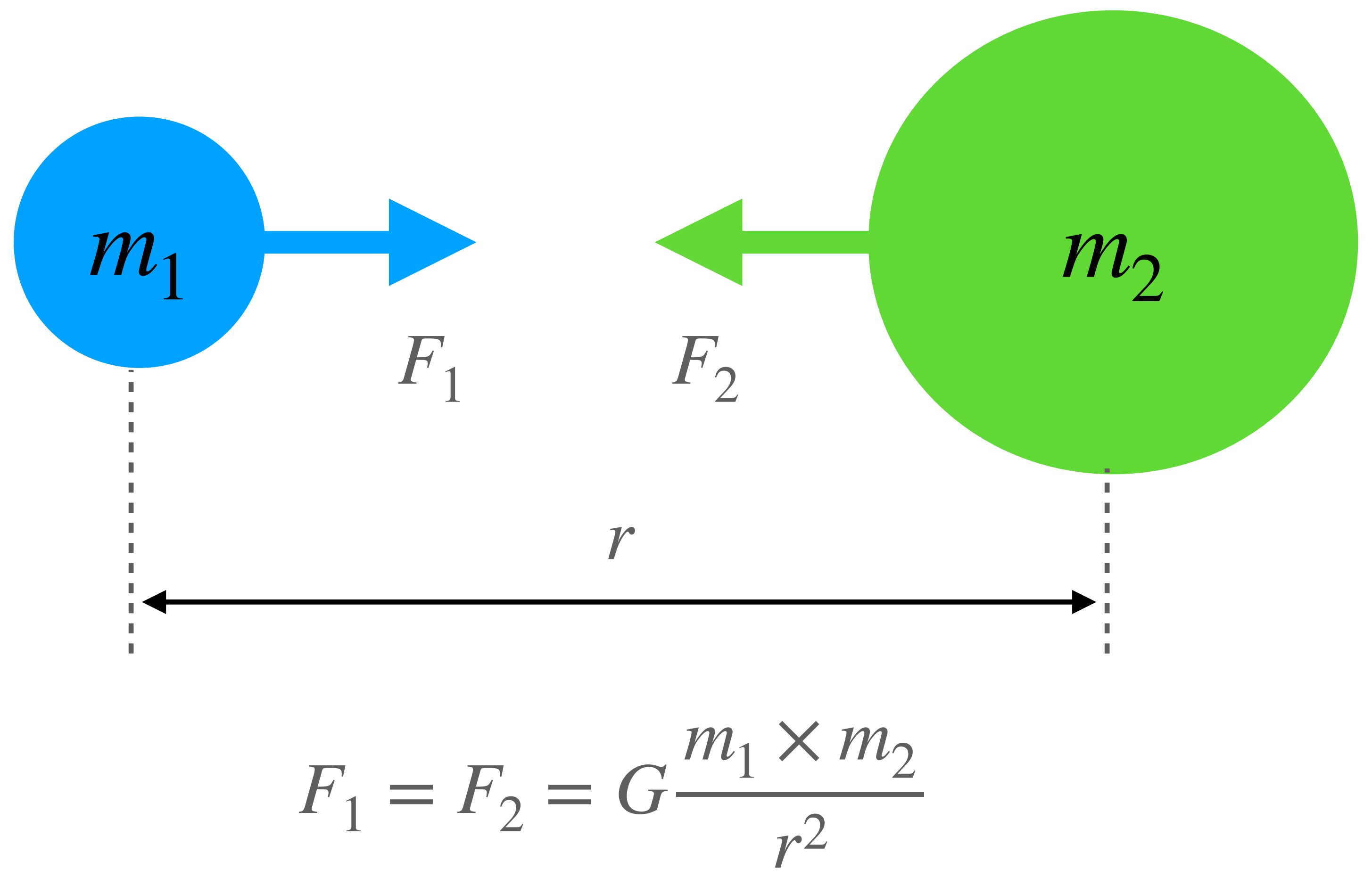
A real number can be represented by

$$x = (-1)^s \sum_{i=0}^{\infty} d_i b^{-i} \cdot b^e,$$

where $s \in \{0,1\}$, $b \geq 2$, $i \in \{0,1,2,...\}$, $d_i \in \{0,...,b-1\}$
and $d_0 > 0$ when $x \neq 0$, b and e are integers

For example: $G \approx 6.674 \times 10^{-11} m^3 \cdot kg^{-1} \cdot s^{-2}$

$$(6 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2} + 4 \times 10^{-3}) \times 10^{-11}$$



Sets of Floating-point Data

finite computer representation

- Floating-point number system is defined by the four natural numbers:
  - $b \geq 2$, the radix, (2 or 10)
  - $p \geq 1$, the precision (the number of of digits in the significand)
  - e_{max}, the largest possible exponent
  - e_{min}, the smallest possible exponent, (shall be $1 - e_{max}$ for all formats)
- Notation:

$$F(b, p, e_{min}, e_{max})$$

In computing, the binary (base-2), octal (base-8) and hexadecimal (base-16) bases are most commonly used.

Floating-point Number Systems

a finite subset of $\mathbb{R}$

The set $F(b, p, e_{min}, e_{max})$ of real numbers represented by this system consists of all floating-point numbers of the form:

$$(-1)^s \sum_{i=0}^{\boxed{p-1}} d_i b^{-i} \cdot b^e,$$

$s \in \{0,1\}$, $d_i \in \{0,...,b-1\}$ for all i, $e \in \{e_{min}, \ldots, e_{max}\}$.

represented in radix b:

$$\pm d_0 \,.\, d_1 \ldots d_{p-1} \times b^e,$$

Floating-point Number Systems

Representations of the decimal number 0.1 (with $b = 10$)

1.0×10^{-1}, 0.1×10^{0}, 0.01×10^{1}, …

Different representations due to choice of exponent

Normalized Representation

Normalized number:

$$\pm d_0 \,.\, d_1 \ldots d_{p-1} \times b^e, \; d_0 \neq 0$$

The normalized representation is unique and therefore preferred

The number 0, as well as all numbers smaller than $b^{e_{min}}$, have no normalized representation

Set of normalized numbers:

$$F^*(b, p, e_{min}, e_{max})$$

Quiz:

How many floating point numbers do the systems $F^*(b, p, e_{min}, e_{max})$ and $F(b, p, e_{min}, e_{max})$ contain?

For each exponent, $F^*(b, p, e_{min}, e_{max})$ has $b - 1$ possibilities for the first digit, and b possibilities for the remaining $p - 1$digits

The size of $F^*(b, p, e_{min}, e_{max})$ is therefore

$$2(e_{max} - e_{min} + 1)(b - 1)b^{p-1},$$

if we take the two possible signs into account.

$F(b, p, e_{min}, e_{max})$ has extra nonnegative numbers of the form

$$0.d_1 \dots d_{p-1} 2^{e_{min}},$$

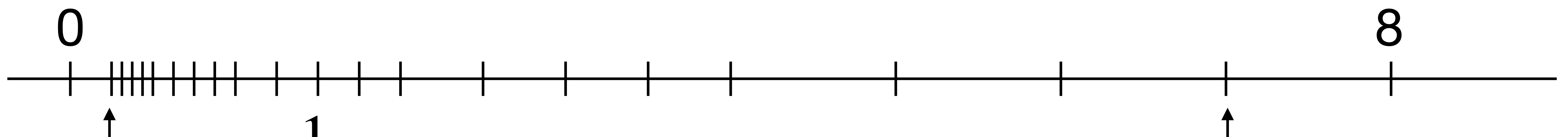
and there are b^{p-1}. Adding the non-positive ones and subtracting 1 for counting 0 twice, we get

$$2b^{p-1} - 1 \text{ extra numbers.}$$

Normalized Representation

$F^*(2,3,-2,2)$

$d_0 . d_1 d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7



$1.00 \cdot 2^{-2} = \frac{1}{4}$

$1.11 \cdot 2^{2} = 7$

Value Range

notebook exercises

- *Exercise 1:* Check machine limits for integer and floating-point types
- *Exercise 2:* Write a function
  - input two floating-point numbers and their difference
  - check whether this is indeed the correct difference
  - test it for the following types:
    - float, np.float32, np.float64
    - and values: 1. and 1.5; 1.1 and 1.

Value Range

notebook exercises solution

- *Exercise 2:*

```
import numpy as np

def diff(n1, n2, d):
    print("First number:", n1)
    print("Second number:", n2)
    print("Their difference:", d)
    print("Computed difference - input difference = ", n1 - n2 - d)
```

```
x1 = np.float32(1.5)
x2 = np.float32(1.0)
x_diff = np.float32(0.5)

diff(x1, x2, x_diff)
```

```
First number: 1.5
Second number: 1.0
Their difference: 0.5
Computed difference - input difference =  0.0
```

```
x1 = np.float32(1.1)
x2 = np.float32(1.0)
x_diff = np.float32(0.1)

diff(x1, x2, x_diff)
```

```
First number: 1.1
Second number: 1.0
Their difference: 0.1
Computed difference - input difference =  2.2351742e-08
```

Binary and Decimal Systems

and relative error

- Internally the computer computes with $b = 2$ (binary system)
- Literals and inputs have $b = 10$ (decimal system)
- Inputs have to be converted!
- If we are not able to represent a real number x exactly as a binary floating point in the system, it is natural to approximate it by the floating point number nearest to x
- The relative error made in approximating x with its nearest point number $\hat{x}$ is called *machine epsilon*

Conversion Decimal to Binary

exercise: assume, $0 < x < 2$

Binary representation:

$$x = \sum_{i=-\infty}^{0} b_i 2^i = b_0 \,.\, b_{-1} b_{-2} b_{-3} \ldots$$

$$= b_0 + \sum_{i=-\infty}^{-1} b_i 2^i = b_0 + \sum_{i=-\infty}^{0} b_{i-1} 2^{i-1}$$

$$= b_0 + \underbrace{\left(\sum_{i=-\infty}^{0} b_{i-1} 2^i \right)}_{x' = b_{-1}.b_{-2}b_{-3}b_{-4}} /2$$

Hence: $x' = b_{-1} \,.\, b_{-2} b_{-3} b_{-4} \ldots = 2 \cdot (x - b_0)$

- Step 1 (for x): Compute b_0:

$$b_0 = \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

- Step 2 (for x): Compute $b_{-1}, b_{-2}, \ldots$
- Go to Step 1 (for $x' = 2 \cdot (x - b_0)$)

Binary Representation of 1.1_{10}

exercise solution

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

$\implies$ $1.0\overline{0011}$, periodic, *not* finite

Binary Number Representation of 1.1 and 0.1

notebook exercises

- Binary number representation of 1.1 and 0.1 are not finite, hence there are errors when converting into a finite binary floating-point system

- 1.1f and 0.1f do not equal 1.1 and 0.1, but are slightly inaccurate approximation of these numbers

Binary Number Representation of 1.1 and 0.1

notebook exercises solution

```
0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

```
x1 = 0.1
x2 = 0.10000000000000001
x3 = 0.1000000000000000055511151231257827021181583404541015625
```

```
eval(repr(x1)) == x1
```

```
True
```

```
eval(repr(x1)) == x2
```

```
True
```

```
eval(repr(x1)) == x3
```

```
True
```

```
a = 1.0
b = 0.1
c = 1.1
c - a - b
```

```
8.326672684688674e-17
```

```
from fractions import Fraction

Fraction(1.1)
```

```
Fraction(2476979795053773, 2251799813685248)
```

```
Fraction(1.1).limit_denominator()
```

```
Fraction(11, 10)
```

PRINCETON UNIVERSITY

Arithmetic Operations

exercise: add two binary numbers $1.111 \cdot 2^{-2}$ and $1.011 \cdot 2^{-1}$

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ +1.011 \cdot 2^{-1} \\ \hline \end{array}$$

- $b = 2, p = 4$
- Align the two numbers such that they have the same exponent: $$\begin{array}{r} 1.111 \cdot 2^{-2} \\ +10.110 \cdot 2^{-2} \end{array}$$
- Add up the two significands: $100.101 \cdot 2^{-2}$
- Renormalize: $1.00101 \cdot 2^{0}$
- The exact result is not representable with $p = 4$ $\quad \boxed{1.001 \cdot 2^{0}}$

PRINCETON UNIVERSITY

The IEEE Standard 754

- Defines floating-point number systems and their rounding behaviour (radix binary and decimal [IEEE 2019])
- Is used nearly everywhere

$F(b, p, e_{min}, e_{max})$

- Single precision (float) numbers:

$$F^*(2, 24, -126, 127) \text{ (32-bit) plus } 0, \infty, \ldots$$

- Double precision (double) numbers:

$$F^*(2, 53, -1022, 1023) \text{ (64-bit) plus } 0, \infty, \ldots$$

- All arithmetic operations round the exact result to the next representable number

IEEE 754-2019 Binary Interchange Formats

- Interchange formats support the exchange of floating-point data between implementations
- This standard defines binary interchange formats of widths 16, 32, 64, and 128 bits, and in general for any multiple of 32 bits of at least 128 bits

Parameter	**binary16**	**binary32**	**binary64**	**binary128**	**binary{k}**$(k \geq 128)$
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - round(4 \times \log_2(k)) + 13$
emax, maximum exponent *e*	15	127	1023	16383	$2^{(k-p-1)} - 1$
Encoding parameters					
bias, E - e	15	127	1023	16383	*emax*
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$round(4 \times \log_2(k)) - 13$
t, trailing significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width i bits	16	32	64	128	$1 + w + t$

The IEEE Standard 754

single-precision value range

$$F^*(2, 24, -126, 127)$$

$F(b, p, e_{min}, e_{max})$

- 1 sign bit
- 23 bits for the significand: the leading bit is 1 and not stored - in a normalized binary floating point system the first digit is always $2^0 = 1$
- 8 bits for the exponent: $254 = 2^8 - 2$ possible exponents, 2 special values: $0, \infty, \ldots$

$\implies$ 32 bits in total

The IEEE Standard 754

double value range

$$F^*(2, 53, -1022, 1023)$$

$F(b, p, e_{min}, e_{max})$

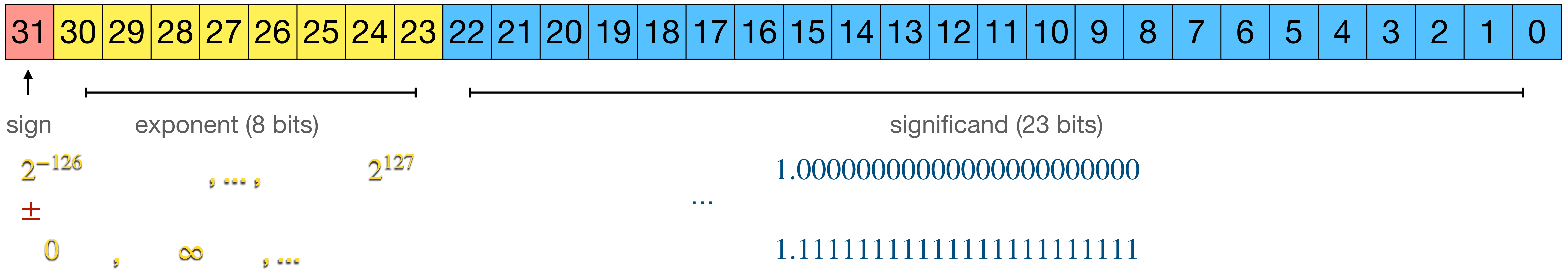
- 1 sign bit
- 52 bits for the significand - leading bit is 1 and not stored
- 11 bits for the exponent: $2046 = 2^{11} - 2$ possible exponents, 2 special values: $0, \infty, \ldots$

$\Longrightarrow$ 64 bits in total

Floating Point Representation

example: binary32

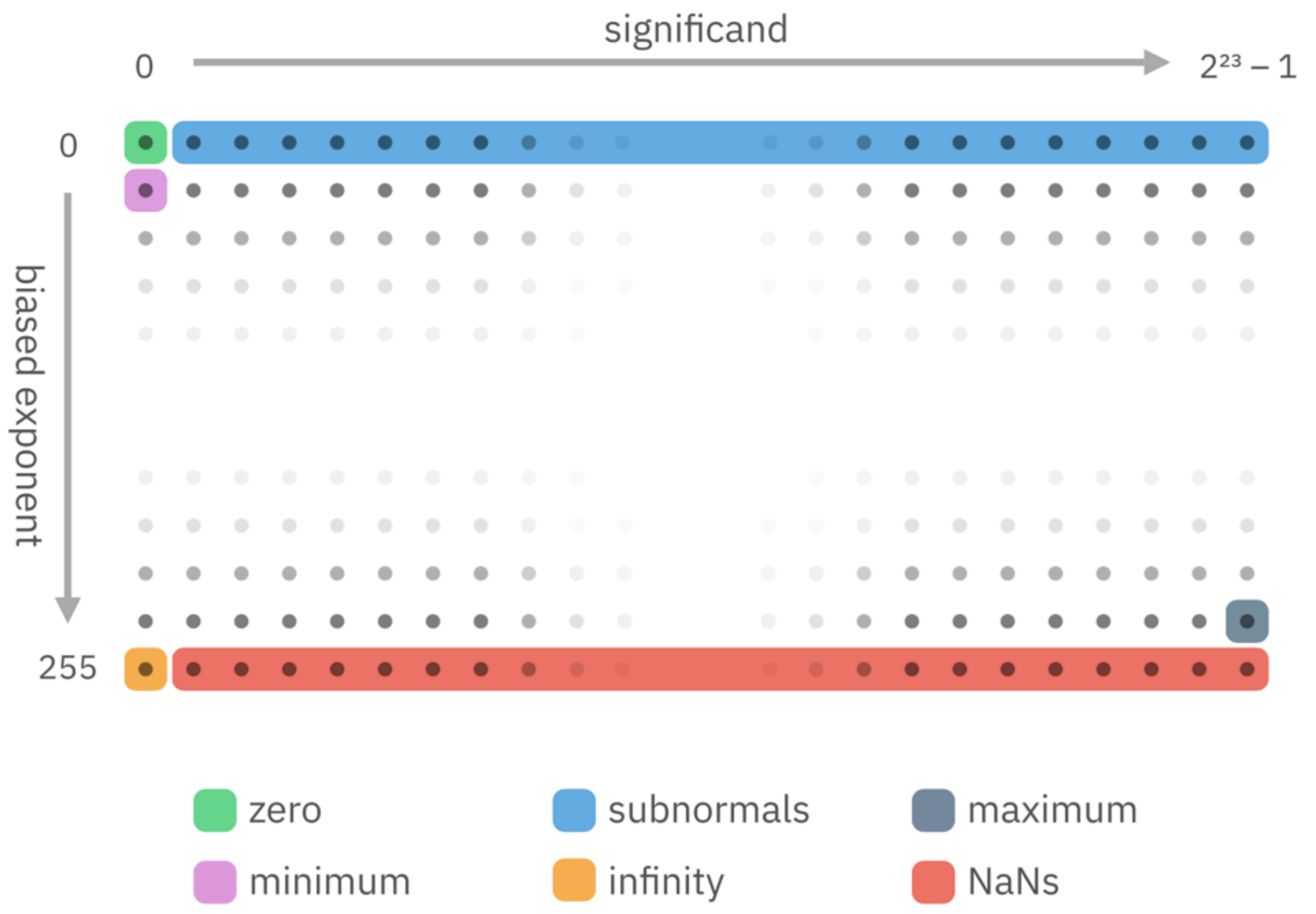
- The exponent is *biased* to avoid special handling of negative exponent values: a fixed *bias* value of 127 is added



Exponent	Fraction zero	Fraction non-zero	Numerical value represented
00000000	± 0	Subnormal numbers	$(-1)^{sign} \times 2^{-126} \times 0.$fraction
not all 0 or 1	Normalized numbers	Normalized numbers	$(-1)^{sign} \times 2^{exponent-127} \times 1.$fraction
11111111	$\pm\infty$	NaN	error pattern

Special Values

a map of floats



https://ciechanow.ski/exposing-floating-point/#special-values

Special Numbers

- The encoding of special numbers uses exponent values that do not occur in normalized numbers
- The *denormalized* numbers of the form:

$$\pm d_0 \,.\, d_1 \ldots d_{p-1} \cdot b^{e_{min}} \,,\; d_0 = 0$$

A denormalized number has smaller absolute value than any normalized number

0 is a denormalized number

The other special numbers cannot really be called numbers: $+\infty$ and $-\infty$ are returned by overflowing operations; there are values called NaNs for “not a number” - either quiet or signaling - that are returned by operations with undefined result, like taking square root of a negative number. They provide more flexibility in dealing with exceptional situations.

Arithmetic Operations Requirements

- The result of any addition, subtraction, multiplication, or division is the representable value *nearest* to the true value
- If there are two nearest values - meaning that the true value is halfway between them, the one that has least significant digit $d_{p-1} = 0$ is chosen
- This is called *round-to-even*; other rounding modes can be enabled if necessary
- The same rule applies to the conversion of decimal values like 1.1 to their binary floating point representation
- Comparisons of values have to be exact for all relational operators $(< , > , \leq , \geq , == , \neq)$

Accuracy of Floating-Point Arithmetic

intrinsic errors

- Rounding
- Differences in addend exponents
- Cancellation
- Near over- and underflow errors

Relative Error

the concept of "significant figures"

The constant ϵ_m provides an upper bound on the relative size of the error in approximating a real number x by its finite floating point representation, which we call $f(x)$. In our example with 0.1 we found that $|x - f(x)| = 6 \times 10^{-9}$and the relative error in the approximation is $|x - f(x)|/|x| = 6 \times 10^{-8}$, which is less than $2^{-23} = 1.2 \times 10^{-7}$. Note that for our single precision case,

$$\begin{aligned} \frac{|x - f(x)|}{|x|} &\leq 2^{-23} 2^k / |x| \\ &\leq 2^{-23} 2^k 2^{-k} \\ &= 2^{-23} \end{aligned}$$

which establishes that the relative error is at most ϵ_m

The IEEE standard for floating point arithmetics calls for basic arithmetics operations to be performed to higher precision and then rounded to the nearest representable floating point number

Rounding Modes

IEEE standard

- Rounding-direction attributes to nearest
  - Round to nearest, ties to even – rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time; this is the default algorithm for binary floating-point and the recommended default for decimal
  - Round to nearest, ties away from zero – rounds to the nearest value; if the number falls midway it is rounded to the nearest value above (for positive numbers) or below (for negative numbers); is not required for a binary format implementation
- Directed roundings
  - Round toward 0 – directed rounding towards zero (also known as truncation).
  - Round toward +∞ – directed rounding towards positive infinity (also known as rounding up or ceiling).
  - Round toward −∞ – directed rounding towards negative infinity (also known as rounding down or floor).

Correctly Rounded Arithmetic

- The IEEE standard requires that the result of addition, subtraction, multiplication and division be **exactly rounded**
  - Exactly rounded means the results are calculated exactly and then rounded. For example: assuming $p = 23$, $x = (1.00..00)_2 \times 2^0$ and $z = (1.00..01)_2 \times 2^{-25}$, then $x - z$ is

$$\begin{array}{rll}
 & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|01000000000000000000000001 &)_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|10111111111111111111111111 &)_2 \times 2^0 \\
\text{Normalize :} & (\ 1.11111111111111111111111|01111111111111111111111110 &)_2 \times 2^{-1} \\
\text{Round to} & & \\
\text{Nearest :} & (\ 1.11111111111111111111111 &)_2 \times 2^{-1}
\end{array}$$

  - Compute the result exactly is very expensive if the operands differ greatly in size
  - The result of two or more arithmetic operations are NOT exactly rounded
- How is **correctly rounded** arithmetic implemented?
  - Using two additional guard bits plus one sticky bit guarantees that the result will be the same as computed using exactly rounded [Goldberg 1990]. The above example can be done as

$$\begin{array}{rll}
 & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|011 &)_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|101 &)_2 \times 2^0 \\
\text{Normalize :} & (\ 1.11111111111111111111111|01 &)_2 \times 2^{-1} \\
\text{Round to Nearest :} & (\ 1.11111111111111111111111 &)_2 \times 2^{-1}
\end{array}$$

Rounding

absolute and relative rounding error

- A positive REAL number in the *normalized range* ($x_{min} \leq x \leq x_{max}$) can be represented as

$$(x)_2 = (1 . b_1 b_2 ... b_{p-1} ...) \text{ x } 2^e,$$

where $x_{min} (= 2^{e_{min}})$ and $x_{max} (= (1 - 2^{-p}) 2^{e_{max}+1}$) are the smallest and largest normalized floating point numbers. (Subscript 2 for binary representation is omitted since now.)

- The nearest floating point number less than or equal to x is

$$x_- = (1.b_1 b_2 ... b_{p-1}) \text{ x } 2^e$$

- The nearest floating point number larger than x is

$$x_+ = (1.b_1 b_2 ... b_{p-1} + 0.00...1) \text{ x } 2^e$$

- The gap between x_+ and x_-, called **unit in the last place** (ulp) is

$$2^{-(p-1)} 2^e$$

- The **absolute rounding error** is

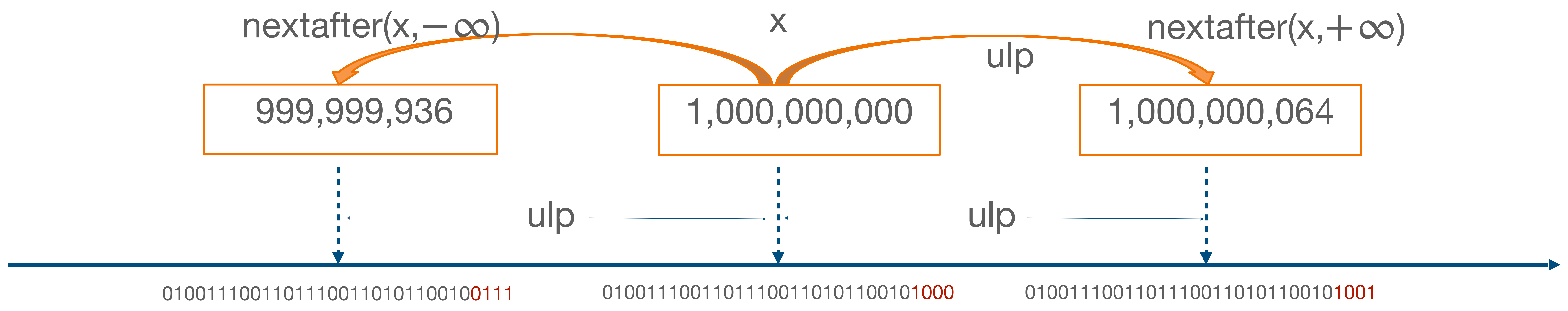
$$abserr(x) = \left| round(x) - x \right| < 2^{-(p-1)} 2^e = \text{ulp}$$

- The **relative rounding error** is

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

Quiz:

a toy rounding example



- **Example**:

```
float x=1000000000;
float y=1000000032;
float z=1000000033;

std::cout << std::scientific << std::setprecision(8) << x << ‘ ‘ << y << ‘ ‘ << z << std::endl;
```

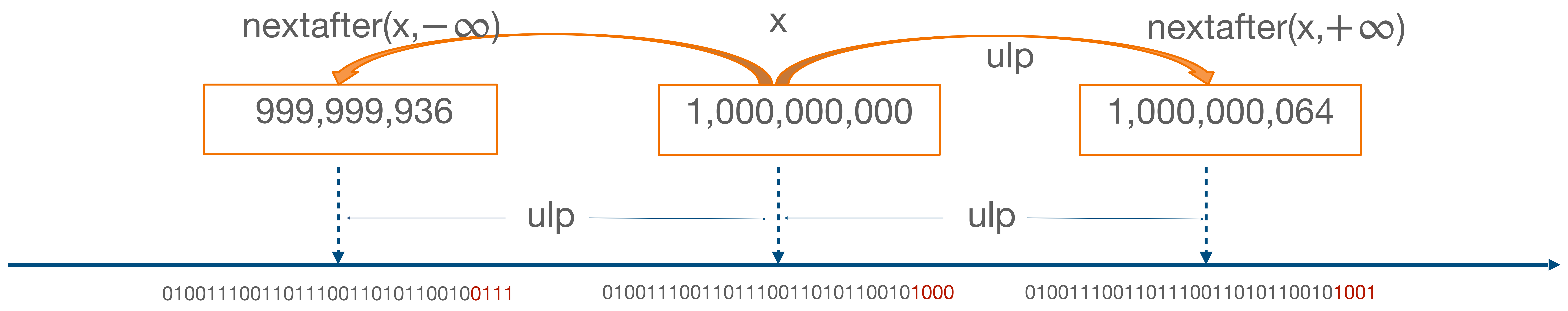
- **Question**: What’s the output?

Answers: 1.00000000e+09 1.00000000e+09 1.00000006e+09

When rounding to the nearest, $\text{abserr}(x) \le \frac{1}{2}\text{ulp}$ and $\text{relerr}(x) \le \frac{1}{2}\epsilon$

Quiz:

Adding a Small and a Large Number



- **Example**:
  float x=1000000000;

  std::cout << std::scientific << std::setprecision(8)
                << x << ‘ ‘ << x + 32.f << ‘ ‘ << x + 33.f << std::endl
                << x + 32.f - x << ‘ ‘ << x + 33.f - x << std::endl;

- **Question**: What is the output?

- **Answers:**

```
1.00000000e+09 1.00000000e+09 1.00000006e+09
0.00000000e+00 6.40000000e+01
```

Cancellation

- Cancellation occurs when we operate with numbers that are not in floating point format
- For every $x \in \mathbb{R}$, there exists $|\varepsilon| < \varepsilon_{mach}$ such that $round(x) = x\,(1 + \varepsilon)$
- Thus

$$round\Big(round(x) - round\big(y\big)\Big) = \Big(round(x) - round\big(y\big)\Big)\big(1 + \varepsilon_3\big)$$

$$= (x(1 + \varepsilon_1 - y(1 + \varepsilon_2))\big(1 + \varepsilon_3\big)$$

$$= (x - y)(1 + \varepsilon_3) + (x\varepsilon_1 - y\varepsilon_2)(1 + \varepsilon_3),$$

and if $\big(x - y\big) \neq 0,$

$$\left|\frac{(round(round(x) - round\big(y\big))}{x - y}\right| = \left|\varepsilon_3 + \frac{x\,\varepsilon_1 - y\varepsilon_2}{x - y}(1+\varepsilon_3)\right|$$

when $x\,\varepsilon_1 - y\varepsilon_2 \neq 0$, and x-y is small, the error could be $\gg\ \varepsilon_{mach}$

Cancellation

- Cancellation occurs when we subtract two almost equal numbers
- The consequence is the error could be much larger than the machine epsilon
- For example, consider two numbers

$x = 3.141592653589793$ (16-digit approximation to π)

$y = 3.141592653585682$ (12-digit approximation to π)

Their difference is

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

In a C program, if we store x, y in single precision and display z in single precision, the difference is

0.000000e+00 $\boxed{\text{Complete loss of accuracy}}$

If we store x, y in double precision and display z in double precision, the difference is

4.110933815582030e-12 $\boxed{\text{Partial loss of accuracy}}$

Cancellation:

The Solution of a Quadratic Equation

- Consider the quadratic equation $ax^2 + bx + c = 0$, the roots are

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Two sources of cancellation

- A better solution will be

$$x_1 = \frac{-b - sign(b)\sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{2c}{-b - sign(b)\sqrt{b^2 - 4ac}} = \frac{c}{ax_1}$$

- When $a = 1$, $b = 1.786737589984535$ and $c = 1.149782767465722 \times 10^{-8}$, in double precision, the first formula yields

$$x_1 = \frac{(1.786737601482363 + 1.786737578486707)}{2} = 1.786737589984535$$

$$x_2 = \frac{(1.786737601482363 - 1.786737578486707)}{2} = 0.000000011497828$$

Cancellation

- The second formula yields

$$x_1 = \frac{(1.786737601482363 + 1.786737578486707)}{2} = 1.786737589984535$$

$$x_2 = \frac{2.054360090947453 \times 10^{-8}}{1.786737589984535} = 1.149782767465722 \times 10^{-8}$$

Floating Point Numbers

- Again, let us consider the floating point representation (assume x $\neq 0$)

$$x = -1^s \left(\sum_{i=0}^{p-1} d_i B^{-i} \right) B^e$$

where s $\in \{0, 1\}$, $B \geq 2$, $d_i \in \{0, \ldots, B-1\}$ with $d_0 > 0$, $i \in \{0, \ldots, p-1\}$, $e \in \{e_{min}, \ldots, e_{max}\}$.

- What is the **largest number** in the system?

$$x_{max} = \left(\sum_{i=0}^{p-1} (B-1) B^{-i} \right) B^{e_{max}} = (1 - B^{-p}) B^{e_{max}+1}, \ (x_{max})_2 = (1 - 2^{-p}) 2^{e_{max}+1}$$

- What is the **smallest positive normalized** in the system

$$x_{min} = B^{e_{min}}, \ (x_{min})_2 = 2^{e_{min}}$$

- The gap between number 1.0 and the next larger floating point number is called **machine epsilon.** What is the machine epsilon in the system?

$$\varepsilon = B^{-(p-1)}, \ \varepsilon_2 = (1.00..1)_2 - (1.00..0)_2 = (0.00\ldots1)_2 = 2^{-(p-1)}$$

- The gap between B^E and the next larger floating point number is called **unit in the last place** (ulp). What is the upl in the system?

$$ulp(x) = B^{-(p-1)} B^e = \varepsilon\ B^e, \ ulp(x)_2 = (1.00\ldots1)_2 2^e - (1.00\ldots0)_2 2^e = (0.00\ldots1)_2 2^e = 2^{-(p-1)} 2^e$$

Quiz: Binary Representation

When $b = 2$:

$$(x)_2 = (-1)^s(1.d_1d_2\ldots d_{p-1})2^e$$

- $d_0 = 1$ is a *hidden bit* (in a *normalized* binary system)
- $d_1d_2\ldots d_{p-1}$ is called the *fractional* part of the significand
- $e \in \{e_{min}, \ldots, e_{max}\}$
- The gap between 1 and the next larger floating-point number is called *machine epsilon, ϵ*

- **Questions:** In this binary system
- What is the largest number?

$$(x_{max})_2 = (1 - 2^{-p})2^{e_{max}+1}$$

- What is the smallest positive normalized number?

$$(x_{min})_2 = 2^{e_{min}}$$

- What is the ϵ?

$$\epsilon_2 = (1.00..1) \times 2^0 - (1.00..0) \times 2^0 = (0.00..1)_2 \times 2^0 = 2^{-(p-1)}$$

Precision

- Floating point arithmetic CANNOT precisely represent true arithmetic operations
  - The operands are rounded
    - They exist in a finite number (~2^{32} for single precision)
    - The space between two floating point numbers differs by one ulp
  - Results of operations are rounded
    - $x + \varepsilon - x \neq \varepsilon$
  - Algebra is NOT necessarily associative nor distributive
    - $(a + b) + c \neq a + (b + c)$
    - $\frac{a}{b} \neq a * \frac{1}{b}$
    - (a + b) * (a – b) $\neq$ a^2 - b^2
  - Example: what will be the result of 0.1^2?
    - In single precision, 0.1 is rounded and represented as 0.100000001490116119384765625 exactly
    - Squaring it with single-precision floating point hardware (with rounding) gives 0.010000000707805156707763671875
    - It is neither 0.01 nor the representable number closest to 0.01 (the representable number closest to 0.01 is 0.009999999776482582092285156250)

Exceptions

- The IEEE floating point standard defines several exceptions that occur when the result of a floating point operation is unclear or undesirable. Exceptions can be ignored, in which case some default action is taken, such as returning a special value. When trapping is enabled for an exception, an error is signaled whenever that exception occurs.

- Possible floating point exceptions:

    - **Underflow**: The result of an operation is too small to be represented as a normalized float in its format. If trapping is enabled, the floating-point-underflow condition is signaled. Otherwise, the operation results in a denormalized float or zero.

    - **Overflow**: The result of an operation is too large to be represented as a float in its format. If trapping is enabled, the floating-point-overflow exception is signaled. Otherwise, the operation results in the appropriate infinity.

    - **Divide-by-zero**: A float is divided by zero. If trapping is enabled, the divide-by-zero condition is signaled. Otherwise, the appropriate infinity is returned.

    - **Invalid**: The result of an operation is ill-defined, such as (0.0/0.0). If trapping is enabled, the floating-point-invalid condition is signaled. Otherwise, a quiet NaN is returned.

    - **Inexact**: The result of a floating point operation is not exact, i.e. the result was rounded. If trapping is enabled, the floating-point-inexact condition is signaled. Otherwise, the rounded result is returned.

- Trapping of these exceptions can be enabled through compiler flags, but be aware that the resulting code will run slower.

Floating-Point Arithmetic Guidelines

computing with floating point numbers

- Do not compare floating point numbers for equality, if at least one of them results from *inexact* floating point computations
    - Q: How to tell if a particular floating point operation is exact or not?
- Avoid adding two numbers that considerably differ in size
- Avoid subtracting two numbers of almost the equal size, if these numbers are the results of other floating point computations

PRINCETON UNIVERSITY

Algorithm Considerations

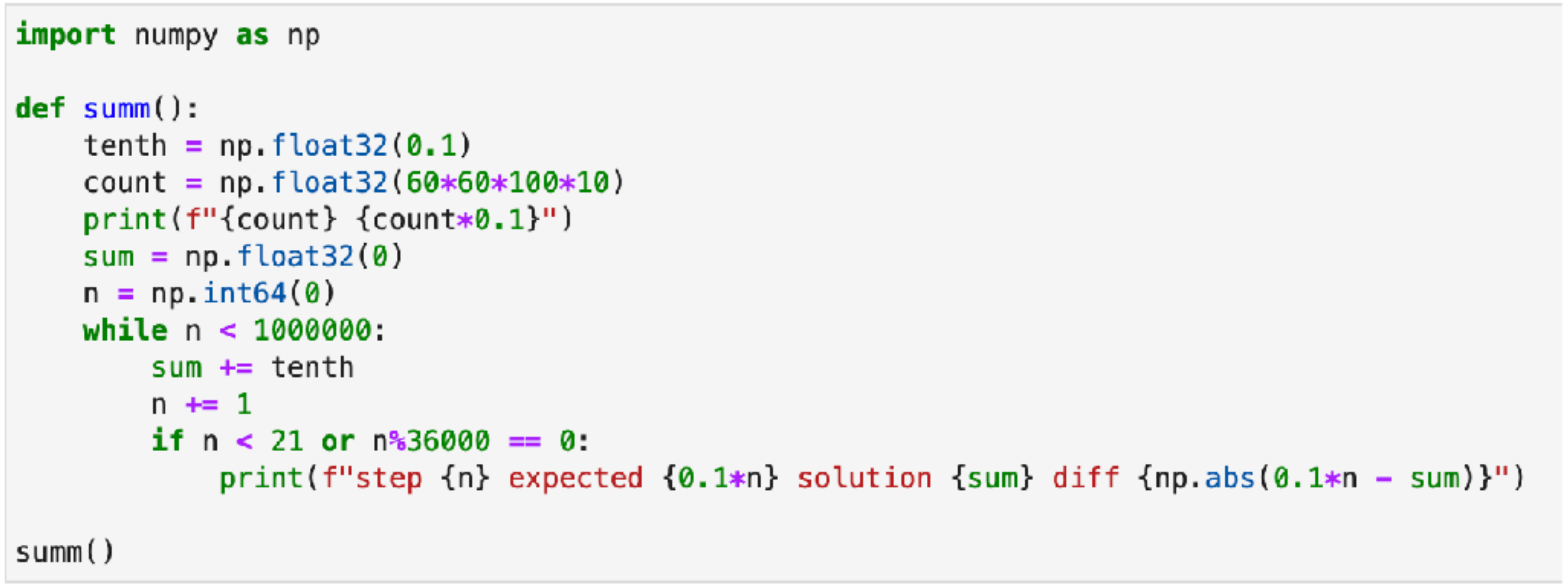
- Numerical algorithms often need to sum up a large number of values (e.g. matrix multiplication)
- The problem gets even more complicated on parallel computers
- A common technique to maximize floating point arithmetic accuracy is to pre-sort the data. On a parallel computer,
    - Divide up the numbers in groups
    - Sort the data in each group and sum them sequentially by one thread
    - A reduction for the partial sum from each thread

Hands-on

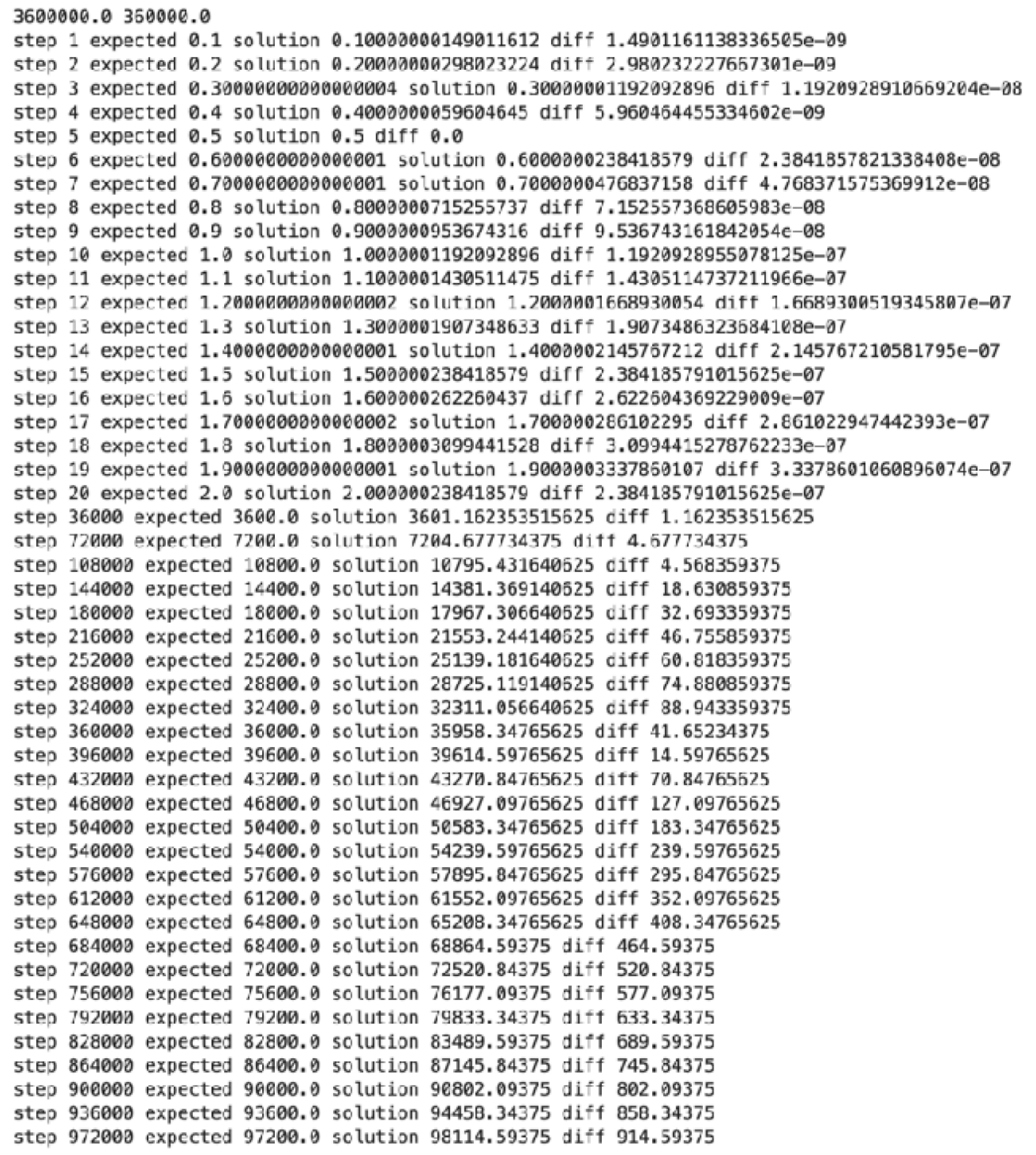
https://github.com/ianna/CoDaS-HEP2023-FloatingPoint/blob/main/exercises/Floating-point-arithmetics.ipynb

Hands-on: Summing Many Numbers

What are the potential arithmetic issues when summing many numbers?



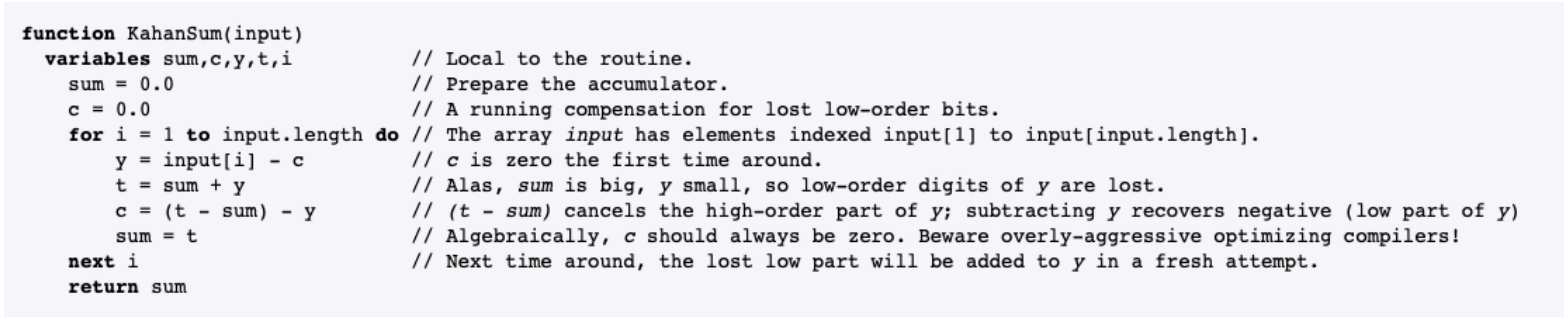
```python
import numpy as np

def summ():
    tenth = np.float32(0.1)
    count = np.float32(60*60*100*10)
    print(f"{count} {count*0.1}")
    sum = np.float32(0)
    n = np.int64(0)
    while n < 1000000:
        sum += tenth
        n += 1
        if n < 21 or n%36000 == 0:
            print(f"step {n} expected {0.1*n} solution {sum} diff {np.abs(0.1*n - sum)}")

summ()
```



```
3600000.0 360000.0
step 1 expected 0.1 solution 0.10000000149011612 diff 1.4901161138336505e-09
step 2 expected 0.2 solution 0.20000000298023224 diff 2.9802322276673010e-09
step 3 expected 0.30000000000000004 solution 0.30000001192092896 diff 1.1920928910669204e-08
step 4 expected 0.4 solution 0.40000000596046450 diff 5.9604644553346020e-09
step 5 expected 0.5 solution 0.5 diff 0.0
step 6 expected 0.6000000000000001 solution 0.6000000238418579 diff 2.3841857821338408e-08
step 7 expected 0.7000000000000001 solution 0.7000000476837158 diff 4.7683715753699120e-08
step 8 expected 0.8 solution 0.8000000715255737 diff 7.1525573686059830e-08
step 9 expected 0.9 solution 0.9000000953674316 diff 9.5367431618420540e-08
step 10 expected 1.0 solution 1.0000001192092896 diff 1.1920928955078125e-07
step 11 expected 1.1 solution 1.1000001430511475 diff 1.4305114737211966e-07
step 12 expected 1.2000000000000002 solution 1.2000001668930054 diff 1.6689300519345807e-07
step 13 expected 1.3 solution 1.3000001907348633 diff 1.9073486323684108e-07
step 14 expected 1.4000000000000001 solution 1.4000002145767212 diff 2.1457672105817950e-07
step 15 expected 1.5 solution 1.500000238418579 diff 2.384185791015625e-07
step 16 expected 1.6 solution 1.600000262260437 diff 2.622604369229009e-07
step 17 expected 1.7000000000000002 solution 1.700000286102295 diff 2.861022947442393e-07
step 18 expected 1.8 solution 1.8000003099441528 diff 3.099441527876223e-07
step 19 expected 1.9000000000000001 solution 1.900000333786010 diff 3.3378601060896074e-07
step 20 expected 2.0 solution 2.000000238418579 diff 2.384185791015625e-07
step 36000 expected 3600.0 solution 3601.162353515625 diff 1.162353515625
step 72000 expected 7200.0 solution 7204.677734375 diff 4.677734375
step 108000 expected 10800.0 solution 10795.431640625 diff 4.568359375
step 144000 expected 14400.0 solution 14381.369140625 diff 18.630859375
step 180000 expected 18000.0 solution 17967.306640625 diff 32.693359375
step 216000 expected 21600.0 solution 21553.244140625 diff 46.755859375
step 252000 expected 25200.0 solution 25139.181640625 diff 60.818359375
step 288000 expected 28800.0 solution 28725.119140625 diff 74.880859375
step 324000 expected 32400.0 solution 32311.056640625 diff 88.943359375
step 360000 expected 36000.0 solution 35958.34765625 diff 41.65234375
step 396000 expected 39600.0 solution 39614.59765625 diff 14.59765625
step 432000 expected 43200.0 solution 43270.84765625 diff 70.84765625
step 468000 expected 46800.0 solution 46927.09765625 diff 127.09765625
step 504000 expected 50400.0 solution 50583.34765625 diff 183.34765625
step 540000 expected 54000.0 solution 54239.59765625 diff 239.59765625
step 576000 expected 57600.0 solution 57895.84765625 diff 295.84765625
step 612000 expected 61200.0 solution 61552.09765625 diff 352.09765625
step 648000 expected 64800.0 solution 65208.34765625 diff 408.34765625
step 684000 expected 68400.0 solution 68864.59375 diff 464.59375
step 720000 expected 72000.0 solution 72520.84375 diff 520.84375
step 756000 expected 75600.0 solution 76177.09375 diff 577.09375
step 792000 expected 79200.0 solution 79833.34375 diff 633.34375
step 828000 expected 82800.0 solution 83489.59375 diff 689.59375
step 864000 expected 86400.0 solution 87145.84375 diff 745.84375
step 900000 expected 90000.0 solution 90802.09375 diff 802.09375
step 936000 expected 93600.0 solution 94458.34375 diff 858.34375
step 972000 expected 97200.0 solution 98114.59375 diff 914.59375
```

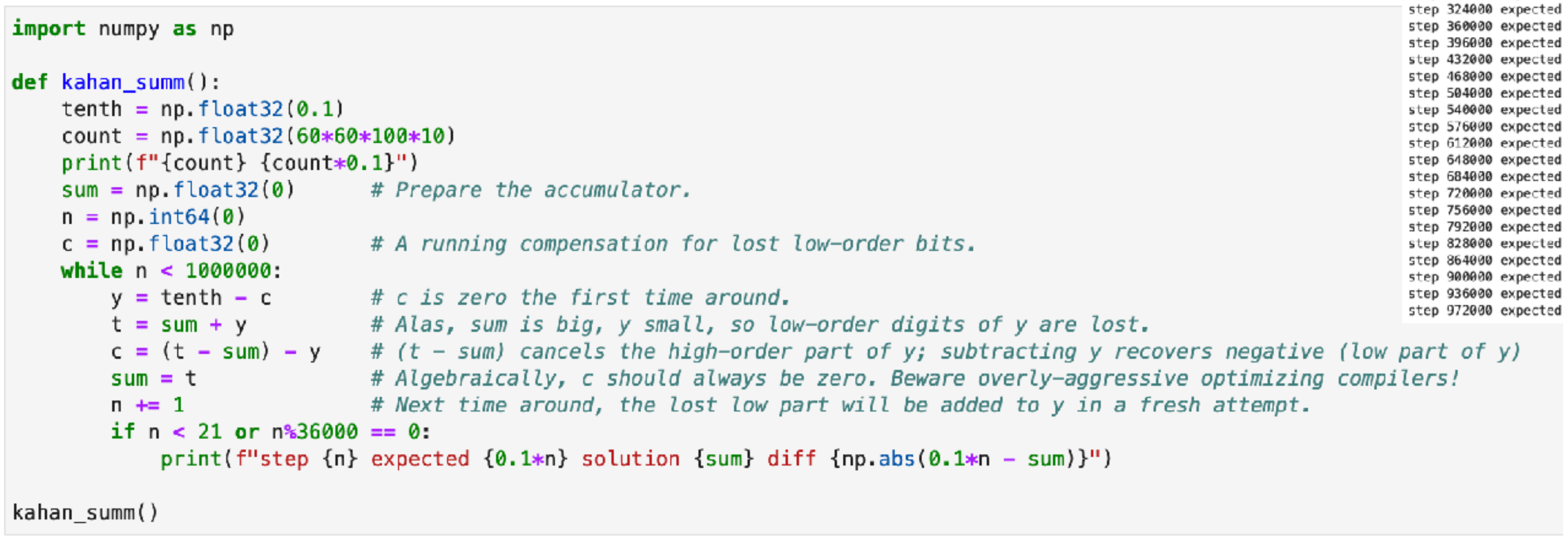
Inspired by the patriot missile failure problem:
http://www-users.math.umn.edu/~arnold/disasters/patriot.html

Hands-on: Kahan Summation Algorithm

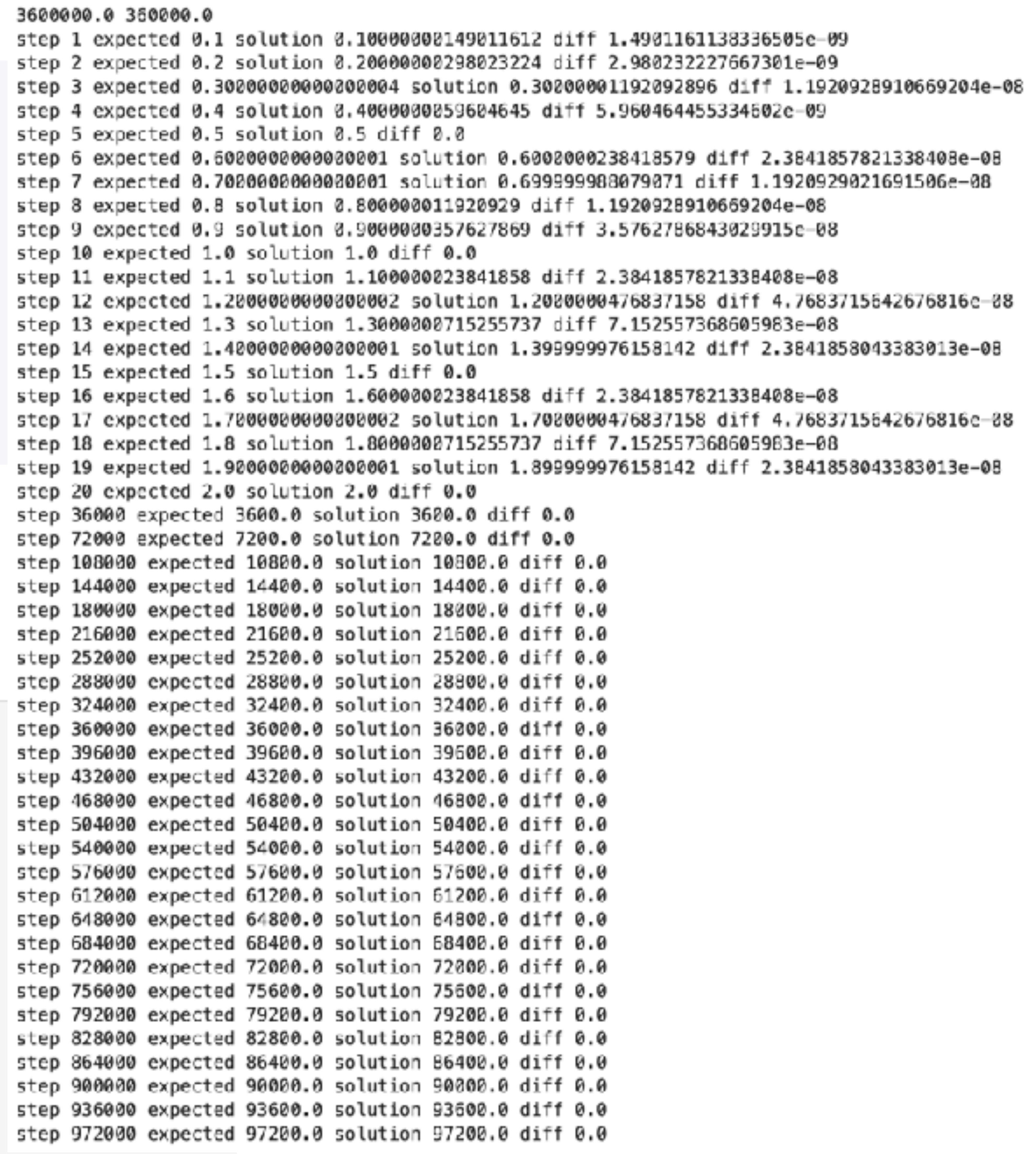


```
function KahanSum(input)
  variables sum,c,y,t,i          // Local to the routine.
    sum = 0.0                    // Prepare the accumulator.
    c = 0.0                      // A running compensation for lost low-order bits.
    for i = 1 to input.length do // The array input has elements indexed input[1] to input[input.length].
        y = input[i] - c         // c is zero the first time around.
        t = sum + y              // Alas, sum is big, y small, so low-order digits of y are lost.
        c = (t - sum) - y        // (t - sum) cancels the high-order part of y; subtracting y recovers negative (low part of y)
        sum = t                  // Algebraically, c should always be zero. Beware overly-aggressive optimizing compilers!
    next i                       // Next time around, the lost low part will be added to y in a fresh attempt.
    return sum
```

See: https://en.wikipedia.org/wiki/Kahan_summation_algorithm



```python
import numpy as np

def kahan_summ():
    tenth = np.float32(0.1)
    count = np.float32(60*60*100*10)
    print(f"{count} {count*0.1}")
    sum = np.float32(0)       # Prepare the accumulator.
    n = np.int64(0)
    c = np.float32(0)         # A running compensation for lost low-order bits.
    while n < 1000000:
        y = tenth - c         # c is zero the first time around.
        t = sum + y           # Alas, sum is big, y small, so low-order digits of y are lost.
        c = (t - sum) - y     # (t - sum) cancels the high-order part of y; subtracting y recovers negative (low part of y)
        sum = t               # Algebraically, c should always be zero. Beware overly-aggressive optimizing compilers!
        n += 1                # Next time around, the lost low part will be added to y in a fresh attempt.
        if n < 21 or n%36000 == 0:
            print(f"step {n} expected {0.1*n} solution {sum} diff {np.abs(0.1*n - sum)}")

kahan_summ()
```



```
3600000.0 360000.0
step 1 expected 0.1 solution 0.10000000149011612 diff 1.4901161138336505e-09
step 2 expected 0.2 solution 0.20000000298023224 diff 2.9802322276673010e-09
step 3 expected 0.30000000000000004 solution 0.30000001192092896 diff 1.1920928910669204e-08
step 4 expected 0.4 solution 0.40000000596046450 diff 5.9604644553345020e-09
step 5 expected 0.5 solution 0.5 diff 0.0
step 6 expected 0.6000000000000001 solution 0.6000000238418579 diff 2.3841857821338408e-08
step 7 expected 0.7000000000000001 solution 0.6999999880790710 diff 1.1920929021691506e-08
step 8 expected 0.8 solution 0.800000011920929 diff 1.1920928910669204e-08
step 9 expected 0.9 solution 0.9000000357627869 diff 3.5762786843029915e-08
step 10 expected 1.0 solution 1.0 diff 0.0
step 11 expected 1.1 solution 1.100000023841858 diff 2.3841857821338408e-08
step 12 expected 1.2000000000000002 solution 1.2000000476837158 diff 4.768371564267681 6e-08
step 13 expected 1.3 solution 1.3000000715255737 diff 7.152557368605983e-08
step 14 expected 1.4000000000000001 solution 1.399999976158142 diff 2.3841858043383013e-08
step 15 expected 1.5 solution 1.5 diff 0.0
step 16 expected 1.6 solution 1.600000023841858 diff 2.384185782133840 8e-08
step 17 expected 1.7000000000000002 solution 1.7000000476837158 diff 4.768371564267681 6e-08
step 18 expected 1.8 solution 1.8000000715255737 diff 7.152557368605983e-08
step 19 expected 1.9000000000000001 solution 1.899999976158142 diff 2.3841858043383013e-08
step 20 expected 2.0 solution 2.0 diff 0.0
step 36000 expected 3600.0 solution 3600.0 diff 0.0
step 72000 expected 7200.0 solution 7200.0 diff 0.0
step 108000 expected 10800.0 solution 10800.0 diff 0.0
step 144000 expected 14400.0 solution 14400.0 diff 0.0
step 180000 expected 18000.0 solution 18000.0 diff 0.0
step 216000 expected 21600.0 solution 21600.0 diff 0.0
step 252000 expected 25200.0 solution 25200.0 diff 0.0
step 288000 expected 28800.0 solution 28800.0 diff 0.0
step 324000 expected 32400.0 solution 32400.0 diff 0.0
step 360000 expected 36000.0 solution 36000.0 diff 0.0
step 396000 expected 39600.0 solution 39600.0 diff 0.0
step 432000 expected 43200.0 solution 43200.0 diff 0.0
step 468000 expected 46800.0 solution 46800.0 diff 0.0
step 504000 expected 50400.0 solution 50400.0 diff 0.0
step 540000 expected 54000.0 solution 54000.0 diff 0.0
step 576000 expected 57600.0 solution 57600.0 diff 0.0
step 612000 expected 61200.0 solution 61200.0 diff 0.0
step 648000 expected 64800.0 solution 64800.0 diff 0.0
step 684000 expected 68400.0 solution 68400.0 diff 0.0
step 720000 expected 72000.0 solution 72000.0 diff 0.0
step 756000 expected 75600.0 solution 75600.0 diff 0.0
step 792000 expected 79200.0 solution 79200.0 diff 0.0
step 828000 expected 82800.0 solution 82800.0 diff 0.0
step 864000 expected 86400.0 solution 86400.0 diff 0.0
step 900000 expected 90000.0 solution 90000.0 diff 0.0
step 936000 expected 93600.0 solution 93600.0 diff 0.0
step 972000 expected 97200.0 solution 97200.0 diff 0.0
```

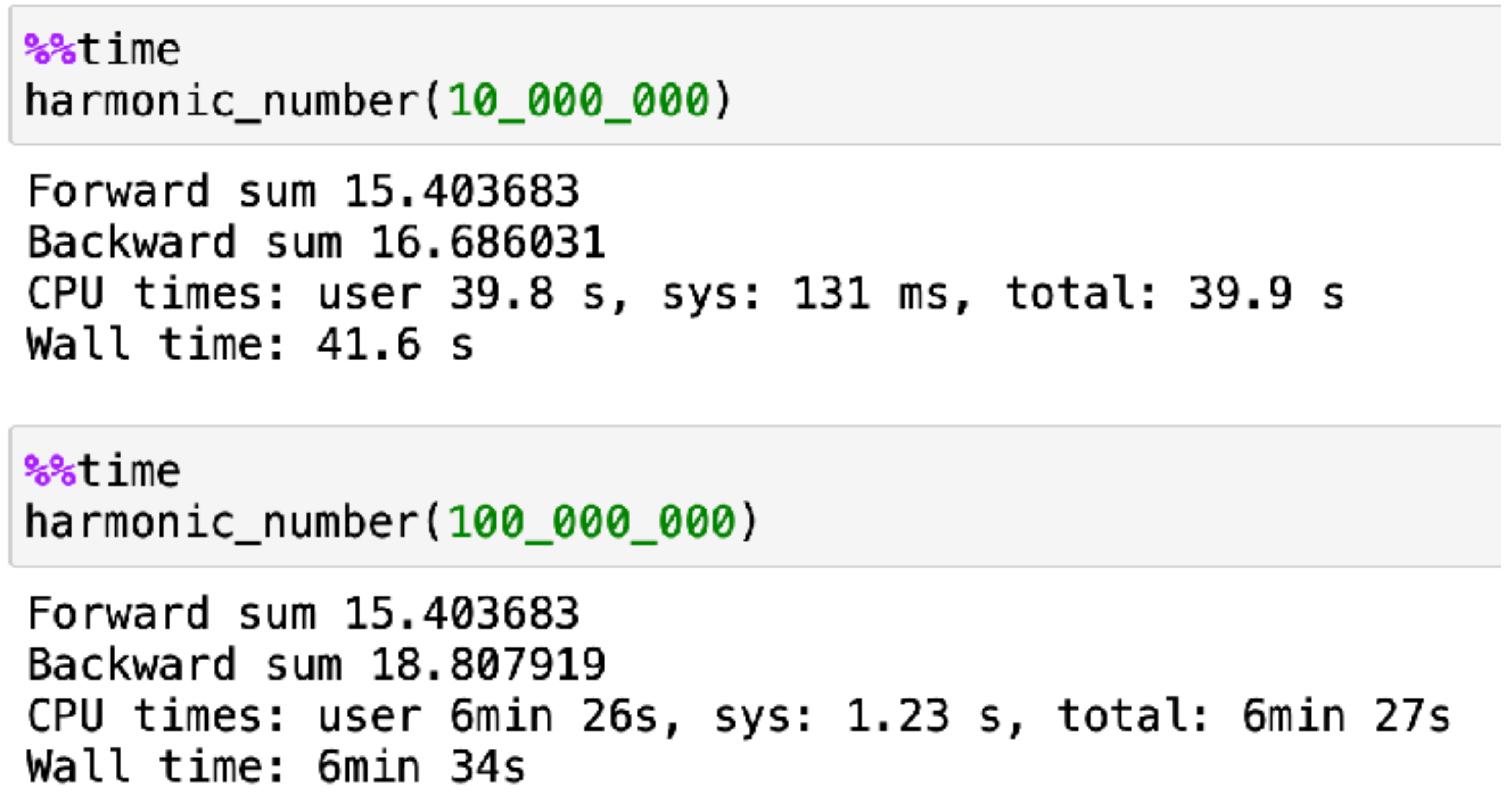
Harmonic Numbers

homework exercise

- Using the approximation:

$$\frac{1}{2(n+1)} < H_n - \ln n - \gamma < \frac{1}{2n},$$

where $\gamma = 0.57721666...$ is a Euler-Mascheroni constant, we get $H_n \approx 18.998$ for $n = 10^8$



```
%%time
harmonic_number(10_000_000)
```

```
Forward sum 15.403683
Backward sum 16.686031
CPU times: user 39.8 s, sys: 131 ms, total: 39.9 s
Wall time: 41.6 s
```

```
%%time
harmonic_number(100_000_000)
```

```
Forward sum 15.403683
Backward sum 18.807919
CPU times: user 6min 26s, sys: 1.23 s, total: 6min 27s
Wall time: 6min 34s
```

The forward sum differs: as the larger summands are added up first, the intermediate value of the sum to be compared grows fast.
At some point, the size difference between the partial sum and the summand $\frac{1}{i}$ to be added is so large that the addition does not change the partial sum anymore.

Approximate Math

Optimization levels and related options

- GCC has a rich optimization pipeline that is controlled by approximately a hundred command line options
- The default is to not optimize. You can specify this optimization level on the command line as -O0. It is often used when developing and debugging a project. This means it is usually accompanied with the command line switch -g so that debug information is emitted. As no optimizations take place, no information is lost because of it. No variables are optimized away, the compiler only inlines functions with special attributes that require it, and so on. As a consequence, the debugger can almost always find everything it searches for in the running program and report on its state very well. On the other hand, the resulting code is big and slow
- The most common optimization level for release builds is -O2 which attempts to optimize the code aggressively but avoids large compile times and excessive code growth.
- Optimization level -O3 instructs GCC to simply optimize as much as possible, even if the resulting code might be considerably bigger and the compilation can take longer.
- Note that neither -O2 nor -O3 imply anything about the precision and semantics of floating-point operations. Even at the optimization level -O3 GCC implements math functions so that they strictly follow the respective IEEE and/or ISO rules. This often means that the compiled programs run markedly slower than necessary if such strict adherence is not required. The command line switch -ffast-math is a common way to relax rules governing floating-point operations.
- The most aggressive optimization level is `-Ofast` which does imply `-ffast-math` along with a few options that disregard strict standard compliance. In GCC 11 this level also means the optimizers may introduce data races when moving memory stores which may not be safe for multithreaded applications. Additionally, the Fortran compiler can take advantage of associativity of math operations even across parentheses and convert big memory allocations on the heap to allocations on stack. The last mentioned transformation may cause the code to violate maximum stack size allowed by **`ulimit`** which is then reported to the user as a segmentation fault.

Optimization level recommendation

- Usually we(*) recommend using -O2, because at this level the compiler makes balanced size and speed trade-offs when building a general-purpose operating system
- However, we suggest using -O3 if you know that your project is compute-intensive and is either small or an important part of your actual workload
- Moreover, if the compiled code contains performance-critical floating-point operations, we strongly advise that you investigate whether -ffast-math or any of the fine-grained options it implies can be safely used

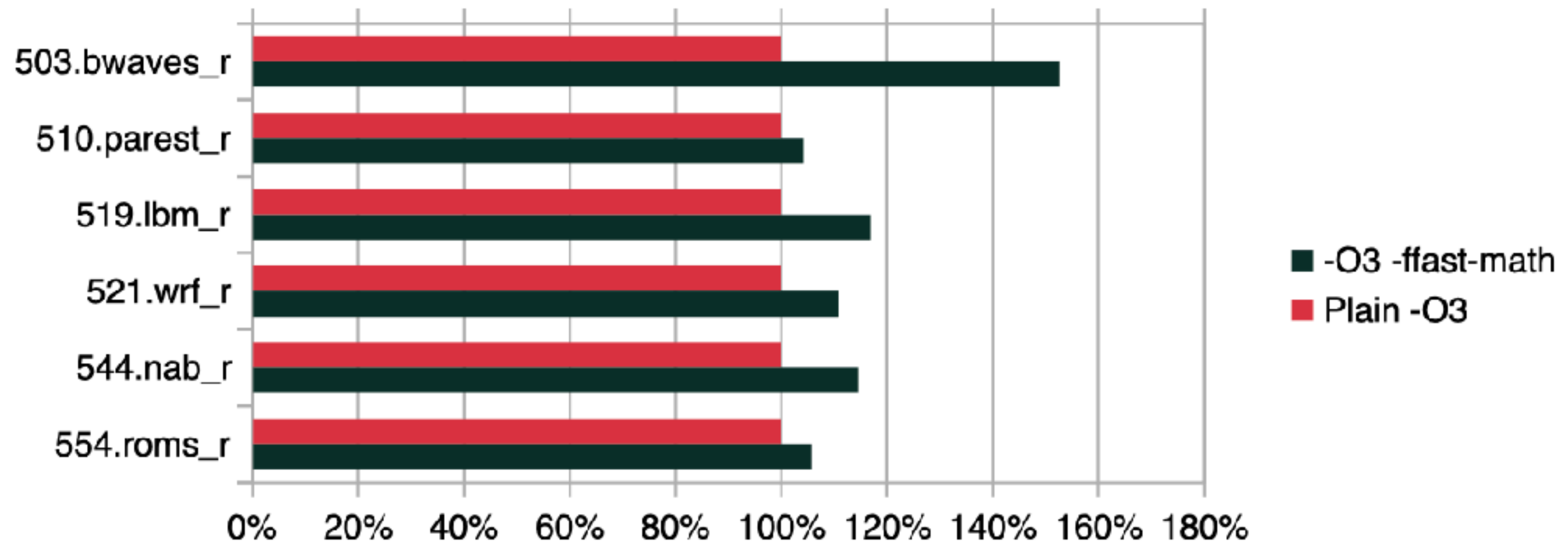


FIGURE 18: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF SELECTED FLOATING-POINT BENCHMARKS BUILT WITH GCC 11.2 AND -O3 -MARCH=NATIVE, WITHOUT AND WITH -FFAST-MATH

- (*) https://documentation.suse.com/sbp/server-linux/single-html/SBP-GCC-11/

Other Optimizing Compilers

- All optimizing compilers have the concept of optimization levels and their optimization levels often have the same names as those in GCC, but they do not necessarily make the same trade-offs

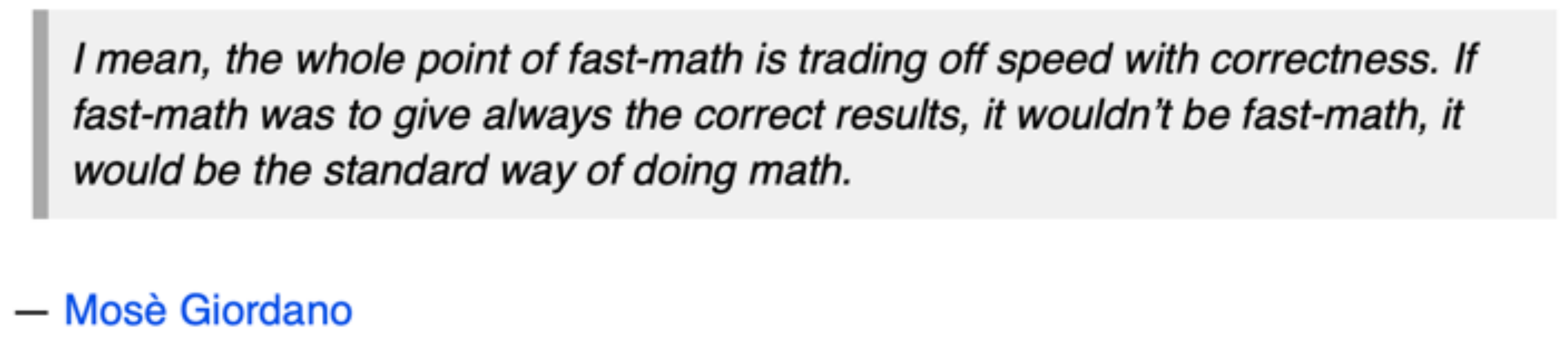
- Famously, GCC's `-Os` optimizes for size much more aggressively than LLVM/Clang's level with the same name. Therefore, it often produces slower code; the more equivalent option in Clang is `-Oz` which GCC does not have

- Similarly, –O2 can have different meanings for different compilers

- For example, the difference between –O2 and –O3 is much bigger in GCC than in LLVM/Clang

Beware of fast-math

https://simonbyrne.github.io/notes/fastmath/



I mean, the whole point of fast-math is trading off speed with correctness. If fast-math was to give always the correct results, it wouldn't be fast-math, it would be the standard way of doing math.

— Mosè Giordano

- A typical process might be:
- Develop reliable validation tests
- Develop useful benchmarks
- Enable fast-math and compare benchmark results
- Selectively enable/disable fast-math optimizations to identify
  a. which optimizations have a performance impact,
  b. which cause problems, and
  c. where in the code those changes arise.
- Validate the final numeric results

PRINCETON UNIVERSITY

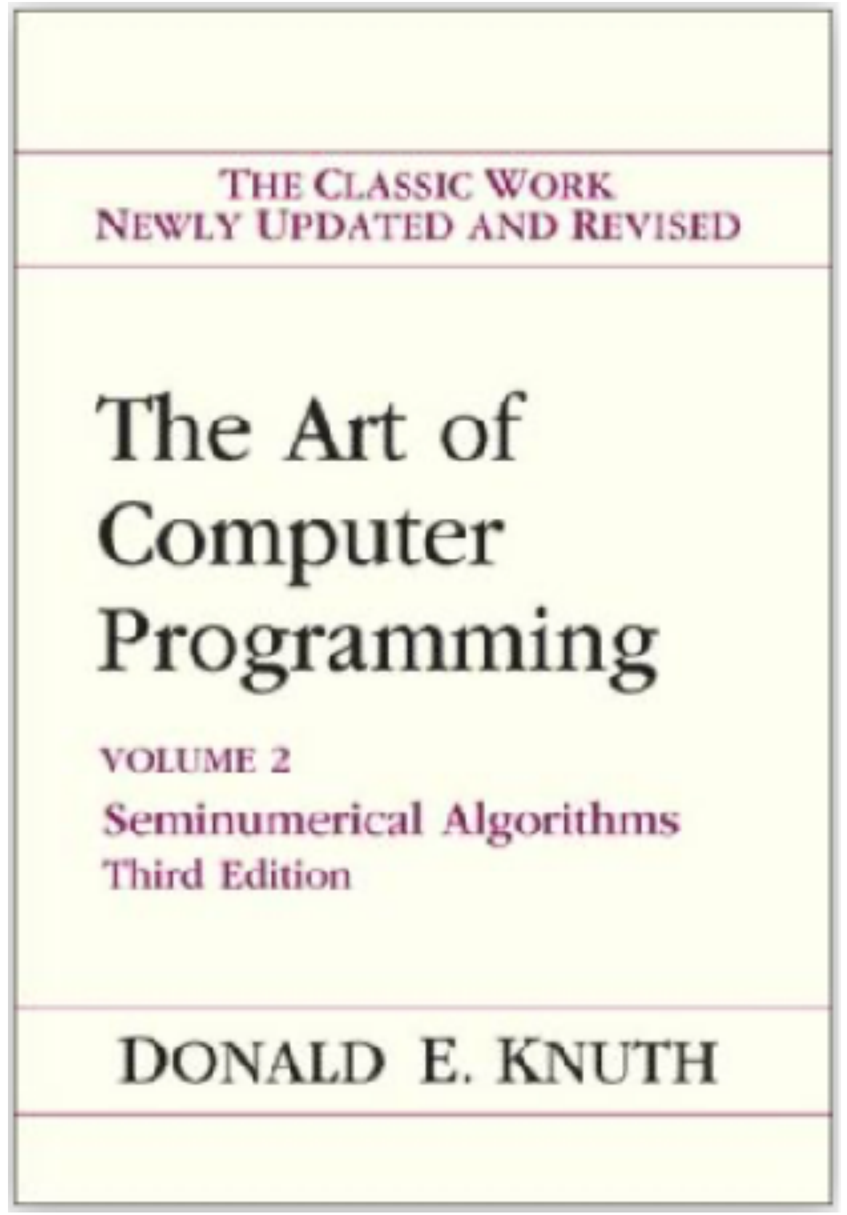
iris hep

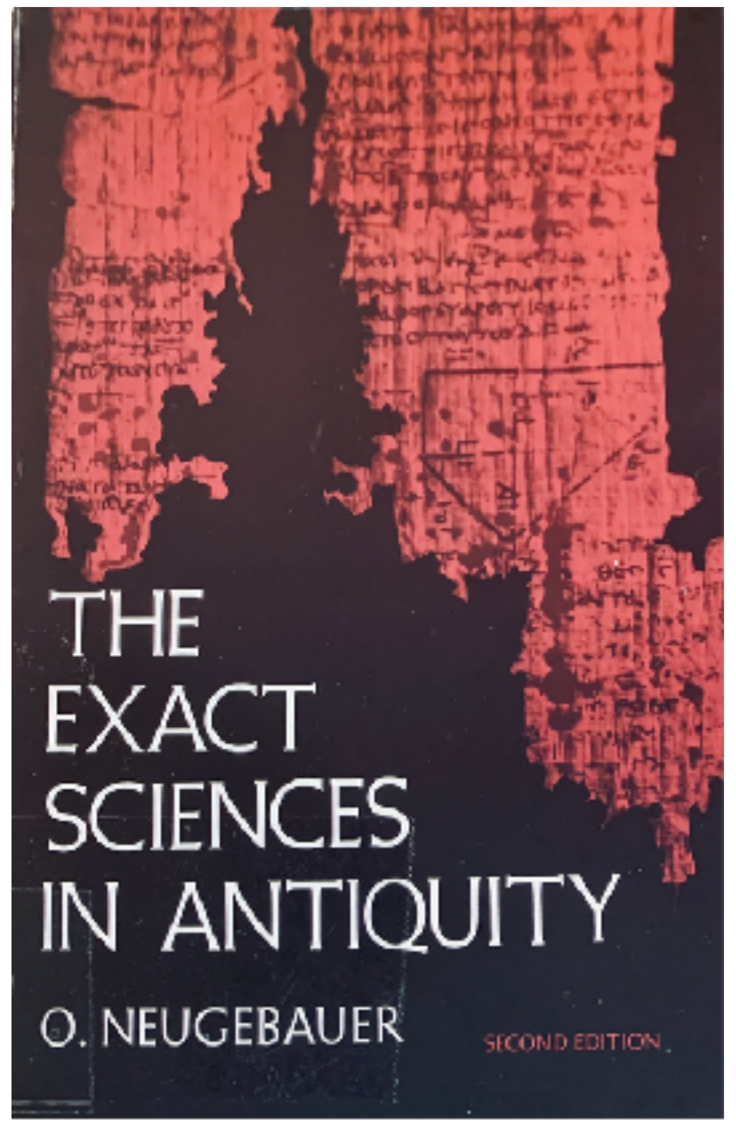
Lessons Learned

- Representing real numbers in a computer always involves an approximation and a potential loss of significant digits
- Testing for the equality of two real numbers is not a realistic way to think when dealing with the numbers in a computer. It is more realistic to test the difference of two numbers with respect to machine epsilon.
- Performing arithmetic on very small or very large numbers can lead to errors that are not possible in abstract mathematics. We can get underflow and overflow, and the order in which we do arithmetic can be important. This is something to be aware of when writing low-level software.
- The more bits we use to represent a number, the greater the precision of the representation and the more memory we consume.
https://www.stat.berkeley.edu/~nolan/stat133/Spr04/chapters/representations.pdf

References

- Donald E. Knuth "The Art of Computer Programming", volume 2 / Seminumerical Algorithms
- What Every Computer Scientist Should Know About Floating-Point Arithmetic. Goldberg. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- Numerical Computing with IEEE Floating Point Arithmetic. Overton, SIAM 2001
- INFORMATIK fur Mathematiker und Physiker, Eine Einfuhrung in C++, Bernd Gartner, Michael Hoffmann
- Chapter 6, numerical considerations, Programming Massively Parallel Processors, A hand-on approach, 3rd edition, David B. Kirk and Wen-wei W. Hwu
- ESC18, "Optimal Floating Point Computation", Vincenzo Innocente, https://agenda.infn.it/event/16941/contributions/34831/attachments/24523/27966/ Vincenzo_OptimalFloatingPoint2018.pdf
- CoDas-HEP 2018, "Floating Point is Not Real", Matthieu Lefebvre, https://indico.cern.ch/event/707498/contributions/2916937/attachments/1691892/27 24587/codas_fpa.pdf
- CoDas-HEP 2022, "Floating Point Arithmetic is Not Real", Bei Wang, https://indico.cern.ch/event/1151367/contributions/4969874/attachments/2488652/4273517/FloatingPoint_CoDaS-HEP2022.pdf


THE CLASSIC WORK
NEWLY UPDATED AND REVISED
The Art of Computer Programming
VOLUME 2
Seminumerical Algorithms
Third Edition
DONALD E. KNUTH


THE EXACT SCIENCES IN ANTIQUITY
O. NEUGEBAUER
SECOND EDITION