How accurate we are? A refresher on the floating-point arithmetic and the standard library.

Bogusław Cyganek

AGH University of Science and Technology, Cracow, Poland

Floating-point presentation

- Simple experiments with FP numbers
- Specifics of FP representation
- Basic facts about FP
- FP distribution
- Round-off errors
- FP representation – IEEE 754 standard
- **Computational aspects – recipes, standard algorithms, etc.**

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 0.1 + 0.2 == 0.3 ? "absolutely" : "& U R welcome" ) << std::endl;
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 0.1 + 0.2 == 0.3 ? "absolutely" : "& U R welcome" ) << std::endl;
```

```
Come to my talk & U R welcome
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 0.1 + 0.2 == 0.3 ? "absolutely" : "& U R welcome" ) << std::endl;
```

```
Come to my talk & U R welcome
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 0.1 + 0.2 == 0.3 ? "absolutely" : "& U R welcome" ) << std::endl;
```

```
Come to my talk & U R welcome
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 0.1 + 0.2 == 0.3 ? "absolutely" : "& U R welcome" ) << std::endl;
```

```
Come to my talk & U R welcome
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( std::fabs( ( 0.1 + 0.2 ) - 0.3 ) < 1e-12 ? "absolutely" : "& U R welcome" );
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( std::fabs( ( 0.1 + 0.2 ) - 0.3 ) < 1e-12 ? "absolutely" : "& U R welcome" );
```

```
Come to my talk absolutely
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( std::fabs( ( 0.1 + 0.2 ) - 0.3 ) < 1e-12 ? "absolutely" : "& U R welcome" );
```

```
Come to my talk absolutely
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( std::fabs( ( 0.1 + 0.2 ) - 0.3 ) < 1e-12 ? "absolutely" : "& U R welcome" );
```

```
Come to my talk absolutely
```

?

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 1.0 + 2.0 == 3.0 ? "absolutely" : "& U R welcome" ) << std::endl;
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 1.0 + 2.0 == 3.0 ? "absolutely" : "& U R welcome" ) << std::endl;
```

```
Come to my talk absolutely
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 1.0 + 2.0 == 3.0 ? "absolutely" : "& U R welcome" ) << std::endl;
```

```
Come to my talk absolutely
```

Floating-point presentation – a simple test

```
std::cout << "Come to my talk " <<
+ ( 1.0 + 2.0 == 3.0 ? "absolutely" : "& U R welcome" ) << std::endl;
```

```
Come to my talk absolutely
```

Problem reformulation

Floating-point presentation – another example

```
// Let's generate some float values
std::vector< double > vec;

// How many times this loop iterates?

// Generate 10 values in the interval [0, 0.9] with step 0.1
for( double x { 0.0 }; x != 1.0; x += 0.1 )
        vec.push_back( x );
```

infinite…

Floating-point presentation – another example

```
// Let's generate some float values
std::vector< double > vec;

// How many times this loop iterates?

// Generate 10 values in the interval [0, 0.9] with step 0.1
for( double x { 0.0 }; x != 1.0; x += 0.1 )
        vec.push_back( x );
```

infinite…

A fix

```
// Generate 10 values in the interval [0, 0.9] with step 0.1
for( double x { 0.0 }; x < 1.0; x += 0.1 )
        vec.push_back( x );

for( auto a : vec )
        std::cout << std::setprecision( 24 ) << a << "\n";
```

A fix?

```
// Generate 10 values in the interval [0, 0.9] with step 0.1
for( double x { 0.0 }; x < 1.0; x += 0.1 )
        vec.push_back( x );

for( auto a : vec )
        std::cout << std::setprecision( 24 ) << a << "\n";
```

```
1       0
2       0.100000000000000005551115
3       0.200000000000000001110223
4       0.300000000000000044408921
5       0.400000000000000002220446
6       0.5
7       0.599999999999999977795540
8       0.699999999999999955591079
9       0.799999999999999933386619
10      0.899999999999999911182158
11      0.999999999999999888977698
```

A fix

```
// So, if we want exactly 10 successive values we can do
double x { 0.0 };
for( int i { 0 }; i < 10; ++ i )
        vec.push_back( x ), x += 0.1;

for( auto a : vec )
        std::cout << std::setprecision( 24 ) << a << "\n";
```

Floating-point presentation – another example

A fix

```
// So, if we want exactly 10 successive values we can do
double x { 0.0 };
for( int i { 0 }; i < 10; ++ i )
        vec.push_back( x ), x += 0.1;

for( auto a : vec )
        std::cout << std::setprecision( 24 ) << a << "\n";
```

```
1   0
2   0.10000000000000005551115
3   0.20000000000000001110223
4   0.30000000000000044408921
5   0.40000000000000002220446
6   0.5
7   0.59999999999999997779554
8   0.69999999999999955591079
9   0.79999999999999933386619
10  0.89999999999999911182158
```

How to compute a sum in C++?

```
1   0
2   0.10000000000000005551115
3   0.20000000000000001110223
4   0.30000000000000044408921
5   0.40000000000000002220446
6   0.5
7   0.59999999999999997779554
8   0.69999999999999955591079
9   0.79999999999999933386619
10  0.89999999999999911182158
```

```
std::cout     << "sum = "
              << std::accumulate( vec.begin(), vec.end(), 0 ) << "\n";
```

sum = 0

?

How to compute a sum in C++?

```
1  0
2  0.10000000000000005551115
3  0.20000000000000001110223
4  0.30000000000000044408921
5  0.40000000000000002220446
6  0.5
7  0.59999999999999997779554
8  0.69999999999999955591079
9  0.79999999999999933386619
10 0.89999999999999911182158
```

```
std::cout << "sum = "
          << std::accumulate( vec.begin(), vec.end(), 0.0 ) << "\n";
```

```
sum = 4.5
```

Rounding, this time worked for us…

How serious it is?

How serious it is?

Launch of the Ariane 5
in 1996

How serious it is?

Launch of the Ariane 5 in 1996

30 seconds later…

How serious it is?

- a floating-point roundoff error
- an unhandled hardware trap

explosion

Launch of the Ariane 5 in 1996

30 seconds later…

from Wikipedia…

37 seconds later…

Floating-point presentation

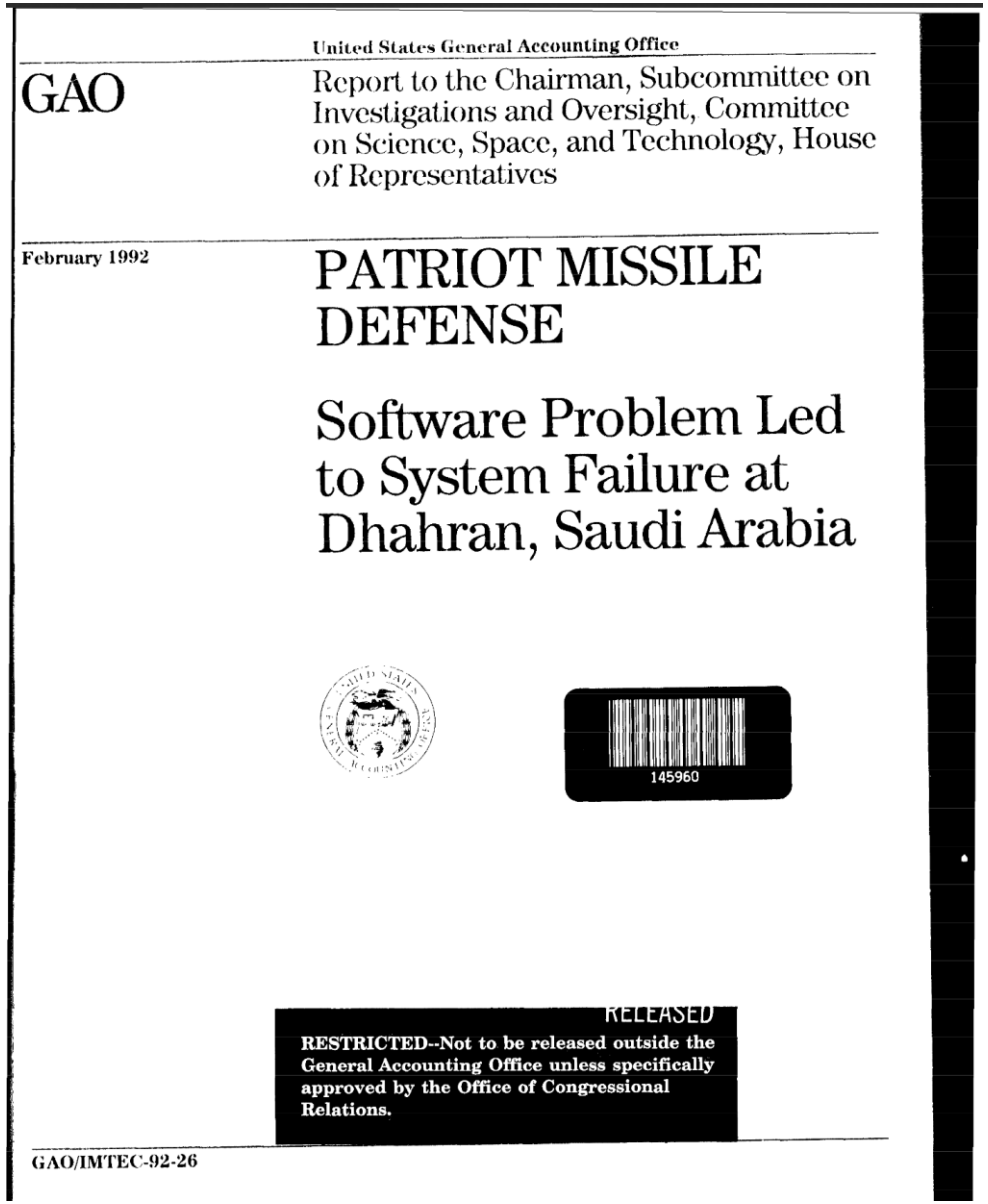
A data conversion from 64-bit floating point value to 16-bit signed integer value to be stored in a variable representing horizontal bias caused **a processor trap** (operand error) because **the floating point value was too large to be represented by a 16-bit signed integer**.

The software was originally written for the Ariane 4 where efficiency considerations (the computer had an 80% maximum workload requirement) led to four variables being protected with a handler while three others, including the horizontal bias variable, **were left unprotected** because it was thought that they were "physically limited or that there was a large margin of safety"…

(The software was written in Ada)

from Wikipedia…

Floating-point presentation



United States General Accounting Office

GAO

Report to the Chairman, Subcommittee on Investigations and Oversight, Committee on Science, Space, and Technology, House of Representatives

February 1992

PATRIOT MISSILE DEFENSE

Software Problem Led to System Failure at Dhahran, Saudi Arabia

GAO/IMTEC-92-26

https://www.gao.gov/assets/220/215614.pdf

Floating-point presentation

Deadly round-off error failure of the Patriot system in Dhahran 1991:

On February 25, 1991, a Patriot missile defence system operating in Dhahran, Saudi Arabia, failed to engage an incoming Scud missile. The missile struck U.S. Army barracks **killing 28 soldiers and injuring 98**. The reason for the failure of the Patriot was a fixed-point round-off error in the range-gate algorithm of the Patriot radar unit's tracking system.

From: http://www-users.math.umn.edu/~arnold//disasters/patriot.html and from from the paper by Timur Sağlam

Floating-point presentation

The radar of a Patriot missile system is designed in a way that it has to detect an incoming missile twice in order to avoid false alarms.

Once an incoming missile is detected, the system calculates where the incoming missile is expected to be after a certain time. If the incoming missile is detected at that position after the time expired it is confirmed that the target is actually a missile.

The Patriot missile is only launched to intercept after the incoming missile is detected a second time. This is a mechanism to avoid false alarms and to avoid shooting down other flying targets (e.g. airplanes).

Floating-point presentation

The problem of the Patriot system was that **a 24 bit number was used to measure time**, and it **was incremented by 1/10**.

Floating-point presentation

The problem of the Patriot system was that **a 24 bit number was used to measure time**, and it **was incremented by 1/10**.

When converting 1/10 to binary, it results in 0011110111001100110011001101… with an infinite number of bits. When cut off after 24 bits, the number is 001111011100110011001100, resulting in an error of the remaining bits 0000000000000000000000110011001… which is about 0.000000095 in decimal.

Floating-point presentation

The problem of the Patriot system was that **a 24 bit number was used to measure time**, and it **was incremented by 1/10**.

When converting 1/10 to binary, it results in 0011110111001100110011001101… with an infinite number of bits. When cut off after 24 bits, the number is 001111011100110011001100, resulting in an error of the remaining bits 0000000000000000000000110011001… which is about 0.000000095 in decimal.

This means that every second, the time was off by 10*0.000000095, which results in about **a third of a second after 100 hours** system operation time. Since the speed of a Scud missile is over 1500 m/s, it can travel **about 500 metres within a third of a second**. This error in the time calculation caused the Patriot system to expect an incoming missile at a wrong location for the second detection, causing it to consider the first detection as false alarm. The incoming Scud missile was not intercepted and it hit some barracks, killing 28 soldiers…

Floating-point presentation

Two weeks before the incident, Army officials received Israeli data indicating some loss in accuracy after the system had been running for 8 consecutive hours.

Consequently, Army officials modified the software to improve the system's accuracy.

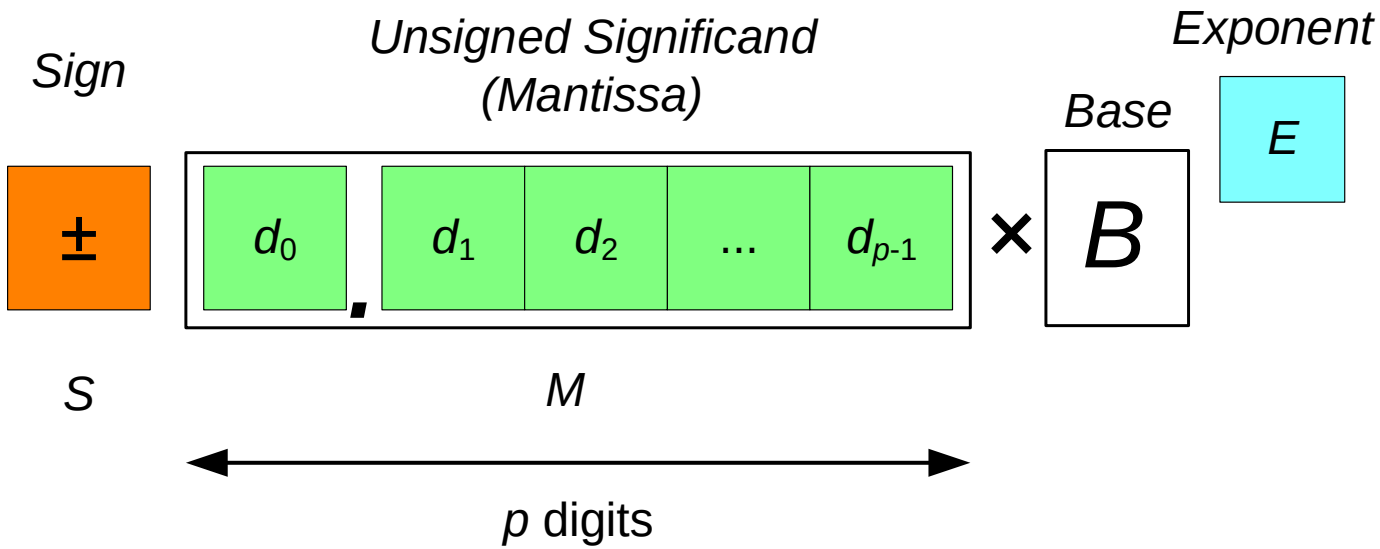
However, the modified software did not reach Dhahran until February 26,1991 - *the day after the Scud incident...*

From: https://www.gao.gov/assets/220/215614.pdf

Let us understand the FP

Let us do conscious computations with the FP

Number representation in the floating-point format



A value D of a number is given as follows

$$D = (-1)^S \cdot M \cdot B^E = (-1)^S \cdot (d_o \bullet d_1 \ldots d_{p-1}) \cdot B^E$$

M is unsigned **significand** (**mantissa**, fraction), *B* is a base, and *E* denotes the exponent

$$E_{\min} \le E \le E_{\max}$$

For *p* digits and the base *B*, a value of the significand is given as follows

$$M = d_0 + d_1 \cdot B^{-1} + \cdots + d_{p-1} \cdot B^{-(p-1)}$$

Floating-point presentation – basic facts

- In the FP domain, **some real values cannot be exactly represented**.
- Due to the **roundoff errors**, in the FP domain some algebraic conditions do not always hold. For example it might happen that *the commutative low does not hold*, that is

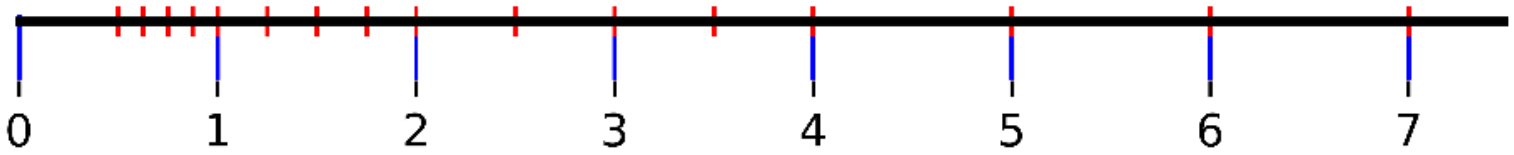
$$(a + b) + c \neq a + (b + c) \quad \text{for} \quad a, b, c \in \text{FP}.$$

- Floating point **representation of numbers is not unique**. The preferable representation of significand is with no leading zeros to retain the maximum number of significant bits, that is $d_0>0$. This is so called a **normalized form**.
- Normalization makes some problems with convenient representation of zero (preferably with all bits set to 0). Hence, *a special encoding is necessary (➔ denormal)*.
- Usually the exponent *E*, which can be negative, is represented in *a biased format* $E=E_{true}+bias$, in which its value is shifted, so *E is always positive* (this is so called the excess method). Such representation simplifies comparison since with this representation we can compare a number [*S*,*E*,*M*] as signed-magnitude numbers.

Floating-point number distribution

$$D = (-1)^S \cdot M \cdot B^E = (-1)^S \cdot (d_o \bullet d_1 \ldots d_{p-1}) \cdot B^E$$

p=3, B=2, E=[-1,2]

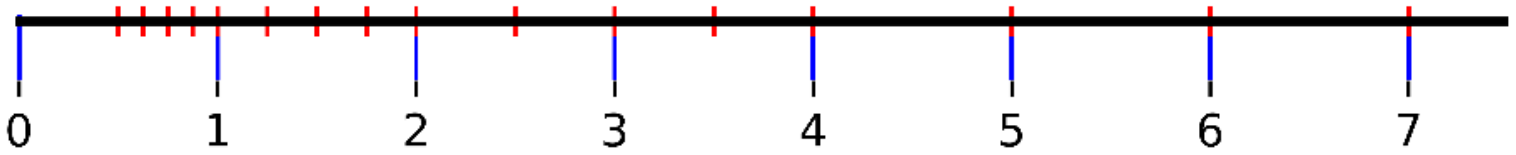


- Floating-point values are shown in **red**. It is easy to observe different groups of number "concentrations", corresponding to different exponents E.
- 4 groups are well visible which correspond to E=-1, 0, 1, and 2, respectively.
- Spacing within a group is the same.
- However, spacing between the groups increases by a factor of the base B.

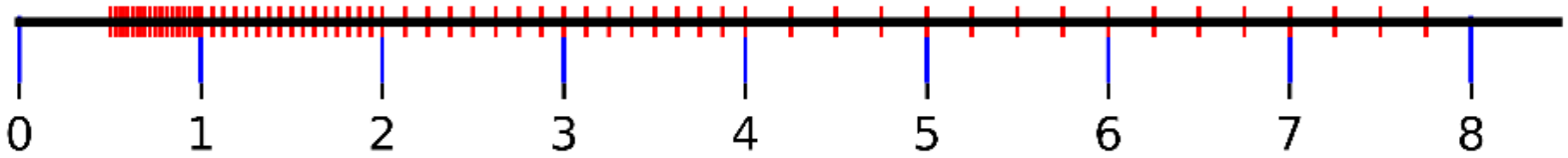
Floating-point number distribution

$$D = (-1)^S \cdot M \cdot B^E = (-1)^S \cdot (d_o \bullet d_1 \ldots d_{p-1}) \cdot B^E$$

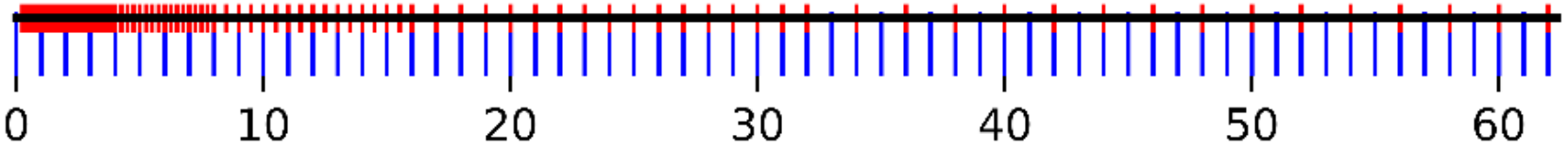
p=3, B=2, E=[-1,2]



p=5, B=2, E=[-1,2]



p=5, B=2, E=[-2,5]



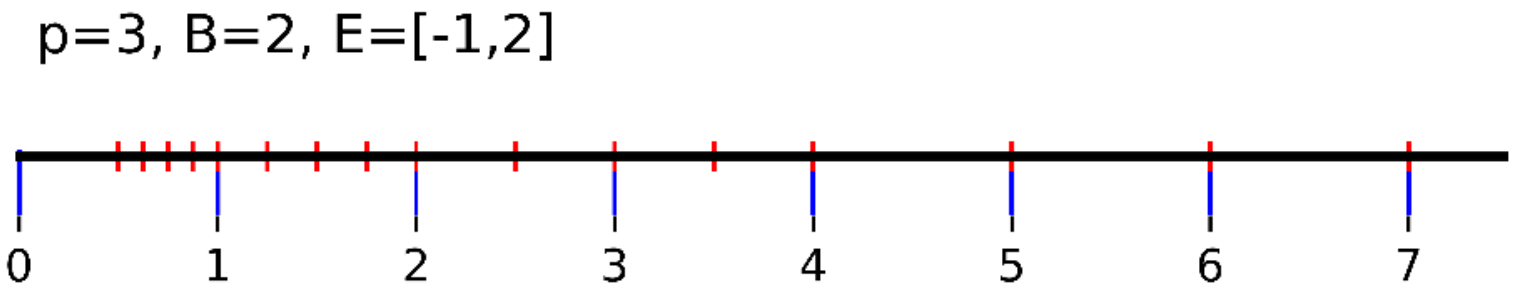
What are the most characteristic parameters?

The smallest possible positive value on the last position is simply for $d_{p-1}=1$

$$\varepsilon = 1 \cdot B^{-(p-1)} = B^{1-p}$$

The above constant ε, called *a machine epsilon*, is one of the most important values characterizing computations with FP numbers.
Since FP numbers are represented in the **normalized representation**, ε can be interpreted as a distance between value 1 and the closest larger value than 1 (but not from a 0).

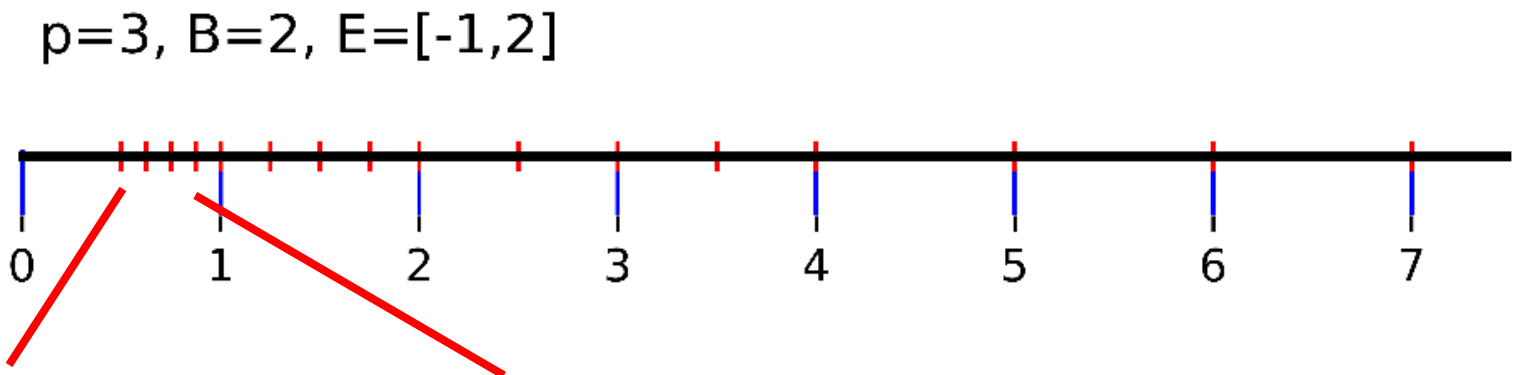
Floating-point representation – an illustrative experiment



$$D = (-1)^S \cdot M \cdot B^E$$

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 … 0.21875	0.25 … 0.4375	0.5 … 0.875	1 … 1.75

Floating-point representation – an illustrative experiment

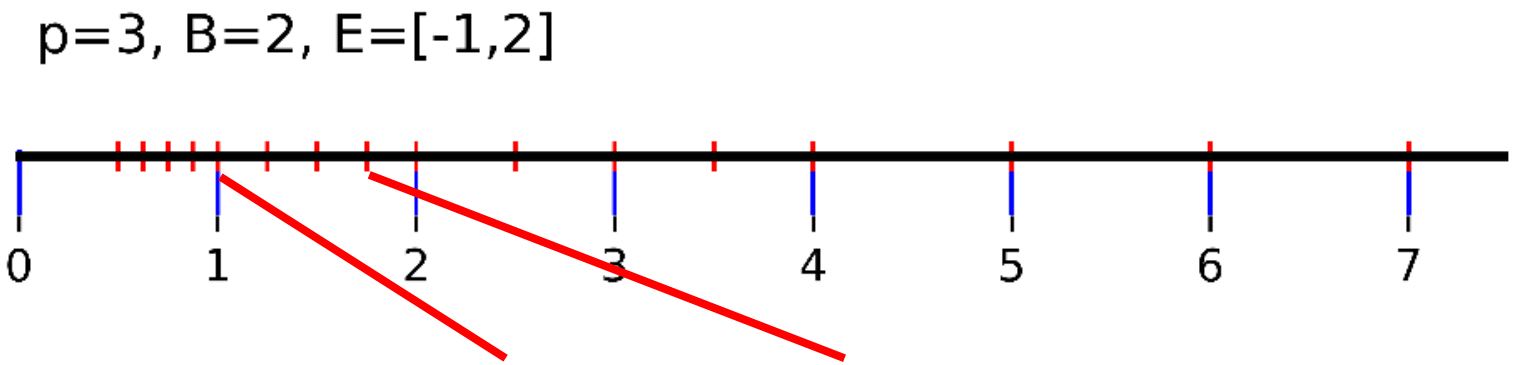


$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

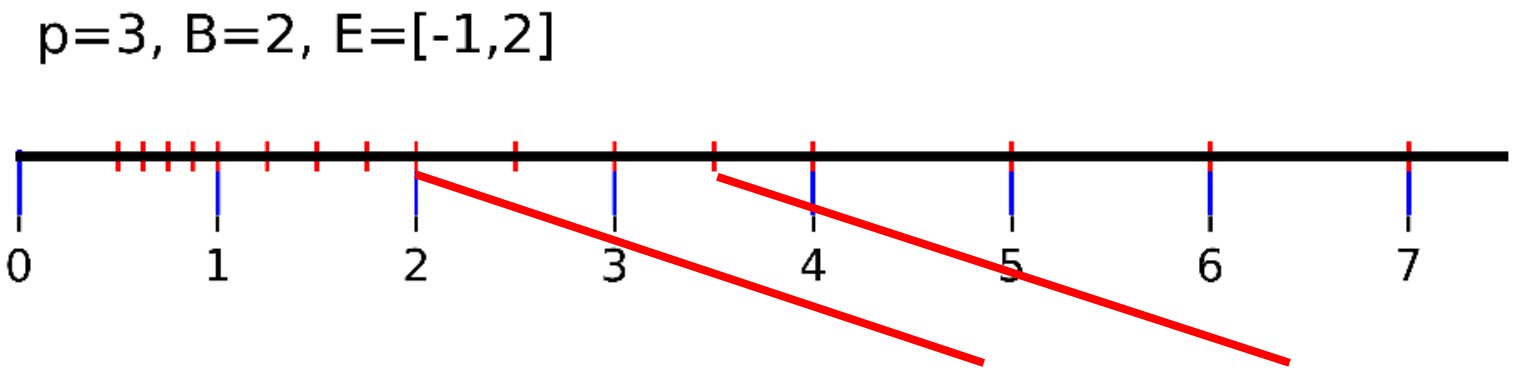


$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

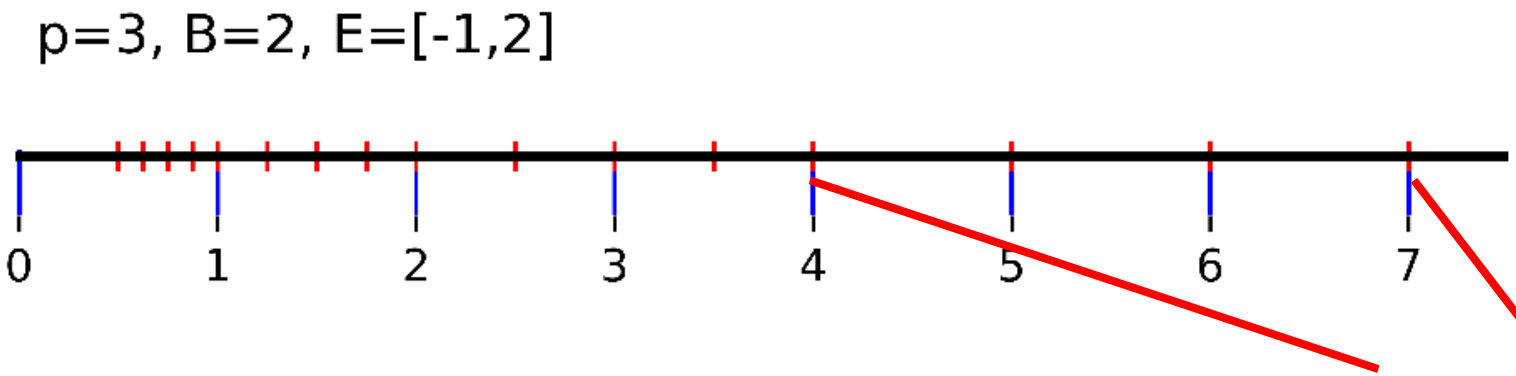


$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment



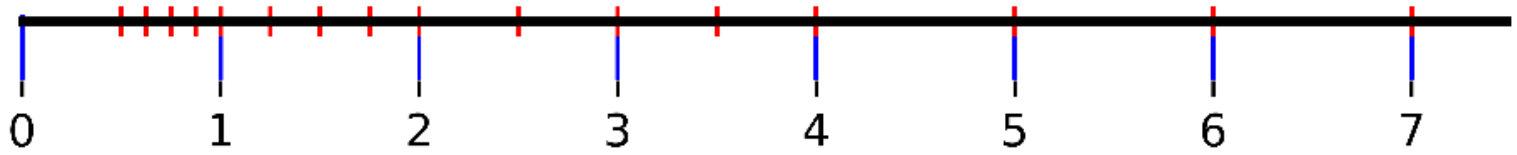
$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

p=3, B=2, E=[-1,2]



$$D = (-1)^S \cdot M \cdot B^E$$

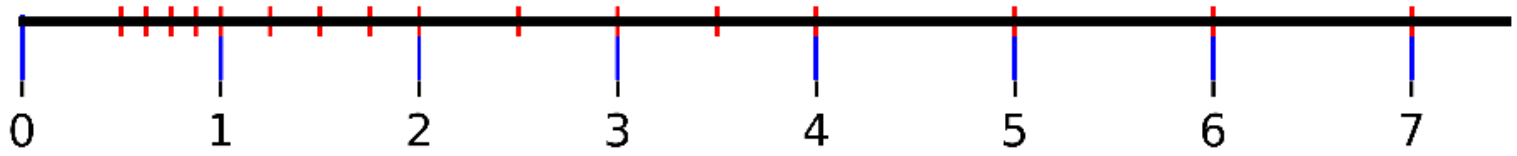
Exponent

machine epsilon

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon=B^{1-p}$	0.25	0.25	0.25	0.25
$\delta=\varepsilon\cdot B^E$	0.125	0.25	0.5	1
$s=\varepsilon\cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

p=3, B=2, E=[-1,2]



$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

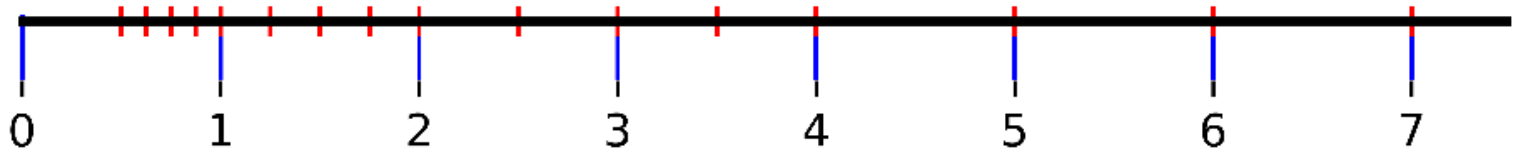
machine epsilon

spacing in a group

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

p=3, B=2, E=[-1,2]



$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

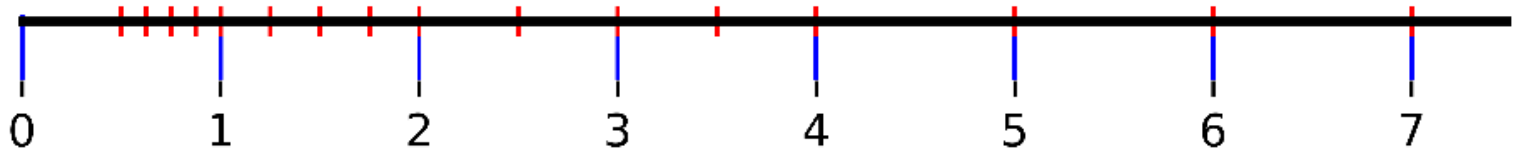
machine epsilon

spacing in a group

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

p=3, B=2, E=[-1,2]



$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

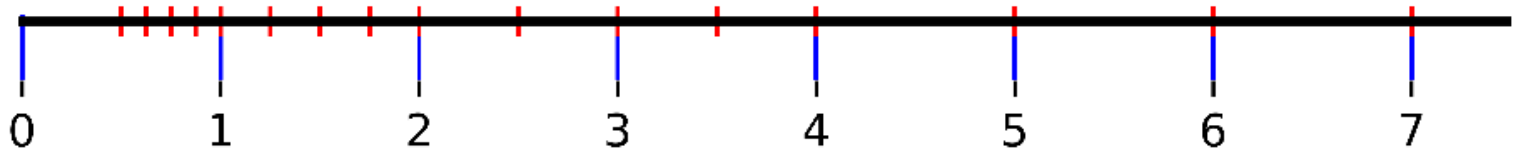
machine epsilon

spacing in a group

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

p=3, B=2, E=[-1,2]



$$D = (-1)^S \cdot M \cdot B^E$$

Exponent

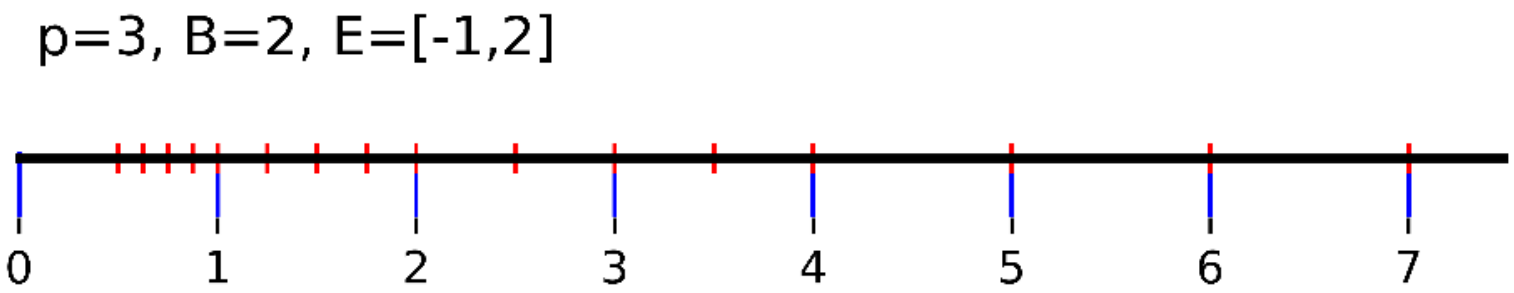
machine epsilon

spacing in a group

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 ... 0.21875	0.25 ... 0.4375	0.5 ... 0.875	1 ... 1.75

Floating-point representation – an illustrative experiment

Bingo!
We've a threshold



$$D = (-1)^S \cdot M \cdot B^E$$

i	0	1	2	3
D	0.5 0.625 0.75 0.875	1 1.25 1.5 1.75	2 2.5 3 3.5	4 5 6 7
E	-1	0	1	2
B^E	2^{-1}	2^0	2^1	2^2
$\varepsilon = B^{1-p}$	0.25	0.25	0.25	0.25
$\delta = \varepsilon \cdot B^E$	0.125	0.25	0.5	1
$s = \varepsilon \cdot D$	0.125 … 0.21875	0.25 … 0.4375	0.5 … 0.875	1 … 1.75

Exponent

machine epsilon

Max spacing in a group

Measuring distance between the floating-point numbers

Bingo!
We've a threshold

$$\delta \leq \varepsilon \cdot |D|$$

For a given value *D*, the product gives us a good upper approximation of the minimal FP value than can fit into the FP representation. Now we can use this property for a more conscious choice of a threshold:

```
double first_val{ 0.1 }, scnd_val{ 0.2 }, expected_result { 0.3 };

double sum = first_val + scnd_val;
double max_val = std::max( std::fabs( sum ), std::fabs( expected_result ) );

// Let us modify the threshold to be at least as the second argument
double kThresh { eps * max_val };

std::cout << "Come to my talk " <<
+ ( std::fabs( sum - expected_result ) <= kThresh ? "absolutely" : "& U R welcome" );
```

Measuring distance between the floating-point numbers

In numerical analysis many iterative algorithms follow a similar scheme of subtracting two FP values and checking the result. Usually, the following condition is checked

$$\left|x_{n+1} - x_n\right| < \tau$$

and if fulfilled, then iterations are stopped.

$$\boxed{\delta \leq \varepsilon \cdot \left|D\right|}$$

gives us a hint how to use the threshold τ

$$\boxed{\left|x_{n+1} - x_n\right| < \varepsilon \cdot \left|x_n\right|}$$

Measuring distance between the floating-point numbers

$$\left|x_{n+1} - x_n\right| < \varepsilon \cdot \left|x_n\right|$$

```
double x_n {}, x_n_1 {};

double thresh { 1e-12 }; // An anticipated convergence threshold
```

Measuring distance between the floating-point numbers

$$\left| x_{n+1} - x_n \right| < \varepsilon \cdot \left| x_n \right|$$

```
double x_n {}, x_n_1 {};

double thresh { 1e-12 }; // An anticipated convergence threshold

const size_t kMaxIters { 1000 }; // A fuse if computations do not converge

for( size_t n = 0; n < kMaxIters; ++ n )
{




}
```

Measuring distance between the floating-point numbers

$$\left|x_{n+1} - x_n\right| < \varepsilon \cdot \left|x_n\right|$$

```
double x_n {}, x_n_1 {};

double thresh { 1e-12 }; // An anticipated convergence threshold

const size_t kMaxIters { 1000 }; // A fuse if computations do not converge

for( size_t n = 0; n < kMaxIters; ++ n )
{
        // do computations, x_n_1 is a new value of x_n ...




}
```

Measuring distance between the floating-point numbers

$$\left| x_{n+1} - x_n \right| < \varepsilon \cdot \left| x_n \right|$$

```
double x_n {}, x_n_1 {};

double thresh { 1e-12 }; // An anticipated convergence threshold

const size_t kMaxIters { 1000 }; // A fuse if computations do not converge

for( size_t n = 0; n < kMaxIters; ++ n )
{
        // do computations, x_n_1 is a new value of x_n ...

        thresh = std::max( thresh, eps * std::fabs( x_n ) );




}
```

Measuring distance between the floating-point numbers

$$\left|x_{n+1} - x_n\right| < \varepsilon \cdot \left|x_n\right|$$

```
double x_n {}, x_n_1 {};

double thresh { 1e-12 }; // An anticipated convergence threshold

const size_t kMaxIters { 1000 }; // A fuse if computations do not converge

for( size_t n = 0; n < kMaxIters; ++ n )
{
        // do computations, x_n_1 is a new value of x_n ...

        thresh = std::max( thresh, eps * std::fabs( x_n ) );

        if( std::fabs( x_n_1 - x_n ) <= thresh )
                break;          // x_n_1 and x_n are approximately equal


}
```

Measuring distance between the floating-point numbers

$$\left|x_{n+1} - x_n\right| < \varepsilon \cdot \left|x_n\right|$$

```
double x_n {}, x_n_1 {};

double thresh { 1e-12 }; // An anticipated convergence threshold

const size_t kMaxIters { 1000 }; // A fuse if computations do not converge

for( size_t n = 0; n < kMaxIters; ++ n )
{
        // do computations, x_n_1 is a new value of x_n ...

        thresh = std::max( thresh, eps * std::fabs( x_n ) );

        if( std::fabs( x_n_1 - x_n ) <= thresh )
                break;          // x_n_1 and x_n are approximately equal

        x_n = x_n_1;// copy for the next iteration
}
```

What is and how to compute the machine ε

```
// Let us find epsilon for the single precision floating point
const float kBase { 2.0f };

// We will start from this and then this will be successively halved
const float kEpsInit { 1.0f };

// Stores the lastly computed epsilon
float store_eps {};

// Iterate as far as adding eps adds nothing
for( float eps = kEpsInit; 1.0f + eps != 1.0f; eps /= kBase )
        store_eps = eps;// We need to catch the one before the last

cout << "Machine epsilon = " << store_eps << endl;
```

```
Machine epsilon = 1.19209e-07
```

What is and how to compute the machine ε

```
// Let us find epsilon for the single precision floating point
const float kBase { 2.0f };

// We will start from this and then this will be successively halved
const float kEpsInit { 1.0f };

// Stores the lastly computed epsilon
float store_eps {};

// Iterate as far as adding eps adds nothing
for( float eps = kEpsInit; 1.0f + eps != 1.0f; eps /= kBase )
        store_eps = eps;// We need to catch the one before the last

cout << "Machine epsilon = " << store_eps << endl;
cout << "Machine epsilon = " << numeric_limits< float >::epsilon() << endl;
```

```
Machine epsilon = 1.19209e-07
Machine epsilon = 1.19209e-07
```

What is and how to compute the machine ε - **numeric_limits< T >**

```
#include <limits>
```

```
cout << "numeric_limits< double >::epsilon() = " << numeric_limits< double >::epsilon() << endl;
```

FP parameters with the **numeric_limits< T >**

```
#include <limits>
```

```
cout << "numeric_limits< double >::epsilon() = " << numeric_limits< double >::epsilon() << endl;

cout << "numeric_limits< double >::radix = " << numeric_limits< double >::radix << endl;
cout << "numeric_limits< double >::digits (mantissa) = " << numeric_limits< double >::digits << endl;
```

FP parameters with the numeric_limits< T >

```
#include <limits>
```

```
cout << "numeric_limits< double >::epsilon() = " << numeric_limits< double >::epsilon() << endl;

cout << "numeric_limits< double >::radix = " << numeric_limits< double >::radix << endl;
cout << "numeric_limits< double >::digits (mantissa) = " << numeric_limits< double >::digits << endl;

cout << "numeric_limits< double >::min() = " << numeric_limits< double >::min() << endl;
cout << "numeric_limits< double >::max() = " << numeric_limits< double >::max() << endl;
```

FP parameters with the **numeric_limits< T >**

```
#include <limits>
```

```
cout << "numeric_limits< double >::epsilon() = " << numeric_limits< double >::epsilon() << endl;

cout << "numeric_limits< double >::radix = " << numeric_limits< double >::radix << endl;
cout << "numeric_limits< double >::digits (mantissa) = " << numeric_limits< double >::digits << endl;

cout << "numeric_limits< double >::min() = " << numeric_limits< double >::min() << endl;
cout << "numeric_limits< double >::max() = " << numeric_limits< double >::max() << endl;

cout << "numeric_limits< double >::has_denorm = " << numeric_limits< double >::has_denorm << endl;
cout << "numeric_limits< double >::denorm_min() = " << numeric_limits< double >::denorm_min() << endl;
cout << "numeric_limits< double >::lowest() = " << numeric_limits< double >::lowest() << endl;
```

FP parameters with the numeric_limits< T >

```
#include <limits>
```

```
cout << "numeric_limits< double >::epsilon() = " << numeric_limits< double >::epsilon() << endl;

cout << "numeric_limits< double >::radix = " << numeric_limits< double >::radix << endl;
cout << "numeric_limits< double >::digits (mantissa) = " << numeric_limits< double >::digits << endl;

cout << "numeric_limits< double >::min() = " << numeric_limits< double >::min() << endl;
cout << "numeric_limits< double >::max() = " << numeric_limits< double >::max() << endl;

cout << "numeric_limits< double >::has_denorm = " << numeric_limits< double >::has_denorm << endl;
cout << "numeric_limits< double >::denorm_min() = " << numeric_limits< double >::denorm_min() << endl;
cout << "numeric_limits< double >::lowest() = " << numeric_limits< double >::lowest() << endl;

cout << "numeric_limits< double >::has_infinity = " << numeric_limits< double >::has_infinity << endl;
```

FP parameters with the numeric_limits< T >

```
#include <limits>
```

```
cout << "numeric_limits< double >::epsilon() = " << numeric_limits< double >::epsilon() << endl;

cout << "numeric_limits< double >::radix = " << numeric_limits< double >::radix << endl;
cout << "numeric_limits< double >::digits (mantissa) = " << numeric_limits< double >::digits << endl;

cout << "numeric_limits< double >::min() = " << numeric_limits< double >::min() << endl;
cout << "numeric_limits< double >::max() = " << numeric_limits< double >::max() << endl;

cout << "numeric_limits< double >::has_denorm = " << numeric_limits< double >::has_denorm << endl;
cout << "numeric_limits< double >::denorm_min() = " << numeric_limits< double >::denorm_min() << endl;
cout << "numeric_limits< double >::lowest() = " << numeric_limits< double >::lowest() << endl;

cout << "numeric_limits< double >::has_infinity = " << numeric_limits< double >::has_infinity << endl;

// round_to_nearest or round_toward_zero
cout << "numeric_limits< double >::round_style = " <<
( numeric_limits< double >::round_style == std::round_to_nearest ? "round_to_nearest" : "round_toward_zero"
) << endl;
```

FP parameters with the numeric_limits< T >

```
#include <limits>
```

```
cout << "numeric_limits< double >::epsilon() = " << numeric_limits< double >::epsilon() << endl;

cout << "numeric_limits< double >::radix = " << numeric_limits< double >::radix << endl;
cout << "numeric_limits< double >::digits (mantissa) = " << numeric_limits< double >::digits << endl;

cout << "numeric_limits< double >::min() = " << numeric_limits< double >::min() << endl;
cout << "numeric_limits< double >::max() = " << numeric_limits< double >::max() << endl;

cout << "numeric_limits< double >::has_denorm = " << numeric_limits< double >::has_denorm << endl;
cout << "numeric_limits< double >::denorm_min() = " << numeric_limits< double >::denorm_min() << endl;
cout << "numeric_limits< double >::lowest() = " << numeric_limits< double >::lowest() << endl;

cout << "numeric_limits< double >::has_infinity = " << numeric_limits< double >::has_infinity << endl;

// round_to_nearest or round_toward_zero
cout << "numeric_limits< double >::round_style = " <<
( numeric_limits< double >::round_style == std::round_to_nearest ? "round_to_nearest" : "round_toward_zero"
) << endl;

cout << "numeric_limits< double >::round_error() = " << numeric_limits< double >::round_error() << endl;
```

FP parameters with the **numeric_limits< T >**

```
numeric_limits< double >::epsilon() = 2.22045e-16
numeric_limits< double >::radix = 2
numeric_limits< double >::digits (mantissa) = 53
numeric_limits< double >::min() = 2.22507e-308
numeric_limits< double >::max() = 1.79769e+308
numeric_limits< double >::has_denorm = 1
numeric_limits< double >::denorm_min() = 4.94066e-324
numeric_limits< double >::lowest() = -1.79769e+308
numeric_limits< double >::has_infinity = true
numeric_limits< double >::round_style = round_to_nearest
numeric_limits< double >::round_error() = 0.5
```

Returns the largest possible rounding error in ULPs (units in the last place).
This can vary from 0.5 (rounding to the nearest digit) to 1.0 (rounding to zero or to infinity).

FP parameters with the **numeric_limits< T >** - examples

```
// Play with denorms
const auto kInfinity = numeric_limits< double >::infinity();

double val = 0.0;
double next_from_val = nextafter( val, kInfinity );
cout    << "nextafter( " << setprecision( 20 ) << val
        << " ) = " << next_from_val
        << hexfloat << " (" << next_from_val << ")\n" << defaultfloat;

val = 1.0;
next_from_val = nextafter( val, kInfinity );
cout    << "nextafter( " << setprecision( 20 ) << val
        << " ) = " << next_from_val
        << hexfloat << " (" << next_from_val << ")\n" << defaultfloat;
```

```
nextafter( 0 ) = 4.9406564584124654418e-324 (0x0.0000000000010000000p-1022)
nextafter( 1 ) = 1.0000000000000222 (0x1.0000000000010000000p+0)
```

FP parameters with the numeric_limits< T > - examples

Be aware of a hardware trap...

```
// Play with special values
double zero { 0.0 };

double inf { 1.0 / zero };// let us generate infinity, no exception

cout << "infinity = " << inf << endl;
cout << "inf == inf ? " << ( inf == kInfinity ) << endl;
cout << "123.0 + inf == " << 123.0 + inf << endl;
cout << "123.0 / inf == " << 123.0 / inf << endl;

// Let us generate NaN

double nan = sqrt( -0.1 );

cout << "123.0 + nan == " << 123.0 + nan << endl;
```

Use with assert(…)

```
infinity = inf
inf == inf ? true
123.0 + inf == inf
123.0 / inf == 0
123.0 + nan == -nan(ind)
```

cout << pow(-3.0, (1.0 / 3.0))

The fractional power of a negative number gives a complex num

"not a number" or "indeterminate"

Floating-point rounding modes

Trying to fit any conceivable real number to FP representation, taking 32 or 64 bits, **inevitably requires choosing the closest FP representation**.

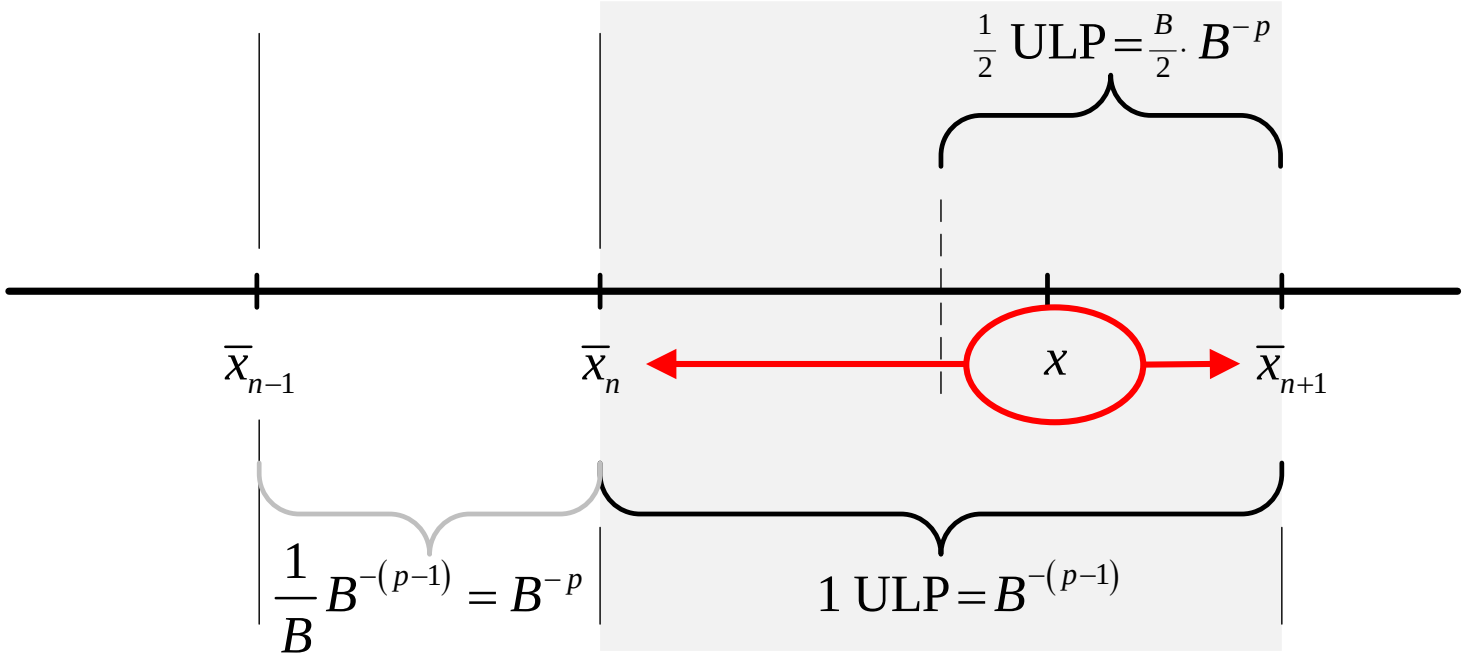
➔ *rounding error* (a *roundoff error*) is the characteristic feature of FP computations. However, even choosing the closest FP representation is not that straightforward, so there are many rounding strategies.

The following situations need to be properly signaled in the FP arithmetic:

- Overflow – means that the result is too large to be correctly represented.
- Underflow – the result is too small to be correctly represented.
- Inexact – the result cannot be represented exactly, so a rounded value is used instead.
- Invalid operation such as dividing by 0 or computing square root from a negative value.

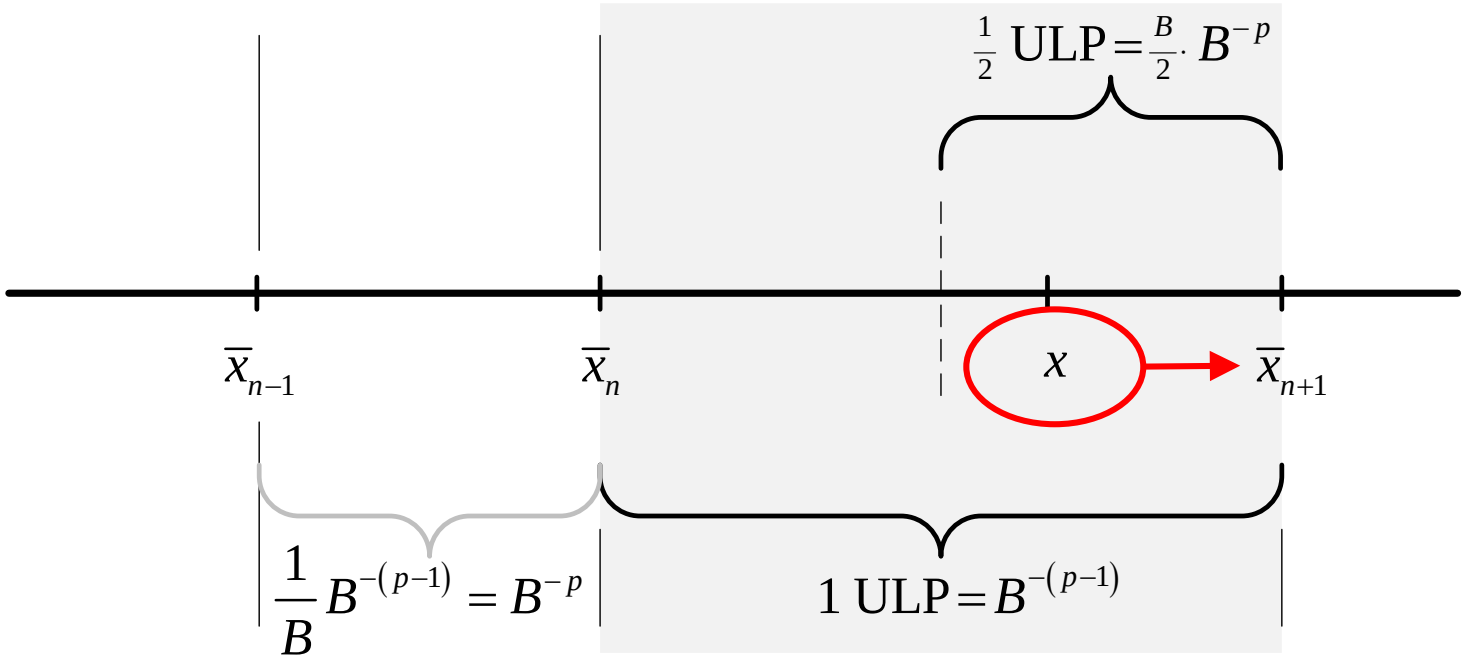
Floating-point rounding modes

When choosing **the nearest FP value** the maximum error is not greater than ½ of **units in the last place** (ULP).

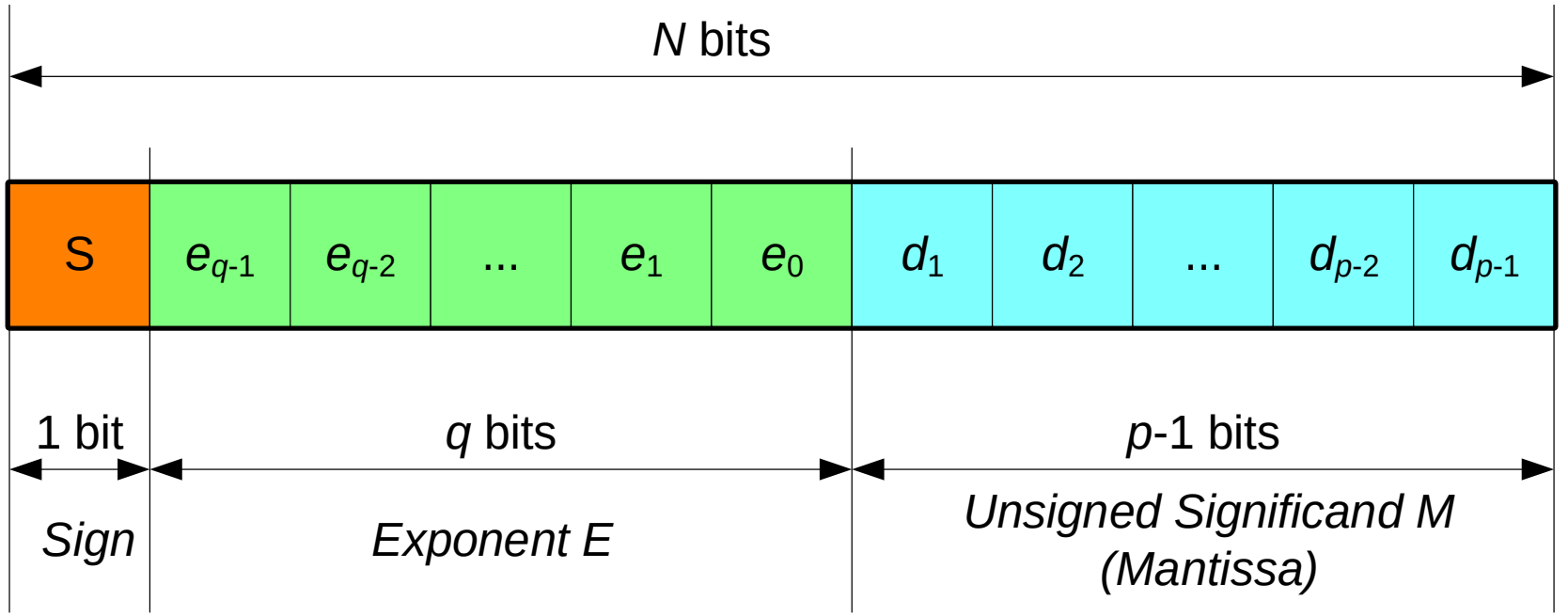


Floating-point rounding modes

When choosing **the nearest FP value** the maximum error is not greater than ½ of **units in the last place** (ULP).



The IEEE 754 Standard for Floating-Point Arithmetic



Bit representation of the floating-point values in the IEEE 754 standard. There are two formats: short with the total length of N=32, precision p=24 of the significand, and q=8 bits for the exponent, and long with N=53, p=53, q=11. The most significant bit d_0 of the significand is always 1 and does not need to be stored. This is the so called hidden bit.

The IEEE 754 Standard for Floating-Point Arithmetic

- IEEE 754 defines many FP formats from which the *single precision* and *double precision* are probably the most commonly encountered in C/C++ compilers.
  - Single precision format occupies 4 bytes (32 bits) and has p=24 and q=8 bits. In some C/C++ compilers represented with float.
  - Double precision format occupies 8 bytes (64 bits) and has p=53 and q=11 bits. In some C/C++ compilers represented with double.
  - Extended precision (double extended) format occupies 10 bytes (79 bits), p=64, q=15. In some C/C++ compilers represented with long double.
  - Quadruple precision occupies 16 bytes (128 bits) with p=113 bits (supported by some C/C++ compilers with long double or special types/flags).

The IEEE 754 Standard for Floating-Point Arithmetic

- The most-important-bit of the significand is not stored since in the normalized FP representation it is always 1. This is so called *a hidden bit* trick (Goldberg, 1991). This also explains why using the binary base B=2 is beneficial (recall also the smallest wobble). This feature explains also how the above bit partitions are organized considering also the S sign bit.
- Standard requires only that double **is at least as precise** as float and long double as double, respectively.

The IEEE 754 Standard for Floating-Point Arithmetic

- The standard defines some special values which are especially encoded:
  - Positive infinity (+∞).
  - Negative infinity (−∞).
  - Negative zero (0^-).
  - Positive zero (0^+).
  - Not-a-number (NaN) –used to represent for example results of forbidden mathematical operations avoiding an exception, such as a square root of a negative value, etc.
- The significand and exponent are especially encoded to represent the above special values. For other than above special encodings, a value 127 in a single, and 1023 in a double precision, are added to the exponent, respectively.

The IEEE 754 Standard for Floating-Point Arithmetic

- The ‘standard’ FP formats there is a problem with precise representation of 0 and values close to 0. To remedy this, there is the special encoding (all 0’s in the exponent bit and d_0=0 of the significand) to represent this group of values, which are called *denormals* (*subnormals*). However, in some systems using denormals comes with a significant run-time cost.

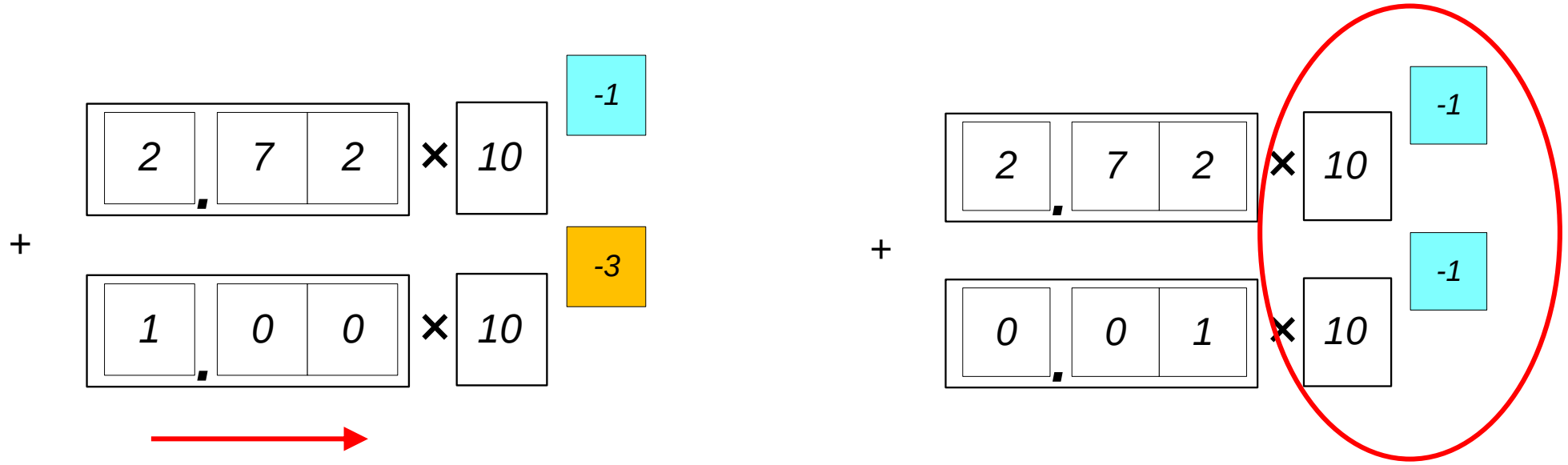
The IEEE 754 Standard for Floating-Point Arithmetic

- The IEEE 754 standard defines all necessary mathematical operations on FP values, such as addition, multiplication, but also roundings. Rounding is inevitable if the result cannot fit in the FP representation, e.g. due to insufficient number of bits for precision. There are five rounding modes, as follows:
  - **Default round to nearest** (ties round to the nearest even digit, i.e. to a value that makes the significand end in an even digit). It can be shown that rounding to nearest even leads to lowest errors.
  - Optional round to nearest (ties round away from 0).
  - Round up (toward $+\infty$).
  - Round down (toward $-\infty$).
  - Round toward 0 (cuts off fractional). This is also **the default rounding for the integer types**.

The IEEE 754 Standard for Floating-Point Arithmetic

- The IEEE 754 standard allows benign propagation of the exceptional conditions, such as overflow or division by zero, in the software controlled fashion rather than hardware interrupts.

Addition/subtraction of the FP numbers



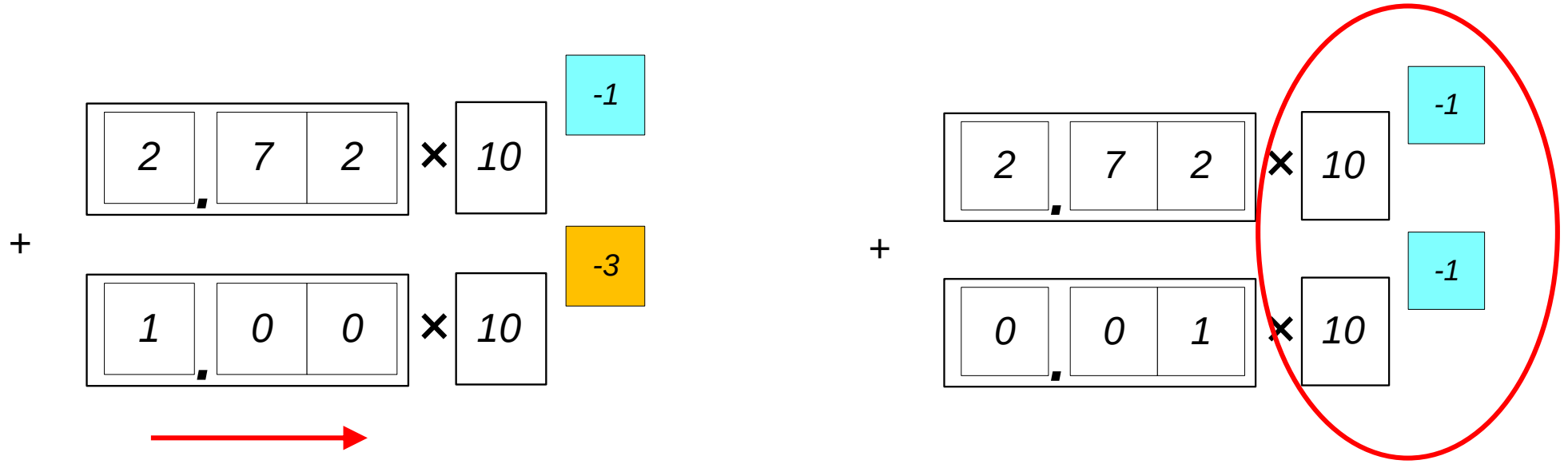
To perform addition the values must be scaled to have the same exponent. In the case of large differences in the exponents, this can lead to loss of the significant digits

In our example the following is obtained:

$2.72 \cdot 10^{-1} + 1.00 \cdot 10^{-3} = 2.72 \cdot 10^{-1} + 0.01 \cdot 10^{-1}$

$= 2.73 \cdot 10^{-1}$.

Addition/subtraction of the FP numbers



To perform addition the values must be scaled to have the same exponent. In the case of large differences in the exponents, this can lead to loss of the significant digits

However, if the latter operand was $1.00 \cdot 10^{-4}$ then, after the right shifting to obtain the common exponent 10^{-1}, the value could not be longer represented with only p=3 digits. The conclusion is that if we have a choice, e.g. when adding a series of FP numbers, then **we always should order the addition in such a way as to add values of as much as possible similar exponents**.

Addition/subtraction of the FP numbers

When doing FP computations we should be well aware of many peculiarities leading to imprecise or even totally incorrect results. The most obvious are the mentioned overflows and underflows. However, similarly dangerous are **subtractions of numbers which are nearly equal in magnitude.**

In such a case, majority of the leading digits vanish and the rounding error may occur which promotes the digit at the last place to a more significant position. For example, when subtracting 2.71828 – 2.71829 we end up with 1*e*-05 which reflects only the last digit since all the other cancelled out.

However, if 2.71828 and 2.71829 were already rounded off, then the left after subtraction last digits can be the less accurate ones, since they might be already inaccurate due to the previous roundings. That is, the left from subtraction digits convey wrong information. Even worse, it was also promoted to higher significant positions of the significand (in our example, 1 would in the first position).

Errors arising from such situations, called **catastrophic cancellations**, can be very severe.

Addition/subtraction of the FP numbers

Catastrophic cancellations can arise even in simple computations, such as when finding roots of the quadratic equation $ax^2+bx+c=0$:

$$x_1 = \frac{1}{2a}\left(-b-\sqrt{b^2-4ac}\right) \qquad x_2 = \frac{1}{2a}\left(-b+\sqrt{b^2-4ac}\right)$$

The problem may arise however if $ac \ll b^2$ and $b<0$. Then: $\sqrt{b^2-4ac} \approx b$

and x_1 will suffer from subtraction of two almost identical values.
A simple way out in this case is first to observe that

$$x_1 x_2 = c$$

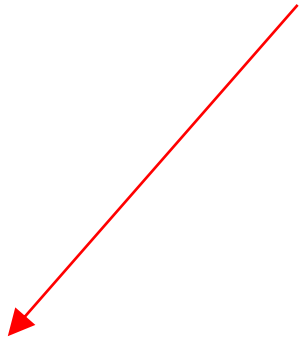
and to compute x_2, then finally x_1 from

Problem reformulation

Addition/subtraction of the FP numbers

```
double a {}, b {}, c {};
// Read in a, b, c ...

double x1 {}, x2 {};

double d = b * b - 4.0 * a * c;
if( d >= 0.0 )
{
        if( b < 0.0 )
        {
                x1 = ( -b + sqrt( d ) ) / ( a + a ); // –b becomes positive – addition
                x2 = c / x1;
        }
        else
        {
                x1 = ( -b - sqrt( d ) ) / ( a + a ); // addition of two negative values
                x2 = c / x1;
        }
}
```

std::fabs(a) > kThresh



Addition/subtraction of the FP numbers

There are many expressions which need closer look in the light of FP computations before implementation.

Yet another example is

$$x^2-y^2$$

which almost always is better to implement using the well known equivalent form

$$(x-y)(x+y)$$

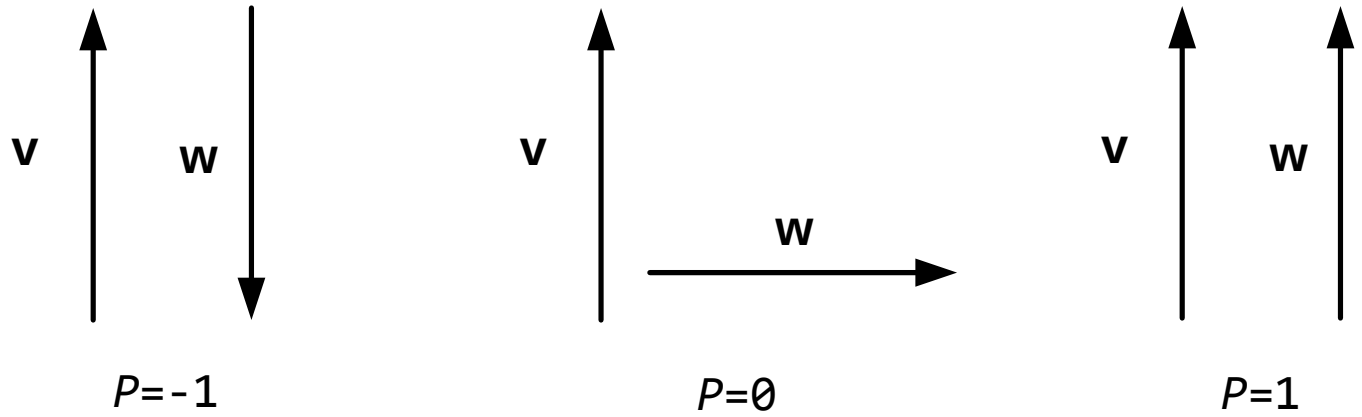
Not only we save on one multiplication, but most of all we avoid catastrophic cancellation errors which are most likely in its first representation.

CASE STUDY – summation, multiplication, inner product with the FP

Despite a great advances in hardware and software engineering, the problems of fast and exact multiplication and summation (MAC) belongs to the most fundamental ones in computational science (signal processing, pattern recognition, etc.)
So, let us focus upon computation of the MAC between two vectors **v** and **w**

$$P = \mathbf{v} \cdot \mathbf{w} = \sum_{i=0}^{N-1} v[i] \cdot w[i]$$

The inner product is that it conveys an information on a distance between two vectors



CASE STUDY – summation, multiplication, inner product with the FP

Let us test some algorithms:

- The simple multiply-and-add method.
- The method which computes the elementwise products, **sorts** them and does summation.
- The **Kahan summation** method, which applies **a correction factor** on the underflow bits.

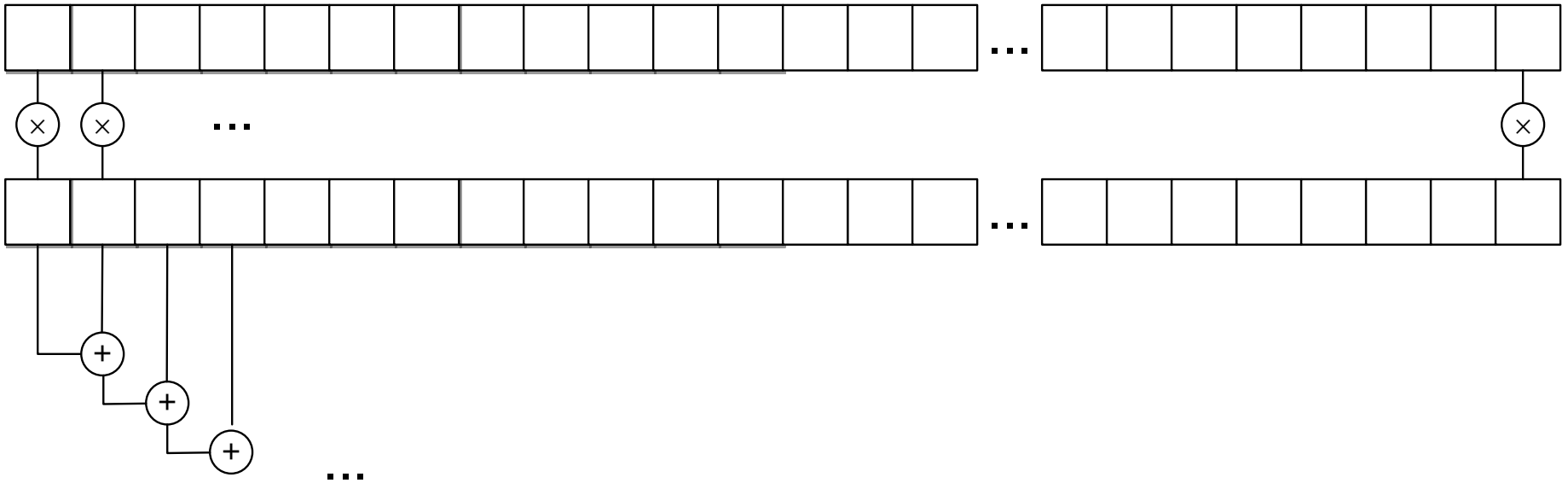
And two modes of operation:

- Sequential.
- Parallel.

$$(a + b) + c \neq a + (b + c)$$

CASE STUDY – summation, multiplication, inner product with the FP

The simple sequential multiply-and-add method



CASE STUDY – summation, multiplication, inner product with the FP

```
using DVec = std::vector< double >;
using DT = DVec::value_type;
using ST = DVec::size_type;

using std::inner_product;
using std::transform;
using std::accumulate;
using std::sort;

using std::cout, std::endl;

auto InnerProduct_StdAlg( const DVec & v, const DVec & w )
{
        // The last argument is an initial value
        return std::inner_product( v.begin(), v.end(), w.begin(), DT() );
}
```

CASE STUDY – summation, multiplication, inner product with the FP

An interesting improvement is first **to arrange the elements in order of their magnitudes**

CASE STUDY – summation, multiplication, inner product with the FP

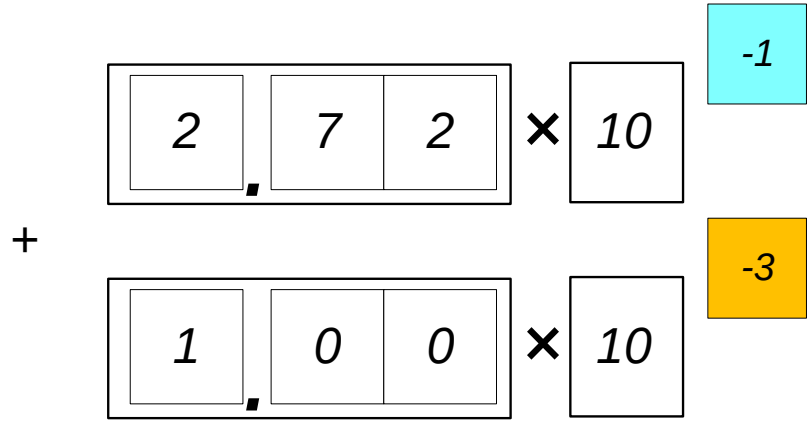
An interesting improvement is first **to arrange the elements in order of their magnitudes**, and then **to add them up**.

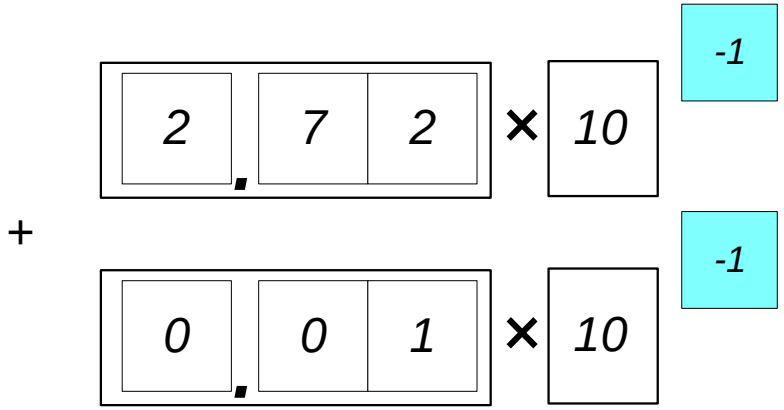
CASE STUDY – summation, multiplication, inner product with the FP

An interesting improvement is first **to arrange the elements in order of their magnitudes**, and then **to add them up**. Let us recall the scaling problem:

CASE STUDY – summation, multiplication, inner product with the FP

An interesting improvement is first **to arrange the elements in order of their magnitudes**, and then **to add them up**. Let us recall the scaling problem:





So, by arranging the elements due to their magnitude, we can avoid excessive shifts of significand which happens when adding FP numbers of highly different exponent.

In effect, the consecutive sums do not require much shifts of the summands and the result is usually more accurate compared to summation with arbitrary magnitudes.

The additional cost of this method comes mostly from the sorting...

CASE STUDY – summation, multiplication, inner product with the FP

An interesting improvement is first **to arrange the elements in order of their magnitudes**, and then **to add them up**. Let us recall the scaling problem:

```
auto InnerProduct_SortAlg( const DVec & v, const DVec & w )
{
        DVec z;         // Stores element-wise products

        // Elementwise multiplication: c = v .* w
        std::transform(         v.begin(), v.end(), w.begin(),
                                back_inserter( z ),
                [] ( const auto & v_el, const auto & w_el) { return v_el * w_el; } );

        // Sort in descending order
        std::sort(      z.begin(), z.end(),   // Is it magic?
                        [] ( const DT & p, const DT & q ) { return fabs(p) < fabs(q); } );

        // The last argument is an initial value
        return std::accumulate( z.begin(), z.end(), DT() );
}
```

CASE STUDY – summation, multiplication, inner product with the FP

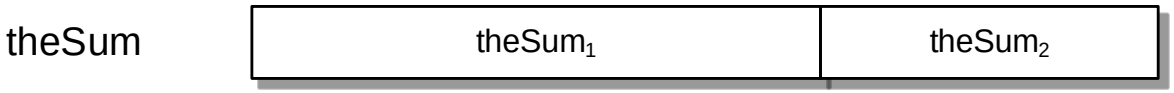
Yet another improvement comes in a form of the compensated summation (Kahan):

CASE STUDY – summation, multiplication, inner product with the FP

```
auto InnerProduct_KahanAlg( const DVec & v, const DVec & w )
{
        DT theSum {};
        volatile DT c {};// a "correction" coefficient
        const ST kElems = std::min( v.size(), w.size() );

        for( ST i = 0; i < kElems; ++ i )
        {
                DT y = v[ i ] * w[ i ] - c; // From y subtracts the correction factor

                DT t = theSum + y; // Add corrected summand to the running sum theSum
                // But theSum is big, y is small, so its lower bits will be lost

                c = ( t - theSum ) - y; // Low order bits of y are lost in the summation.
                // High order bits of y are computed in ( t - theSum ). Then, when y
                // is subtracted from this, the low order bits of y are recovered.

                theSum = t;
        }
        return theSum;
}
```

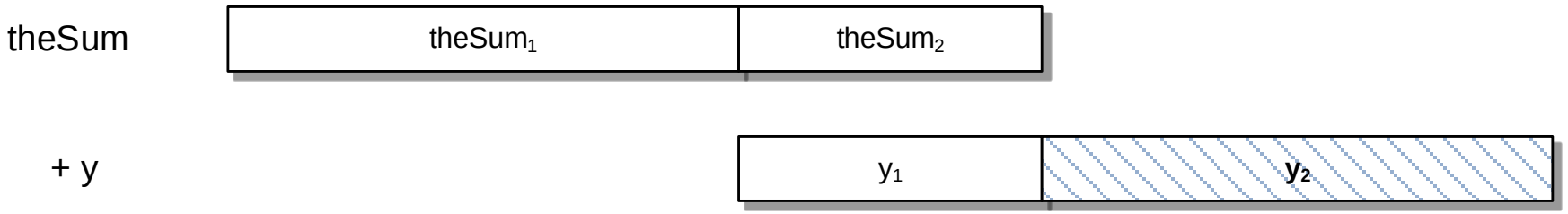
CASE STUDY – summation, multiplication, inner product with the FP

Yet another improvement comes in a form of the compensated summation (Kahan):



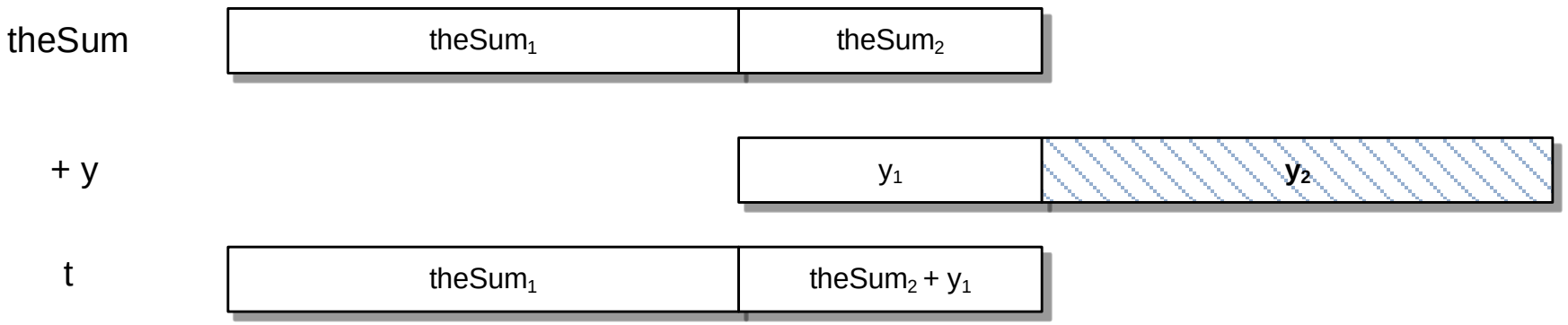
CASE STUDY – summation, multiplication, inner product with the FP

Yet another improvement comes in a form of the compensated summation (Kahan):



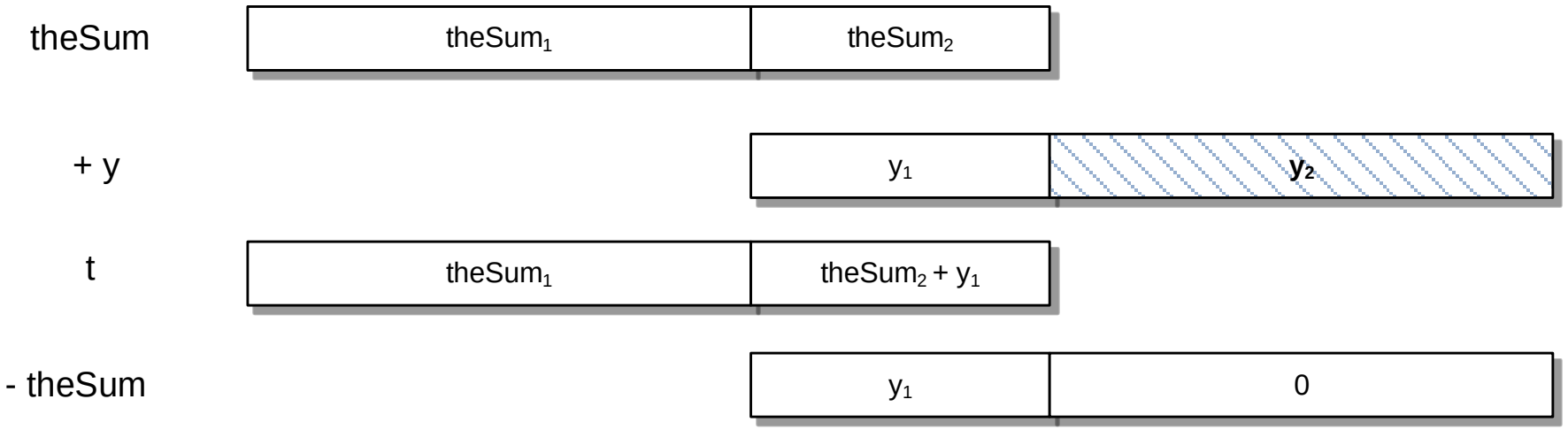
CASE STUDY – summation, multiplication, inner product with the FP

Yet another improvement comes in a form of the compensated summation (Kahan):



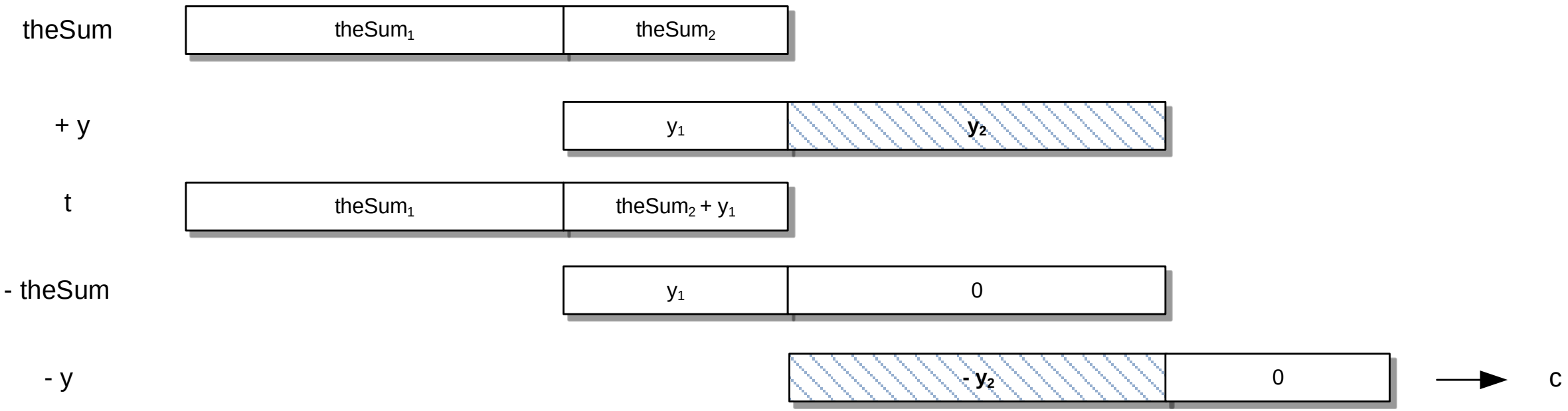
CASE STUDY – summation, multiplication, inner product with the FP

Yet another improvement comes in a form of the compensated summation (Kahan):



CASE STUDY – summation, multiplication, inner product with the FP

Yet another improvement comes in a form of the compensated summation (Kahan):



Explanation of the recovery of the rounding error y_2 in the compensated summation scheme implemented in the Kahan compensated summation algorithm.

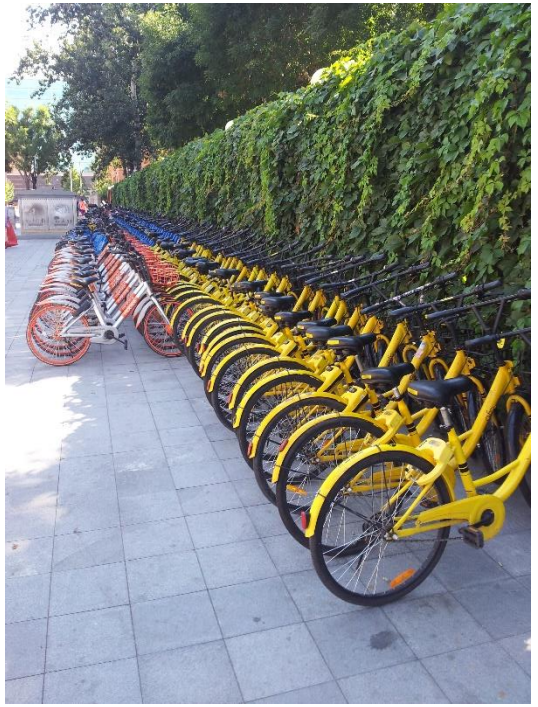
CASE STUDY – summation, multiplication, inner product with the FP

What can be done more?

Parallel implementation!

SL comes with the *Execution Policy* **std::execute::par**

However, not for all algorithms ...



CASE STUDY – summation, multiplication, inner product with the FP

STL comes with the *Execution Policy* **std::execute::par**

However, instead of the std::inner_product the std::transform_reduce has to be used.

```
// The transform-reduce parallel version
auto InnerProduct_TR_Alg( const DVec & v, const DVec & w )
{
	return std::transform_reduce(
			std::execution::par,
			v.begin(), v.end(), w.begin(), DT(),
			[] ( const auto a, const auto b ) { return a + b; },
			[] ( const auto a, const auto b ) { return a * b; }
		);
}
```

CASE STUDY – summation, multiplication, inner product with the FP

STL comes with the *Execution Policy* **std::execute::par**

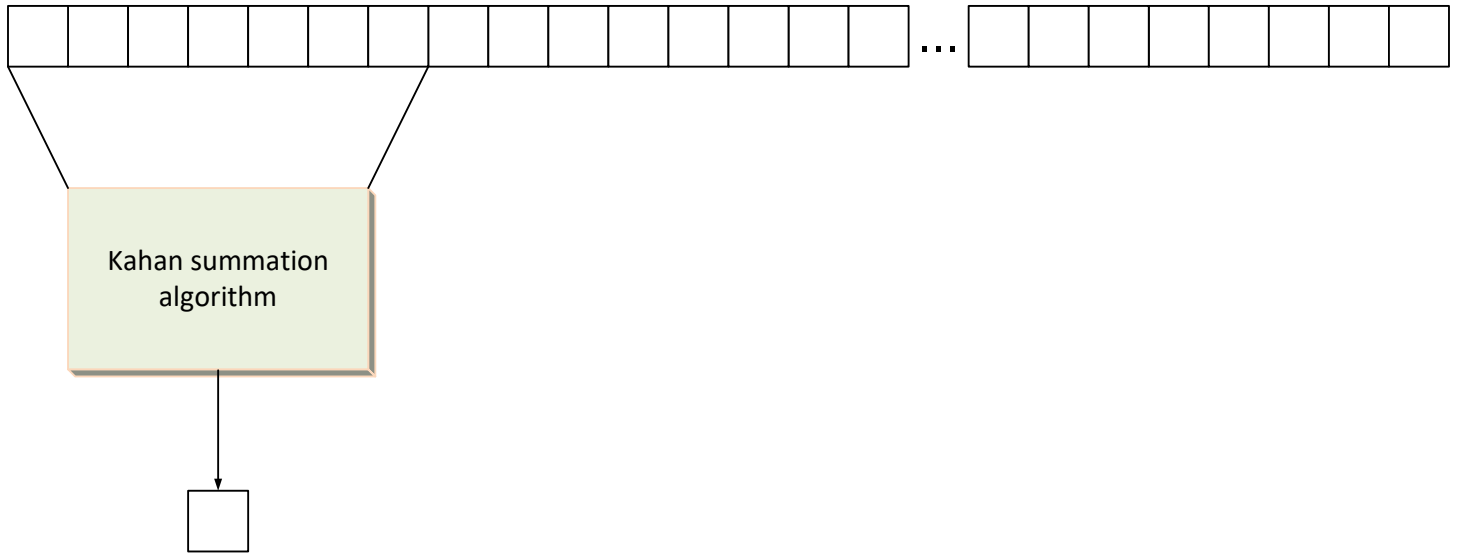
```
auto InnerProduct_SortAlg( const DVec & v, const DVec & w )
{
        DVec z( std::min( v.size(), w.size() ) ); // Stores element-wise products

        // Elementwise multiplication: c = a .* b
        std::transform( std::execution::par, v.begin(), v.end(), w.begin(), z.begin(),
                [] ( const auto & v_el, const auto & w_el) { return v_el * w_el; } );

        // Sort in descending order.
        std::sort( std::execution::par, z.begin(), z.end() ); // Is it magic?

        // The last argument is an initial value
        return std::accumulate( z.begin(), z.end(), DT() );
}
```

CASE STUDY – summation, multiplication, inner product with the FP

A hybrid algorithm is one of the best in terms of performance vs. accuracy.

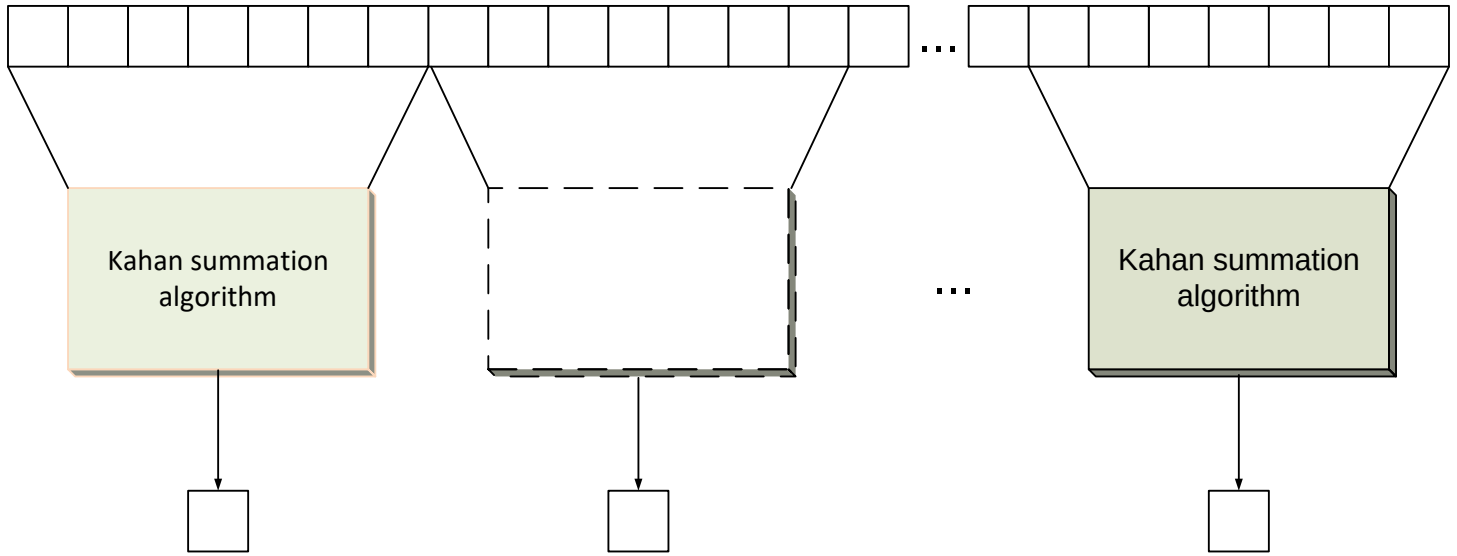
CASE STUDY – summation, multiplication, inner product with the FP

A hybrid algorithm is one of the best in terms of performance vs. accuracy.



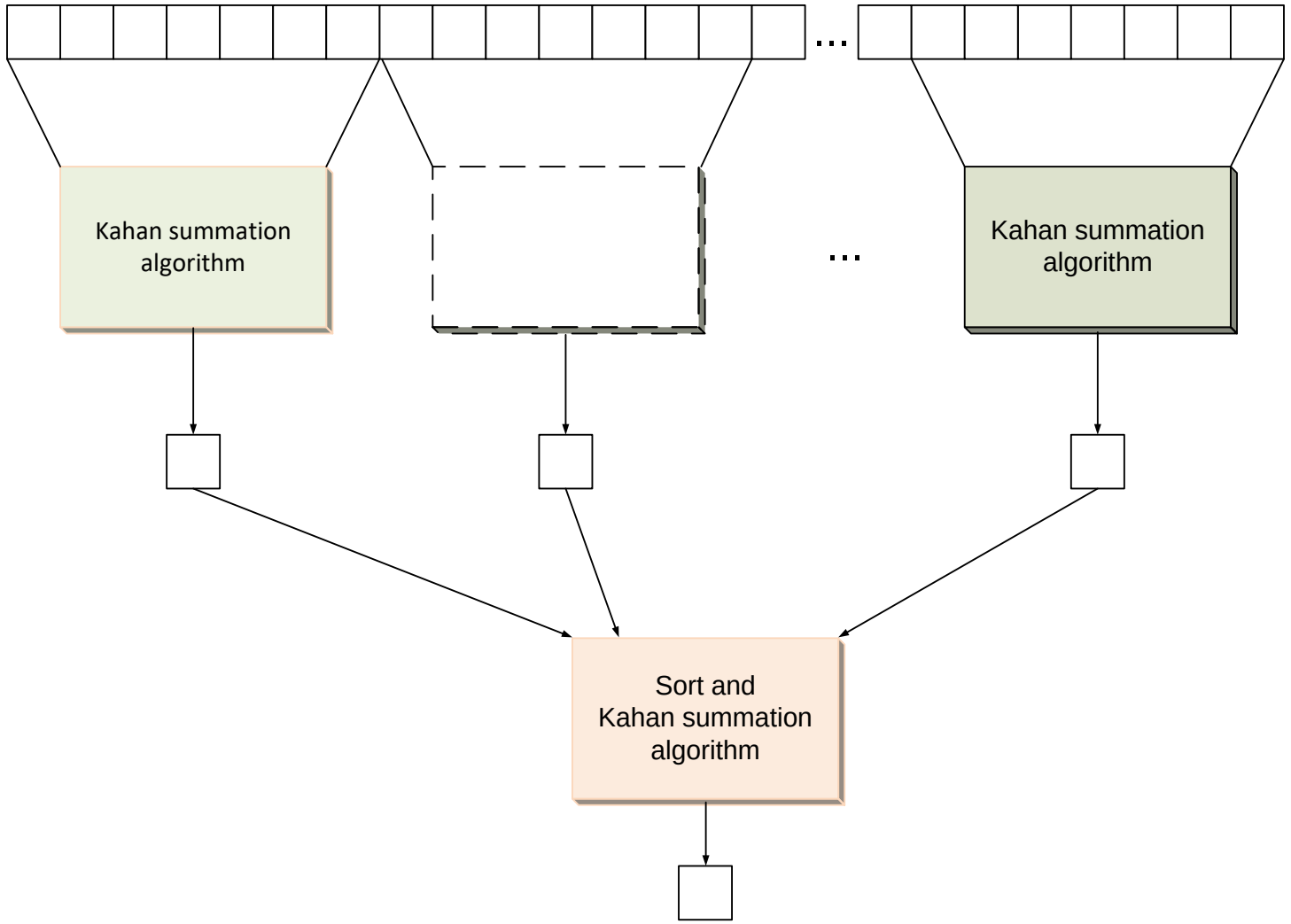
CASE STUDY – summation, multiplication, inner product with the FP

A hybrid algorithm is one of the best in terms of performance vs. accuracy.



CASE STUDY – summation, multiplication, inner product with the FP

A hybrid algorithm is one of the best in terms of performance vs. accuracy.



CASE STUDY – summation, multiplication, inner product with the FP

Some results:

Our custom dataset composed of vectors [**v**,–**v**] and [**w**,**w**], where components of **v** and **w** are random values of the Mersenne twister (*kMersenneRand_InnerZero*). Theoretically, their inner product should be 0, as explained in formula

$$P = \left[\mathbf{v}, -\mathbf{v}\right] \cdot \left[\mathbf{w}, \mathbf{w}\right] = \mathbf{v} \cdot \mathbf{w} - \mathbf{v} \cdot \mathbf{w} = 0$$

```
void Fill_Numerical_Data_MersenneUniform( DVec & inVec, ST num_of_data, DT kDataMag )
{
        inVec.resize( num_of_data );
        mt19937  rand_gen{ random_device{}() }; // Random Mersenne twister
        uniform_real_distribution< double > dist( - kDataMag, + kDataMag );
        std::generate( inVec.begin(), inVec.end(), [&](){ return dist( rand_gen ); } );
}
```

CASE STUDY – summation, multiplication, inner product with the FP

Some results:

Our custom dataset composed of vectors [**v**,–**v**] and [**w**,**w**], where components of **v** and **w** are random values of the Mersenne twister (*kMersenneRand_InnerZero*). Theoretically, their inner product should be 0, as explained in formula

$$P = \left[\mathbf{v}, -\mathbf{v}\right] \cdot \left[\mathbf{w}, \mathbf{w}\right] = \mathbf{v} \cdot \mathbf{w} - \mathbf{v} \cdot \mathbf{w} = 0$$

```
void Duplicate( DVec & inVec, DT multFactor = 1.0 )
{
        ST kElems { inVec.size() };
        inVec.resize( 2 * kElems );
        std::generate( inVec.begin() + kElems, inVec.end(),
        [ n = 0, & inVec, multFactor ] () mutable { return multFactor * inVec[ n++ ]; } );
}
```

CASE STUDY – summation, multiplication, inner product with the FP

Some results:

Our custom dataset composed of vectors [**v**,–**v**] and [**w**,**w**], where components of **v** and **w** are random values of the Mersenne twister (*kMersenneRand_InnerZero*). Theoretically, their inner product should be 0, as explained in formula

$$P = \left[\mathbf{v}, -\mathbf{v}\right] \cdot \left[\mathbf{w}, \mathbf{w}\right] = \mathbf{v} \cdot \mathbf{w} - \mathbf{v} \cdot \mathbf{w} = 0$$

```
data_generator.Fill_Numerical_Data_MersenneUniform( v, kElems / 2, pow( 2.0, dExp ) );
data_generator.Duplicate( v, + 1.0 );

data_generator.Fill_Numerical_Data_MersenneUniform( w, kElems / 2, pow( 2.0, dExp ) );
data_generator.Duplicate( w, - 1.0 );
```

CASE STUDY – summation, multiplication, inner product with the FP

Some results:

Our custom dataset composed of vectors [**v**,–**v**] and [**w**,**w**], where components of **v** and **w** are random values of the Mersenne twister (*kMersenneRand_InnerZero*). Theoretically, their inner product should be 0, as explained in formula

$$P = [\mathbf{v}, -\mathbf{v}] \cdot [\mathbf{w}, \mathbf{w}] = \mathbf{v} \cdot \mathbf{w} - \mathbf{v} \cdot \mathbf{w} = 0$$

N=20000000

δ	SimpAccu	TrRedPar	Sort	Kahan	SortKahan	KahanPar	SortKahanPar	908Par	908	ttmath
10	7.58568e-05	9.62019e-05	0	2.86673e-09	0	0	0	0	0	0
	55	8	793	217	942	9	140	11	128	6905
30	1.1727e+08	1.34939e+08	0	3766	0	0	0	0	0	0
	55	8	781	217	945	10	140	10	127	6893
50	1.04494e+20	4.55674e+19	0	1.46895e+15	0	0	0	0	0	0
	55	8	808	217	956	10	141	10	127	6883
100	3.05903e+50	1.09567e+50	0	6.77943e+45	0	0	0	0	0	0
	55	8	808	218	1000	10	139	10	126	6872
300	3.28593e+170	3.75038e+170	0	3.45517e+165	0	0	0	0	0	0
	55	8	828	218	981	10	151	11	126	7055
500	1.73055e+291	9.44077e+289	0	4.40157e+286	0	**0**	0	**0**	0	0
	55	8	754	217	937	**9**	143	**10**	128	7070

Cyganek B., Wiatr K.: How orthogonal are we? A note on fast and accurate inner product computation in the floating-point arithmetic. Int. Conference Societal Automation Technological & Architectural Frameworks, 2019.

CASE STUDY – summation, multiplication, inner product with the FP

Some results:

Our custom dataset composed of vectors [**v**,–**v**] and [**w**,**w**], where components of **v** and **w** are random values of the Mersenne twister (*kMersenneRand_InnerZero*). Theoretically, their inner product should be 0, as explained in formula

$$P = [\mathbf{v}, -\mathbf{v}] \cdot [\mathbf{w}, \mathbf{w}] = \mathbf{v} \cdot \mathbf{w} - \mathbf{v} \cdot \mathbf{w} = 0$$

N=20000000

δ	SimpAccu	TrRedPar	Sort	Kahan	SortKahan	KahanPar	SortKahanPar	908Par	908	ttmath
10	7.58568e-05	9.62019e-05	0	2.86673e-09	0	0	0	0	0	0
	55	8	793	217	942	9	140	11	128	6905
30	1.1727e+08	1.34939e+08	0	3766	0	0	0	0	0	0
	55	8	781	217	945	10	140	10	127	6893
50	1.04494e+20	4.55674e+19	0	1.46895e+15	0	0	0	0	0	0
	55	8	808	217	956	10	141	10	127	6883
100	3.05903e+50	1.09567e+50	0	6.77943e+45	0	0	0	0	0	0
	55	8	808	218	1000	10	139	10	126	6872
300	3.28593e+170	3.75038e+170	0	3.45517e+165	0	0	0	0	0	0
	55	8	828	218	981	10	151	11	126	7055
500	1.73055e+291	9.44077e+289	0	4.40157e+286	0	**0**	0	**0**	0	0
	55	8	754	217	937	**9**	143	**10**	128	7070

Cyganek B., Wiatr K.: How orthogonal are we? A note on fast and accurate inner product computation in the floating-point arithmetic. Int. Conference Societal Automation Technological & Architectural Frameworks, 2019.

CASE STUDY – summation, multiplication, inner product with the FP

Some results:

Our custom dataset composed of vectors [**v**,–**v**] and [**w**,**w**], where components of **v** and **w** are random values of the Mersenne twister (*kMersenneRand_InnerZero*). Theoretically, their inner product should be 0, as explained in formula

$$P = [\mathbf{v}, -\mathbf{v}] \cdot [\mathbf{w}, \mathbf{w}] = \mathbf{v} \cdot \mathbf{w} - \mathbf{v} \cdot \mathbf{w} = 0$$

N=20000000

δ	SimpAccu	TrRedPar	Sort	Kahan	SortKahan	KahanPar	SortKahanPar	908Par	908	ttmath
10	7.58568e-05	9.62019e-05	0	2.86673e-09	0	0	0	0	0	0
	55	8	793	217	942	9	140	11	128	6905
30	1.1727e+08	1.34939e+08	0	3766	0	0	0	0	0	0
	55	8	781	217	945	10	140	10	127	6893
50	1.04494e+20	4.55674e+19	0	1.46895e+15	0	0	0	0	0	0
	55	8	808	217	956	10	141	10	127	6883
100	3.05903e+50	1.09567e+50	0	6.77943e+45	0	0	0	0	0	0
	55	8	808	218	1000	10	139	10	126	6872
300	3.28593e+170	3.75038e+170	0	3.45517e+165	0	0	0	0	0	0
	55	8	828	218	981	10	151	11	126	7055
500	1.73055e+291	9.44077e+289	0	4.40157e+286	0	**0**	0	**0**	0	0
	55	8	754	217	937	**9**	143	**10**	128	7070

Cyganek B., Wiatr K.: How orthogonal are we? A note on fast and accurate inner product computation in the floating-point arithmetic. Int. Conference Societal Automation Technological & Architectural Frameworks, 2019.

CASE STUDY – summation, multiplication, inner product with the FP

Some results:

Our custom dataset composed of vectors [**v**,–**v**] and [**w**,**w**], where components of **v** and **w** are random values of the Mersenne twister (*kMersenneRand_InnerZero*). Theoretically, their inner product should be 0, as explained in formula

$$P = [\mathbf{v}, -\mathbf{v}] \cdot [\mathbf{w}, \mathbf{w}] = \mathbf{v} \cdot \mathbf{w} - \mathbf{v} \cdot \mathbf{w} = 0$$

N=20000000

δ	SimpAccu	TrRedPar	Sort	Kahan	SortKahan	KahanPar	SortKahanPar	908Par	908	ttmath
10	7.58568e-05	9.62019e-05	0	2.86673e-09	0	0	0	0	0	0
	55	8	793	217	942	9	140	11	128	6905
30	1.1727e+08	1.34939e+08	0	3766	0	0	0	0	0	0
	55	8	781	217	945	10	140	10	127	6893
50	1.04494e+20	4.55674e+19	0	1.46895e+15	0	0	0	0	0	0
	55	8	808	217	956	10	141	10	127	6883
100	3.05903e+50	1.09567e+50	0	6.77943e+45	0	0	0	0	0	0
	55	8	808	218	1000	10	139	10	126	6872
300	3.28593e+170	3.75038e+170	0	3.45517e+165	0	0	0	0	0	0
	55	8	828	218	981	10	151	11	126	7055
500	1.73055e+291	9.44077e+289	0	4.40157e+286	0	**0**	0	**0**	0	0
	55	8	754	217	937	**9**	143	**10**	128	7070

Cyganek B., Wiatr K.: How orthogonal are we? A note on fast and accurate inner product computation in the floating-point arithmetic. Int. Conference Societal Automation Technological & Architectural Frameworks, 2019.

Floating-point presentation - Conclusions

- **In the FP domain we always work with approximations of real values**.
- Computations with the FP numbers can result in overflow, underflow, or are burdened by **roundoff errors**.
- **In FP the commutative low does not hold:**

$$(a + b) + c \neq a + (b + c) \quad \text{for} \quad a, b, c \in \text{FP}.$$

- Machine epsilon conveys a value represented by the lowest bit of the significand. This is a difference between 1.0 and the next closest higher value representable in the FP format. A product with D provides a spacing assessment ➔ **thresholds** in iterations.
- **Adding values with different exponents leads to large errors.**
- **Subtracting close values can lead to severe cancellation errors**.

Floating-point presentation - Conclusions

- IEEE 754 defines standards of FP representation. The most common are the single precision (`float`) and double precision (`double`) formats.
- In C++ FP values are rounded to the nearest, whereas integer values are rounded toward 0 (cutting off the fractional).
- The `std::numeric_limits< T >` class conveys information on numerical properties of a type T, such as minimal, maximal values, etc.

Floating-point presentation - Conclusions

- Special care must be taken when summing up long series of FP numbers (roundoff, overflow).
- SL provides the **std::accumulate**, **std::inner_product**, **std::transform_reduce** – however, they do a ”simple” additions (no compensation).
- A simple value sorting with **std::sort** greatly improves additions, at an additional cost.
- SL provides the **std::execution::par** execution policy (paralel for free!)
- A simple compensated summation (Kahan) method greatly improves additions.
- A hybrid compensated algorithm performs accurately and fast.

Floating-point presentation - Literature

- D. E. Knuth, *The Art of Computer Programming*, Volume 2, Seminumerical Algorithms. Addison-Wesley, Boston, MA, USA, 1997.
- W. Kahan, *Further Remarks on Reducing Truncation Errors*. Communications of the ACM, Vol. 8, No. 1, 1965, pp. 40.
- N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd Ed. Society for Industrial and Applied Mathematics, USA, 2002.
- D. Goldberg, *What every computer scientist should know about floating-point arithmetic*. CSUR, 1991, pp. 5–48.
- Y.-K. Zhu and W. B. Hayes, *Algorithm 908: Online Exact Summation of Floating-Point Streams*. ACM Transactions on Mathematical Software, Vol. 37, No. 3, 2010, pp. 1–13.

Floating-point presentation - Literature

- 754-2008 - *IEEE Standard for Floating-Point Arithmetic*, DOI: 10.1109/IEEESTD.2008.5976968, ISBN: 978-0-7381-6981-1, 2008.
- GMP a library for arbitrary precision arithmetic (https://gmplib.org/).
- ttmath – A library for math operations on big integer/floating point numbers (www.ttmath.org).
- Cyganek B., Wiatr K.: *How orthogonal are we? A note on fast and accurate inner product computation in the floating-point arithmetic*. Int. Conference Societal Automation Technological & Architectural Frameworks, 2019.
- **Cyganek B.: *Introduction to Programming with C++ for Engineers*, Wiley, 2020**

Thank you!

The new book for beginners and advanced programmers:

Cyganek B.: ***Introduction to Programming with C++ for Engineers*****, Wiley, 2020**

Software available:
InnerProductMeasurement, GitHub repository, 2019.
https://github.com/BogCyg/InnerProductMeasurement

home.agh.edu.pl/~cyganek/BCProjects.zip

Thank you!

The new book for beginners and advanced programmers:

Cyganek B.: ***Introduction to Programming with C++ for Engineers*****, Wiley, 2020**

Software available:
InnerProductMeasurement, GitHub repository, 2019.
https://github.com/BogCyg/InnerProductMeasurement

home.agh.edu.pl/~cyganek/BCProjects.zip

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

1

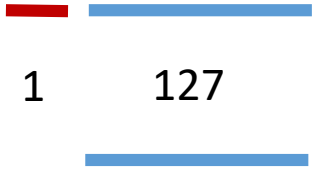
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

1 127

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

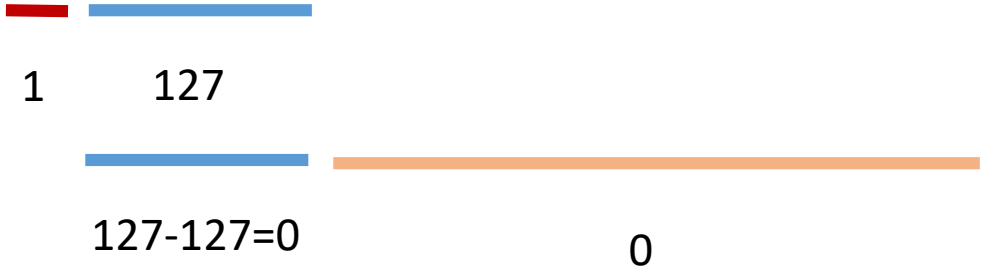


1 127

127-127=0

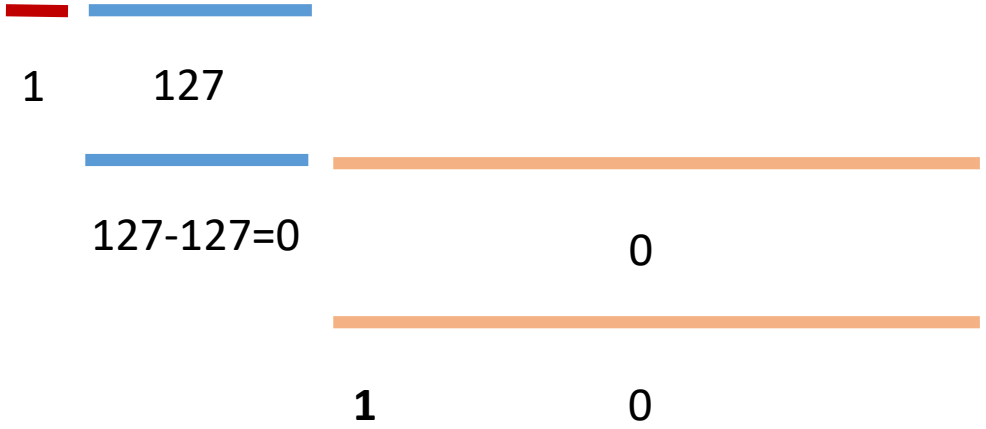
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000



The IEEE 754 Standard for Floating-Point Arithmetic

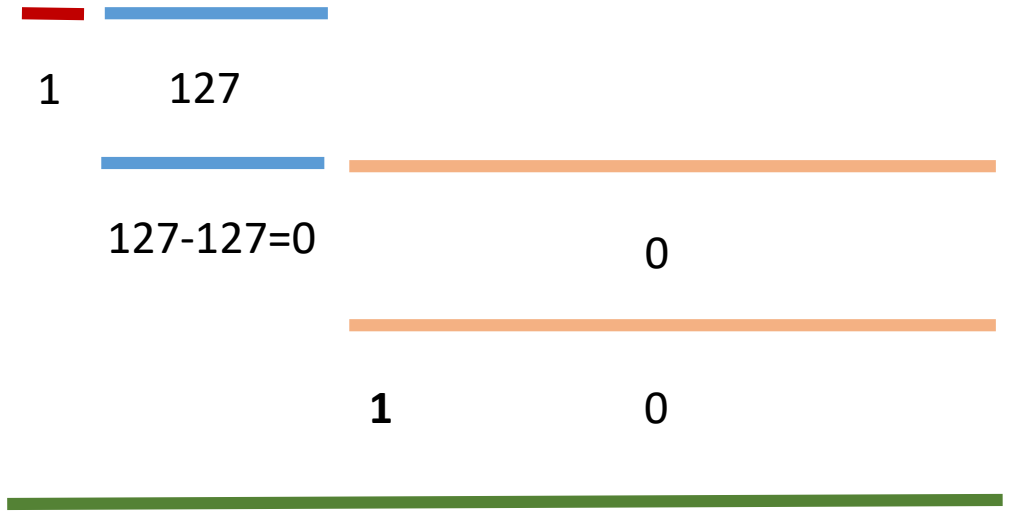
IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000



Hidden bit

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000



$$D = (-1)^S \cdot M \cdot B^E = -1 \cdot (1.0\ldots0) \cdot 2^0 = -1$$

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

0

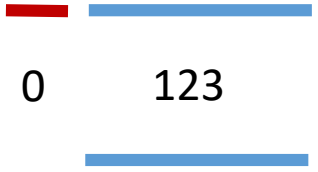
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

0 123

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

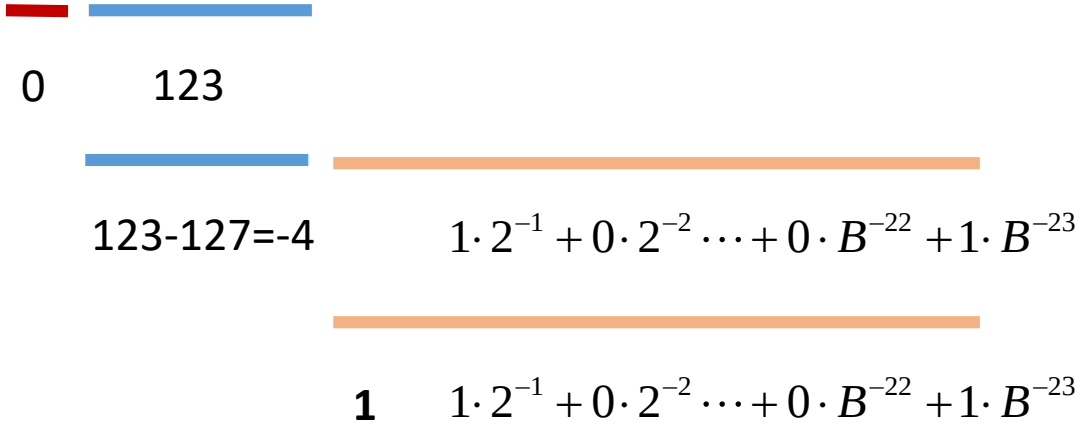


The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

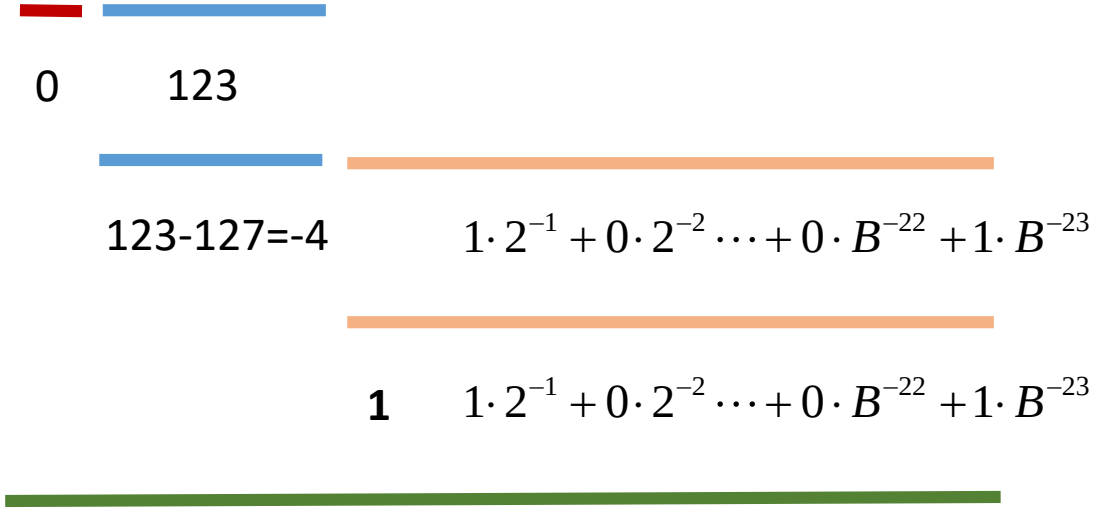
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101



The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101



$$D=(-1)^{S}\cdot M\cdot B^{E}=1\cdot\left(1+1\cdot 2^{-1}+0\cdot 2^{-2}\cdots+0\cdot B^{-22}+1\cdot B^{-23}\right)\cdot 2^{-4}\approx 0.1$$