# Efficient floating point arithmetic

Efficient School of Computing, Bertinoro, 2022
Wahid Redjeb
wahid.redjeb@cern.ch

# Why Floating-Point Arithmetic?

- Floating point instead of fixed point
  - They give you a wider range
  - You can represent very big numbers as well as very small numbers
- All of you are using floating point numbers
  - They need some attention, to avoid big failure in your code
- It is important to understand how they work
  - Finding the best way to solve a problem
    - How do you find the“best” approach to solve your problem?
    - Sometime you want something precise, some other time you want something fast
-

# Thinking of floating point numbers

- Sometime they are considered:
  - Not well defined
  - Full of mysteries and undefined behaviour
- But actually you can:
  - Write proof, like in standard math!
  - You can determine if your algorithm is going to fail
  - Or if it is going to work

- New hardware new challenges!
  - Parallelism make floating point calculation less deterministic!

# Defining the floating-point system.

Some desirable properties for floating-points

- Speed
- Accuracy
  - We want to be fast, but we don’t want a wrong answer
- Range
  - We want represent very small and very big numbers
- Portability
  - The code we write have to run on all the machine
- Something easy
  - We don’t want an extremely complicated set of rules

# J-M Muller sequence

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}$$

$$u_n = \frac{\alpha \cdot 100^{n+1} + \beta \cdot 6^{n+1} + \gamma \cdot 5^{n+1}}{\alpha \cdot 100^n + \beta \cdot 6^n + \gamma \cdot 5^n}$$

- if α ≠0 —> converges to 100
- if α = 0, □ ≠ 0 —> converges to 6

- u0 = 2 , u1 = -4 → α = 0, □ = -3, ɣ = 4
    - should converge to 6!

- … but sometimes, things don't work as expected
    - Try it youself! There is a sequence, that should converge to 6.

```
cd hands-on/floatingpoints/
g++ muller.cc -o muller
./muller
32
2
-4
```

| $n$ | Exact value |
|---|---|
| 3 | 18.5 |
| 4 | 9.378378378378378378384 |
| 5 | 7.801152737752161383 3 |
| 6 | 7.154414480975249353 5 |
| 11 | 6.274438598216327913 8 |
| 12 | 6.218695739802397788 3 |
| 16 | 6.094739439333681128 3 |
| 17 | 6.077722304847242736 3 |
| 18 | 6.063940322499808755 3 |
| 19 | 6.052721761016152193 4 |
| 20 | 6.043552110189268867 8 |
| 21 | 6.036031881081856780 0 |
| 22 | 6.029847325023901856 7 |
| 23 | 6.024749652366847898 7 |
| 30 | 6.006786093031205758 5 |
| 31 | 6.005648688771420267 9 |

# J-M Muller sequence

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}$$

$$u_n = \frac{\alpha \cdot 100^{n+1} + \beta \cdot 6^{n+1} + \gamma \cdot 5^{n+1}}{\alpha \cdot 100^{n} + \beta \cdot 6^{n} + \gamma \cdot 5^{n}}$$

- if **α** ≠0 —> converges to 100
- if **α** = 0, □ ≠ 0 —> converges to 6

- u0 = 2 , u1 = -4 → **α** = 0, □ = -3, ɣ = 4
  - should converge to 6!

- … but sometimes, things don't work as expected
  - Try it youself! There is a sequence, that should converge to 6.

```
cd hands-on/floatingpoints/
g++ muller_quad.cc -o muller_quad -lquad
./muller_quad
32
2
-4
```

| $n$ | Exact value |
|---|---|
| 3 | 18.5 |
| 4 | 9.378378378378378378384 |
| 5 | 7.801152737752161383 3 |
| 6 | 7.154414480975249353 5 |
| 11 | 6.274438598216327913 8 |
| 12 | 6.218695739802397788 3 |
| 16 | 6.094739439333681128 3 |
| 17 | 6.077722304847242736 3 |
| 18 | 6.063940322499808755 3 |
| 19 | 6.052721761016152193 4 |
| 20 | 6.043552110189268867 8 |
| 21 | 6.036031881081856780 0 |
| 22 | 6.029847325023901856 7 |
| 23 | 6.024749652366847898 7 |
| 30 | 6.006786093031205758 5 |
| 31 | 6.005648688771420267 9 |

# Building a floating-point system - Some properties

- Floating-points don’t behave as real numbers!
- All floating-points are rational numbers
- Common rules of arithmetic, in general, are not valid for floating-points
  - Distributivity, associativity
- There are a finite number of floating points!
  - Doesn’t matter how many bits you can store, you’ll always have a finite number of floating points
  - There are rational numbers that are not floating-points!
  - And irrational numbers can not be represented by floating-points

# IEEE 754-2008 standard

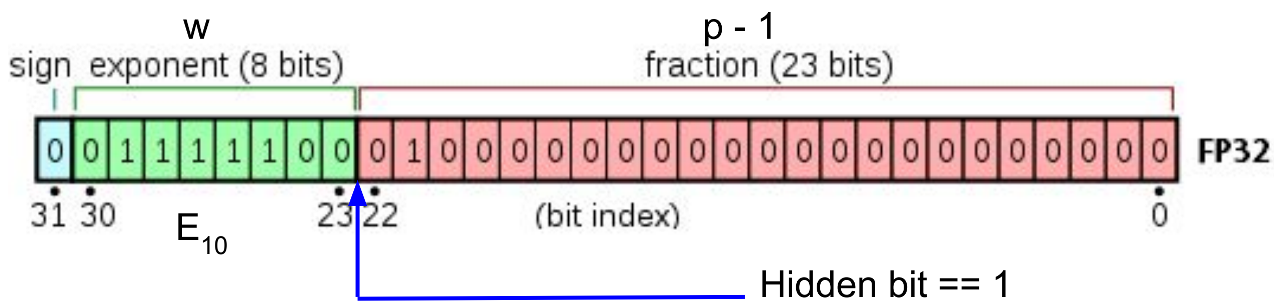
- We need something that tells us some rules for building a floating-point systems
  - Standardize formats
  - Conversions between different formats
  - Conversion between floating-points and integers
  - Standardize some operations
  - Rounding modes
  - Special values
    - Zero, Infinity, subnormals, NaN (not a Number)
      - NaN → 0/0, infinity / 0, infinity + infinity
  - Exceptions
    - Underflow
      - Value is less than the smallest non-zero floating-point number
      - Return 0
    - Overflow
      - Value is greater than the largest floating-point number
      - Return infinity
    - Division by zero

# Building a floating-point system - Value of a floating-point number

A floating point number is characterized by the following numbers

- A radix (base) **β**
- A sign bit, $s \in \{0,1\}$
- Exponent $e$
  - Integer such that $e_{min} \leq e \leq e_{max}$
- A precision $p$

# Building a floating-point system - Storage Format




| IEEE Name | Precision | N bits | Exponent w | Fraction p | $e_{min}$ | $e_{max}$ |
|---|---|---|---|---|---|---|
| Binary32 | Single | 32 | 8 | 24 | -126 | +127 |
| Binary64 | Double | 64 | 11 | 53 | -1022 | +1023 |
| Binary128 | Quad | 128 | 15 | 113 | -16382 | +16383 |

- Exponent: $E = e - e_{min} + 1$, *w* bits
- $e_{max} = -e_{min} + 1$

# Building a floating-point system - Value of a floating-point number

A floating point number is characterized by the following numbers

- A radix (base) **β**
- A sign, ***s*** $\in \{0,1\}$
- Exponent ***e***
  - Integer such that $e_{min} \leq e \leq e_{max}$
- A precision ***p***

The **value** of a floating-point number is given by

- Its format
- The digits in the number
  - $x_i$ such that $0 \leq i \leq p$ and $0 \leq x_i < \beta$

We can express its value:

$$\boxed{x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}}$$

# Building a floating-point system - Value of a floating-point number

$$x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}$$

Let's try to represent the number 0.5 (in binary radix $\beta$ = 2 )

# Building a floating-point system - Value of a floating-point number

$$\boxed{x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}}$$

Let’s try to represent the number 0.5 (in binary radix $\beta = 2$ )

- e = -1 → $2^{-1}$ x 1 · $2^{0}$ → ($x_0$ = 1 )

# Building a floating-point system - Value of a floating-point number

$$x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}$$

Let's try to represent the number 0.5 (in binary radix $\beta$ = 2 )

- e = -1 → $2^{-1}$ x 1 · $2^{0}$ → ($x_0$ = 1 )
- e = 0 → $2^{0}$ x 1 · $2^{-1}$ → ($x_0$ = 1, $x_1$ = 1)

# Building a floating-point system - Value of a floating-point number

$$\boxed{x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}}$$

$$m = \sum_{i=0}^{p-1} x_i \beta^{-i} \quad 0 \leq m < \beta$$

Let's try to represent the number 0.5 (in binary radix $\beta = 2$ )

- e = -1 → $2^{-1}$ x 1 · $2^{0}$ → ($x_0$ = 1 )
- e = 0 → $2^{0}$ x 1 · $2^{-1}$ → ($x_0$ = 0, $x_1$ = 1)
- e = 1 → $2^{1}$ x 1 · $2^{-2}$ → ($x_0$ = 0, $x_1$ = 0, $x_2$ = 1)
- …

**Multiple (m, e) representation**

# Value of a floating-point number - Uniqueness

$$\boxed{x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}} \qquad m = \sum_{i=0}^{p-1} x_i \beta^{-i} \quad 0 \leq m < \beta$$

- In order to have a unique representation we want to **normalize** the floating-point number
  - We can choose to represent the floating-point number with a (m,e) representation, such that *e* is minimum ($e_{min} \leq e \leq e_{max}$)
    - $1 \leq |m| < \boldsymbol{\beta}$
    - That also means to require $x_0 \neq 0$, first bit = 1

- If minimizing the exponent results in $e < e_{min}$
  - then $x_0$ must be 0, $e = e_{min}$ , first bit = 0
- These numbers are called **subnormal** numbers

# Value of a floating-point number - Normals and Subnormals

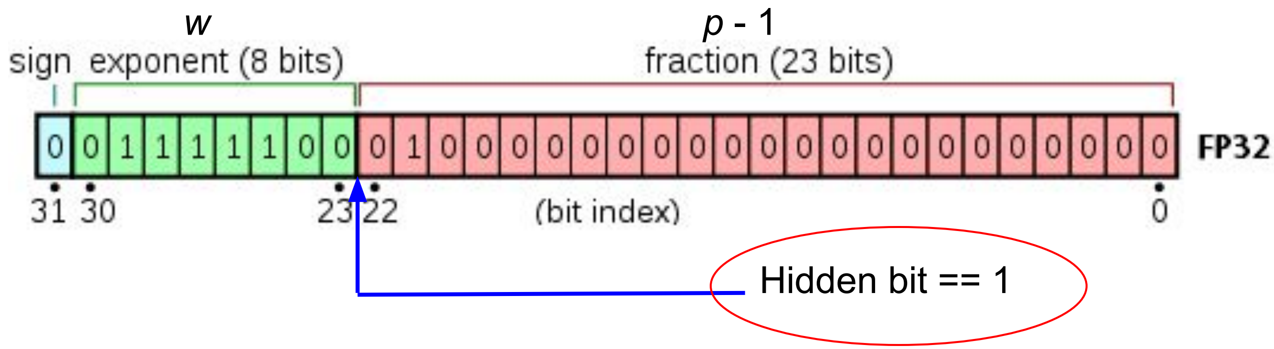
$$\boxed{x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}}$$

$$m = \sum_{i=0}^{p-1} x_i \beta^{-i} \qquad 0 \leq m < \beta$$

To summarize, we can have three different cases:

- m = 0 —-> $x_0 = x_1 = \ldots = x_{p-1} = 0$ —> Value: ±0
- m ≠ 0 and $x_0 \neq 0$ —> Normal number
  - $1 \leq m < \beta$
- m ≠ 0 **but** $x_0 = 0$ —-> Subnormal number
  - $0 < m < 1$ and $e = e_{min}$

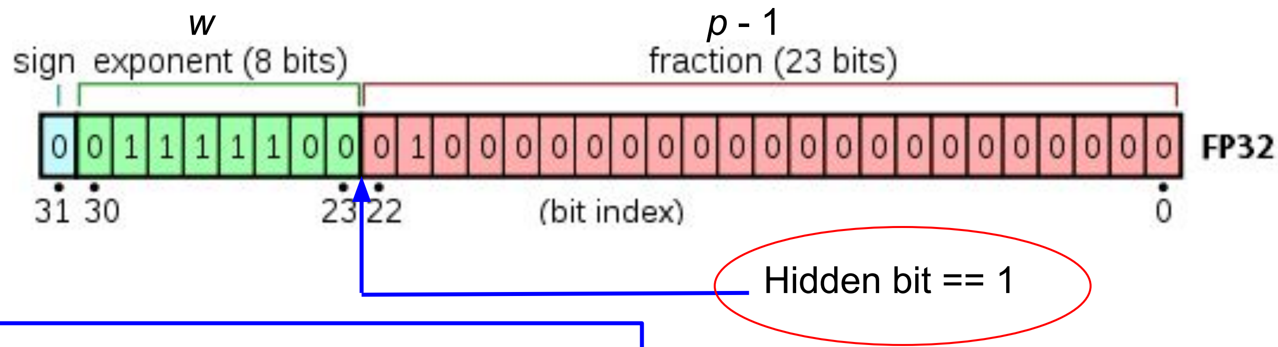
# Value of a floating-point number - Some additional points




- Exponent: $E = e - e_{min} + 1$, $w$ bits
- $e_{max} = -e_{min} + 1$

- Many more normal numbers than subnormal numbers
    - Let's just assume $x_0 = 1$
- Use special exponent to trigger special treatment for subnormals
    - $e = e_{min} - 1$

# Value of a floating-point number - Some additional points




- Exponent: $E = e - e_{min} + 1$, *w* bits
- $e_{max} = -e_{min} + 1$

- In principle the exponent is a signed integer -126 < e < +127
  - But, it is easier to compare floating point numbers with an unsigned exponent
  - We don't store the exponent, but something else

- $e = (E_{10} + 127_{10})_2$
- $E = (e_2)_{10} - 127_{10}$

- Many more normal numbers than subnormal numbers
  - Let's just assume $x_0 = 1$
- Use special exponent to trigger special treatment for subnormals
  - $e = e_{min} - 1$

# Exercise!

```
> cd esc22/hands-on/floatingpoints/
> g++ float-rep.c -o float-rep
> ./float-rep
```

- Play a bit with the program!
- Try to extract the floating point representation for the number 17.625
- Try to extract the base 10 value of the value
  - 0 10000010 00011100000000000000000

# IEEE 754-2008 - Rounding Modes

- The standard defines 5 rounding modes:
    - **Round to nearest - Ties to even**
        - Round to nearest, in case of a tie, the breaking rule is to select the result with an even significand
        - **It is the default rounding mode!**
    - Round to nearest - Ties away from zero
        - Round to nearest, in case of a tie, round to the nearest value above (if positive) or below (if negative)
    - Round towards 0 - Direct rounding
    - Round towards $+\infty$ - Direct rounding
    - Round towards $-\infty$ - Direct rounding

# IEEE 754-2008 - Rounding Modes

- IEEE 754-2008 **requires** these operations to be correctly rounded
    - Addition
    - Subtraction
    - Multiplication
    - Division
    - Fused multiply add (FMA)
    - Square root
    - Comparison

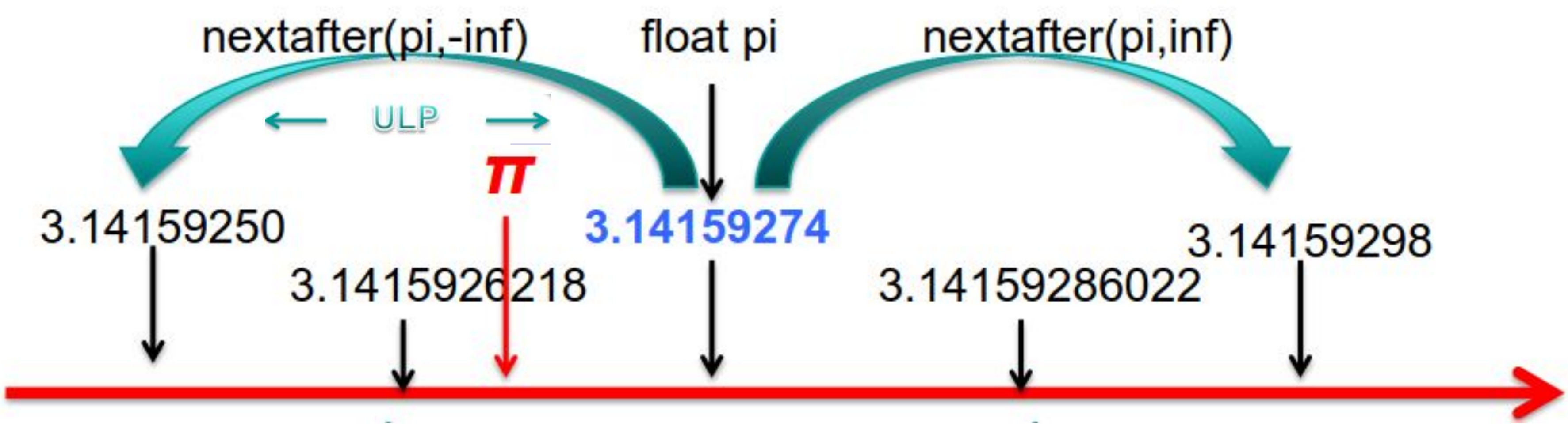
# Notation

- For floating points operation we use:
  - $\oplus$ for addition
  - $\ominus$ for subtraction
  - $\otimes$ for multiplication
  - $\oslash$ for division
- `fl(x)` is the result of an operation using the the current rounding mode

# Rounding - Error measures - ULP

$$m = \sum_{i=0}^{p-1} x_i \beta^{-i} \quad 0 \leq m < \beta$$

- Unit in the last place (ULP)
- Place value of the least bit of the significand of x
- Represent the distance between two floating points
- If `x` is the infinite-precise result and `y` is the rounded results
    - `|x - y| ≤ 0.5·ulp(y)`




**Tools**

```
#include <limits> //std::numeric_limits
float x = //value;
float ulp = std::nextafter(x,std::numeric_limits<float>::max())-x;
```

# Math with floating-point numbers

- Addition and multiplication are guaranteed to be commutative
- **BUT** associativity and distributivity are in general lost
  - (a ⊕ b) ⊕ c may not be equal to a ⊕ (b ⊕ c)
    - Similar for ⊖,⊗,⊘
  - a ⊗ (b ⊕ c) may not be equal to (a ⊗ b) ⊕ (a ⊕ c)
  - (1⊘a)⊗a may not be equal to a

- Write a program to sum all the numbers from 1 to N in single precision
- Write a program to sum all the numbers from N to 1 in single precision
- You see any difference?
- What happens if you use double precision?

Note that: $\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$

# Fused Multiply-add instruction (FMA)

- Standardized by IEEE-754-2008
- Allows to compute `(a x b ) + c` in a single instruction
- Only one rounding instead of two
- May produce faster and accurate calculation
  - Matrix multiplication
  - Polynomial evaluation

**BUT**

- FMA may change floating point results
  - `FMA(a*b+c)` might be different from $(a \otimes b) \oplus c$
- The compiler is allowed to rearrange the terms of an expression to generate a single instruction: **Contractions**

```
double a, b, c, d;
a = b*c + d
```

```
double a,b,c,d;
a = b;
b *= c;
c += d;
```

**IMPORTANT!**
Different compilers might have contractions enabled or disabled!

- If you want to be fully reproducible
  - Disable contractions
  - use std::fma explicitly
    - `-O2 -mfma`

**Tools**: There are compiler switches and `#pragmas`

- `-fpp-contract=off|fast`
- `#pragma STDC FP_CONTRACT ON|OFF`

# Rounding - Approximation error

```
#include <cmath>
int main () {
printf("%1.17g\n", std::sin(M_PI));
}
```

```
[wa@T470]$ ./a.out
1.2246467991473532e-16
```

- The value of `M_PI` is less than π by `~1.2x10`$^{-16}$
- `sin(M_PI) ≠ 0`

- Try yourself!
  - Edit `approx_err.cc`
- Is 0.1 a floating point?
- Is 0.01 a floating point?
- is 0.01 = 0.1*0.1 ?
- You can use ulps to understand the differences!

https://www.iro.umontreal.ca/~mignotte/IFT2425/Disasters.html

# Rounding - Associativity and Catastrophic Cancellation

```cpp
#include <cstdio>

int main () {
   const double a = +1.0E+300;
   const double b = -1.0E+300;
   const double c = 1.0;
   double x = ( a + b ) + c ;
   double y = a + ( b + c );

   printf("x = %1.10g\n", x);
   printf("y = %1.10g\n", y);

return 0;
}
```

```
[wa@T470 ~]$ ./a.out
x = 1
y = 0
```

- **Catastrophic cancellation** occurs when two nearly equal floating-point numbers are subtracted.
  - If $x \approx y$, their significands are nearly identical.
  - When they are subtracted, only a few low-order digits remain. I.e., the result has very few significant digits left.

# Rounding - Catastrophic Cancellation - Quadratic Equation

$$ax^2 + bx + c = 0 \longrightarrow x_\pm = \left(-b \pm \sqrt{b^2 - 4ac}\right)/(2a)$$

- Write a program that gives you back the roots of the quadratic equation with single precision
  - a = $5*10^{-4}$
  - b = 100
  - c = $5*10^{-3}$
- What happens?
- How can we treat the problem?

# Rounding - Catastrophic Cancellation - Quadratic Equation

$$ax^2 + bx + c = 0$$

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
$$= -\frac{b}{2a}(1 \mp \sqrt{1 - \frac{4ac}{b^2}})$$

- Let's rewrite the solutions
- Let's define □ = 4ac / $b^2$

$$x_+ = -\frac{b}{2a}(1 - \sqrt{1-\delta})$$

- When $b^2$ >> 4ac → □ << 1
  - $(1 - \sqrt{1-\delta})$ contains a possible cancellation!
- We can remove one cancellation rationalizing the expression
  - Multiply by $1 + \sqrt{1-\delta}$ numerator and denominator
- Now no catastrophic cancellation can occur!

$$x_+ = -\frac{b}{2a}\left(\frac{1-(1-\delta)}{1+\sqrt{1-\delta}}\right)$$
$$= -\frac{2c}{b}\left(\frac{1}{1+\sqrt{1-\delta}}\right)$$

# Rounding - Sterbenz's Lemma

Provided a floating-point system and subnormal numbers, if $a$ and $b$ are floating points numbers such that:

$$b/2 \leq a \leq 2b$$

Then:

$$a \ominus b = a - b$$

Thus:

There is no rounding error associated with $a \ominus b$

- The Sterbenz lemma asserts that if x and y are sufficiently close floating-point numbers then their difference x − y is computed exactly by floating-point arithmetic

$x \ominus y = fl(x - y)$ with no rounding needed.

# Error Free Transformation - EFT

Error-free transformation is a concept that makes it possible to compute accurate results within a floating point arithmetic. (*Error-free transformations in real and complex floating point arithmetic - Stef Graillat and Valérie Ménissier-Morain* )

- Algorithms that transforms a set of floating-point numbers in a new set of floating-point numbers without any loss of information
  - Useful for computing round-off errors
  - Obtain accurate operations
- There EFT for
  - Addition
  - Multiplication
  - Splitting
  - Derived: Combination of the previous ones

```
f(x,y) -> (s,t)
```

# Addition EFT - Fast2Sum

```
Fast2Sum algorithm

a,b floating-points numbers such that

Requires |a|≥|b|
1.  s ←a⊕b
2.  t ←b⊖(s⊖a)
3.  return (s,t)
```

$a+b = s+t$, where $s = a \oplus b$ and $t$ are floating points

Intuitive explanation → *Rigorous proof on Handbook of Floating Point arithmetic*

- a > b
  - $s = a +$ (part of $b$ that contributed to the sum)
  - $s \ominus a$ = part of $b$ that contributed to the sum
  - $b \ominus (s \ominus a) =$ Rounding error

**NOTE: This algorithm requires a branching**

# Addition EFT - TwoSum

```
TwoSum algorithm

a,b floating-points numbers
1.  s ←a⊕b
2.  z ←(s⊖a)
3.  t ← (a⊖(s⊖z)⊕(b⊖z)
4.  return (s,t)
```

$a+b = s+t$, where $s=a\oplus b$ and $t$ are floating points

- No branching
  - But 6 floating-points instead of 3
- TwoSum usually faster

# Addition EFT - Precise Multiplication

```
TwoProduct algorithm

a , b floating-points numbers
1.   s ←a⊗b
2.   t ←FMA(a,b,-s)
3.   return (s,t)
```

$a \times b = s + t$, where $s = a \otimes b$ and $t$ are floating points

# Condition numbers

- Given a $x$ number you want to compute $f(x) = y$
  - But there is the rounding in place
    - Most of the time you have $\ddot{x} = x + \Delta x$
    - $f(x + \Delta x) = y + \Delta y$
- We can compute the following number

$$C = \frac{|\frac{\Delta y}{y}|}{|\frac{\Delta x}{x}|} = \frac{|x \cdot f'(x)|}{|f(x)|}$$

Example: $ln(x) \quad \text{with} \quad x = 1$

$$C = \frac{1}{lnx} \longrightarrow \infty$$

- Small condition number means:
  - Small $\Delta x$ produces small $\Delta y$
  - **Well Conditioned**
- Big condition number means:
  - Small $\Delta x$ produces big $\Delta y$
  - **Ill conditioned**

# Summation Techniques - Condition Number

Condition number for addition $$C_{sum} = \frac{\sum\limits_{i=0} |x_i|}{|\sum\limits_{i} x_i|}$$

- Numerator without cancellations
- Denominator contains cancellations
- If C is big → you want to tackle the problem carefully
  - Using higher precision
  - We need to apply some techniques to obtain a results as if we were in higher precision but without actually using higher precision

# Summation Techniques - Compensated Sum

- Developed by William Kahan
- Exploits `TwoSum/Fast2Sum` algorithms
    - Use the knowledge on the exact rounding error to recover the summation!

```
Kahan sum
s ←x_0
t ←0
for i = i to n -1 :
1.  y ←x_i - t
2.  z ←s + y
3.  t ←(z - s) + y
4.  s ← z
end
return s
```

- This is the simplest version of the Kahan summation
- There are a lot of different variations

- Apply correction
- Calculate new sum
- Update correction (contribution of y)
- Update sum

# Dot product techniques - Condition Number and Traditional algorithm

Condition number for dot product

$$C_{\text{dot product}} = \frac{\sum_{i=0} |x_i \cdot y_i|}{|\sum_i x_i \cdot y_i|}$$

- If C is not too large —> Traditional algorithms can be used
- If C is large more accurate techniques are needed

Compute C while you are computing the dot product

```
Traditional Algorithm
s ←0
for i = i to n -1 :
1.  s ← s ⊕ ( xi⊗yi)
end
return s
```

# Dot product techniques - Condition Number and Traditional algorithm

Condition number for dot product

$$C_{\text{dot product}} = \frac{\sum_{i=0} |x_i \cdot y_i|}{|\sum_i x_i \cdot y_i|}$$

- If C is not too large —> Traditional algorithms can be used
- If C is large more accurate techniques are needed

Compute C while you are computing the dot product

```
Traditional Algorithm
s ←0
for i = i to n -1 :
1.  s ← s ⊕ ( x_i⊗y_i)
end
return s
```

Looks familiar?

# Dot product techniques - Condition Number and Traditional algorithm

Condition number for dot product

$$C_{\text{dot product}} = \frac{\sum_{i=0} |x_i \cdot y_i|}{|\sum_i x_i \cdot y_i|}$$

- If C is not too large —> Traditional algorithms can be used
- If C is large more accurate techniques are needed

Compute C while you are computing the dot product

```
Traditional Algorithm
s ←0
for i = i to n -1 :
1.   s ← s ⊕ ( x_i⊗y_i)
end
return s
```

```
Traditional Algorithm
s ←0
for i = i to n -1 :
1.   s ← FMA(x_i,y_i,s)
end
return s
```

- You can use FMA in traditional algorithms
  - Even with one less rounding
  - But it does not improve the accuracy

# Dot product techniques - Compensated Dot Product

- Dot product is just the combination of an addition and a multiplication
  - We already have some EFT for them!
  - Just apply them!

**Dot2Product Algorithm**

```
Dot2 (x, y, N)
  [p, s] = TwoProduct (x_0, y_0);
  for i = 1 to N
    [h, r] = TwoProduct (x_i, y_i);
    [p, q] = TwoSum (p, h);
    s = s ⊕ (q ⊕ r);
  end
  p = p ⊕ s;
end
```

# Compilers options!

- **There are many compiler options which affect floating point results!**
- Some of them can be enabled/disabled by other options!
- Different compilers might have different options!

- gcc default mode is “Strict IEEE 754 mode”
- `-O1, -O2, -O3, -Ofast , -ffast-math, -funsafe-math-optimizations`

https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Optimize-Options.html

Tool for inspecting assembly code
http://gcc.godbolt.org

# Take Away Message

- Floating points arithmetic needs some attention
- It depends on the problem you are trying to solve
    - The accuracy you want to achieve
    - Tradeoff between accuracy and speed
- There are some techniques to achieve accuracy without increasing the precision
    - Depends again on your problem
        - Condition number
- Compilers can help you speeding up your math
    - But again, be careful, sometime you can lose accuracy
    - In general, try to avoid square roots, division or trigonometric function
        - Try to use linear algebra when possible
- Reproducibility of the results
    - Keep in mind that floating point arithmetic is different from real number arithmetic
    - Associativity, Distributivity are in general loss in floating point arithmetics

# Reference

- https://link.springer.com/book/10.1007/978-0-8176-4705-6


- Floating point workshop @ CERN by Jeffrey Arnold
- D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surverys, 23(1):5-47, March 1991
- IEEE, IEEE Standard for Floating-Point Arithmetic, IEEE Computer Society, August 2008.