



Dealing with real numbers

Sistemi Digitali M

A.A. 2024/2025

Stefano Mattoccia

Università di Bologna

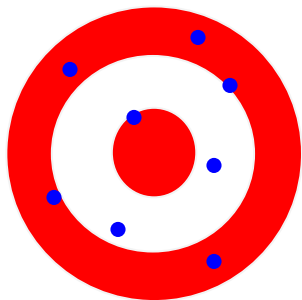
Introduction

- Before dealing with strategies to manage real numeric values, this section introduces/recalls some basic concepts about precision and accuracy
- Then, we'll introduce the two primary methodologies for encoding and processing data representing real numeric values: fixed-point and floating-point
- For the latter, we'll also introduce widely used yet not standardized formats and issues related to the deployment of floating-point data to increase efficiency and memory footprint
- Finally, we'll consider techniques to perform computations involving computations with real numbers using only basic arithmetic and logic operations with integer data

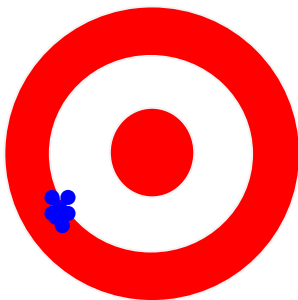
Precision and Accuracy

Before dealing with real numbers encoding, let's recall **precision** and **accuracy**

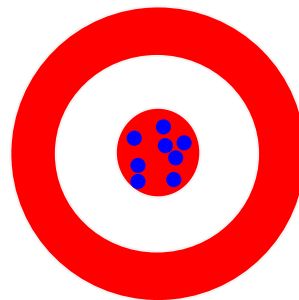
- Precision quantifies how well a measuring device can obtain consistent values of the same entity across different measurements (*repeatability*)
- Another relevant concept relevant for our purposes is accuracy; it quantifies how well we can approximate the actual value with a specific data encoding or the capability of a particular sensing device



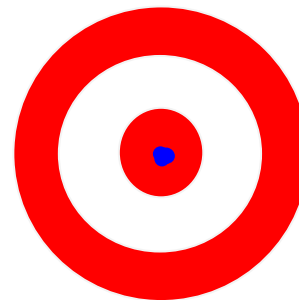
Low Precision
Low accuracy



High Precision
Low accuracy



Low Precision
High accuracy



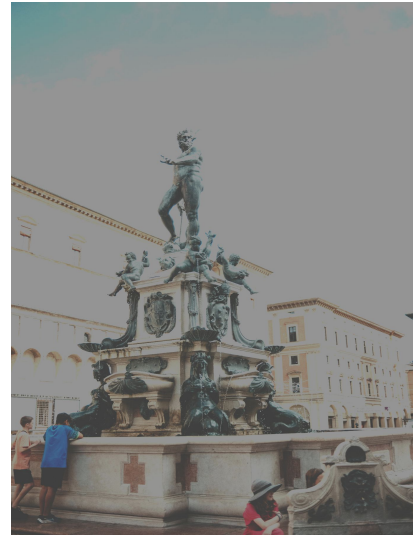
High Precision
High accuracy

Camera: Accuracy (Resolution)



Different imaging sensors can frame the same scene with various amounts of details (accuracy) according to their resolution (ie, the smaller the pixel, the higher the accuracy)

Camera: Precision (Repeatability)

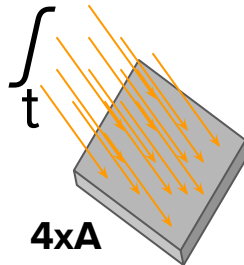
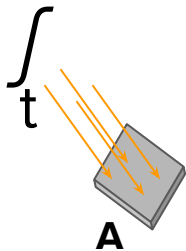


t

However, the same imaging sensor can frame a scene from the same position with the same settings, generating different (unprecise) outcomes for the same pixel over time

Camera: how pixel size can affect precision and accuracy

- The spatial resolution (ie, the number of photosensitive elements present in the imaging sensor), which sets the accuracy, is often considered the crucial parameter
- However, the camera acquisition process relies on the acquisition of light photons
- The larger is the pixel, the higher is the light gathered through the photosensitive device
- Thus, in low light conditions, smaller sensing devices would receive fewer photons than a device with larger ones, which may lead to a noisier image, reducing the *precision* (ie, increasing the noise) despite the potentially higher *accuracy* (ie, smaller pixels)



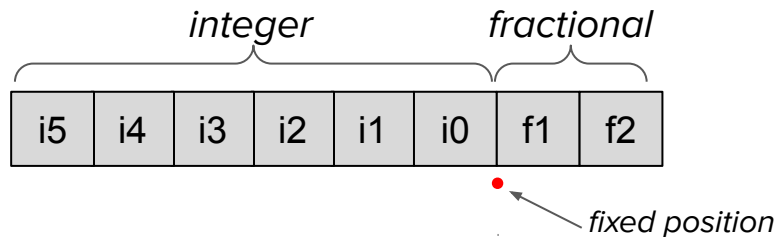
Dealing with real numbers

The need for real numbers

- Integer data types and computations are helpful, but managing real numbers is pivotal for many scientific purposes and other tasks, such as gaming
- Most applications, including AI, heavily rely on real data, encompassing integers, fractions, and irrationals. Hence, a representation of such data (even if in an approximate format) and the ability to perform mathematical computations is crucial
- Nonetheless, performing computation using real data type (eg, with floating-point encoding) is generally expensive and requires many bits to encode such data
- Therefore, we also introduce a less popular yet helpful data representation that can speed up computations in some relevant circumstances.

Fixed-point: introduction 1/2

Fixed-point representation assumes that we can encode numbers with a constant number of digits for the integer and fractional part with the **decimal point in a fixed position**



For instance, 0.75 and 1.5 can be encoded (unsigned, 6+2 digits) as:

$$000000.11_2 = 0.2^5 + 0.2^4 + 0.2^3 + 0.2^2 + 0.2^1 + 0.2^0 + 1.2^{-1} + 1.2^{-2} = 0 + \frac{1}{2} + \frac{1}{4} = 0.75_{10}$$

$$000001.10_2 = 0.2^5 + 0.2^4 + 0.2^3 + 0.2^2 + 0.2^1 + 1.2^0 + 1.2^{-1} + 0.2^{-2} = 1 + \frac{1}{2} = 1.5_{10}$$

For unsigned values, the data range with 6+2 digits is [0, +63.75]

For signed values, the data range with 6+2 digits is [-32.0, +31.75]

Fixed-point: introduction 2/2

- Not surprisingly, with this specific encoding (two fractional digits), 0.20_{10} would be approximated to 0.25_{10} (ie, 0.01_2), and 0.1_{10} would be approximated to 0.0_{10} , being zero the closest number
- The major limitation of fixed-point encoding is the limited data range that spans approximately (signed numbers) $[-2^K, +2^K]$ with k integer digits
- For these reasons, encoding real numbers with a *floating* decimal point position is the preferred solution, even if approximation issues persist
- Does it mean that fixed-point numbers are useless? In some circumstances, they can be valuable in replacing expensive computations with real numbers with more lightweight integer operations, as will be discussed later
- Moreover, compared to floating-point, it allows a constant accuracy within its range, although such range is much smaller using the same number of bits

Floating-point: introduction 1/2

- Despite the advantages previously outlined, fixed-point data is seldom deployed in practice, mainly due to its limited dynamic range
- A more effective and ubiquitous method to represent real values relies on the floating-point paradigm
- First of all, we recall that a number is said in *normalized scientific notation* if it contains a single integer digit, different from zero, and a limited (eg, $\frac{1}{2}$) or unlimited number (eg, $\frac{1}{3}$) of decimal digits
- These are some examples (using base 10):
 - -2.3443434 normalized scientific format
 - $7.26752 \cdot 10^{23}$ normalized scientific format
 - 0.3310 not in normalized scientific format
 - $16.4510 \cdot 10^{-3}$ not in normalized scientific format

Floating-point: introduction 2/2

In contrast to fixed-point, for which the decimal point is in a fixed position, it can move in floating-point according to an exponent term embedded into the same encoding format

Hence, with floating-point, a numeric value N in **scientific format*** is encoded as:

$$N = (-1)^S \cdot \text{VAL}_{\text{BASE}} \cdot \text{BASE}^{\text{EXP}}$$

with **S** the sign (a single digit set to 1 or 0), **VAL** the numeric value encoded according to parameters **BASE** and **EXP**. The two latter terms, **VAL** and **EXP**, are encoded with a certain number of digits called ***mantissa*** and ***exponent***.

For instance, the *float* format uses $\text{BASE}=2$, 23+1 bit for VAL and 8 bit for EXP

* not true for **subnormals** (or denormalized) values

Floating-point: encoding approximation issues 1/4

Let's consider the numerical value 10.75_{10}

We can represent it in the scientific format as: $1.075_{10} \cdot 10^1$

With a binary encoding, the same number is: $1010.11_2 \cdot 2^0$

Hence, its representation in scientific format with 6 bit is: $1.01011_2 \cdot 2^3 = (1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{32}) \cdot 2^3$

Similarly, 0.125_{10} in scientific format is $1.25_{10} \cdot 10^{-1}$

Its binary version is 0.001_2 in scientific format is $1.00000_2 \cdot 2^{-3}$

Regardless of the representation, the **number of available digits is limited**, and despite some *lucky* cases, like those reported before, many numerical values cannot be encoded without approximation/rounding. Indeed, increasing the number of binary digits can yield a better approximation but not solve the problem as discussed next.

Floating-point: encoding approximation issues 2/4

Let's consider the numerical value 1.21_{10} , already in scientific format using a decimal encoding and a single fractional decimal digit. Its (approximated) binary representation with 8 fractional binary digits is:

$$1.00110101_2 = 1 + 0.2^{-1} + 0.2^{-2} + 1.2^{-3} + 1.2^{-4} + 0.2^{-5} + 1.2^{-6} + 0.2^{-7} + 1.2^{-8} = 1.20703125_{10} \neq 1.21_{10}$$

or (with a better approximation/rounding):

$$1.00110110_2 = 1 + 0.2^{-1} + 0.2^{-2} + 1.2^{-3} + 1.2^{-4} + 0.2^{-5} + 1.2^{-6} + 1.2^{-7} + 0.2^{-8} = 1.21093750_{10} \neq 1.21_{10}$$

1.0011 1111 10

We can increase the number of digits to get better-approximated results; however, we can only represent it with errors even by increasing the number of fractional digits to 23 or more. Of course, the higher the number of digits, the better the approximation, but the error persists when we can't represent a number with a finite number of power of 2 terms.

Floating-point: encoding approximation issues 3/4

Extending the number of fractional binary digits from 8 to 23 bit, the numerical value 1.21_{10} is:

$$1.00110101110\ 000101001000_2 = 1.21000003814697265625_{10} \neq 1.21_{10}$$

With **8** fractional bits, the absolute approximation error is:

$$1.21_{10} - 1.21093750_{10} = -0.00093750_{10}$$

With **23** fractional bits, the absolute approximation error is:

$$1.21_{10} - 1.21000003814697265625_{10} = -0.00000003814697265625_{10}$$

We get a much better approximation with 23 fractional bits, but we cannot encode exactly a *simple* numerical value like 1.21_{10} . This evidence is the first critical aspect to consider when using floating-point: **most numbers are just a rounded approximation to the closest available encodable representation.**

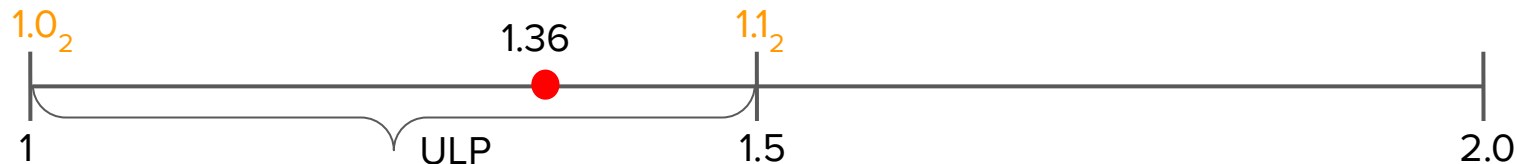
Floating-point: some encoding examples, positive exponent

Let's consider 2.00_{10} ; in binary format, it is 10.0_2 . In scientific format, with 3 fractional digits, 2.00_{10} is: $1.00\textcolor{red}{0}_2 \cdot 2^1$. The red digit is the *Unit in the Last Place (ULP)*, which affects accuracy.

- Hence, the largest number we can represent in scientific format with 3 fractional digits and **exponent 1** is: $1.11\textcolor{red}{1}_2 \cdot 2^1 = 1.875_{10} \cdot 2^1 = 3.75_{10}$ (equal to $2^2 - \textcolor{red}{ULP} \cdot 2^1$)
- With the next **exponent 2**, we can encode values from 4_{10} (100.0_2), in scientific format $1.00\textcolor{red}{0}_2 \cdot 2^2$, to $1.11\textcolor{red}{1}_2 \cdot 2^2 = 1.875_{10} \cdot 2^2 = 7.5_{10}$ (equal to $2^3 - \textcolor{red}{ULP} \cdot 2^2$)
- With the next **exponent 3**, we can encode values from 8_{10} (1000.0_2), in scientific format is $1.00\textcolor{red}{0}_2 \cdot 2^3$, to $1.11\textcolor{red}{1}_2 \cdot 2^3 = 1.875_{10} \cdot 2^3 = 15.0_{10}$ (equal to $2^4 - \textcolor{red}{ULP} \cdot 2^3$)
- Interestingly, with **exponent 4** (values ranging from $1.00\textcolor{red}{0}_2 \cdot 2^4$ (**16**) to $1.11\textcolor{red}{1}_2 \cdot 2^4 = 1.875_{10} \cdot 2^4 = 30$ (equal to $2^5 - \textcolor{red}{ULP} \cdot 2^4$)) even integer values (ie, 17, 19, 21, 23, 25, 27, 29, 31) cannot be encoded without rounding error ($|1/2 \cdot \textcolor{red}{ULP} \cdot 2^4| = 1$)
- Accuracy can be improved by increasing the mantissa length (number of fractional bits)

Floating-point vs Fixed-point and rounding error 1/10

Let's consider the numerical value 1.36_{10} and a single bit for the fractional part



Should we encode it as 1.0_2 or 1.1_2 ?

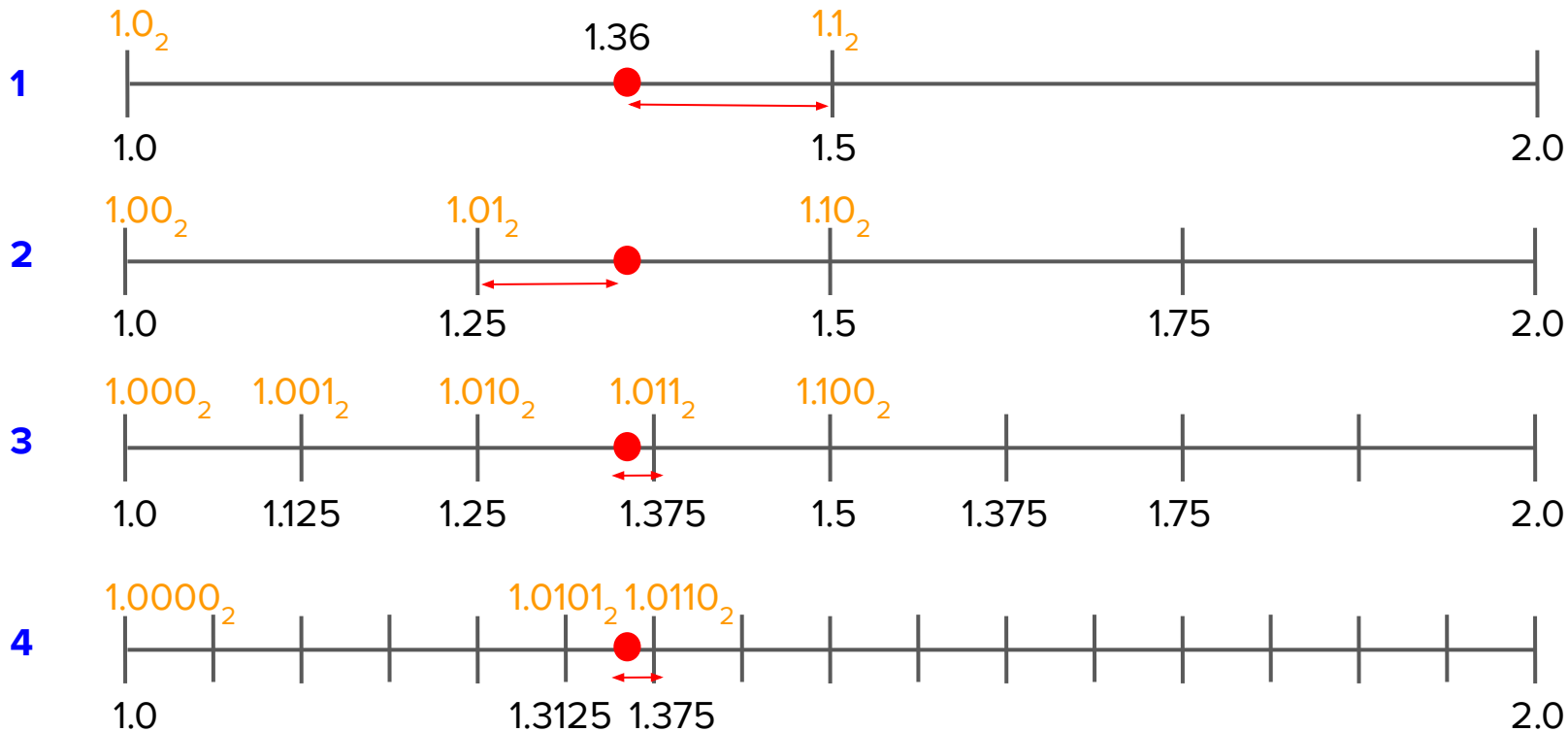
Of course, looking at the picture, we'd say 1.1, since 1.36 is closer to 1.5 than 1.0 and consequently the relative error is lower for 1.5 (ie, $|1.5-1.36|/1.36 \approx 0.10 < 0.26 \approx |1.0-1.36|/1.36$).

Given a specific binary encoding, the error should be as low as possible (eg, smaller than half of the weight given by the ULP (Unit in the Last Place) that for a single bit for the fractional part is $\frac{1}{2} \cdot 0.5$. In the example, similar considerations apply to fixed and floating-point representations.

But how can we handle this during the encoding phase?

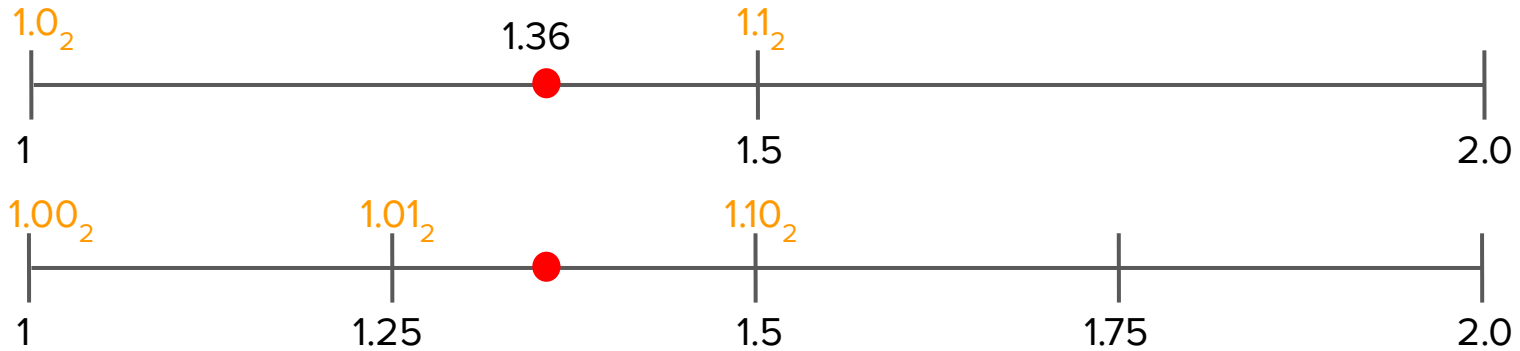
Floating-point vs Fixed-point and rounding error 2/10

Let's consider again the numerical value 1.36_{10} with different numbers of fractional digits:



Floating-point vs Fixed-point and rounding error 3/10

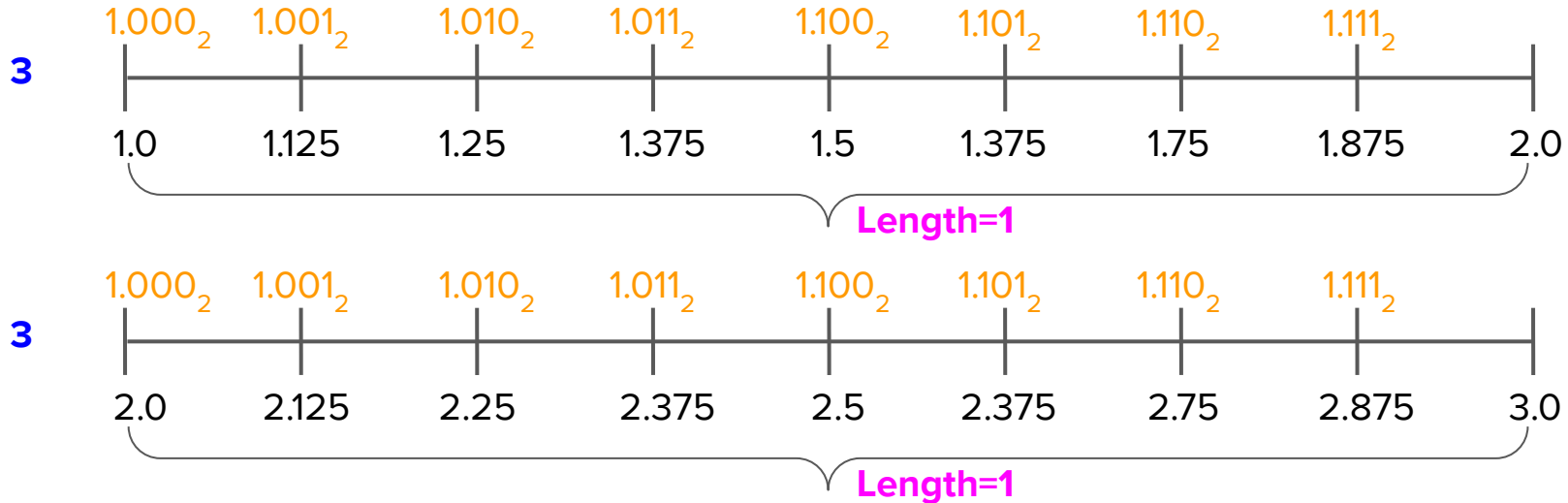
Hence, during the encoding phase, we should use at least one additional bit to better round the actual value to the numeric value encoded, discovering that 1.36 is closer to 1.5 than 1.0. The most significant error is half the weight assigned to the ULP (0.25 with a single using 1+1 digits and 0.125 using 1+2 digits). But what happens when the value is in between (eg, 1.375?)



Again, in this example, the same considerations apply to fixed-point and floating-point. However, there is a major difference in behaviour between the two, as discussed next. Increasing the number of additional bits vs those available enables more accurate rounding

Floating-point vs Fixed-point and rounding error 4/10

With **fixed-point**, intervals of the same length are always split into portions of the same size. For instance, with 3 fractional bits, intervals of the same length (eg, $[1,2]$ and $[2,3]$) are always split into 8 portions of the same size, as depicted below.

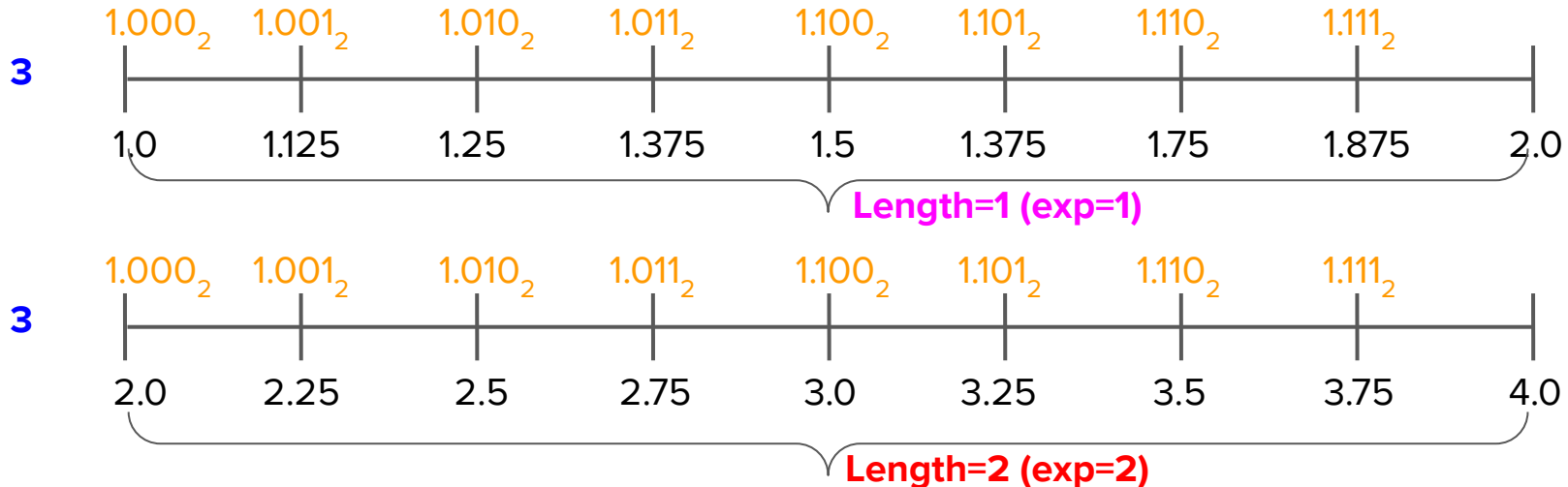


Consequently, the rounding error is constant regardless of the considered interval range.

Floating-point vs Fixed-point and rounding error 5/10

In contrast, with **floating-point**, an interval is always split into the same number of portions, but the length of the intervals changes according to the exponential value.

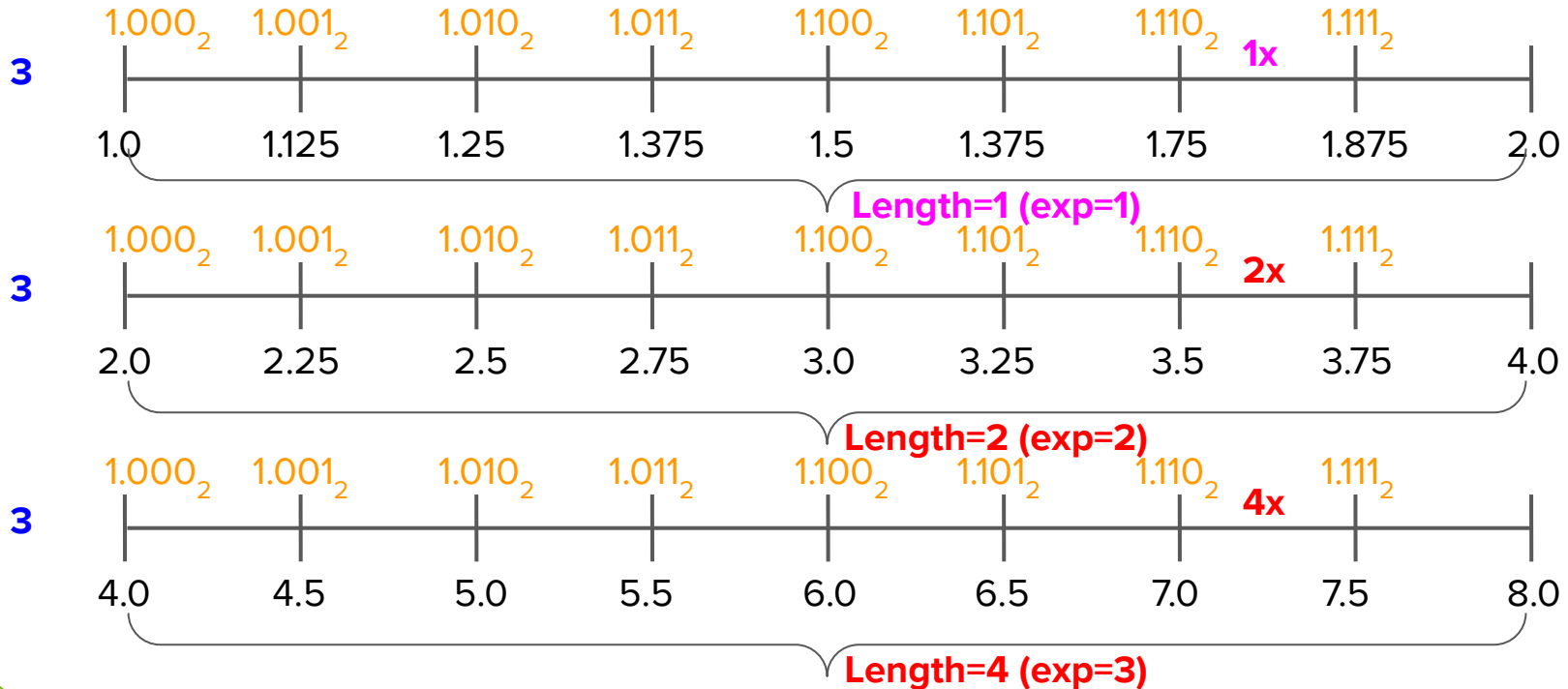
With binary encoding and 3 bits for fractionals, we can encode numeric values falling between* [1,2] and [2,4], respectively, with exponent 0 and 1.



The impact of ULP increases with the magnitude of the number according to the exponent

Floating-point vs Fixed-point and rounding error 6/10

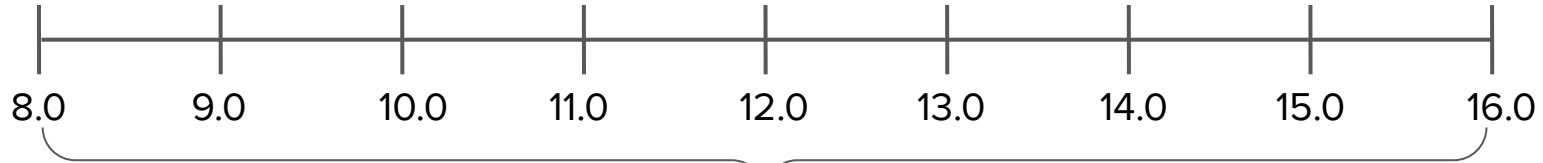
Increasing the magnitude of the numeric value further, the positive exponent gets higher (2x for each unit increase), and the issue outlined becomes even more evident.



Floating-point vs Fixed-point and rounding error 7/10

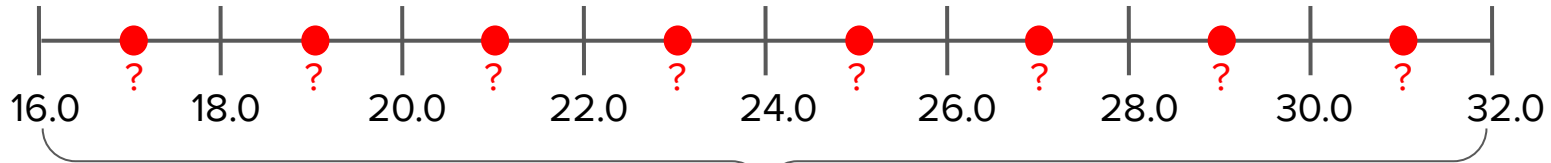
Increasing even further, with 3 fractional digits, we cannot encode odd integers (17, 19, 21, 23, 25, 27 and 31) without rounding when the exponent is 4 (see the graph at the bottom).

3



Length=8 (exp=4)

3



Length=16 (exp=4)

To overcome this issue, the solution consists of increasing the number of fractional bits (mantissa) at the cost of a larger data footprint and energy consumption for computations

Floating-point vs Fixed-point and rounding error 8/10

- Hence, the distance between two encodable values increases with the magnitude of the numeric value (or better, according to the increase of its positive exponent)
- Consequently, the rounding error in the worst cases is higher with larger positive exponents because the weight of ULP gets higher (2x when the exp. increases by 1)
- On the other hand, the issue just highlighted is the side effect that allows encoding much higher numeric values with floating-point vs fixed-point
- In contrast, with fixed-point, the largest rounding error is constant across all the representable ranges and equal to $\frac{1}{2}$ of the constant weight of the ULP
- Regardless of the encoding type (fixed or floating-point), both allow encoding only a subset of numeric values without approximation
- With floating-point: the larger the exponent, the higher the likelihood that this issue occurs (eg, odd integers (eg, 16777217) cannot be encoded with a *float* data type)

Floating-point vs Fixed-point and rounding error 9/10

- Of course, with negative exponents, the behaviour is in the opposite direction and the higher the magnitude of the negative exponent, the better the accuracy
- Intuitively, as outlined for positive exponents, by increasing the magnitude of a negative exponent (ef, from -1 to -2), a smaller portion of the range ($\frac{1}{2}$ in the example by increasing by 1) is divided into the same constant number of segments
- For instance, let's consider the range [0.5, 0.9375] and [0.25, 0.46875]:

$$[0.5_{10} = 0.1_2 = \mathbf{1.000} \cdot 2^{-1}, 0.9375_{10} = 0.1111_2 = \mathbf{1.111} \cdot 2^{-1}]$$

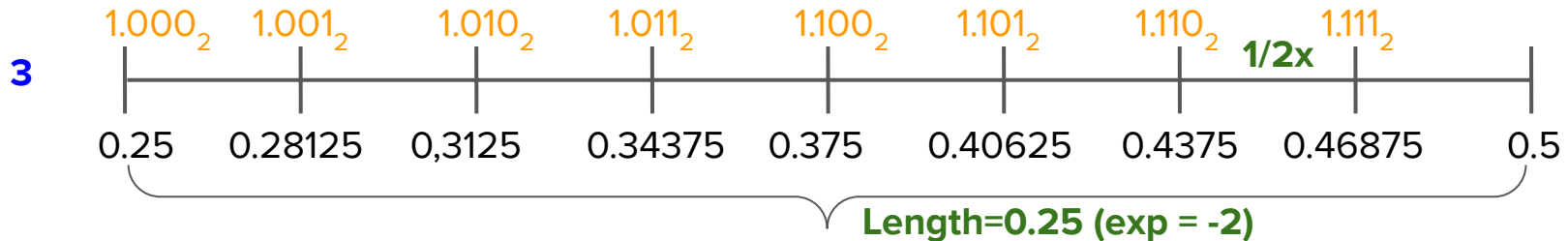
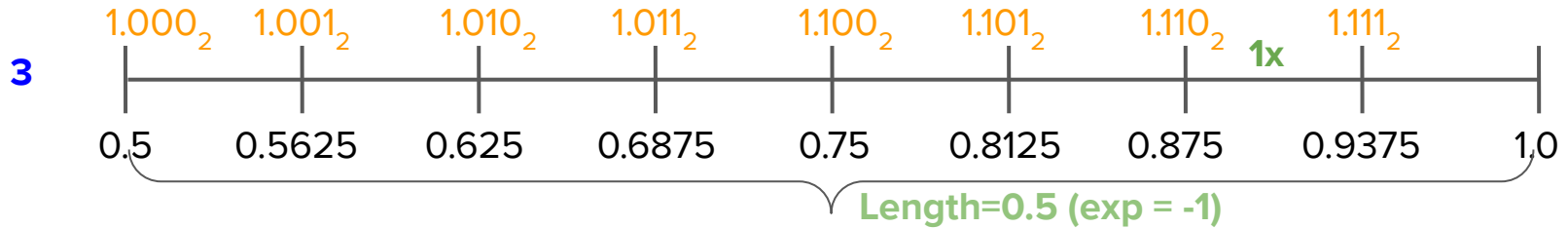
$$[0.25_{10} = 0.01_2 = \mathbf{1.000} \cdot 2^{-2}, 0.46875_{10} = 0.01111_2 = \mathbf{1.111} \cdot 2^{-2}]$$

As outlined in the next slide, decreasing the exponent from -1 to -2, the data interval decreases from 0.5 to 0.25, but both segments are divided into the same 8 portions.

Thus, the weight assigned to the ULP is, respectively, $\frac{1}{8} \cdot 2^{-1} = 0.0625$ and $\frac{1}{8} \cdot 2^{-2} = 0.03125$

Floating-point vs Fixed-point and rounding error 10/10

- Below, we show how increasing the magnitude of a negative exponent yields better accuracy

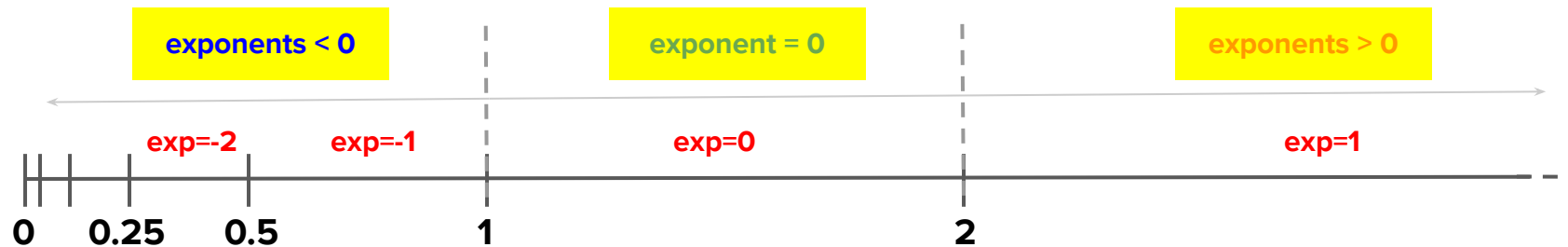


- Using the same scale, the behaviour described so far would appear like this



Floating-point in a nutshell 1/3

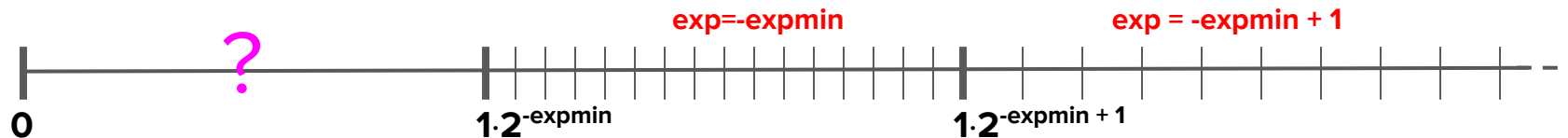
A summary of floating-point encoding



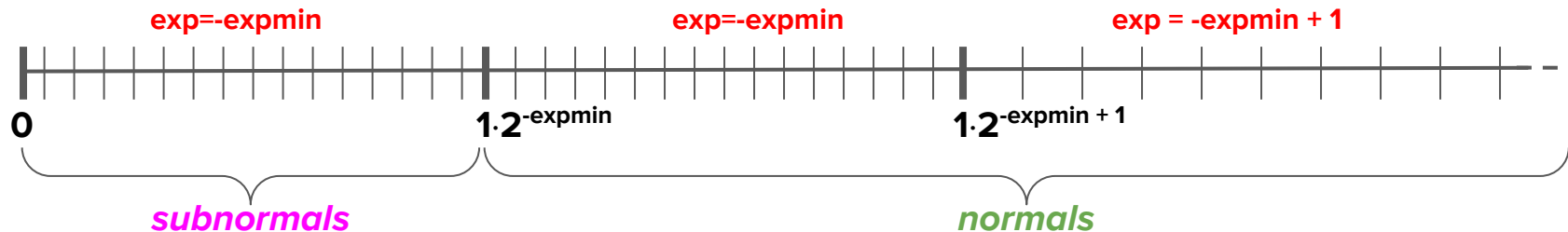
- Each segment has a different length according to the exponent and consists of a constant number of equidistant subsections according to the mantissa length
- There is a single exception to this scheme, the **subnormal numbers** discussed next, placed between 0 and the leftmost segment (ie, the one with the most negative exponents). They have the same most negative exponent, but they are not encoded in scientific notation, enabling increased accuracy around zero
- The graph is specular for negative numbers (Sign=-1)

Floating-point in a nutshell 2/3

When a negative exponent is equal to the minimum value **-expmin**, using the scientific notation would yield a gap between] **0** and $1 \cdot 2^{-\text{expmin}}$ [, as depicted below:



To avoid this weird behaviour (*flush to zero*) and gain accuracy near zero, numeric values between] **0** and $1 \cdot 2^{-\text{expmin}}$ [are not represented in scientific notation (ie, they are encoded as $0.x \cdot 2^{-\text{expmin}}$). For this reason, they are called **denormal** (or **subnormal** numbers), leading to:



Floating-point in a nutshell 3/3

- With a single notable exceptions (ie, the E4M3 minifloat introduced later), floating point reserves codes for **NaN** (Not a Number), **Inf** (Infinite) and **0**
- NaN examples: $\text{Inf} - \text{Inf}$, Inf/Inf , $\text{Inf} \times 0$, $0/0$
- Inf examples: $1/0$, 2^{Inf}
- NaN, Inf and 0 can be positive or negative (ie, $\pm\text{NaN}$, $\pm\text{Inf}$ and ± 0)
- The range for positive and negative number is the same, the only difference is the sign
- For rounding and during computations the IEEE standard defines three extra bits: Guard, Round and Sticky (referred to as GRS)
- Guard and Round are the two additional bits at the right of the least significant bit used with the chosen encoding representation
- The Sticky bit is a third bit (at the right of Round bit) which *remembers* (eg, during right shifts) if at least one bit among those at its right is 1

Floating-point: additions 1/2

- The approximation issues highlighted start at the beginning, during the encoding phase and propagate through arithmetic operations
- Let's consider the addition of two numerical values in scientific notation encoded with 8 fractional bits: **1.21 + 1.22 = 2.43**

$$\begin{array}{r} 1.00110110_2 + \quad 1.2109375 \\ 1.00111000_2 = \quad 1.2187500 \\ \hline 10.01101110_2 \end{array}$$

In scientific notation: **1.0011 0111₂ · 2¹ = 2.4296875₁₀**

<https://www.rapidtables.com/convert/number/binary-to-decimal.html?x=100110111>

Floating-point: additions 2/2

Let's consider again the addition of two numerical values in scientific notation encoded with 8 fractional bits: **1.21 + 0.02** = $1.21 \cdot 10^0 + 2 \cdot 10^{-2} = 1.23 \cdot 10^0$

0.02 in scientific format is: $1.0100\ 011_2 \cdot 2^{-6} = 0.01953125_{10}$

The first phase consists of aligning the two exponents to the largest one (0 in this case):

$$\begin{array}{rcl} 1.00110110\ 000000_2 \cdot 2^0 + & & 1.2109375 \\ 0.00000101\ 000111_2 \cdot 2^0 = & & 0.01953125\ (0.01995849609375\ \text{with extra bits}) \\ \hline 1.00111011\ 000111_2 \cdot 2^0 & & \end{array}$$

In scientific notation: $1.00111011_2 \cdot 2^0 = 1.23046875_{10}$

Adding large values with smaller ones can induce errors due to the exponents' alignment

Floating-point: multiplication

Let's consider the multiplication of two numerical values in scientific notation encoded with 8 fractional bits: **1.21 x 0.02** = $1.21 \cdot 10^0 \times 2 \cdot 10^{-2} = 0.0242 \cdot 10^0 = 2.42 \cdot 10^{-2}$

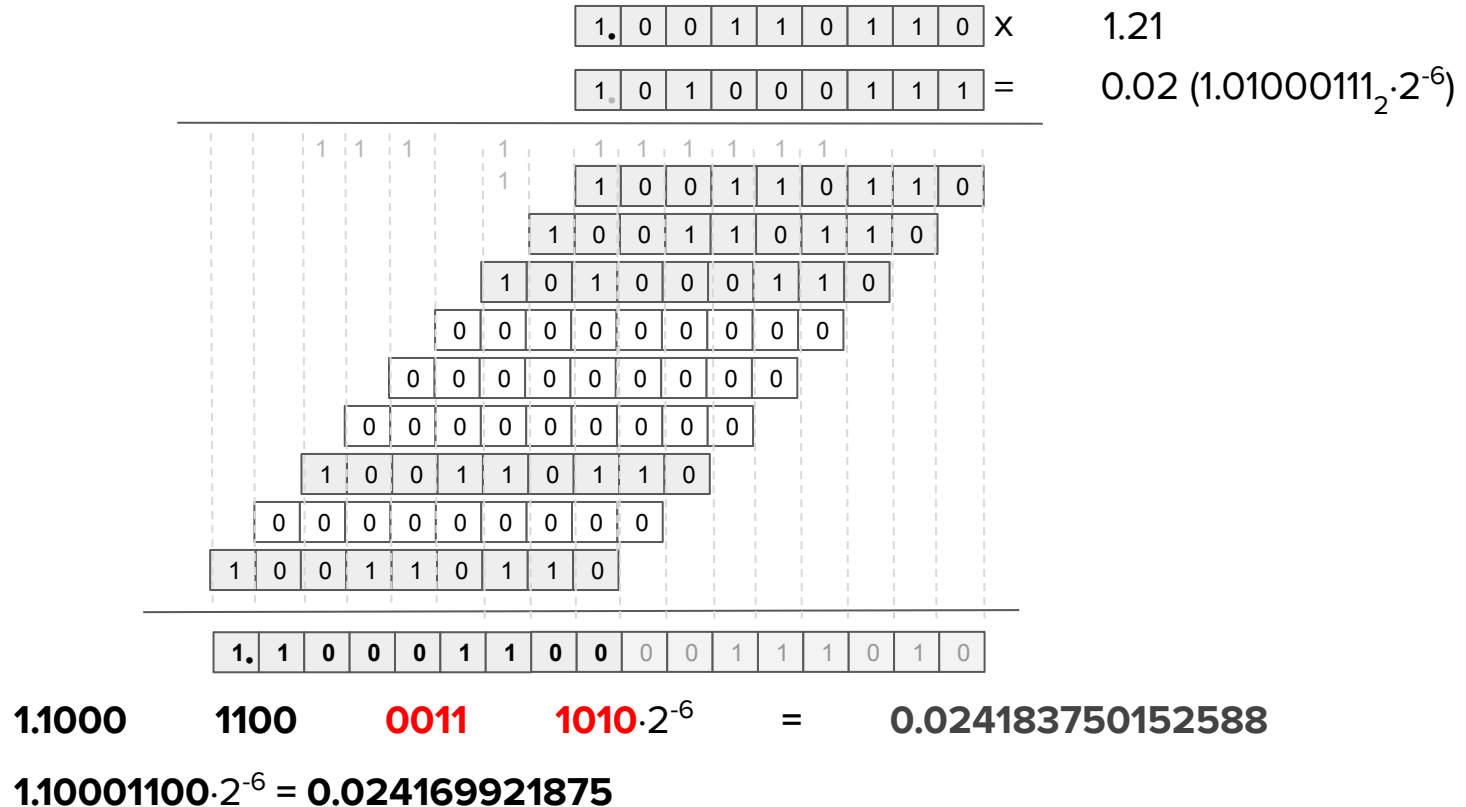
0.02 in scientific format is: $1.0100\ 0111_2 \cdot 2^{-6} = 0.01953125$

The exponent of the results is the sum of the two exponents (0-6=-6 in this case), and the magnitude is the product of the two *significands* (1 + fractional part):

$$\begin{array}{rcl} 1.00110110_2 & \times & 1.2109375 \\ 1.0100\ 0111_2 & = & 0.01953125 \text{ (0.01995849609375 with extra bits, previous slide)} \\ \hline 1.1000110000111010 & \cdot 2^{-6} & \text{(details in the next slide)} \end{array}$$

The result has 2x the number of binary digits than the operands

Floating-point: multiplication details



<https://www.rapidtables.com/convert/number/binary-to-decimal.html?x=100110111>

Multiply and Accumulate (MAC) and Fused MAC (FMA)

A multiplication followed by an addition, $\mathbf{Z} = \mathbf{AxB} + \mathbf{C}$, is a typical sequence of operations performed in many application fields (eg, AI, computer graphics, image/signal processing).

The conventional approach, referred to as **MAC**, relies on two distinct phases/operations:

- i) **Multiply** ; $\text{TEMP} = \mathbf{AxB}$ (1st rounding)
- ii) **Accumulate** ; $\mathbf{Z} = \text{TEMP} + \mathbf{C}$ (2nd rounding)

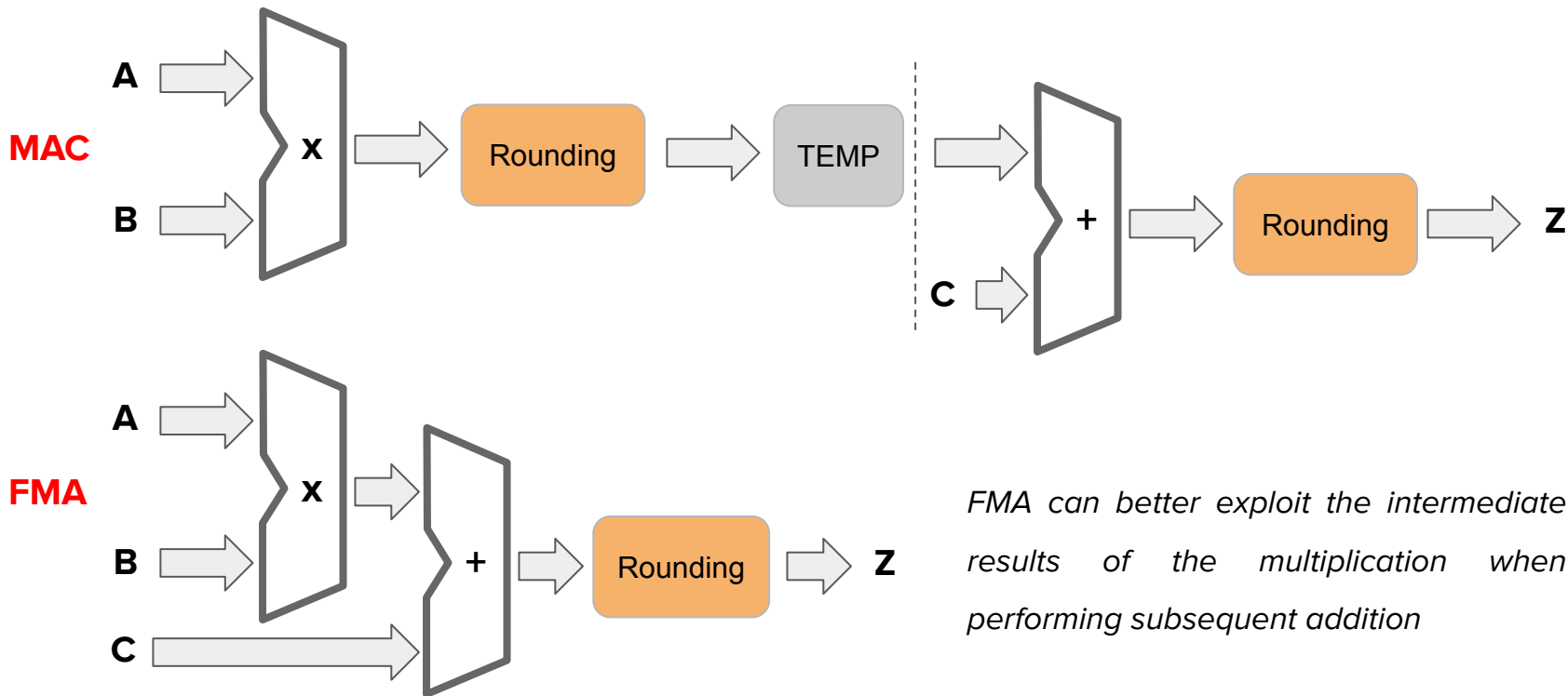
In contrast, Fused Multiply Add (**FMA**) performs the same operations in a single step:

- i) **Multiply and Accumulate** ; $\mathbf{Z} = \mathbf{AxB} + \mathbf{C}$ (single rounding)

Although FMA requires more demanding hardware, it performs a single rounding, is generally faster than MAC and is available in all modern architectures (eg, CPUs, GPUs)

MAC vs FMA: hardware design

- Difference between MAC and FMA from a hardware perspective



MAC vs FMA: example

Let's consider the following example:

$$\mathbf{A} = 1.1011_2 \cdot 2^0 = 1.6875_{10}, \mathbf{B} = 1.1101_2 \cdot 2^0 = 1.8125_{10}, \mathbf{C} = 1.0100_2 \cdot 2^{-2} = 0.3125_{10}$$

$$\mathbf{Z} = \mathbf{A} \times \mathbf{B} + \mathbf{C} = 1.6875_{10} \times 1.8125_{10} + 0.3125_{10} = 3.37109375_{10}$$

Using **MAC**:

$$\text{TEMP} = \mathbf{A} \times \mathbf{B} = 1.1011_2 \times 1.1101_2 = 11.00001111_2 = 1.100001111_2 \cdot 2^1 \approx 1.1000_2 \cdot 2^1 \approx 3.00_{10} \quad (\text{1st rounding})$$

$$\mathbf{Z} = \text{TEMP} + \mathbf{C} = 1.1000_2 \cdot 2^1 + 1.0100_2 \cdot 2^{-2} = 1.1000_2 \cdot 2^1 + 0.0010100_2 \cdot 2^1 = 1.1010_2 \cdot 2^1 \approx 3.25_{10} \quad (\text{2nd rounding})$$

Using **FMA**:

$$\mathbf{A} \times \mathbf{B} = 1.1011_2 \times 1.1101_2 = 11.00001111_2 = 1.100001111_2 \cdot 2^1 = 1.529296875_{10} \cdot 2^1 = 3.05859375_{10}$$

$$\mathbf{A} \times \mathbf{B} + \mathbf{C} = 1.100001111_2 \cdot 2^1 + 1.0100_2 \cdot 2^{-2} = 1.100001111_2 \cdot 2^1 + 0.0010100_2 \cdot 2^1 = 1.10101111_2 \cdot 2^1 =$$

$$= 1.685546875_{10} \cdot 2^1 = 3.37109375_{10} \approx 1.1011_2 \cdot 2^1 = 1.6875_{10} \cdot 2^1 = 3.375_{10} \quad (\text{Single rounding})$$

Floating-point pitfalls

If we add 1.11_{10} to 1.21_{10} , we do not obtain the expected result 2.32_{10} due to approximations:

```
double temp_1 = 1.11, temp_2 = 1.21, temp_3 = 2.32;  
if ( (temp_1 + temp_2) != temp_3 ) is true...
```

Hence, rather than testing equality it is wiser to evaluate the difference between quantities when dealing with floating-point numbers:

```
if ( fabs((temp_1 + temp_2) - temp_3) < epsilon )
```

However, pay attention when subtracting two very close numbers because the difference can be zero due to approximations.

Moreover, for the same reason, take care even when adding a huge number with a tiny one since the latter can *vanish*

Floating-point: associative and distributive properties

Unfortunately, due to approximation encoding and rounding, the associative property does not (always) hold:

$$(A + B) + C \neq A + (B + C)$$

Moreover, even the distributive property is not always satisfied for the same reason;

$$A \times (B + C) \neq A \times B + B \times C$$

For the above reasons, take extra care when handling floating-point data, even using the large double format.

Floating-point and *fast-math* optimizations

- Compilers have flags to generate optimized codes, such as `-O3` in GCC
- Most compilers also have flags to speed up computations when dealing with math operations and, consequently, with floating points computations
- For instance, the `-ffastmath` flag can significantly speed up computations involving floating-point data
- However, among other things, it does so neglecting the existence of subnormal numbers by flushing to zero all these values for efficiency purposes
- Other simplifications occur; thus, take extra care before applying this kind of optimization, as described in detail in Simon Byrne's post:

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

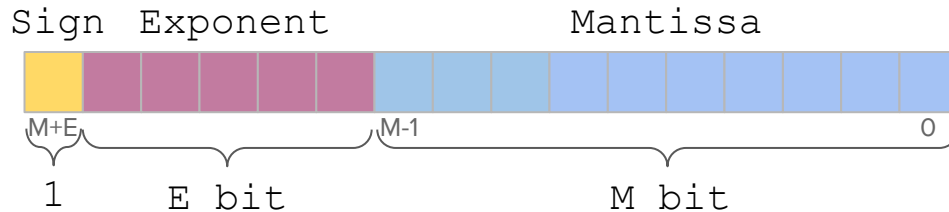
Floating-point encoding: IEEE 754 1/7

- A binary number is expressed in scientific format if it has a single integer digit set to 1 (the other option with the single bit set to 0 is not allowed* for the scientific format)
- Thus, a binary number expressed in normalized scientific format has this form:
 - $(-1)^S \cdot 1.xxxxxx \cdot 2^{yyyy}$, with S a single bit, xxxxxx and yyyy binary digits
- In this notation, the first digit (ie, the integer part) is always* 1 and thus omitted in the encoding
- The fractional part xxxxxx is the *mantissa* **M**
- The yyyy binary digits form the *exponent* **E** (discussed next)
- A single bit **S** encodes the *sign* (S=0 positive sign, S=1, negative sign)
- Consequently, we can represent* a binary value with floating-point notation as:
 $(-1)^S \cdot 1.M \cdot 2^{E-BIAS}$, with **BIAS** a predefined polarization value for each representation

* exceptions for subnormal numbers apply (discussed later)

Floating-point encoding: IEEE 754 2/7

- Standards are crucial to enable seamless interoperability among different architecture, manufacturers, compilers, etc
- Purposely, IEEE 754 defines the standard to represent floating-point numbers using the following scheme:



- Since the integer part is always one*, it is omitted in the encoding, and the mantissa – also referred to as *significand* – is devoted to encoding the fractional part only.
- Hence, the encoded numeric value (excluding sign and exponent and subnormal numbers) corresponds to: **1.**



Floating-point encoding: IEEE 754 3/7

Considering numbers in scientific notation and a format like that defined in the previous slide (sign 1 bit, exponent 5 bit, mantissa 10 bit plus 1 implicit, and setting BIAS = 15):

$$\text{value} = (-1)^S \times 2^{(\sum_{k=0}^4 (b_{10+i} \cdot 2^k) - 15)} \times (1 + \sum_{i=1}^{10} b_{10-i} \cdot 2^{-i})$$

Thus, the following binary digits encodes value 23.265625

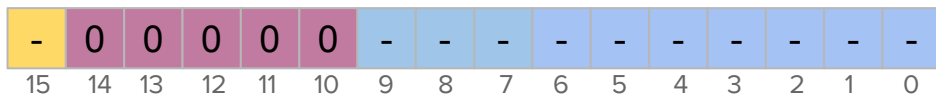
Sign		Exponent					Mantissa									
0	1	0	0	1	1	0	1	1	1	0	1	0	0	0	1	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

$$\begin{aligned} & (-1)^0 \times 2^{(19-15)} \times (1 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6} + 0 \cdot 2^{-7} + 0 \cdot 2^{-8} + 0 \cdot 2^{-9} + 1 \cdot 2^{-10}) = \\ & = 1 \times 2^{(4)} \times (1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64} + \frac{1}{1024}) = 1 \times 16 \times 1.45410156 = 23.265625 \end{aligned}$$

This specific data representation is referred to by IEEE 754 as a *half-precision floating-point* or *floating-point 16* (FP16) since it uses a 16-bit encoding

Floating-point encoding: IEEE 754 4/7

FP16 also allows encoding *subnormal* – numbers around zero ($\pm 0.x$) which are not in scientific format – using an exponent 00000 and a mantissa different from all zeros:

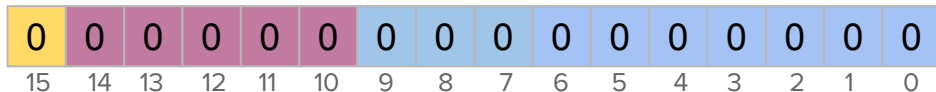


subnormal, “-” means 0 or 1

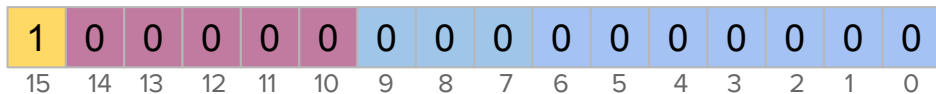
A subnormal number has always exponent $2^{1-\text{BIAS}}$; hence, with BIAS=15, it corresponds to:

$$\text{value} = (-1)^S \times 2^{-14} \times \left(0 + \sum_{i=1}^{10} b_{10-i} \cdot 2^{-i} \right)$$

FP16 encodes zero(s) – indeed, there are two zero – with the following codes:



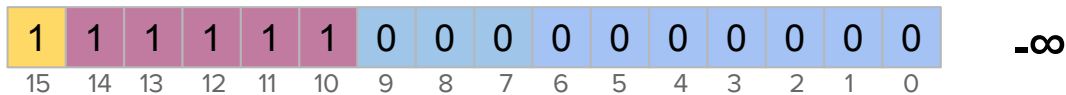
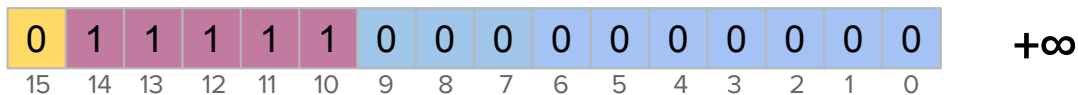
Zero (+0)



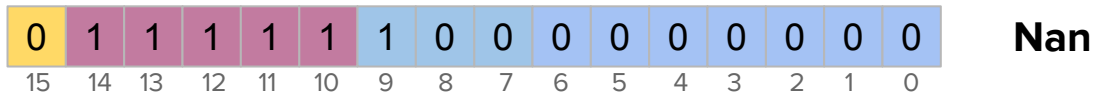
Zero (-0)

Floating-point encoding: IEEE 754 5/7

Additionally, In FP16 some configurations are reserved to represent specific cases such as $+\infty$ (eg, 1.0/0.0) and $-\infty$ (eg, -1.0/0.0):



Finally, FP16 encodes the result of unpredictable operations, referred to as NaN (*Not a Number*), with all exponent bits set to 1 and a mantissa different from all zeros:



Floating-point encoding: IEEE 754 6/7

In FP16, considering the reserved configurations 0 and 31, the exponent E for numbers in scientific/normalized format can assume any integer value between 1 and 30:

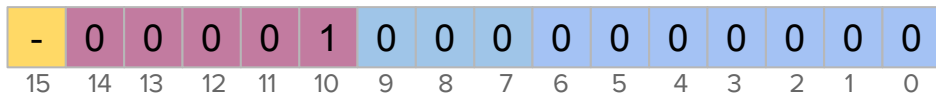


Hence, being in FP16 the exponent polarized according to a BIAS 15 (ie, 2^{E-15}), the exponential factor spans from 2^{-14} to 2^{+15} . Consequently, the dynamic range spans from



- means 0 (+) or 1 (-)

$$\pm 65504 = 1 \times 2^{15} \times (1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256} + \frac{1}{512} + \frac{1}{1024}) = 2^{15} \times 1.9990234375 \text{ to}$$

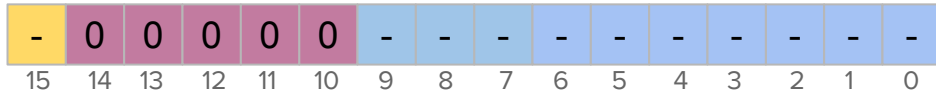


- means 0 (+) or 1 (-)

$$\pm 0.00006103515625 = 1 \times 2^{-14} \times (1 + 0) = 2^{-14}$$

Floating-point encoding: IEEE 754 7/7

- Subnormal numbers can be either positive ($S=0$) or negative ($S=1$)



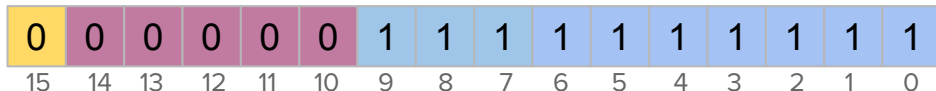
- means 0 (+) or 1 (-)

- Although not in scientific format, subnormal numbers allow to encode values around zero ranging from $[-1 \times 2^{-14}$ to $+1 \times 2^{-14}]$

- Specifically,

$$[-(2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-10})\times 2^{-14}, +(2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-10})\times 2^{-14}]$$

$$[-1023/1024 \times 2^{-14}, +1023/1024 \times 2^{-14}]$$

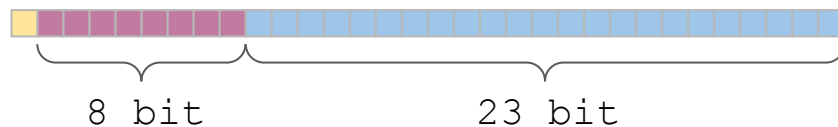


IEEE 754: data formats

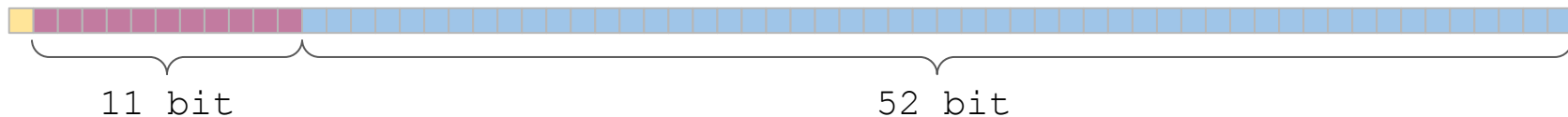
- To facilitate interoperability, IEEE released a first draft to standardize floating-point data and computations in 1985
- Since then, it was updated in 2008 and 2019
- Among other things, currently, it supports the following data types:
 - **FP16** (Half, or binary16) : 16 bit, S=1, E=5, M=10, BIAS=15
 - **FP32** (Float, or binary32): : 32 bit, S=1, E=8, M=23, BIAS=127
 - **FP64** (Double, or binary, 164) : 64 bit, S=1, E=11, M=52, BIAS=1023
 - **FP128** (Quadruple, or binary128) : 128 bit, S=1, E=15, M=112, BIAS=16383
 - **FP256** (Octupule, or binary256) : 256 bit, S=1, E=19, M=236, BIAS=262143

IEEE 754: Float and Double


- In addition to the FP16 format previously described, IEEE 754 defines the following data formats used in most programming languages/compiler: `float` and `double`, respectively, using a 32 and 64 bits
- **Float** (32 bit) – *single precision*: sign 1 bit, exp. 8 bit, mantissa 23 bit (+1), and BIAS = 127. Range [min, max] is: $[1.17549435 \cdot 10^{-38}, 3.40282347 \cdot 10^{+38}]$

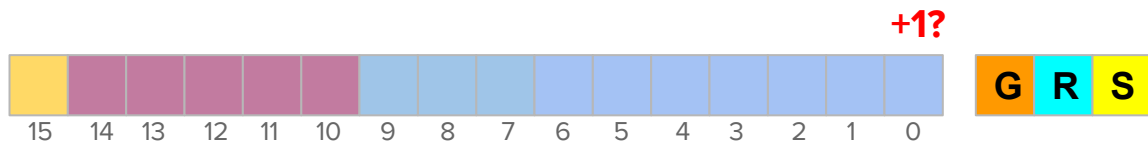





- **Double** (64 bit) – *double precision*: sign 1 bit, exp. 11 bit, mantissa 52 bit (+1), and BIAS = 1023. Range [min, max] is: $[2.2250738585072014 \cdot 10^{-308}, 1.7976931348623158 \cdot 10^{+308}]$



IEEE 754: Guard, Round and Sticky bits for rounding

- IEEE 754 performs rounding using three additional hidden bits (i.e., in addition to the specific format used) during encoding and computations
- These bits are called Guard (**G**), Round (**R**), and Sticky (**S**) 
- They are somehow concatenated to the right end to decide whether the +1 is needed:

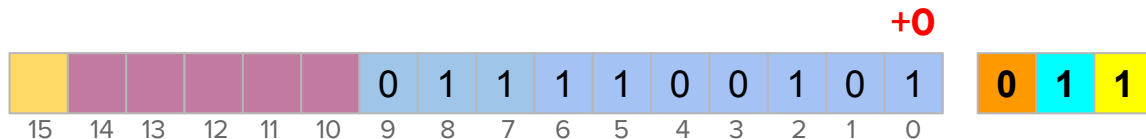


- The S bit is set to 1 (for instance, when shifting right) if at least one of the bits at its right was 1, while G and R are two extra bits which increase the mantissa length by 2
- If **G=0**, the mantissa doesn't change, regardless of R and S (GRS = 0xx) 
- If **G=1, R=1, S=x** or **G=1, R=0, S=1**, then round up (+1) 
- If **G=1, R=0, S=0**, we are in between. If the last bit of the mantissa is 0, do nothing else add +1 to the mantissa. In the IEEE terminology, *round to even* 

IEEE 754: rounding examples 1/2

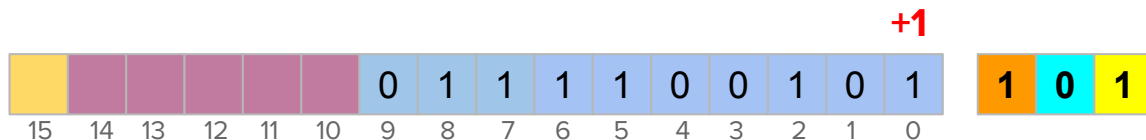
With FP16, suppose that the result of a multiplication is $1.011110010101011010 \cdot 2^{-2}$

GRS bits can summarize the additional bits 01011010 as G=0, R=1, S=1 (the OR between the last 6 bits 011010). Consequently, the mantissa remains unchanged:



With FP16, suppose that the result of a multiplication is $1.011110010110011010 \cdot 2^{-2}$

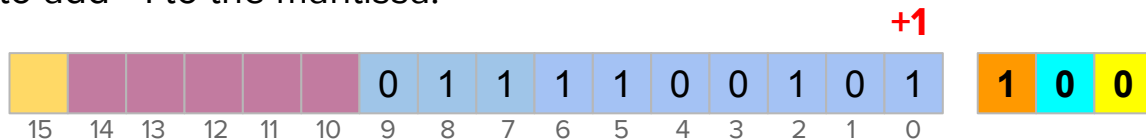
GRS bit can summarize the additional bits 10011010 as G=1, R=0, S=1 (the OR between the last 6 bits 011010); consequently, the mantissa requires a +1 increment:



IEEE 754: rounding examples 2/2

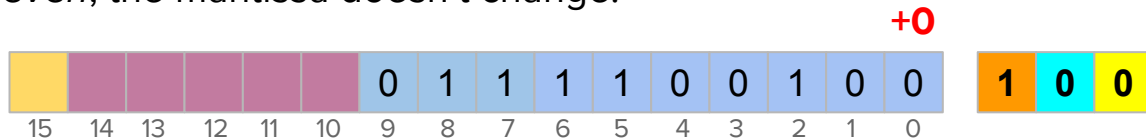
With FP16, suppose that the result of a multiplication is $1.0111100101100000000 \cdot 2^{-2}$

GRS bits can summarize the additional bits 10000000 as $G=1, R=0, S=0$ (the OR between the last 6 bits 0000000). Nonetheless, since the value in the mantissa is odd, the *round to even* requires to add +1 to the mantissa:



Finally, with FP16, suppose that the result of a multiplication is $1.0111100100100000000 \cdot 2^{-2}$

GRS bits can summarize the additional bits 10011010 as $G=1, R=0, S=0$ (the OR between the last 6 bits 0000000). There is a tie, but the last bit of the mantissa is even, and according to *round to even*, the mantissa doesn't change:

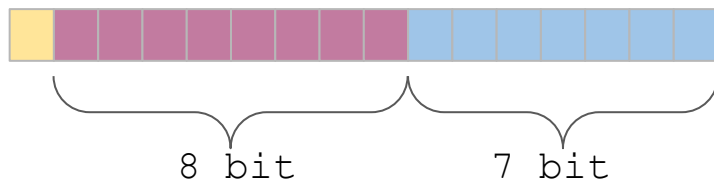


Other floating-points formats

- With the widespread diffusion of AI, some companies have defined other formats; they are typically more compact and better tailored for their purposes
- The main reason concerns efficiency and power consumption that generally *scales as the square of the mantissa length during arithmetic operations* (source, NVidia)
- Another reason for more compact representations is the reduced memory footprint
- In the following slides, we'll introduce the following popular formats, although not (yet) IEEE standard:
 - **BFloat**
 - **Tensore-Core32**
 - **Minifloat E5M2**
 - **Minifloat E4M3**

Brain Float (Google Brain) 1/2

- Brain Float (BFloat) is a floating-point proposed by Google for its devices
- It's gaining interest since it reduces data storage and speeds up AI computations
- To better fit AI needs, it trades dynamic range for precision using a 16 bit encoding
- Specifically, it is a more compact 16 bit version of IEEE 754 FP32; it preserves the dynamic range (ie, same $E=8$ and $BIAS=127$) but has a shortened Mantissa (from 23 to 7 bits), reducing accuracy (or precision, meaning with this the reduced number of bits):

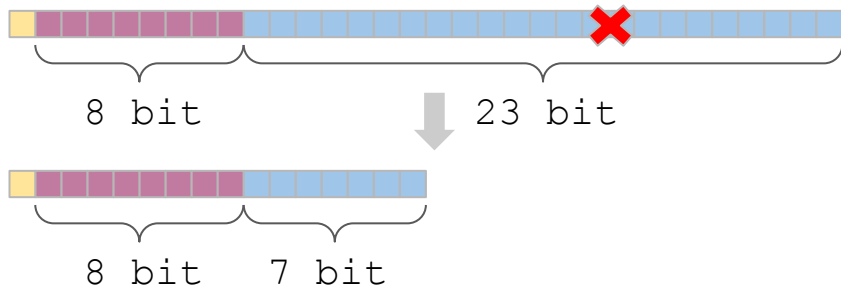


- Data range equivalent to FP32 but with reduced precision
- Adopted by Google (TPUs), Intel (AVX-512), NVidia (CUDA), Apple (CPUs), etc

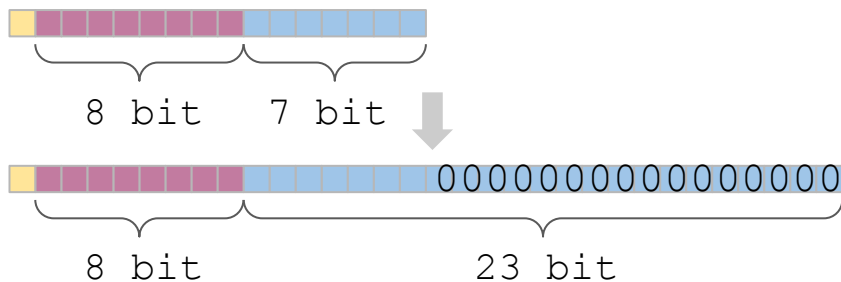
Brain Float 16 (Google Brain) 2/2

- Features: Fast conversion from float (FP32) to BFloat (eg, a simple truncation of M, from 23 to 7 bit, or better with rounding) and from BFloat to FP32 (ie, padding missing mantissa bits with zeros):

- From Float to BFloat:

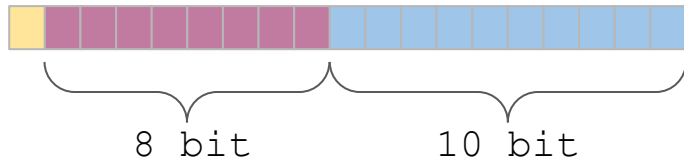


- From BFloat to Float:



TensorFloat-32 (NVIDIA) 1/2

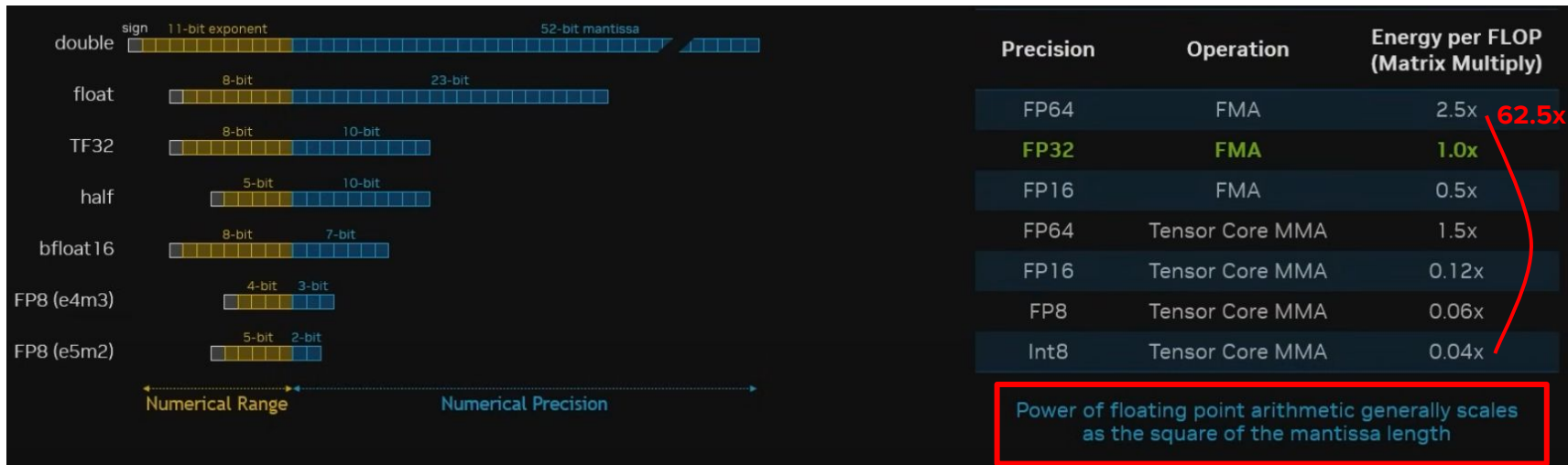
- In its recent GPUs, NVidia has introduced this new floating-point format, enabling a good trade-off between performance and accuracy, fitting very well with tensor cores
- It has the same exponent E of a float (FP32) and BFloat, thus using the same BIAS, but the mantissa is only 10 bit wide. Specifically, TensorFloat-32 encodes numbers with 1-bit sign S , exponent $E=8$ bit, mantissa $M=10$ bit and BIAS = 127.



- Data range equivalent to FP32 with reduced precision (but better than BFloat)

TensorFloat-32 (NVIDIA) 2/2

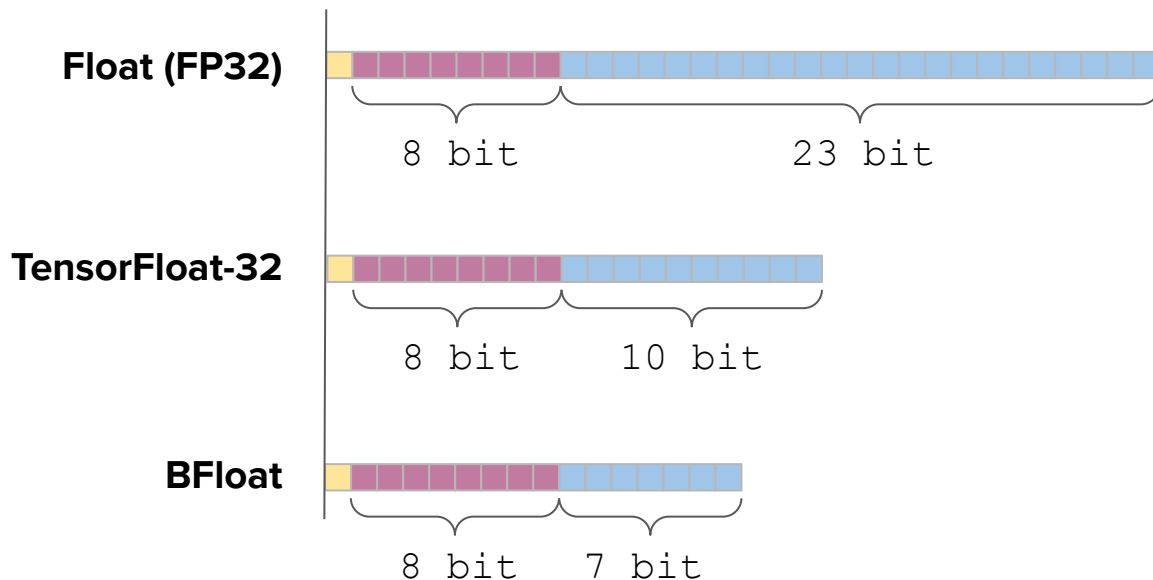
- NVidia claims significant benefits in terms of reduced power consumption vs FP32
- The reduced precision might not represent a crucial issue for some relevant tasks in AI
- The following screenshot, taken from an NVidia [talk](#), reports Energy/FLOPS for Multiply-Add operations using different data formats on NVidia GPUs



MMA: Matrix Multiply-Accumulate (like $D = A * B + C$) – FMA: Fused Multiply-Add

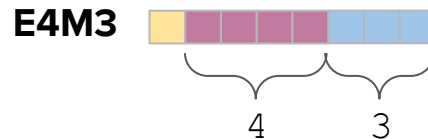
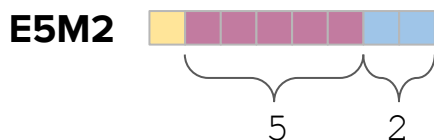
Float vs TensorFloat-32 vs BFloat

Below, we summarize the different encoding of the three widely used floating point formats Float, TensorFloat-32 and BFloat, all using an 8-bit exponent E and the same BIAS=127 but a different number of bits for the mantissa:



Minifloat data FP8

- AI can be surprisingly effective even with reduced accuracy
- Consequently, encoding data with less than 16 bits can be a viable solution in some cases to reduce data storage and computations using mixed-type data
- Below, the 8-bit encoding of two minifloat data types (not yet included in the IEEE 754 standard) named E5M2 (E=5, M=2, BIAS=15) and E4M3 (E=4, M=3, BIAS=7)



- As usual, they follow the IEEE 754 specifications but with some limitations (eg, E4M3)
- Major players use these data types (NVidia, ARM, Intel, AMD, ..)

Minifloat: E5M2 1/3

- *Full-fledged* floating point with S=1, E=5, M=2 and BIAS=15
- It represents subnormal numbers (E=0, M≠0), infinities (E= 1111, M=0), NaNs (E= 1111, M≠0) and two zeros (00000000 and 10000000, respectively, for +0 and -0)
- The dynamic range spans from $\pm 2^{-14} \cdot (0 + \frac{1}{4})$ to $\pm 2^{15} \cdot (1 + \frac{3}{4})$

E5M2



Since E= 1111 is reserved for infinities and NaNs, and E=00000 for subnormal numbers, the exponent spans $[11110_2, 00001_2] = [30_{10}, 1_{10}]$ for values in scientific notation

Therefore, considering the BIAS=15, the actual exponents span the range $[15_{10}, -14_{10}]$, with -14 (ie, 1-BIAS) for subnormal numbers

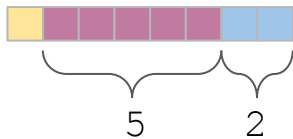
The 2-bit used for the mantissa allows to encode the following values in scientific format:

$1, 1 + \frac{1}{4}, 1 + \frac{1}{2}, 1 + \frac{1}{2} + \frac{1}{4} = 1, 1.25, 1.5, 1.75$ and for subnormals $\frac{1}{4}, \frac{1}{2}, \frac{1}{2} + \frac{1}{4} = 0.25, 0.5, 0.75$

Minifloat: E5M2 2/3

- *Full-fledged* floating point
- It encodes NaN and Inf

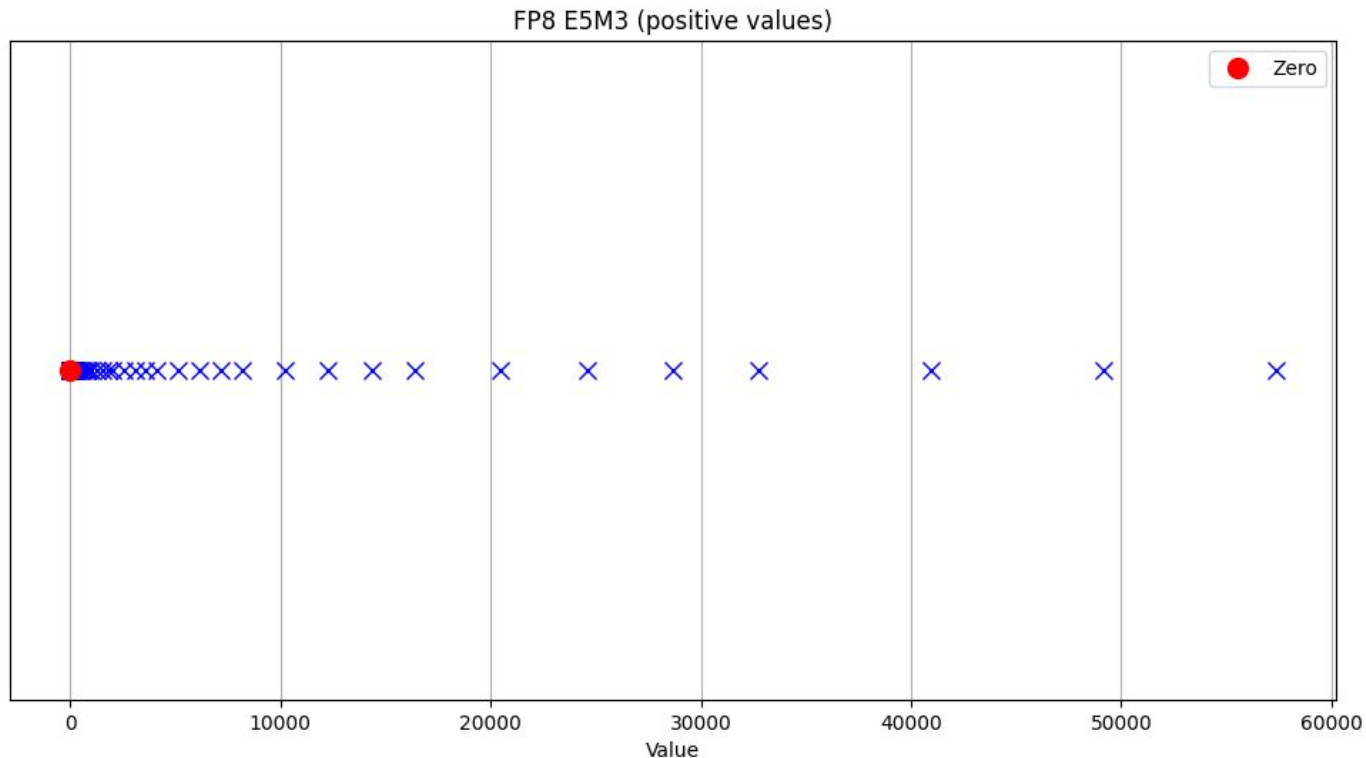
E5M2



E5M2				
Binary	Hex	Value	Notes	
0 00000 00	0	0		
0 00000 01	1	$2^{-14} \times (0 + \frac{1}{4}) \approx 0.000015258$	Smallest positive subnormal number	
0 00000 11	3	$2^{-14} \times (0 + \frac{3}{4}) \approx 0.000045776$	Largest subnormal number	
0 00001 00	4	$2^{-14} \times (1 + \frac{0}{4}) \approx 0.000061035$	Smallest positive normal number	
0 00110 00	18	$2^{-9} \times (1 + \frac{0}{4}) = 0.001953125$	E4M3 - Smallest positive subnormal number	
0 01101 01	35	$2^{-2} \times (1 + \frac{1}{4}) = 0.3125$	Nearest value to 1/3	
0 01110 11	37	$2^{-1} \times (1 + \frac{3}{4}) = 0.875$	Largest number less than one	
0 01111 00	38	$2^0 \times (1 + \frac{0}{4}) = 1$	One	
0 01111 01	39	$2^0 \times (1 + \frac{1}{4}) = 1.25$	Smallest number larger than one	
0 01111 10	3a	$2^0 \times (1 + \frac{2}{4}) = 1.5$	Nearest value to sqrt(2) = 1.414	
0 10000 10	42	$2^1 \times (1 + \frac{2}{4}) = 3.0$	Nearest value to PI = 3.141	
0 10111 11	5f	$2^8 \times (1 + \frac{3}{4}) = 448$	E4M3 - Largest number	
0 11110 11	7b	$2^{15} \times (1 + \frac{3}{4}) = 57344$	Largest normal number	
0 11111 00	7c	<i>inf</i>	Infinity	
0 11111 01	7d	<i>NaN</i>	First of 6 NaN representations	
1 00000 00	80	0		
1 10000 00	c0	-2		
1 11111 00	fc	- <i>inf</i>	Negative Infinity	
1 11111 11	ff	<i>NaN</i>	Last of 6 NaN representations	

Minifloat: E5M2 3/3

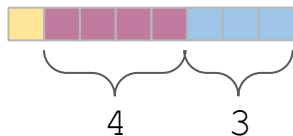
The plot below reports the positive values encodable with the E5M2 format



Minifloat FP8: E4M3

- Compared to E5M2, it has a smaller mantissa and larger exponent
- Notably, it does not encode Inf

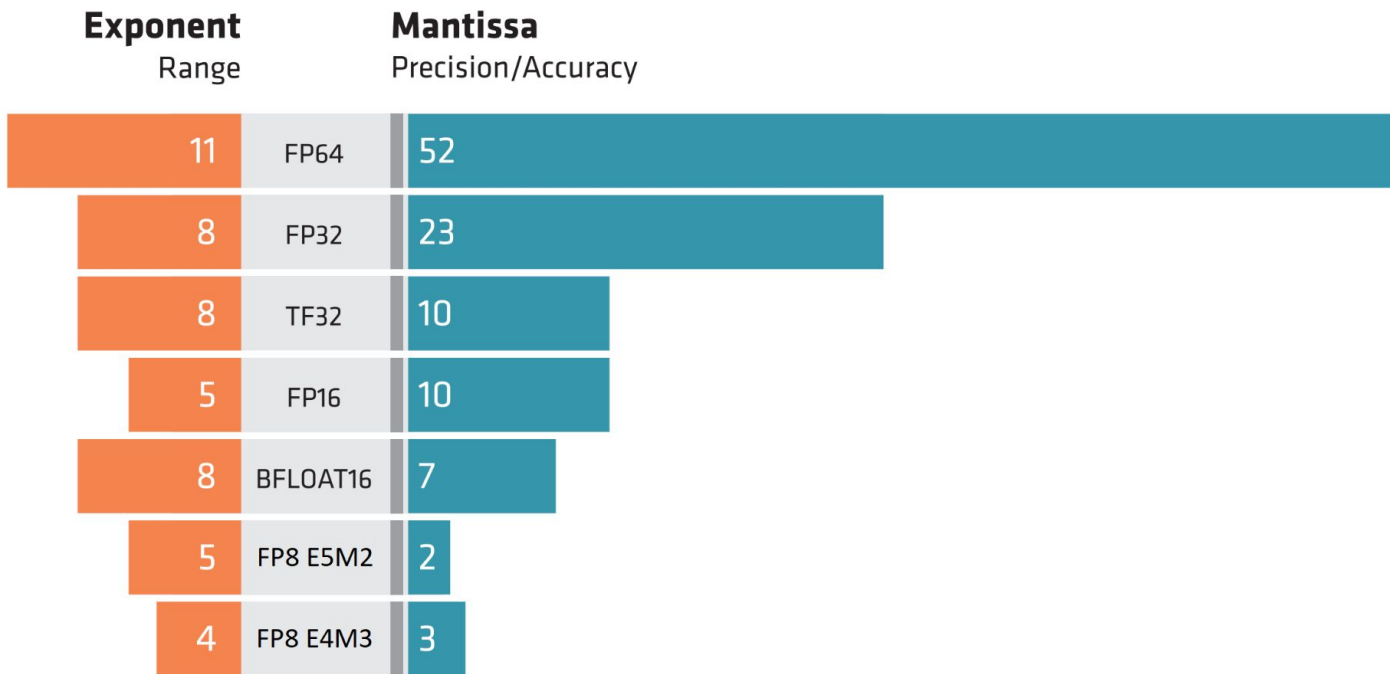
E4M3



E4M3

Binary	Hex	Value	Notes
0 0000 000	0	0	
0 0000 001	1	$2^{-6} \times (0 + \frac{1}{8}) = 0.001953125$	Smallest positive subnormal number
0 0000 111	7	$2^{-6} \times (0 + \frac{7}{8}) = 0.013671875$	Largest subnormal number
0 0001 000	8	$2^{-6} \times (1 + \frac{0}{8}) = 0.015625$	Smallest positive normal number
0 0101 011	2b	$2^{-2} \times (1 + \frac{3}{8}) = 0.34375$	Nearest value to 1/3
0 0110 111	37	$2^{-1} \times (1 + \frac{7}{8}) = 0.9375$	Largest number less than one
0 0111 000	38	$2^0 \times (1 + \frac{0}{8}) = 1$	One
0 0111 001	39	$2^0 \times (1 + \frac{1}{8}) = 1.125$	Smallest number larger than one
0 0111 100	3c	$2^0 \times (1 + \frac{3}{8}) = 1.375$	Nearest value to sqrt(2) = 1.414
0 1000 101	45	$2^1 \times (1 + \frac{5}{8}) = 3.25$	Nearest value to PI = 3.141
0 1110 111	77	$2^7 \times (1 + \frac{7}{8}) = 240$	Largest normal number without range extension
0 1111 110	7e	$2^8 \times (1 + \frac{6}{8}) = 448$	Largest normal number
0 1111 111	7f	NaN	First of 2 NaN representations
1 0000 000	80	0	
1 1000 000	c0	-2	
1 1111 111	ff	NaN	Last of 2 NaN representations

Summary of floating-point data types



<https://rocm.docs.amd.com/en/docs-6.0.2/about/compatibility/data-type-support.html>

Exercise

Exercise

Implement a C function to compute, with basic arithmetic and logic operations, the numerical value of a number encoded in E5M2 floating point format.

Hint: $2^1 = 1 \ll 1$, $2^2 = 1 \ll 2$, $2^{-1} = 1 / (1 \ll 1)$, $2^{-2} = 1 / (1 \ll 2)$

```
struct E5M2 {  
    unsigned char bit[8];  
    float numeric_value;  
    int raw_exponent;  
    int actual_exponent;  
    float fractional;  
    float significand;  
    bool is_negative;  
    bool is_Normal;  
    bool is_Subnormal;  
    bool is_Zero;  
    bool is_Inf;  
    bool is_NaN;  
    int BIAS; };
```

Addition is all you need? 1/2

- A recent paper (Oct. 2024), “Addition is All You Need for Energy-efficient Language Models”, proposes an approximated method (referred to as L-Mul) to speed up and reduce power consumption for floating-point computation in language models
- It relies only on lightweight integer additions, replacing the expensive multiplication with an approximated term
- Given two numbers X and Y encoded in floating-point format, with m bits for mantissa and e bits for exponent, they can be expressed as: $X = (1 + X_m) \cdot 2^{X_e}$ and $Y = (1 + Y_m) \cdot 2^{Y_e}$
- Their product results:

$$(1 + X_m) \cdot 2^{X_e} \cdot (1 + Y_m) \cdot 2^{Y_e} = (1 + X_m + Y_m + X_m \cdot Y_m) \cdot 2^{X_e + Y_e} \approx (1 + X_m + Y_m + 2^{-L(m)}) \cdot 2^{X_e + Y_e}$$

$$\text{with } L(m) = \begin{cases} m, & \text{if } m \leq 3 \\ 3, & \text{if } m = 4 \\ 4, & \text{if } m > 4 \end{cases}$$

<https://arxiv.org/abs/2410.00907>

Addition is all you need? 2/2

- Author claims:
 - *L-Mul is more precise (accurate???) than fp8 e4m3 multiplications but uses less computation resource than fp e5m2*
 - *massive power reduction with marginal or null performance drop in some relevant use cases reported in the paper*
- Independently of these claims, the outlined strategy yields a raw significant accuracy drop compared to the conventional multiplication approach
- Regardless of the effectiveness of this specific method, reducing power consumption with floating-point arithmetic is a vivid topic with significant environmental implications (“ It is estimated that Google’s AI service could consume as much electricity as Ireland (29.3 TWh per year) in the worst-case scenario (de Vries, 2023).”)

Exercise

Exercise

Given two numbers encoded in the floating point format E5M2, evaluate the difference between the conventional and approximated methods proposed in the paper “Addition is All You Need for Energy-efficient Language Models”.

Hint: use the function implemented in the previous exercise.

Computational issues and memory footprint

- Neural networks typically consist of many (million) weights encoded as floating-point data
- Such an amount of data yields computational issues, often related to multiply and add/matrix multiplications and convolutions; to this aim, parallel computational strategy and architecture can allow to cope with it
- However, such models also have massive memory footprints, challenging the memory available even for devices equipped with powerful computing platforms
- Therefore, there are techniques to face these problems, such as:
 - Data quantization to reduce data size
 - Weight sharing or Palletization to reduce the amount of data

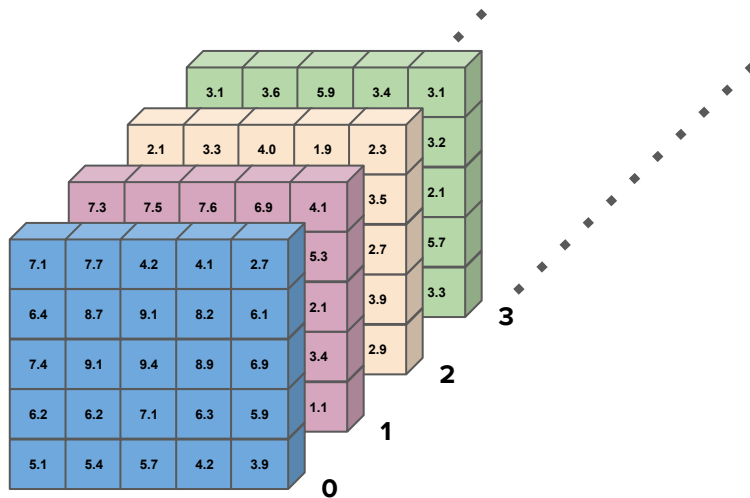
Reducing data size

- A straightforward strategy to reduce computations and memory footprint consists of reducing the data size
- Starting from an existing network model using 32-bit floating-point data, moving to a smaller data representation can improve both issues
- For instance, switching from FP32 to E5M2 would decrease the footprint by 1/4
- Of course, the resulting network is not equivalent to the original one and post-quantization fine-tuning is required to achieve reasonable performance
- A further yet more challenging improvement from the computational point of view would consist of switching to integer data
- An even further step forward is represented by Logic Gate Networks (LGNs), performing high-speed computations with conventional gates (eg, XOR, NAND, AND)

<https://arxiv.org/pdf/2411.04732>

Weight-sharing (or Palettization) 1/3

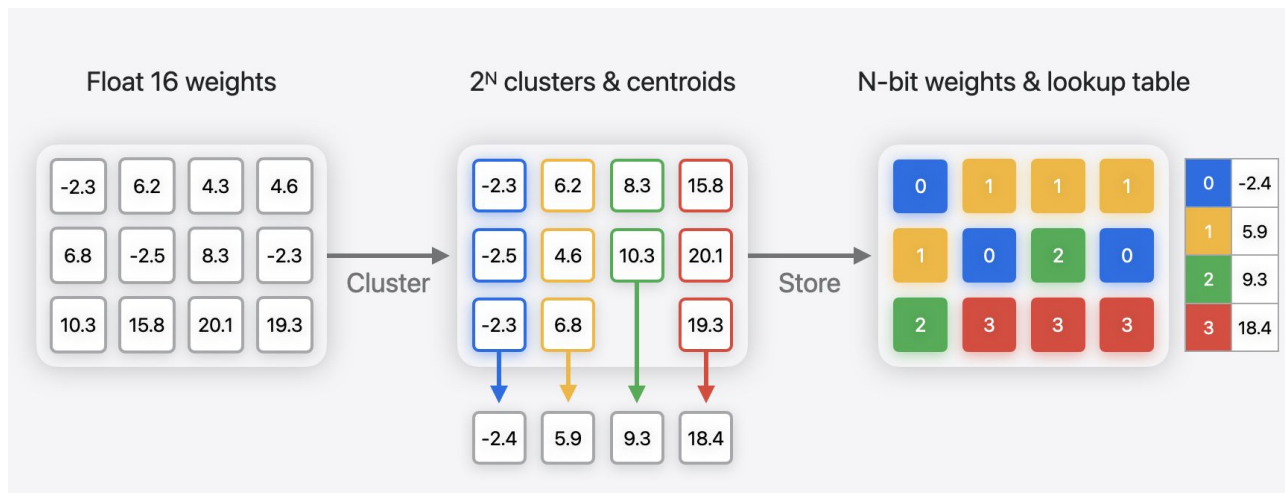
- The massive amount of data in network models is primarily needed to store the floating-point weights needed to perform computations (eg, large amounts of different convolutions, each one with different kernel weights)



- Weight sharing aims to reduce the memory footprint using a smaller yet representative number of weights indexed through a Look-Up-Tables (LUT)

Weight-sharing (or Palettization) 2/3

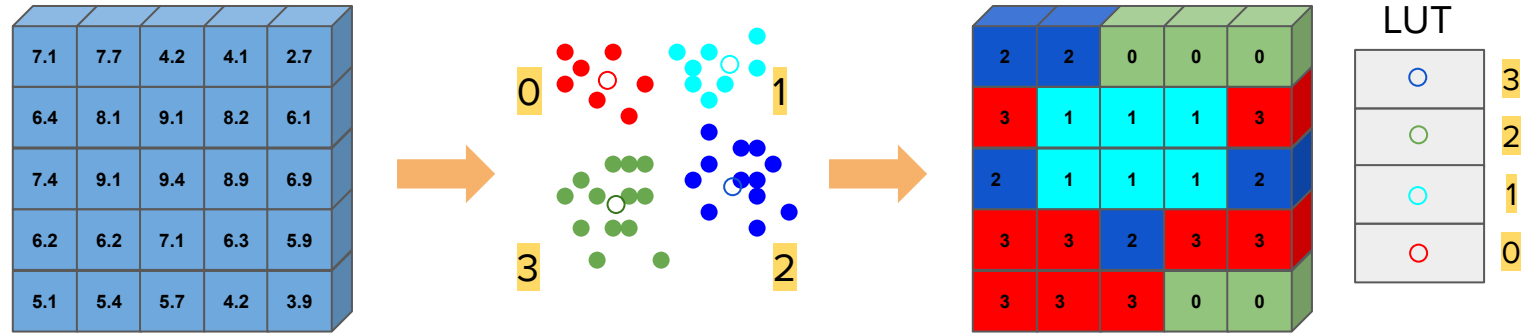
- The key idea is to replace multiple weights with their average values (centroids) using quantization techniques and Look-Up-Tables to index centroids
- For instance, the weights below could be clustered in 4 classes and to each centroid assigned an index [0,3]; overall, 4 weights (centroids) and 2 bit for each element



<https://apple.github.io/coremltools/docs-guides/source/opt-palettization-overview.html>

Weight-sharing (or Palettization) 3/3

- Given the number of classes, K-means is a well-known vector quantization technique suited for this purpose; it clusters the input values around each centroid



- In the figure, with FP 32 weights, the kernel requires 25x32 bit; after quantization in 4 classes, the memory footprint is 25x2 + 4x32 bit – reduced by a factor $800/178 \approx 4.5$
- The lower the number of clusters, the lower the memory footprint and accuracy
- Thus, fine-tuning is required to revolve the loss in accuracy after palettization

Fixed-point arithmetic: use case

So far, we have considered floating-point as the unique method to deal with real values, but can we perform these computations using only integer data/operations?

Such an approach might have significant advantages:

- floating-point modules/features could not be available (eg, microcontrollers) or highly resource-demanding (eg, FPGA)
- computations using floating-point often require longer latency vs integer

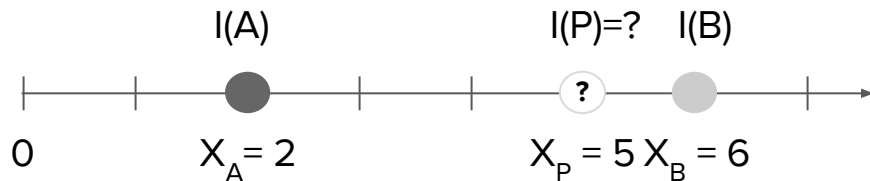
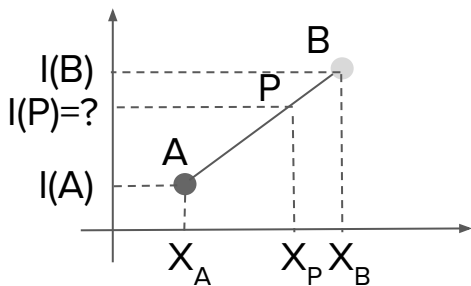
For instance, we can add two numbers like 0.75_{10} and 1.5_{10} , performing the following integer operations: $0.75 \rightarrow 75 \cdot 10^{-2}$, $1.50 \rightarrow 150 \cdot 10^{-2}$ and consequently their sum (up to a scale factor) is: $0.75_{10} + 1.5_{10} = 225 \cdot 10^{-2}_{10}$

Integer operations were feasible by simply scaling up numbers by a constant factor (ie, 10^2)

Example: linear interpolation (real)

Example

- Given two points A and B, located at X_A and X_B and with intensity magnitude $I(A)$, $I(B)$, compute the weighted color intensity $I(P)$ for point P located at $X_P = 5$



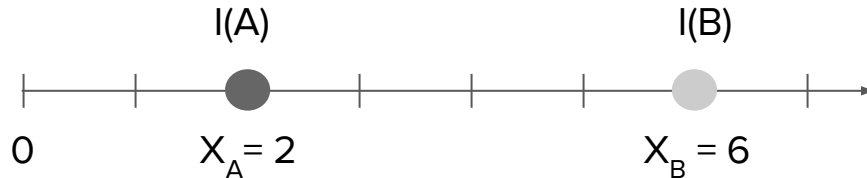
$$I(P) = I(A) \frac{X_B - X_P}{X_B - X_A} + I(B) \frac{X_P - X_A}{X_B - X_A}$$

Assuming $I(A)=20$, and $I(B)=200$: $I(P) = 20 \cdot (0.25) + 200 \cdot (0.75) = 155$

Example: linear interpolation (integer) 1/3

Can we perform this computation using only integer operations?

Assuming, for simplicity, that $I(A)$ and $I(B)$ are integers and point P lies only at discrete positions, the weights $[w_A, w_B]$ are: $[1,0]$, $[0.75,0.25]$, $[0.5,0.5]$, $[0.25,0.75]$, $[0,1]$



For instance, if $P=5$, $I(A)=20$, and $I(B)=200$, by scaling up weights by a factor of 10^2 , we get:

$I^*(P) = 20 \cdot (25) + 200 \cdot (75) = 500 + 15000 = 15500 = 155 \cdot 10^2$ (only integer operations).

Finally, we obtain the expected result (ie, 155) by scaling down (ie, *dividing* by a factor of 10^2).

However, the final division is unnecessary since it suffices a simple truncation – that might yield approximation, although not in this case – of the two last decimal digits.

Example: linear interpolation (integer) 2/3

- Issue 1: Let's consider what happens with $P=5$, $I(A)=23$, and $I(B)=178$

$$I^*(P) = 23 \cdot (25) + 178 \cdot (75) = 575 + 13350 = 13925 = 139.25 \cdot 10^2 \quad (\text{only integer operations})$$

In this case, truncating the last two digits leads to an acceptable approximation in most cases (e.g., with images, the pixel intensity is a discrete value).

Moreover, by analyzing the last two digits, we can perform the appropriate rounding (in this case, $139.25 \approx 139$ equivalent to C function `round()`).

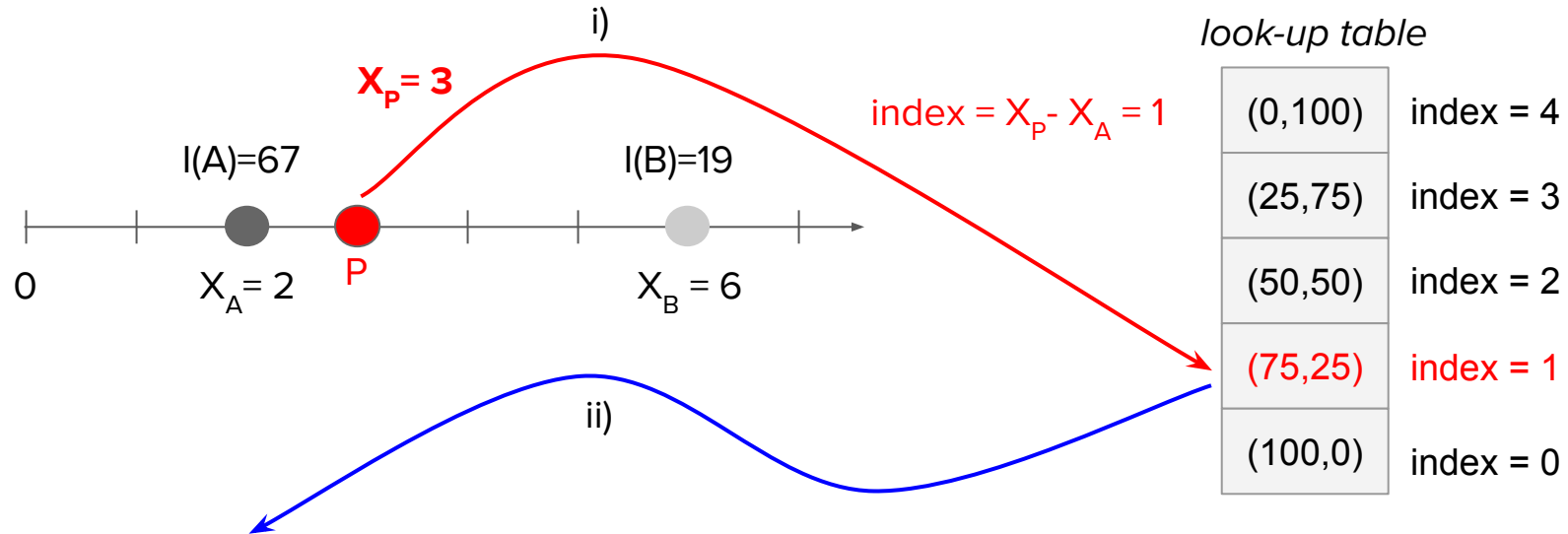
- Issue 2: All operations occur with integer values, but weighting factors need conventional *real* division

That's true, but weighting factors can be computed offline, scaled up by the chosen factor, and stored in a look-up table (LUT) easily index (e.g., in this case, using the index $[XP-XA]$).

The advantage is twofold: all integer operations and no weight computation at run-time.

Example: linear interpolation (integer) 3/3

The overall process involving only integer operations can be summarized as follows:



$$I^*(P) = 67 \cdot (75) + 19 \cdot (25) = 5025 + 475 = 5500 = 55 \cdot 10^2 \quad (\text{only integer operations})$$

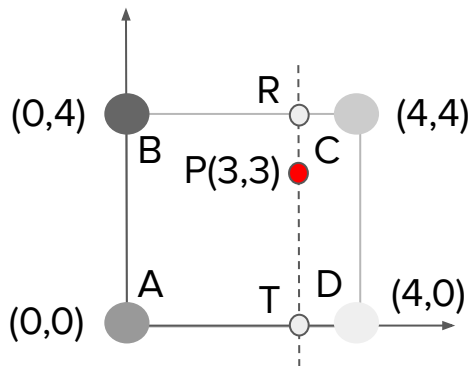
$$I(P) = 55 \quad (\text{truncating last two digits and rounding})$$

Of course, we can use binary numbers as well and scaling by power of 2

Example: bilinear interpolation (real) 1/4

Example

- Given the four points A, B, C, D, with intensity magnitude $I(A)$, $I(B)$, $I(C)$ and $I(D)$, compute the weighted intensity $I(P)$ of point P



We can compute two horizontal linear interpolations (A,D and B,C) and then perform a final vertical bilinear interpolation between T and R as reported in the next slide

Example: bilinear interpolation (real) 2/4

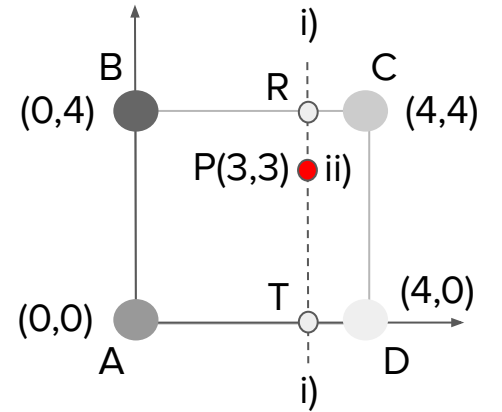
i) Horizontal linear interpolations (A,D and B,C):

$$I(T) = I(A) \frac{X_D - X_T}{X_D - X_A} + I(D) \frac{X_T - X_A}{X_D - X_A}$$

$$I(R) = I(B) \frac{X_C - X_R}{X_C - X_B} + I(C) \frac{X_R - X_B}{X_C - X_B}$$

ii) Vertical linear interpolations (T,R):

$$I(P) = I(T) \frac{Y_R - Y_P}{Y_R - Y_T} + I(R) \frac{Y_P - Y_T}{Y_R - Y_T}$$



$$X_A = X_B, X_C = X_D, X_R = X_T$$

Replacing in the last equation $I(R)$ and $I(T)$, and recalling that $X_A = X_B$, $X_C = X_D$, $X_R = X_T$, we obtain:

$$I(P) = I(A) \frac{X_D - X_T}{X_D - X_A} \frac{Y_R - Y_P}{Y_R - Y_T} + I(D) \frac{X_T - X_A}{X_D - X_A} \frac{Y_R - Y_P}{Y_R - Y_T} + I(B) \frac{X_D - X_T}{X_D - X_A} \frac{Y_P - Y_T}{Y_R - Y_T} + I(C) \frac{X_T - X_A}{X_D - X_A} \frac{Y_P - Y_T}{Y_R - Y_T}$$

Example: bilinear interpolation (real) 3/4

Since the denominator is the same, we can simplify the equation as follows:

$$I(P) = \frac{1}{(X_D - X_A)(Y_R - Y_T)} (I(A)(X_D - X_T)(Y_R - Y_P) + I(D)(X_T - X_A)(Y_R - Y_P) + I(B)(X_D - X_T)(Y_P - Y_T) + I(C)(X_T - X_A)(Y_P - Y_T))$$

Then, renaming the four inner factors as:

$$\left. \begin{aligned} Q_{00} &= (X_D - X_T)(Y_R - Y_P) \\ Q_{01} &= (X_T - X_A)(Y_R - Y_P) \\ Q_{11} &= (X_D - X_T)(Y_P - Y_T) \\ Q_{10} &= (X_T - X_A)(Y_P - Y_T) \end{aligned} \right\} \text{integer or real (see next slides)}$$

finally, we obtain:

$$I(P) = \frac{1}{(X_D - X_A)(Y_R - Y_T)} (I(A)Q_{00} + I(D)Q_{01} + I(B)Q_{11} + I(C)Q_{10})$$

The last equation shows that the computation of $I(P)$ requires real values. However, we can obtain an accurate approximation or the exact result by performing only integer computations.

Example: bilinear interpolation (real) 4/4

Assuming $I(A)=20$, $I(B)=10$, $I(C)=150$, $I(D)=200$:

$$I(P) = \frac{1}{4 \cdot 4} (20 \cdot 1 \cdot 1 + 200 \cdot 3 \cdot 1 + 10 \cdot 1 \cdot 3 + 150 \cdot 3 \cdot 3) = \frac{1}{16} (20 + 600 + 30 + 1350) = 125$$

Assuming $I(A)=20$, $I(B)=10$, $I(C)=151$, $I(D)=200$:

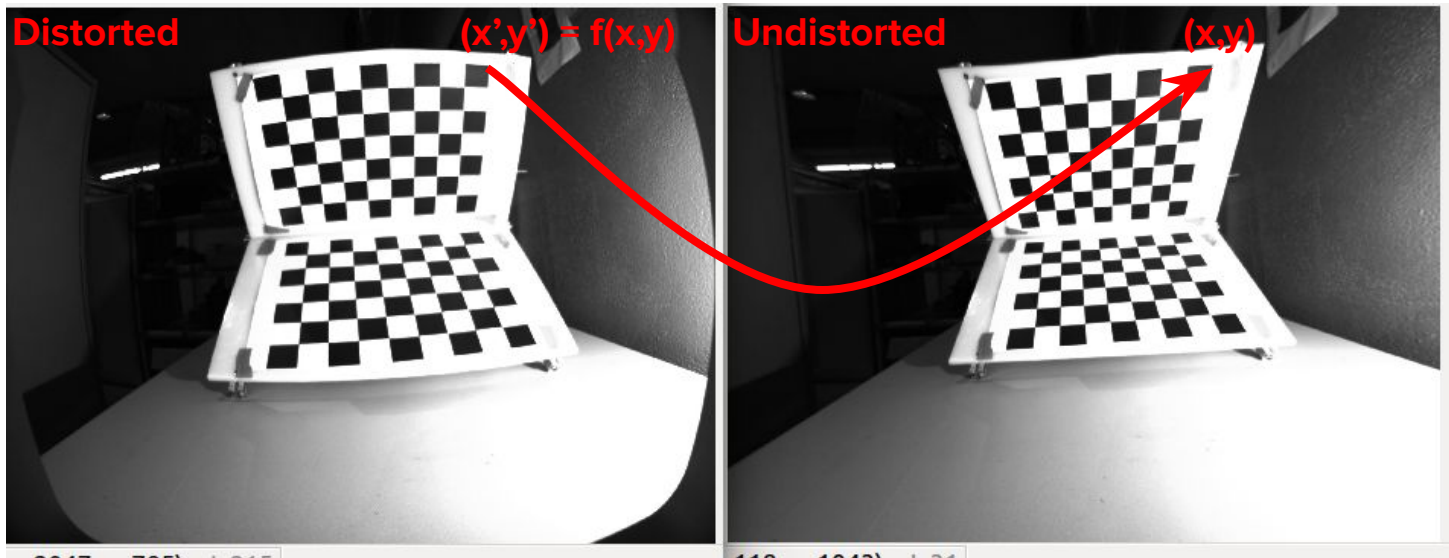
$$I(P) = \frac{1}{4 \cdot 4} (20 \cdot 1 \cdot 1 + 200 \cdot 3 \cdot 1 + 10 \cdot 1 \cdot 3 + 151 \cdot 3 \cdot 3) = \frac{1}{16} (20 + 600 + 30 + 1359) = 125.5625$$

A (integer) division by 16 would be equivalent to truncating the last four binary digits using binary numbers. Despite this lucky circumstance of a denominator that is a power of two, for bilinear interpolation and other tasks as well, we can use scaling factors that allow the deployment of lightweight integer computations to achieve approximated solutions.

Using binary numbers, scaling by power of 2 is the right strategy since we can replace multiplications and divisions with lightweight shifts (or even simple manipulations).

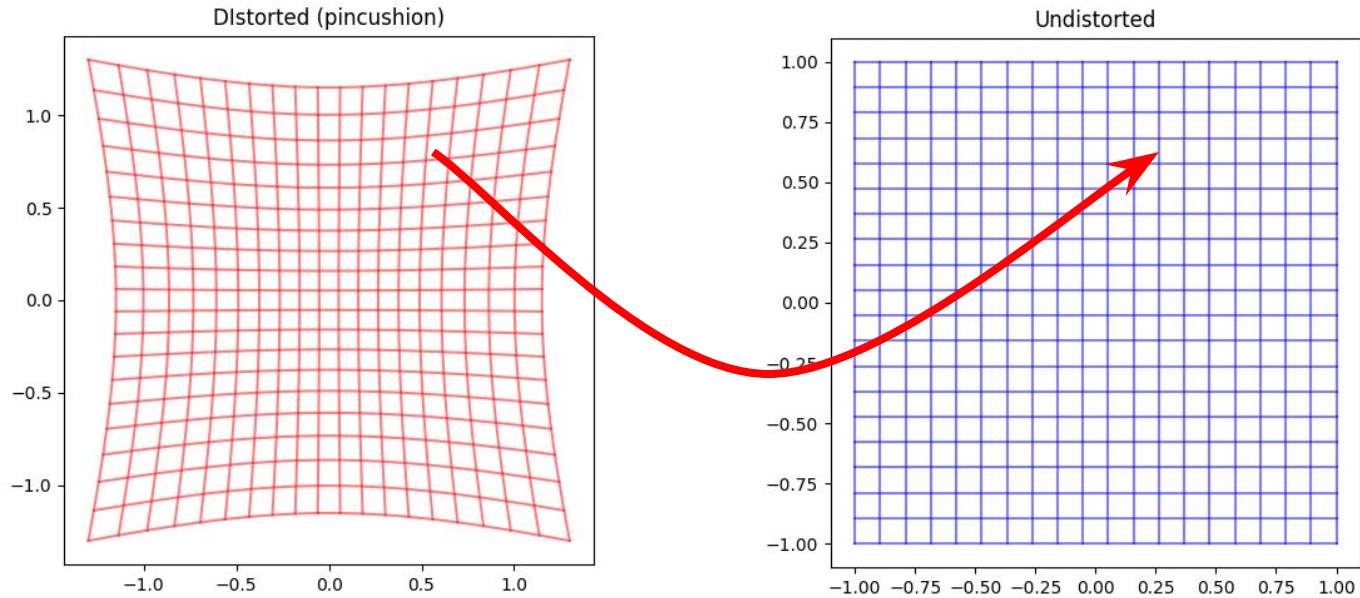
Example: correction of image distortions 1/8

For instance, bilinear interpolation is used to correct lens distortion using a function f that, given the integer coordinates (x,y) of an undistorted point fills its color through bilinear interpolation of four points in proximity of point $(x',y')=f(x,y)$ in the distorted image



Example: correction of image distortions 2/8

The picture below highlights the problem



Example: correction of image distortions 3/8

The target real coordinates (x',y') can be expressed as the sum of an integer offset (eg, the position of the bottom-left point A) and a positive increment $[0,<1]$ from that anchor point:

$$f(x,y) = (x',y') = (Lx',Ly') + (\Delta x',\Delta y')$$

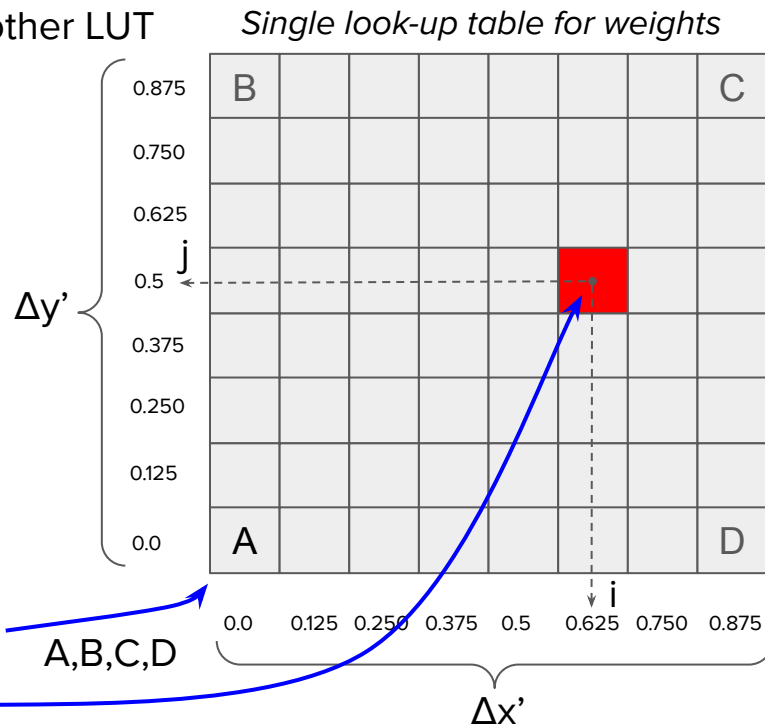
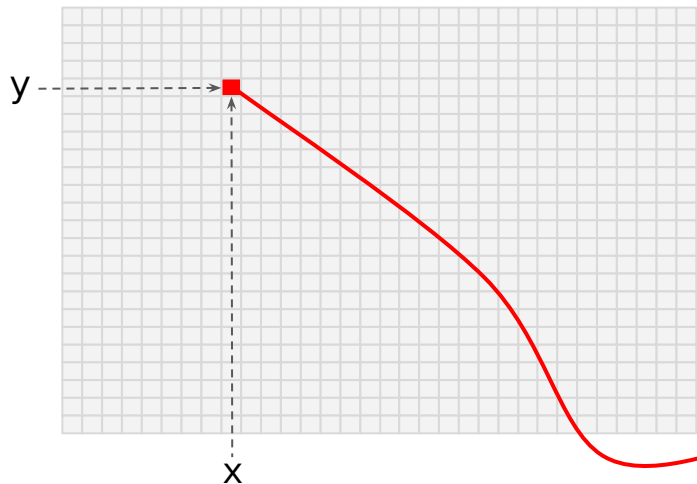
Hence, to reduce computation at runtime, for each pixel (x,y) , we can store (Lx',Ly') and two integer indexes to retrieve $(\Delta x',\Delta y')$ in an appropriate form to enable integer computations – that is, the fixed point coefficients to perform bilinear interpolation. The accuracy relates to the discretization of the interval $[0,1]$ used to encode $\Delta x'$ and $\Delta y'$ (the more finely discretized the interval, the better the accuracy).

However, it is worth highlighting that the size of the LUT to obtain is as large as the input image (2x integer values for each index in the table)

Example: correction of image distortions 4/8

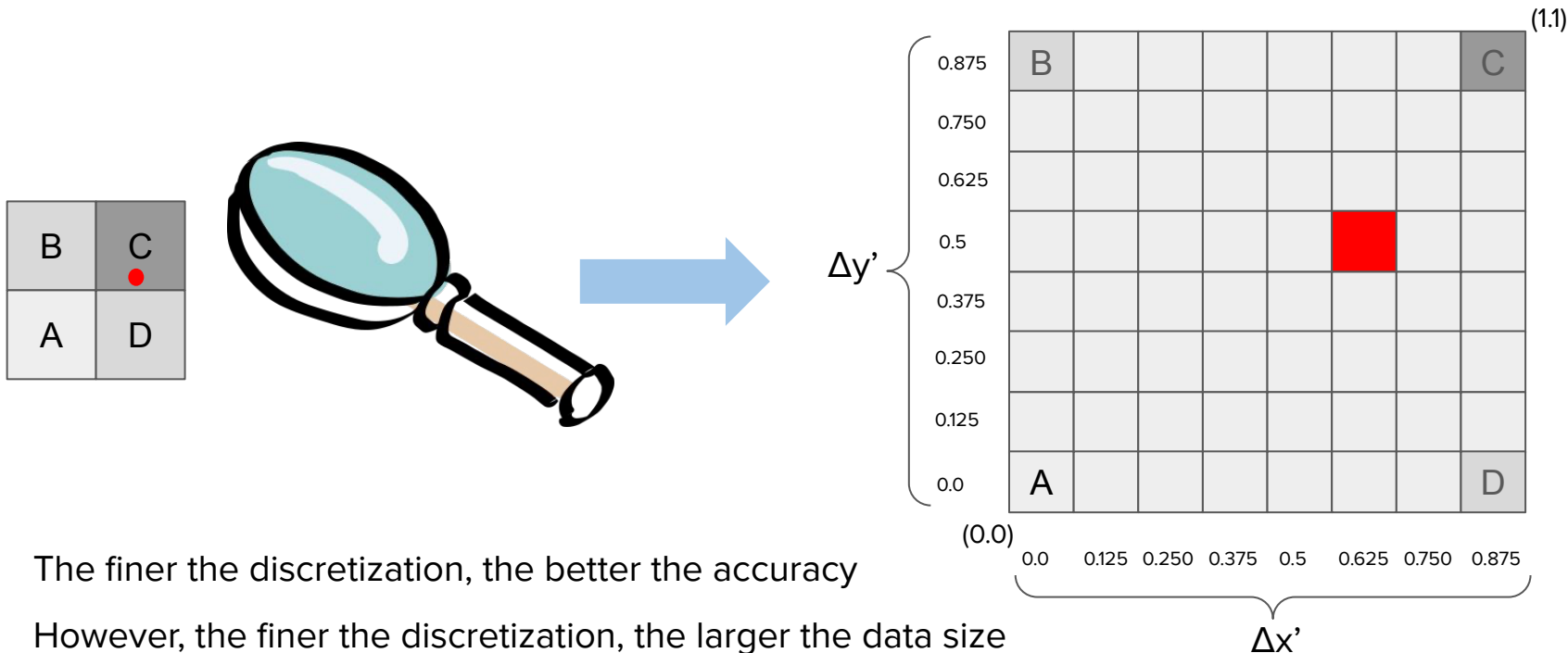
Given the coordinates (x,y) of the point in the target image we want to fill in, accessing a first LUT allows for retrieving Lx',Ly' to find A, B, C, D and indexes (i,j) to retrieve (fractional) weights for that specific position in the other LUT

Global and **HUGE LUT** for Lx',Ly'
and **small LUT** for indexes (i,j)



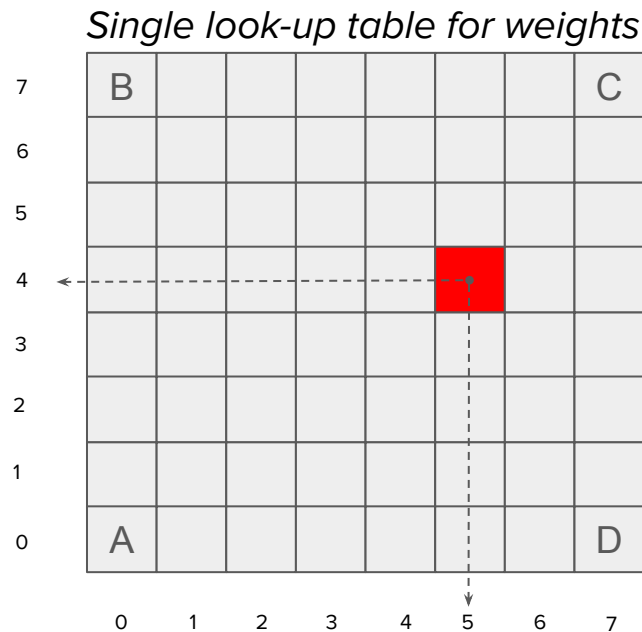
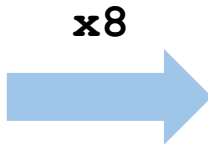
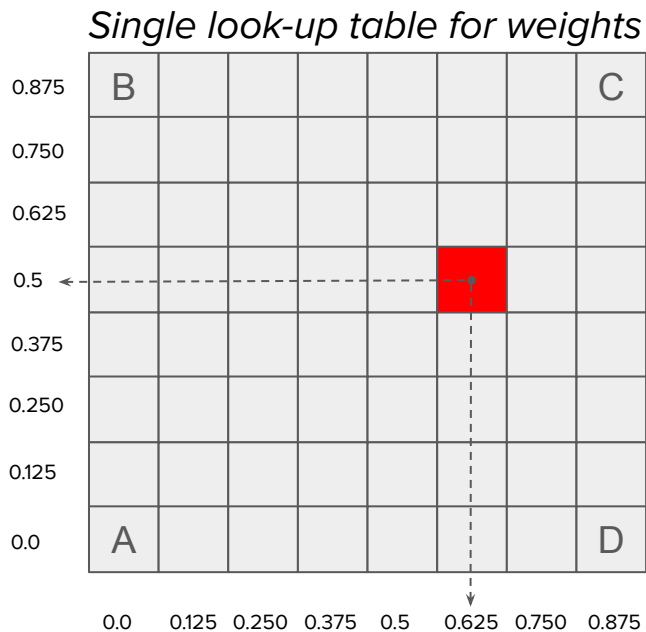
Example: correction of image distortions 5/8

By discretizing equally (eg, 8 portions) the intervals $[0,1]$ we can perform bilinear interpolation with offsets $\Delta x'$ and $\Delta y'$ on a regular grid, as follows:



Example: correction of image distortions 6/8

The coordinates x' and y' are fractional numbers. However, we can scale them up by a factor power of 2 and possibly do the same for denominator needed to compute $I(P)$ to perform only integer multiplication and truncations (or shifts).

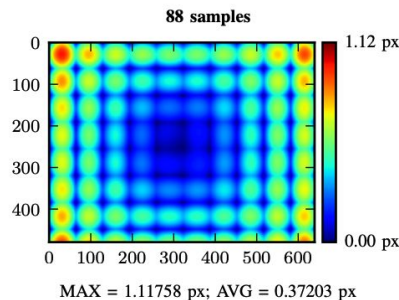


Example: correction of image distortions 7/8

- After scaling up by a factor 2^3 , Q_{00} , Q_{01} , Q_{11} and Q_{10} are all integer values and they can be precomputed offline and stored in the single LUT for weights to speed up computations at run-time
- Moreover, since $I(A)$, $I(B)$, $I(C)$ and $I(D)$ are integer values in the domain $[255,0]$, the numerators require only integer multiplications (ie, $I \cdot Q$)
- The denominator is a power of two ($1 \cdot 1 \cdot 2^3$), and thus the division can be replaced by a truncation – or a right shift of three positions if the truncation is not feasible or more expensive in the target architecture
- Consequently, the overall $I(P)$ computation can be carried out with pure integer operations that are much less demanding than operations with real numbers
- The result is an approximation but it can be tailored to the specific requirements increasing granularity (ie, discretizing more finely the interval)

Example: correction of image distortions 8/8

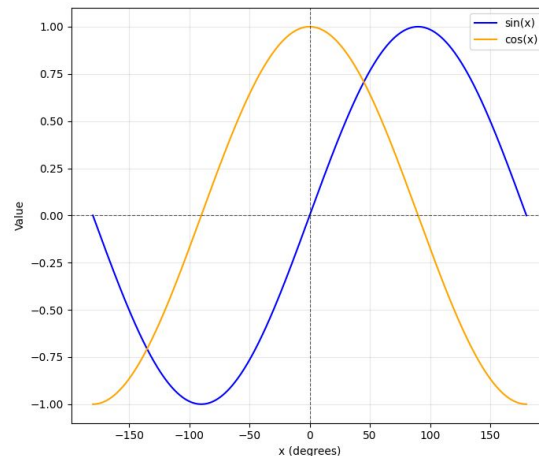
- Since the lens distortion varies across the image position, the main problem with the previous method is the size of the first LUT used to retrieve (Lx', Ly') coordinates
- However, solutions exist to deal with this issue; for instance, the first LUT could be subsampled, and the actual position computed through an additional interpolation process to deal with the radial distortion that is larger near image borders ([link](#))



- Other solutions might approximate the mapping function, although this is not trivial in this specific case. Nonetheless, as discussed next, this strategy might be more effective in other cases, such as when facing computations involving sine and cosine functions

Computing trigonometric functions sine and cosine

- The sine and cosine of an angle is a numeric value ranging from -1 to 1
- Of course, we know that they have a periodic nature that allows us to restrict their computation domain to $[-180^{\circ}, 180^{\circ}]$
- Moreover, they expose additional symmetries that narrow the actual computation domain to $[0^{\circ}, 90^{\circ}]$
- This evidence might be helpful in some cases but does not seem a breakthrough if our purpose is to determine $\sin(x)$ and $\cos(x)$ by deploying only integer data and simple, lightweight computations such as basic arithmetic and logic operations
- How can we achieve that?



Sine and cosine computation: Taylor expansion

- The Taylor series enables sine and cosine approximation with a polynomial; for instance, $\sin(x)$ near $x=0$ is (polynomial of order 9):

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} + O(x^{11})$$

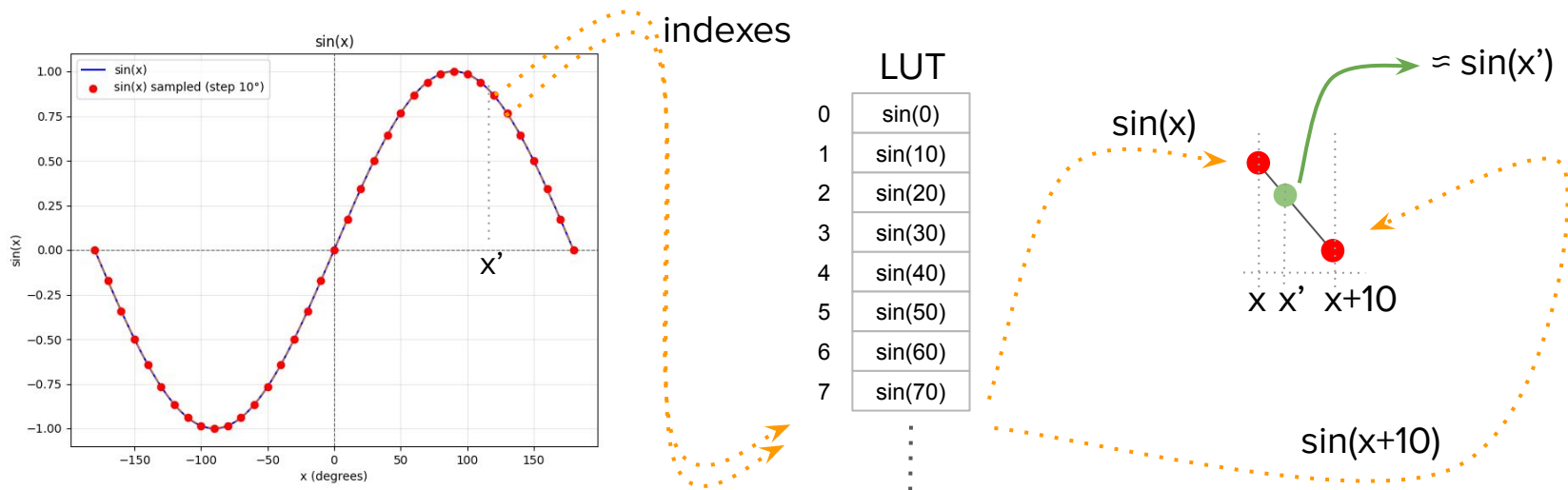
(Taylor series)

- It provides a pretty accurate approximation, It provides a pretty accurate approximation, getting better by increasing the polynomial order
- Unfortunately, even exploiting the reduced domain $[0^0, 90^0]$, this formulation intrinsically involves computations with real numerical values and doesn't seem a viable solution for our purposes, even switching to fixed-point data and arithmetic
- Therefore, let's consider now two potential strategies best suited to our aim

<https://www.wolframalpha.com/input?i=series+of+sin%28x%29+to+order+10+at+x+%3D+0>

Sine and cosine computation: LUT and interpolation

- A first alternative approach consists of approximating sine or cosine values leveraging pre-computed sample values stored in a Look-Up-Table (LUT), for instance, through linear interpolation and fixed-point arithmetic
- This strategy is feasible, but the size of the LUT increases with the number of samples, and scatter access to large tables might not be optimal (eg, caches)



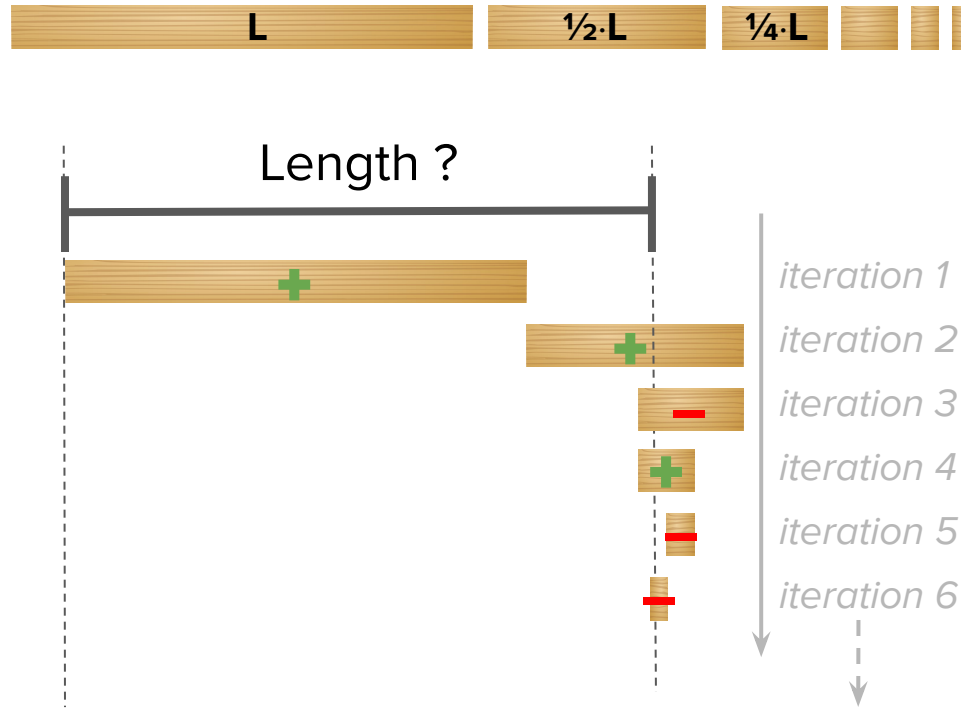
CORDIC

- CORDIC is the acronym for COordinate Rotation DIgital Computer, an algorithm originally proposed by J. E. Volder in 1959
- CORDIC allows computing mathematical functions (eg, sin, cos, tan, multiplications, divisions) using only basic arithmetic and logic operations with integers
- Its extensions can face additional operations like sqrt, $1/x$, and many others
- Iterative approach: the higher the iterations, the better the accuracy
- Useful when floating-point is not available (eg, microcontrollers) or demanding in terms of hardware resources like with FPGAs (Field Programmable Gate Array)
- Notable historical deployments: NASA Apollo mission to the moon and past Intel math co-processors (eg, math co-processor 8087, see link below)

<https://www.righto.com/2020/05/extracting-rom-constants-from-8087-math.html>

CORDIC principle 1/2

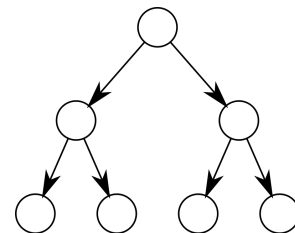
Given some objects of **known length**, scaling by $\frac{1}{2}$; how to compute the segment length?



iteration 1
iteration 2
iteration 3
iteration 4
iteration 5
iteration 6

CORDIC principle 2/2

The CORDIC approximation acts as a binary tree, and we use the same principle to encode an integer number between 0 and 255 with 8 bits.



For instance, consider 243; stop when the number is equal:

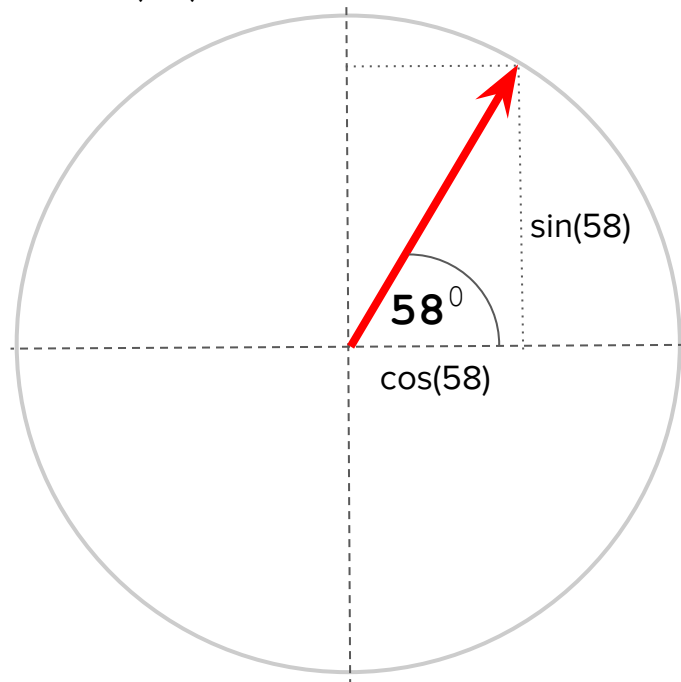
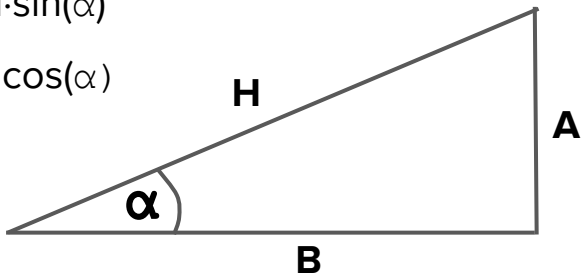
128	is it greater than 243? NO	1	
$128 + 64 = 192$	is it greater than 243? NO	1	
$128 + 64 + 32 = 224$	is it greater than 243? NO	1	
$128 + 64 + 32 + 16 = 240$	is it greater than 243? NO	1	
$128 + 64 + 32 + 16 + 8 = 248$	is it greater than 243? YES	0	
$128 + 64 + 32 + 16 + 8 - 4 = 244$	is it greater than 243? YES	0	
$128 + 64 + 32 + 16 + 8 - 4 - 2 = 242$	is it greater than 243? NO	1	
$128 + 64 + 32 + 16 + 8 - 4 - 2 + 1 = 243$	is it greater than 243? NO	1	(Stop)

CORDIC: sine and cosine computation

- Let's consider the initial problem: compute the sine and cosine at a given angle, restricting the domain to the first quadrant.
- For instance, how can we compute $\sin(58)$ and $\cos(58)$?
- Recall that:

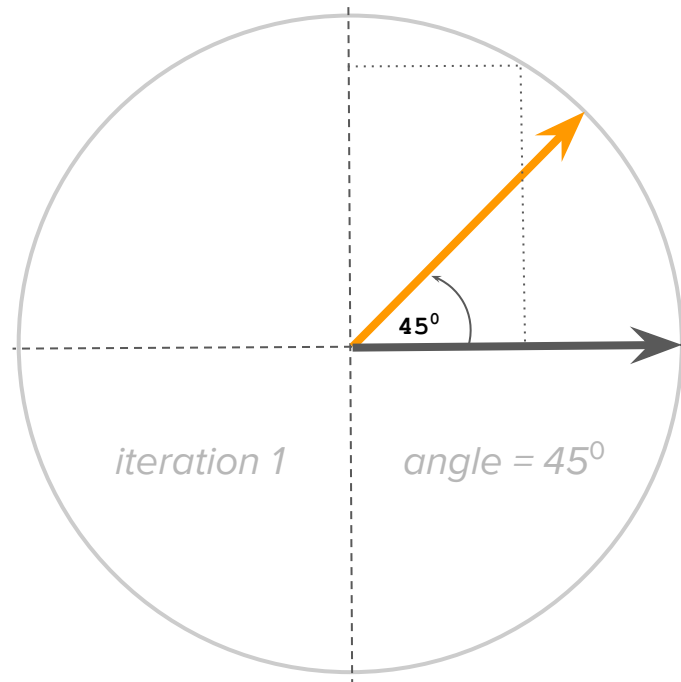
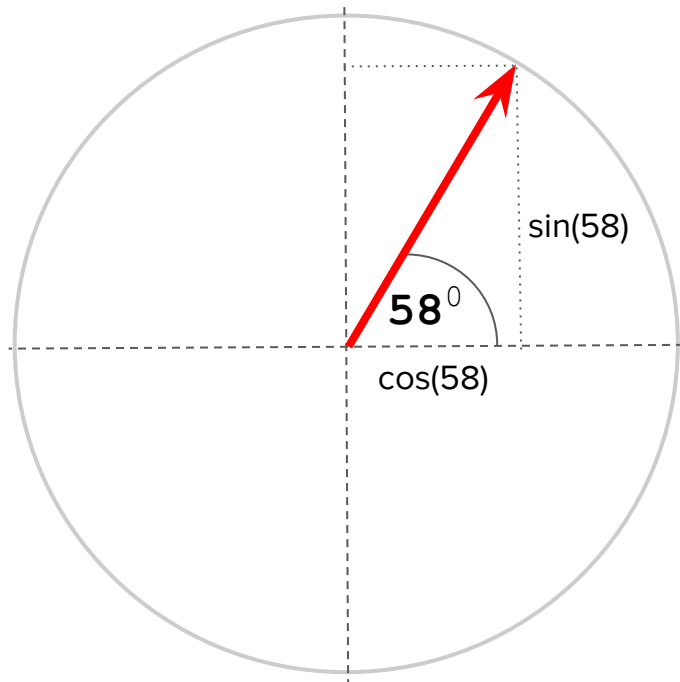
$$A = H \cdot \sin(\alpha)$$

$$B = H \cos(\alpha)$$



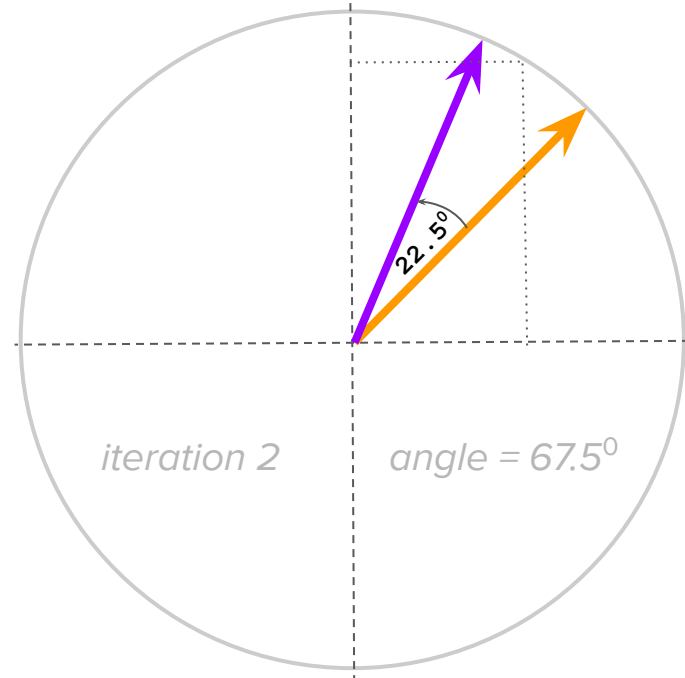
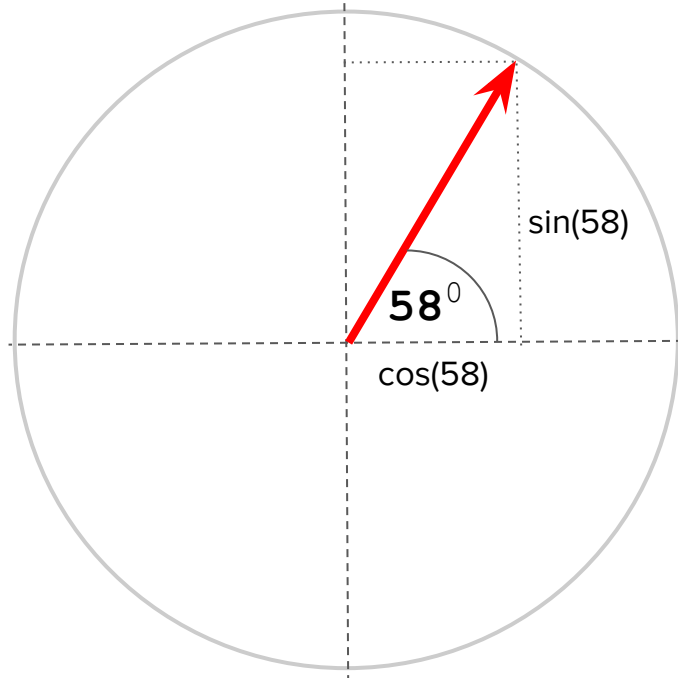
CORDIC: sine and cosine computation

We can compute $\sin(58^\circ)$ and $\cos(58^\circ)$ approximating them through an iterative approach using, for instance, basis angles such as 45° , 22.5° , 11.25° , ...



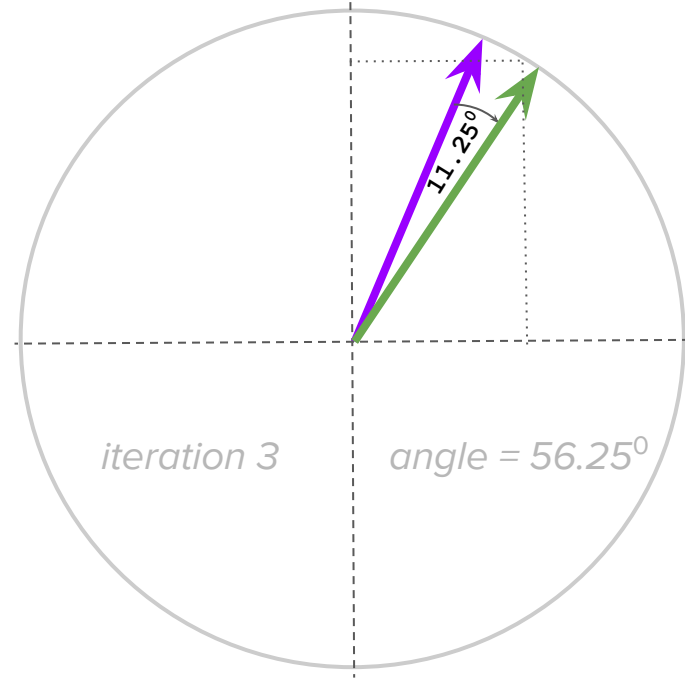
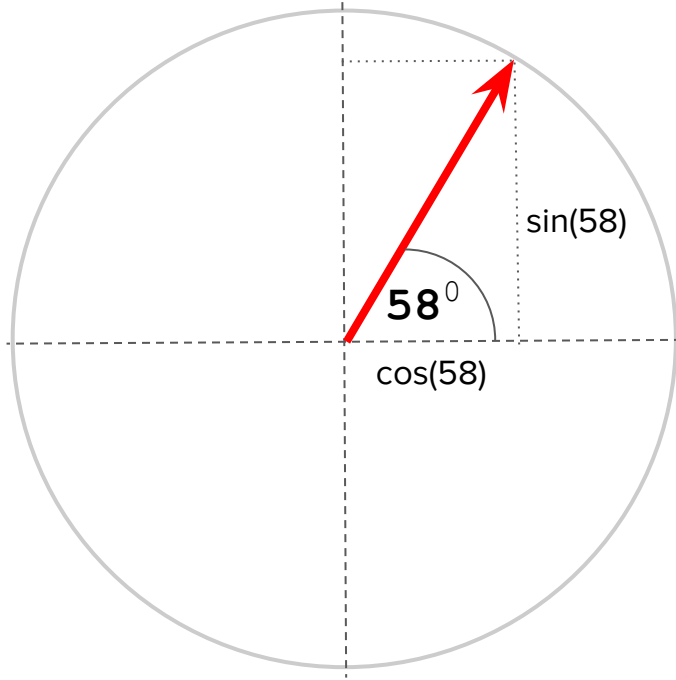
CORDIC: sine and cosine computation

- 45° is smaller than 58° ; add next basis 22.5° to 45°



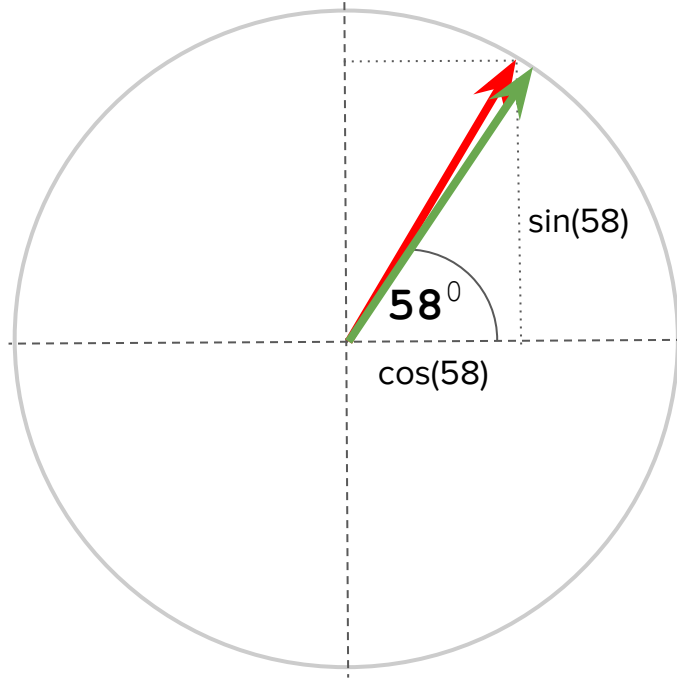
CORDIC: sine and cosine computation

- 67.5° is greater than 58° ; subtract next basis 11.25° to 67.5°



CORDIC: sine and cosine computation

- Using just three iterations, we get: $45^\circ + 22.5^\circ - 11.25^\circ = 56.25^\circ \approx 58^\circ$
- The higher the number of iterations, the better the angle approximation



CORDIC: sine and cosine computation

Recalling that:

$$\sin(a+b) = \sin(a) \cdot \cos(b) + \cos(a) \cdot \sin(b)$$

$$\cos(a+b) = \cos(a) \cdot \cos(b) - \sin(a) \cdot \sin(b)$$

$$\sin(a-b) = \sin(a) \cdot \cos(b) - \cos(a) \cdot \sin(b)$$

$$\cos(a-b) = \cos(a) \cdot \cos(b) + \sin(a) \cdot \sin(b)$$

and, by pre-computing/storing sine and cosine at the known angles, we can write:

$$\sin(45^\circ) = \sin(0^\circ + 45^\circ) = \sin(0^\circ) \cdot \cos(45^\circ) + \cos(0^\circ) \cdot \sin(45^\circ)$$

$$\cos(45^\circ) = \cos(0^\circ + 45^\circ) = \cos(0^\circ) \cdot \cos(45^\circ) - \sin(0^\circ) \cdot \sin(45^\circ)$$

$$\sin(67.5^\circ) = \sin(45^\circ + 22.5^\circ) = \sin(45^\circ) \cdot \cos(22.5^\circ) + \cos(45^\circ) \cdot \sin(22.5^\circ)$$

$$\cos(67.5^\circ) = \cos(45^\circ + 22.5^\circ) = \cos(45^\circ) \cdot \cos(22.5^\circ) - \sin(45^\circ) \cdot \sin(22.5^\circ)$$

$$\sin(56.25^\circ) = \sin(67.5^\circ - 11.25^\circ) = \sin(67.5^\circ) \cdot \cos(11.25^\circ) - \cos(67.5^\circ) \cdot \sin(11.25^\circ)$$

$$\cos(56.25^\circ) = \cos(67.5^\circ - 11.25^\circ) = \cos(67.5^\circ) \cdot \cos(11.25^\circ) + \sin(67.5^\circ) \cdot \sin(11.25^\circ)$$

Since these computations involves only multiplications of sine and cosine terms at known angles, we could easily obtain both values at 56.25° , approximation of the actual angle 58°

CORDIC: sine and cosine computation

However, the previous formulation involves expensive multiplications and doesn't seem a significant step forward. Moreover, although it is a minor issue, it needs two LUTs for sine and cosine at known angles.

Despite these facts, we can rearrange previous equations as follows:

$$\sin(45^\circ) = \cos(45^\circ) \cdot (1 \cdot \sin(0^\circ) + \cos(0^\circ) \cdot \sin(45^\circ) / \cos(45^\circ)) = \cos(45^\circ) \cdot (1 \cdot \sin(0^\circ) + \cos(0^\circ) \cdot \tan(45^\circ))$$

$$\cos(45^\circ) = \cos(45^\circ) \cdot (1 \cdot \cos(0^\circ) - \sin(0^\circ) \cdot \sin(45^\circ) / \cos(45^\circ)) = \cos(45^\circ) \cdot (1 \cdot \cos(0^\circ) - \sin(0^\circ) \cdot \tan(45^\circ))$$

and similarly, for other terms:

$$\sin(67.5^\circ) = \cos(22.5^\circ) \cdot (\sin(45^\circ) + \cos(45^\circ) \cdot \tan(22.5^\circ))$$

$$\cos(67.5^\circ) = \cos(22.5^\circ) \cdot (\cos(45^\circ) - \sin(45^\circ) \cdot \tan(22.5^\circ))$$

$$\sin(56.25^\circ) = \cos(11.25^\circ) \cdot (\sin(67.5^\circ) - \cos(67.5^\circ) \cdot \tan(11.25^\circ))$$

$$\cos(56.25^\circ) = \cos(11.25^\circ) \cdot (\cos(67.5^\circ) + \sin(67.5^\circ) \cdot \tan(11.25^\circ))$$

CORDIC: sine and cosine computation

Even with this new formulation, it's hard to notice a meaningful improvement, and all multiplications remain. How can we move forward?

Here comes the CORDIC intuition, replacing previous basis angles, such as 22.5° , 11.25° and so on, with computational-friendly values (eg, 26.56° and 14.04°) exploiting the evidence that:

$$\tan(45.00^\circ) = 1$$

$$\tan(26.56^\circ) \approx \frac{1}{2}$$

$$\tan(14.04^\circ) \approx \frac{1}{4}$$

$$\tan(7.12^\circ) \approx \frac{1}{8}$$

$$\tan(3.58^\circ) \approx \frac{1}{16}$$

.....

Being $\tan(x)$ very close to a power of two for those angles, we can greatly simplify computations

CORDIC: sine and cosine computation

To replace demanding multiplications by $\tan(x)$ with lightweight shifts, at first, we must approximate the angle (58°) as a combination of CORDIC-friendly values (ie, $58^\circ \approx 45^\circ + 26.56^\circ - 14.04^\circ = 57.52^\circ$), obtaining:

$$\sin(45^\circ) = \cos(45^\circ) \cdot (1 \cdot \sin(0^\circ) + \cos(0^\circ) \cdot \tan(45^\circ)) = \cos(45^\circ) \cdot (1 \cdot \sin(0^\circ) + \cos(0^\circ) \cdot 1) = \cos(45^\circ)$$

$$\cos(45^\circ) = \cos(45^\circ) \cdot (1 \cdot \cos(0^\circ) - \sin(0^\circ) \cdot \tan(45^\circ)) = \cos(45^\circ) \cdot (1 \cdot \cos(0^\circ) - \sin(0^\circ) \cdot 1) = \cos(45^\circ)$$

$$\begin{aligned}\sin(71.56^\circ) &= \cos(26.56^\circ) \cdot (\sin(45^\circ) + \cos(45^\circ) \cdot \tan(26.56^\circ)) = \cos(26.56^\circ) \cdot (\sin(45^\circ) + \cos(45^\circ) \cdot 2^{-1}) = \\ &= \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot (1 + 2^{-1})\end{aligned}$$

$$\begin{aligned}\cos(71.56^\circ) &= \cos(26.56^\circ) \cdot (\cos(45^\circ) - \sin(45^\circ) \cdot \tan(26.56^\circ)) = \cos(26.56^\circ) \cdot (\cos(45^\circ) - \sin(45^\circ) \cdot 2^{-1}) = \\ &= \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot (1 - 2^{-1})\end{aligned}$$

$$\begin{aligned}\sin(57.52^\circ) &= \cos(14.04^\circ) \cdot (\sin(71.56^\circ) - \cos(71.56^\circ) \cdot \tan(14.04^\circ)) = \cos(14.04^\circ) \cdot (\sin(71.56^\circ) - \cos(71.56^\circ) \cdot 2^{-2}) = \\ &= \cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot ((1 + 2^{-1}) - (1 - 2^{-1}) \cdot 2^{-2}) = \cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot (1 + 2^{-1} - 2^{-2} + 2^{-3})\end{aligned}$$

$$\begin{aligned}\cos(57.52^\circ) &= \cos(14.04^\circ) \cdot (\cos(71.56^\circ) + \sin(71.56^\circ) \cdot \tan(14.04^\circ)) = \cos(14.04^\circ) \cdot (\cos(71.56^\circ) + \sin(71.56^\circ) \cdot 2^{-2}) = \\ &= \cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot ((1 - 2^{-1}) + (1 + 2^{-1}) \cdot 2^{-2}) = \cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot (1 - 2^{-1} + 2^{-2} + 2^{-3})\end{aligned}$$

CORDIC: sine and cosine computation

The previous formulation seems much better, but it still has multiplications:

$$\sin(57.52^\circ) = \cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot (1 + 2^{-1} - 2^{-2} + 2^{-3})$$

$$\cos(57.52^\circ) = \cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ) \cdot (1 - 2^{-1} + 2^{-2} + 2^{-3})$$

- However, both factors are identical and constant when performing a fixed number of iterations. The **$\cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ)$** factor is the **CORDIC gain**, and it is constant once the number of iterations is fixed. However, when early stopping, the CORDIC gain must account only for the actual number of iterations, not considering the cosine of angles not involved in the approximation.
- This/these factor/s can be precomputed and stored somewhere (eg, in a ROM)
- Hence, sine and cosine computations rely on simple shifts, adds, and a single multiplication with the constant precomputed CORDIC gain factor(s).

CORDIC: sine and cosine computation

The picture seems much better now, but there are still two issues:

The scaling factor **$\cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ)$** and $(1 + 2^{-1} - 2^{-2} + 2^{-3})$ are real numbers. That's true, but fixed-point computation comes to the rescue and these computations can be efficiently carried out using integer operations.

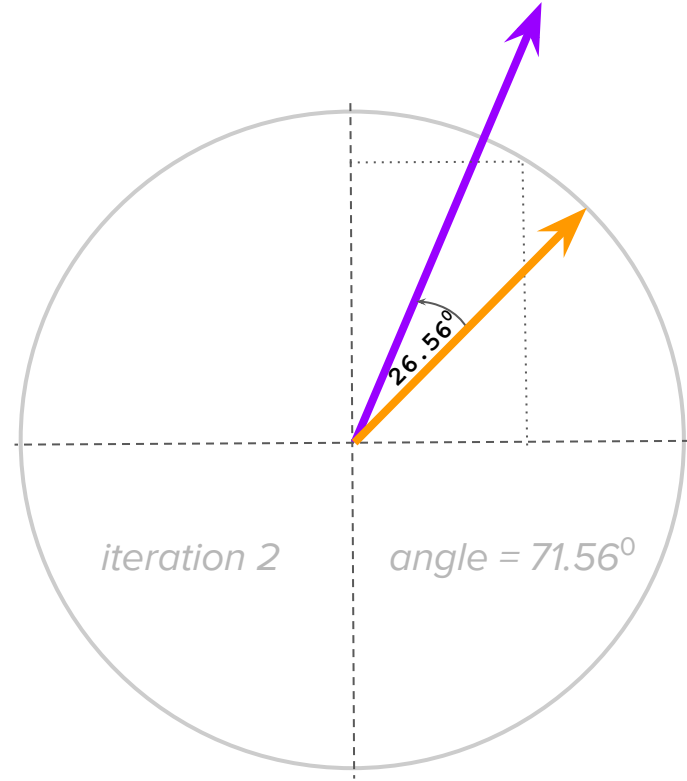
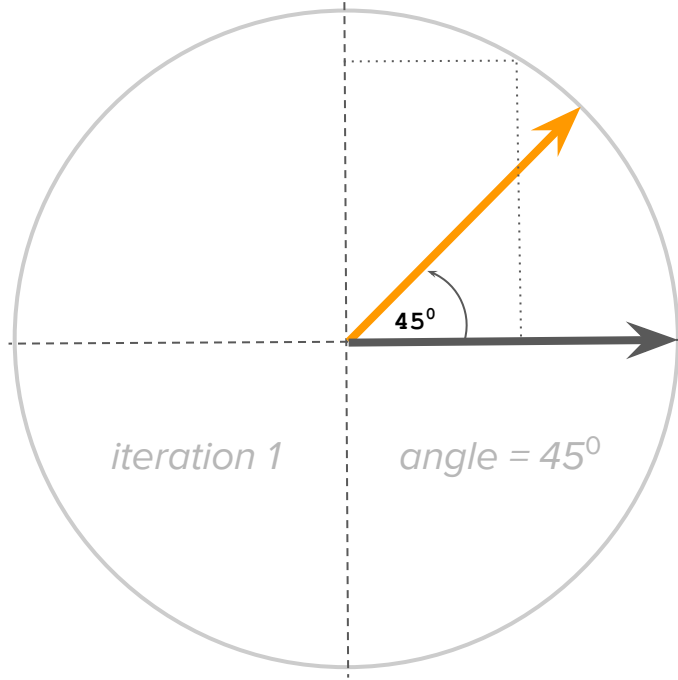
1. There's only a single multiplication, but it might be demanding. If so, even switching to fixed-point, we can provide/use the results up to a scale factor, avoiding the multiplication. In this case, the output wouldn't be the value of the trigonometric function but a scaled version up to the CORDIC gain:

$$\sin(57.52^\circ) / \mathbf{\cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ)}$$

$$\cos(57.52^\circ) / \mathbf{\cos(14.04^\circ) \cdot \cos(26.56^\circ) \cdot \cos(45^\circ)}$$

CORDIC: sine and cosine computation

If we do not normalize with the CORDIC gain, vectors are no longer on the unit circle



Exercise

Exercise

Implement the CORDIC algorithm for sine and cosine in C/C++ and compare its accuracy to conventional C/C++ functions, varying the number of iterations.

Take-home message

Computation with real numeric values is pure abstraction; approximations are always present, even with cumbersome and computationally demanding large floating-point data with a long mantissa

Hence, when facing a new problem, it is crucial to evaluate whether you need floating point encoding to address it:

- If yes, keep their size as small as possible to reduce computational issues, power consumption and footprint, and take care when dealing with them because they hide pitfalls
- Otherwise, it might be better to look for more lightweight data types and computations to deal with real numeric values, as outlined in the previous slides in two practical use cases

References

- [1] David Goldberg, “What every computer scientist should know about floating-point arithmetic”, ACM Computing Surveys (CSUR), Volume 23, Issue 1, 1991
<https://dl.acm.org/doi/pdf/10.1145/103162.103163>
- [2] Jean-Michel Muller , Nicolas Brunie , Florent de Dinechin , Claude-Pierre Jeannerod , Mioara Joldes , Vincent Lefèvre , Guillaume Melquiond , Nathalie Revol , Serge Torre, Handbook of Floating-Point Arithmetic, Springer Birkhäuser 2018
- [3] Jean-Michel Muller, Elementary Functions: Algorithms and Implementation, Springer Birkhäuser 2016