


# REPRÉSENTATION DES NOMBRES RÉELS

À la fin de ce chapitre, je sais :

- ✎ le concept de nombre flottant (signe, mantisse, exposant)
- ✎ expliquer la différence entre simple et double précision
- ✎ calculer une erreur relative et une erreur absolue
- ✎ expliquer le mécanisme d'absorption et ses conséquences

## A Calculs à virgule fixe --> HORS PROGRAMME

La représentation des nombres entiers décrite au chapitre précédent peut servir à encoder des nombres rationnels dont la précision est limitée, des décimaux ou des dyadiques par exemple. On appelle cette représentation à virgule fixe.

 **Vocabulary 1 — Fixed-Point Arithmetics** ↔ En anglais, ce mode de calcul s'énonce Fixed-Point Arithmetics, étant donné l'utilisation du point à la place de la virgule par les anglo-saxons.

■ **Définition 1 — Représentation à virgule fixe** . Il s'agit de représenter un nombre par un entier divisé par un facteur d'échelle choisi, c'est-à-dire  $\frac{a}{b^n}$ ,  $a \in \mathbb{Z}$ ,  $b \in \mathbb{N}$ ,  $n \in \mathbb{N}$ .

On affecte un certain nombre de bits à la **partie entière** et à la **partie fractionnaire** de ce nombre.

La notation standard est la suivante : `fixed<m,n>` est le type d'un nombre à virgule fixe codé sur  $m$  bits pour la partie entière et sur  $n$  bits pour la partie fractionnaire. Le facteur d'échelle vaut donc  $2^n$ .

La base de numération est deux et le facteur d'échelle est une puissance de deux. On peut donc représenter à virgule fixe un sous-ensemble des nombres dyadiques  $\mathcal{D}$ .

■ **Exemple 1 — 7,5 est dyadique** . En base dix, le nombre décimal 7,5 peut être représenté à virgule fixe par le nombre entier 75 et le facteur d'échelle 10, car  $75 = \frac{75}{10}$ .

Pour coder 7,5 en binaire, on observe que  $7,5 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} = 4 + 2 + 1 + 0,5 = 7,5$ . C'est un nombre dyadique. Pour trouver la partie fractionnaire, on procède comme pour la partie entière, par divisions successives : la partie entière du résultat contient

le bit recherché. Dans notre cas :  $\lfloor 0,5/2^{-1} \rfloor = \lfloor 0,5 \times 2 \rfloor = 1$ . Tous les autres bits sont nuls.

On peut alors choisir de représenter 7,5 avec une type `fixed<3, 1>` et l'écrire :  $111,1_2$ . En machine, il peut être stocké sur 4 bits  $1111_2 = 15_{10}$  et le facteur d'échelle vaut 2. On vérifie que  $15/2 = 7,5$ .

■ **Exemple 2 — 3,14 est décimal mais pas dyadique.** ...  $\pi$  est un nombre transcendant. Mais la valeur approchée 3,14 en base dix est un nombre décimal. Il peut être représenté à virgule fixe par le nombre entier 314 et le facteur d'échelle 100, car  $3,14 = \frac{314}{100}$ .

3,14 n'est pas un nombre dyadique. Pour coder exactement la valeur 0,14 en binaire, il faudrait une infinité de bits<sup>a</sup> :  $0,14_{10} = 0,001000111101011100001_2$ . Cependant, on remarque que  $3,14 \approx 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-3} + 1 \times 2^{-6} = 3 + 0,125 + 0,015625 = 3,140625$ . Il s'agit là d'une valeur approchée.

On peut alors choisir de représenter 3,14 en valeur approchée avec un type `fixed<2, 6>` et l'écrire :  $11,001001_2$ . Le facteur d'échelle vaut  $2^6$ . Il sera donc codé en machine  $11001001$ . On vérifie que  $11001001_2 / 1000000_2 = 201/64 = 3,140625$

a. Les chiffres surmontés d'une barre horizontale constituent le motif répété à l'infini.

**R** Tout comme en base 10, certains nombres ne peuvent pas être codés en base 2 par un nombre fini de bits. C'est le cas par exemple de  $0,1_{10} = 0,00011001100110011001_2$ .

La représentation à virgule fixe des nombres permet d'utiliser les circuits dédiés à l'arithmétique des nombres entiers pour effectuer des calculs sur des nombres codés en virgule fixe<sup>1</sup>. On n'a donc pas besoin de modifier les architectures des processeurs qui sont capables de calculer sur les entiers pour calculer en arithmétique à virgule fixe. C'est un avantage dans le cadre des systèmes embarqués.

## Erreurs d'approximation

L'inconvénient de l'arithmétique à virgule fixe est principalement le lien entre la précision obtenue et l'ordre de grandeur des entiers représentés.

**Théorème 1 — Majorant de l'erreur absolue.** L'erreur absolue d'un nombre  $v$  et sa valeur approchée  $\tilde{v}$  encodé par un `fixed<m, n>` est majorée par une constante. On a :

$$|\varepsilon_a| = |\tilde{v} - v| < 2^{-n-1} \quad (1)$$

*Démonstration.* Soit  $a$  une nombre codé en `fixed<m, n>`. Le nombre  $b$  directement supérieur que l'on peut coder est obtenu en ajoutant  $2^{-n}$ , c'est-à-dire en ajoutant 1 au bit de poids faible.  $b - a = 2^{-n}$  : cette différence est la plus petite que l'on puisse coder. Au pire, le nombre réel  $v$  à encoder se situe au milieu de l'intervalle  $[a, b]$  et donc l'erreur commise sera la moitié de la largeur de cet intervalle :  $\frac{1}{2}(b - a) = 2^{-n-1}$ . Cet valeur est un majorant de l'erreur absolue. ■

1. On code souvent une valeur approchée du nombre décimal ou dyadique.

■ **Exemple 3 — Erreur absolue, ordre de grandeur et virgule fixe.** Imaginons que l'on utilise des entiers codés à virgule fixe par un type `fixed<6,3>` non signé. La plage des valeurs que l'on peut atteindre va de 0 à  $63,875_{10} = 111111,111_2$ . On peut tout d'abord noter une limitation : on ne pourra coder des nombres supérieurs à 63,875.

De plus, entre chaque nombre ainsi représenté, l'incrément minimal est de  $1/2^3 = 0,125$ . L'erreur absolue est majorée par la constante  $2^{-n-1}$ , dans notre cas 0,0625.

**Théorème 2 — Majorant de l'erreur relative.** L'erreur relative entre un nombre réel  $v$  et sa valeur approchée  $\tilde{v}$  encodée par un `fixed<m,n>` est majorée par un terme dépendant de  $v$ . On a :

$$|\epsilon_r| = \frac{|\tilde{v} - v|}{|v|} \leq \frac{2^{-n-1}}{|v|} \quad (2)$$

*Démonstration.* Ce résultat est une conséquence de la proposition précédente. ■

■ **Exemple 4 — Erreur relative en `fixed<5,3>`.** L'erreur relative pour des nombres codés en `fixed<5,3>` non signé est beaucoup plus faible pour les valeurs élevées du nombre.

Sur l'intervalle  $[1, 1,125]$ , tout nombre supérieur ou égal à 1,0625 va être arrondi à 1,125, tout nombre strictement inférieur à 1. L'erreur relative en arrondissant à 1 est majorée par celle commise pour le nombre 1,0625. Elle vaut  $(1 - 1,0625)/1,0625 \simeq -0,059$ . Celle commise en arrondissant à 1,125 est majorée par celle commise pour 1,0625. Elle vaut  $(1,125 - 1,0625)/1,0625 \simeq 0,059$ .

Ce résultat est conforme à la proposition précédente et peut s'écrire :

$$\max |\epsilon_r(a \in [1, 1,125])| < 0,059 \simeq \frac{2^{-4}}{1,125}$$

En procédant de même avec l'intervalle  $[63, 63,125]$ , on trouve que :

$$\max |\epsilon_r(a \in [63, 63,125])| < 0,001 \simeq \frac{2^{-4}}{63}$$

Cela signifie que l'erreur relative d'arrondi commise sur un nombre  $a$  compris entre  $[63, 63,125]$  peut être jusqu'à soixante trois fois plus faible que pour un nombre  $a$  appartenant à l'intervalle  $[1, 1,125]$ .

Même si l'arithmétique à virgule fixe est très utilisée dans le calcul embarqué, elle nécessite un savoir faire particulier, car il faut gérer les dépassements correctement. En outre, **une erreur relative variable sur la plage de données manipulée par un algorithme peut être rédhibitoire pour certaines applications.** Heureusement, les nombres flottants permettent de dépasser ces limites.

## B Représentation à virgule flottante

### a Cas général

La représentation d'un nombre à virgule flottante ne fixe pas un nombre exact de bits alloués à la partie fractionnaire. Au contraire, la position de la virgule dépend d'un exposant. IEEE 754 est la norme utilisée pour cette représentation par la plupart des systèmes.

■ **Définition 2 — Notation scientifique.** Exprimer un nombre  $a$  selon la notation scientifique c'est l'écrire sous la forme :

$$a = \pm m \times 10^e \quad (3)$$

où  $m \in [1, 10[ \subset \mathbb{D}$  est un nombre décimal nommé *mantisse* et  $e \in \mathbb{Z}$  l'exposant.

■ **Définition 3 — Représentation normalisé IEEE 754 d'un nombre binaire.** Pour représenter un nombre binaire en utilisant norme IEEE 754, il est nécessaire d'écrire le nombre à représenter sous la forme :  $\pm 1, M.2^e$ .

- On appelle  $M$  la pseudo-mantisse. Le 1 au début du nombre n'est pas codé en machine, il est implicite. On l'appelle le bit caché.
- $\pm$ , le signe de la mantisse est codée par  $s$  qui vaut 0 ou 1.
- $e$  est l'exposant qui va être codé par un exposant biaisé  $E$ . Le biais dépend du format choisi, simple ou double précision :  $c$  est codé sur  $n_E = 8$  bits ou  $n_E = 11$  bits. En simple précision, il vaut  $2^{n_E-1} - 1 = 2^7 - 1 = 127$ . En double précision,  $2^{n_E-1} - 1 = 2^{10} - 1 = 1023$ .



FIGURE 1 – Schématisation du format IEEE 754.

■ **Définition 4 — Nombre IEEE 754 normalisé.** Un nombre IEEE 754 est normalisé lorsque le bit caché de la mantisse est 1 et lorsque l'exposant biaisé appartient à  $\llbracket 1, 2^{n_E} - 2 \rrbracket$ , où  $n_E$  est le nombre de bits qui code l'exposant biaisé  $E$ .

Si le biais vaut  $b$ , cela signifie qu'un nombre normalisé correspond à un exposant  $e \in \llbracket 1 - b, 2^{n_E} - 2 - b \rrbracket$ . En simple précision,  $b = 127$  et cela se traduit par  $e \in \llbracket -126, 127 \rrbracket$ . En double précision,  $b = 1023$  et cela se traduit par  $e \in \llbracket -1022, 1023 \rrbracket$ .

La norme IEEE 754 définit plusieurs formats qui garantissent des précisions différentes. Dans tous les cas,  $s$  est codé sur un seul bit.

**Simple précision - 32 bits**  $M$  est codée sur 23 bits,  $E$  sur 8 bits.

**Double précision - 64 bits**  $M$  est codée sur 52 bits,  $E$  sur 11 bits.

■ **Exemple 5 — Représenter  $39,125_{10}$  par un nombre flottant IEEE 754.** Tout d'abord, il est nécessaire de convertir le nombre en base 2, comme expliqué dans l'exemple 2. On obtient  $39,125_{10} = 100111,001_2$ . Une fois ce résultat obtenu, il est nécessaire d'écrire ce nombre sous la forme  $\pm 1, M \cdot 2^e$ . Cela donne, en binaire,  $1,00111001 \cdot 2^5$ .

Dans cette notation, l'exposant n'est pas biaisé. Pour le représenter en machine, il faut donc le translater et la translation dépend du format choisi, simple ou double précision. Au format simple précision,  $E$  est codé sur 8 bits et on translate de 127. Donc on a  $E = 127 + 5 = 132_{10} = 10000100_2$ .

Le nombre étant positif, le bit de signe  $s$  vaut 0.

La représentation IEEE 754 simple précision de  $39,125_{10}$  est donc  $01000010000111001000000000000000_2$  comme le montre la figure 2.

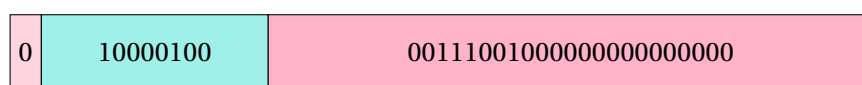


FIGURE 2 – Représentation IEEE 754 simple précision de  $39,125_{10}$ .

**R** L'intérêt de placer l'exposant avant la mantisse et de le biaiser est de faciliter la comparaison de deux nombres flottants. Si les exposants sont différents, il suffit de comparer ces valeurs positives pour connaître le nombre le plus grand.

**R** Pourquoi biaiser l'exposant? --> HORS PROGRAMME Il semble que ce soit pour faire en sorte que l'opération d'inversion du plus petit et du plus grand normalisé ne produise ni dépassement ni sous-dépassement (cf. définitions b).

En simple précision, le plus petit normalisé est  $2^{-126}$ . L'inverse de ce nombre vaut  $2^{126}$  : c'est un nombre normalisé. Le plus grand normalisé est  $(2 - 2^{-23}) \cdot 2^{127}$ . Son inverse arrondi vaut  $2^{-128}$  : ce nombre est dénormalisé mais il ne s'agit pas d'un sous-dépassement.

Un autre biais ne permettrait pas d'obtenir ce résultat.

## b Cas particuliers --> HORS PROGRAMME

■ **Définition 5 — Nombre IEEE 754 dénormalisé.** Si  $n_E$  est le nombre de bits qui code l'exposant biaisé  $E$ , un nombre IEEE 754 est dénormalisé lorsque :


$E = 0$  : Alors l'exposant  $e$  vaut **par convention**  $-2^{n_E-1} + 2$  et le nombre représenté  $0, M \cdot 2^{-2^{n_E-1}+2}$  (noter le 0 au début).

$E = 2^{n_E} - 1$  : la plus grande valeur possible de l'exposant biaisé. Le nombre représente alors des valeurs spéciales comme l'infini ou NaN.

**R** En simple précision, un nombre dénormalisé correspond à un exposant  $e$  non biaisé de  $-126$  commençant par le bit 0 ou de 127.



2. les nombres négatifs supérieurs à  $-2^{-149}$  constituent un sous-dépassement par valeurs négatives,
3. les nombres positifs inférieurs à  $2^{-149}$  constituent un sous-dépassement par valeurs positives,
4. les nombres positifs supérieurs à  $(2 - 2^{-23}) \cdot 2^{127}$  constituent un dépassement par valeurs positives.

 **Vocabulary 2 — Overflow, underflow** ~~~ En anglais on désigne par *overflow* les dépassements et *underflow* les sous-dépassements. On peut ainsi désigner un sous-dépassement par valeurs négatives par *negative underflow*.

**R** Les sous-dépassements n'engendrent qu'une perte de précision dont l'ordre de grandeur est très faible. Les dépassements sont plus problématiques car ils sont la manifestation d'une incapacité à représenter un nombre en flottant.

### Valeurs accessibles aux flottants

Le tableau 1 précise les valeurs extrêmes accessibles à la norme IEEE 754. Le tableau 2 donne un aperçu de la plage de valeurs accessibles via des flottants en simple et double précision.

Précision	Normalisé	Dénormalisé
<b>simple</b>	de $\pm 2^{-126}$ à $(2 - 2^{-23}) \cdot 2^{127}$	de $\pm 2^{-149}$ à $(1 - 2^{-23}) \cdot 2^{-126}$
<b>double</b>	de $\pm 2^{-1022}$ à $(2 - 2^{-52}) \cdot 2^{1023}$	de $\pm 2^{-1074}$ à $(1 - 2^{-52}) \cdot 2^{-1022}$

TABLE 1 – Plus petit (dé)normalisé et plus grand (dé)normalisé.

Précision	Binaire	Décimal
<b>simple</b>	$[-(2 - 2^{-23}) \cdot 2^{127}, (2 - 2^{-23}) \cdot 2^{127}]$	$\sim [-10^{38,53}, 10^{38,53}]$
<b>double</b>	$[-(2 - 2^{-52}) \cdot 2^{1023}, (2 - 2^{-52}) \cdot 2^{1023}]$	$\sim [-10^{308,25}, 10^{308,25}]$

TABLE 2 – Plage de valeurs accessibles aux flottants.

### Opérations spéciales

La norme IEEE 754 définit le résultat des opérations spéciales qui comportent des valeurs dénormalisées. Elles sont décrites dans le tableau 3 et servent à faire en sorte que les résultats classiques sur les limites des fonctions réelles soient respectés par les flottants.

Opération	Résultat
$n / \pm\infty$	0
$n \times \pm\infty$	$\pm\infty$
$\pm nz / 0, nz \neq 0$	$\pm\infty$
$\pm 0 / \pm 0$	NaN
$\infty + \infty$	$\infty$
$\infty - \infty$	NaN
$\pm\infty \times \pm\infty$	$\pm\infty$
$\pm\infty / \pm\infty$	NaN
$\pm\infty \times 0$	NaN
<b>NaN=NaN</b>	Faux

TABLE 3 – Opérations spéciales sur les flottants.

### Synthèse de l'écriture des flottants

Le tableau 4 résume l'écriture normée IEEE754 des flottants et associe à chaque représentation une valeur ou un sens particulier. La figure 3 montre les nombres flottants IEEE754 codables sur l'axe des réels.

Signe	Exposant	Mantisse	Sens / valeur
0	0	0	0
0	0	$M \neq 0$	Nombre positif dénormalisé
0	$0 < E < 2^{n_E} - 1$	$M$	Nombre positif normalisé
0	$2^{n_E} - 1$	0	$+\infty$
0	$2^{n_E} - 1$	$0 < M < 2^{n_M-1} - 1$	SNaN
0	$2^{n_E} - 1$	$M \geq 2^{n_M-1}$	QNaN
1	0	0	-0
1	0	$M \neq 0$	Nombre négatif dénormalisé
1	$0 < E < 2^{n_E} - 1$	$M$	Nombre négatif normalisé
1	$2^{n_E} - 1$	0	$-\infty$

TABLE 4 – Synthèse de l'écriture des flottants.



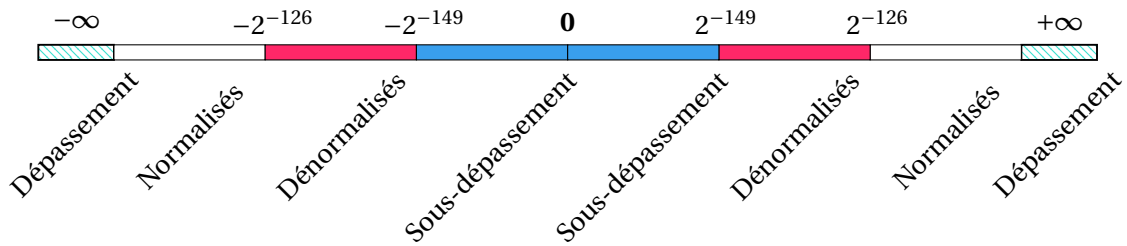


FIGURE 3 – Illustration de la répartition des nombres flottants IEEE754 en simple précision sur la droite des réels.

### c Erreur d'approximation

#### Erreur absolue

Si on cherche à majorer l'erreur absolue que l'on commet lors de la représentation binaire IEEE 754, il suffit de considérer un incrément de 1 sur le bit de poids faible par rapport à la valeur considérée : celui-ci dépend à la fois du nombre de bits avec lequel est codée la mantisse et de l'exposant.

On trouve alors :

$$\epsilon_a < 2^{-n_M} \times 2^e \quad (4)$$

si  $e$  est l'exposant non biaisé et  $n_M$  le nombre de bits qui code la pseudo-mantisse.

Pour l'exemple 5, l'erreur absolue est donc majorée :  $\epsilon_a < 2^{-23} \times 2^5 = 2^{-18} \simeq 3,18.10^{-6}$ .

Dans le cas du nombre :

$$500000,875_{10} = 1111010000100100000,111_2 = 1,111010000100100000111_2 \times 2^{18}$$

On a :  $\epsilon_a < 2^{-23} \times 2^{18} = 2^{-5} \simeq 3,125.10^{-2}$ .

**L'erreur absolue varie donc selon la plage de valeurs considérée.**

#### Erreur relative

L'intérêt de la représentation à virgule flottante est qu'elle garantit une erreur relative constante quelque soit l'ordre de grandeur des nombres représenté. Avec la norme IEEE 754, on peut représenter  $2^{n_M}$  nombres entre chaque puissance de 2.

L'erreur relative entre chaque nombre flottant en simple précision est donc majorée par l'inverse de ce nombre. On a :

$$\epsilon_r < 2^{-23} \quad (5)$$

**et ce, quelque soit l'exposant.**

**R**  $\epsilon_r < 2^{-23} \simeq \frac{10^{-6}}{8}$ . Cela nous garantit 6 chiffres significatifs en base 10.

En double précision,  $\epsilon_r < 2^{-52} \simeq \frac{10^{-15}}{4,5}$ . IEEE754 nous garantit donc 15 chiffres significatifs en base 10.

### Modes d'arrondi --> HORS PROGRAMME

La norme IEEE 754 définit plusieurs modes pour arrondir les résultats des calculs si ceux-ci ne sont pas exacts. Le résultat correct se trouve la plupart du temps entre deux valeurs représentable par la norme. Il faut en choisir un. On utilise alors une des méthodes suivantes :

**arrondir au plus proche** on choisit de prendre le résultat représentable le plus proche. Si le résultat correct est exactement au milieu de l'intervalle des représentables, on choisit le résultat dont la mantisse se termine par 0 et on parle d'arrondi pair. C'est le mode par défaut. On note qu'il est différent de celui adopté classiquement pour le calcul sur les décimaux.

**arrondir au dessus** on choisit le résultat représentable le plus grand, éventuellement infini ou zéro.

**arrondir au dessous** on choisit le résultat représentable le plus petit, éventuellement infini ou zéro.

**arrondir en troncant** on choisit le résultat représentable le plus proche de zéro dans tous les cas.

**Le mode par défaut est arrondir au plus proche.**

## C Calcul avec les flottants --> HORS PROGRAMME

■ **Exemple 7 — Addition en simple précision.** Considérons l'opération  $(0,1 + 0,2) - 0,3$  en simple précision.

Tout d'abord, il faut remarquer que ni 0,1 ni 0,2 ni 0,3 ne sont des nombres dyadiques. Cela signifie que leur représentation en machine ne peut être qu'une approximation. Le résultat d'une opération avec ces opérandes est donc toujours une valeur approchée...

La valeur  $0,1_{10}$  en binaire fait apparaître le premier 1 à la quatrième décimale. Donc l'exposant sera  $-4$ . De plus,  $0,1_{10}$  fait l'objet d'un arrondi au plus près en binaire. En l'occurrence, c'est la valeur supérieure qui est plus proche car le milieu de l'intervalle  $[0,1]$  est 0,1.

Pour aboutir à ce résultat, on utilise les trois bits GRS après le dernier bit de poids faible.

**On nomme ces trois bits GRS pour Guard, Round, Sticky.** Dans ce cas, ils valent 110 et  $110 > 100$ . Par conséquent, en mode par défaut, on arrondit à la valeur supérieure et on

obtient :

$$1.10011001100110011001100\dots \quad GRS \quad (6)$$

$$1.10011001100110011001100.2^{-4} \quad 110 \quad (7)$$

$$0,1_{10} \mapsto 1.10011001100110011001101.2^{-4} \quad (8)$$

Il est de même pour la valeur  $0,2_{10}$ , mais avec un exposant valant  $-3$  :

$$0,2_{10} \mapsto 1.10011001100110011001101.2^{-3} \quad (9)$$

L'addition de  $0,1 + 0,2$  nécessite tout d'abord la mise à l'échelle de  $0,1$  par rapport à  $0,2$  : pour les additionner, on les représente avec le même exposant en l'occurrence  $-3$ . On ne conserve que les trois GRS des bits de  $0,1$  qui sont rejetés à droite du nouveau bit de poids faible.

$$1.10011001100110011001101.2^{-4} \quad GRS \quad (10)$$

$$\mapsto 0.11001100110011001100110.2^{-3} \quad 100 \quad (11)$$

$$(12)$$

On peut alors procéder à l'addition :

$$0.11001100110011001100110.2^{-3} \quad GRS \quad (13)$$

$$+ 1.10011001100110011001101.2^{-3} \quad 100 \quad (14)$$

$$+ 1.10011001100110011001101.2^{-3} \quad 000 \quad (15)$$

$$----- \quad (16)$$

$$10.01100110011001100110011.2^{-3} \quad 100 \quad (17)$$

$$\mapsto 1.00110011001100110011001.2^{-2} \quad 110 \quad (18)$$

$$\mapsto 1.00110011001100110011010.2^{-2} \quad (19)$$

Le passage de 17 à 18 se justifie par le maintien de la norme : on choisit l'exposant de la notation IEEE754 d'après le premier bit à 1. Les bits décalés vers la droite se retrouvent positionnés sur les bits GRS.

On effectue ensuite l'opération d'arrondi 19 en tenant compte des bits GRS. Le résultat obtenu se termine par 01|110 (en notant  $b_1 b_0 | GRS$ ). On essaie donc de savoir comment se positionne cette valeur par rapport au milieu de l'intervalle de représentation possible qui est  $[01, 10]$ . Ce milieu vaut  $1.100$ . Le résultat obtenu  $1.110$  étant supérieur à cette valeur, on arrondit au plus près au nombre strictement supérieur  $10$ . D'où le résultat.

La valeur  $0.3_{10}$  est codée par :

$$0,3_{10} \mapsto 1.00110011001100110011010.2^{-2}. \quad (20)$$

**En simple précision**, on a donc bien  $(0,1 + 0,2) - 0,3 = 0$ .

Cependant, les limites de précision des flottants évoquées précédemment engendrent des calculs parfois déroutants et entachés d'erreur. L'associativité de l'addition peut ne plus être vérifiée, la distributivité de la multiplication par rapport à l'addition non plus.

■ **Exemple 8 — Erreur d'arrondis en double précision.** Considérons de nouveau l'opération  $(0,1 + 0,2) - 0,3$  mais cette fois-ci **en double précision**. Le résultat de cette opération devrait être 0, mais il n'en est rien et on trouve  $5,551115123125783.10^{-17}$ .

L'erreur commise provient de la représentation de  $0,3_{10}$  en double précision :

$$0,3_{10} \mapsto 1.00110011001100110011001100110011001100110011.2^{-2} \quad (21)$$

Pour  $0,3_{10}$ , les trois derniers bits valent 011 et les bits GRS 001. L'intervalle considéré pour l'arrondi est  $[11, 100]$ , son milieu vaut  $11.100$  et  $11.001 < 11.100$ . La valeur choisie pour les trois derniers bits est donc la valeur inférieure de l'intervalle, soit 011 : l'arrondi ne modifie pas la valeur codée en machine. Cette représentation se note 0,2999999999999999 sur 17 décimales.

Par ailleurs, l'addition de  $0,1 + 0,2$  aboutit à  $0,30000000000000004$  en représentant 17 décimales. La soustraction de la représentation de  $0,3$  en double précision aboutit donc à une erreur d'arrondi. **En double précision**, on a  $(0,1 + 0,2) - 0,3 = 5,551115123125783.10^{-17}$ . La différence entre les deux valeurs porte sur les trois derniers bits.

**Bien entendu, les chiffres significatifs du résultat, c'est-à-dire les 15 premiers, sont corrects. Il faut donc juste être vigilant sur l'interprétation des résultats.**

## D Conséquence des erreurs et des arrondis

■ **Définition 8 — Mécanisme d'absorption.** Le mécanisme d'absorption apparaît lorsqu'on additionne deux valeurs dont l'écart relatif est très important : cela engendre l'absorption de la plus petite valeur par la plus grande.

■ **Exemple 9 — Perte de l'associativité de l'addition.** Considérons l'addition suivante en simple précision :

$$2^{24} + 1 + 1 \quad (22)$$

Supposons que le calcul soit mené de la gauche vers la droite, c'est-à-dire  $(2^{24} + 1) + 1$ . Comme l'écart entre deux nombres codables en IEEE 754 est de 2 lorsque l'exposant non biaisé  $e$  vaut 24, on obtient que  $(2^{24} + 1) + 1 = 2^{24} + 1 = 2^{24}$ . Or, si on commence le calcul par la droite,  $2^{24} + (1 + 1) = 2^{24} + 2$ , car le nombre 2 est représentable avec l'exposant 24. L'addition n'apparaît donc plus associative.

■ **Exemple 10 — Multiplication non distributive par rapport à l'addition.** Considérons l'opération  $100 \times (0,1 + 0,2)$  en simple précision.

Si le calcul est effectué en commençant par l'addition, le résultat est faux et vaut 30,000000000000004. Par contre, si on développe le calcul et qu'on l'effectue après développement, on trouve exactement  $100 \times 0,1 + 100 \times 0,2 = 30,0$ .

Lorsqu'on utilise les flottants, il faut donc éviter d'additionner des calculs sur des nombres dont l'écart relatif est très important. De même, on évitera de soustraire deux valeurs très proches au risque de perdre énormément de chiffres significatifs dans le résultat.

De plus, il faut se poser la question de la nécessité d'utiliser une simple ou une double précision en fonction de l'application.

**R** L'impact du changement de précision des flottants sur la complexité mémoire est conséquent car il double systématiquement l'espace mémoire nécessaire au stockage des données. Par ailleurs, il a également un impact important sur la complexité temporelle, car les unités de calculs sur les flottants (FPU) sont spécifiques, plus complexes et moins nombreuses que les unités arithmétiques sur les entiers (ALU). La puissance de calcul nécessaire pour changer de précision n'est donc pas à négliger non plus et dépend fortement de l'architecture du processeur.

**R** Certains processeurs ne disposent même pas d'unités arithmétiques pour les flottants. Dans ce cas, il faut soit utiliser l'arithmétique fixée soit émuler le calcul des flottants. La première solution, l'arithmétique fixée, est la plus souhaitable pour minimiser l'impact sur la complexité temporelle. Dans le domaine des systèmes embarqués elle est souvent à privilégier.

## E Bilan

**M** **Méthode 1 — Bien manipuler les flottants** Pour bien calculer avec les flottants, il est préférable de :

1. ne **jamais** tester l'égalité exacte ou la différence entre deux `float`, préférer `<` ou `>`
2. additionner les petits avant les grands,
3. éviter de soustraire deux données quasi-égales (perte de précision),
4. se poser la question de la précision nécessaire au calcul,
5. calculer sur les entiers si c'est possible plutôt que sur les `float`,
6. s'il est possible de renoncer à un calcul rapide, les bibliothèques `decimal` et `mpmath` permettent de calculer avec une précision arbitraire.

**R** Pour connaître les limites de sa machine en Python, consulter la documentation [ici](#) :

```

import math
import sys

m = sys.float_info.max
# nombre maximum positif fini flottant encodable
eps = sys.float_info.epsilon
# différence entre 1.0 et la plus petite valeur plus grande que 1.0 et encodable en
# flottant
print(m)
# 1.7976931348623157e+308
print(eps)
# 2.220446049250313e-16
print(m == m-(m*(eps/10))) # True
print(m == m-(m*eps))     # False

```

---

**R** Attention : `10e23` est un flottant qui vaut 10 que multiplie  $10^{23}$ , soit  $10^{24}$ . Alors que `10**23` représente un entier : l'entier 10 à la puissance 23. D'ailleurs l'assertion `10e23 == 10**23` renvoie faux;-)