

REPRÉSENTATION DES NOMBRES FLOTTANTS

I. Arithmétique flottante IEEE 754

1) Nombres binaires à virgule

Par analogie avec l'écriture décimale, on peut définir les nombres binaires à virgule : les bits après la virgule correspondent à des puissances négatives de 2.

Par exemple, 100,011 désigne :

$$\begin{aligned} 1.2^2 + 0.2^1 + 0.2^0 + 0.2^{-1} + 1.2^{-2} + 1.2^{-3} &= 4 + \frac{1}{4} + \frac{1}{8} \\ &= 4,375 \end{aligned}$$

2) Arithmétique flottante

En écriture décimale, pour écrire des nombres très petits, comme 0,000000001, ou très grands, comme 100000000, on doit écrire beaucoup de zéros non significatifs. On utilise alors la notation scientifique : 1.10^{-9} , 1.10^8 .

Il existe un équivalent en base 2, défini dans la norme IEEE¹ 754.

Un nombre non nul est alors représenté sous la forme $s m . 2^e$ où :

- s est le *signe* du nombre,
- e est son *exposant*, un entier relatif,
- m est sa *mantisse*, un nombre compris entre 1 (inclus) et 2 (exclu).

On parle alors de nombre flottant (*float* en anglais).

Sur 64 bits (flottants double précision) :

- Le signe s est représenté sur 1 bit : 0 pour +, 1 pour −.
- L'exposant e est un entier compris entre −1022 et 1023
 - On représente sur 11 bits l'entier naturel $e + 1023$ (décalage), et $1 \leq e + 1023 \leq 2046$
 - Les mots de 11 bits correspondant aux entiers naturels 0 (00...0) et 2047 (11...1) servent pour des cas particuliers.
- L'unique chiffre avant la virgule de la mantisse m est forcément un 1. Par conséquent, on ne l'écrit pas (on parle de bit caché).
Les 52 bits restants sont utilisés pour représenter les 52 bits après la virgule de la mantisse.

Finalement, le mot binaire

$$s e_1 e_2 \dots e_{11} m_1 m_2 \dots m_{52}$$

représente le nombre :

$$(-1)^s \times 2^{n-1023} \times \left(1 + \sum_{i=1}^{52} m_i 2^{-i} \right)$$

où n est l'entier naturel dont l'écriture binaire est $e_1 e_2 \dots e_{11}$

Exemple

Quel est le nombre flottant représenté par

$$1 \text{ 10010001101 101101001000 } \dots 0$$

- **Signe** : 1, donc nombre négatif.
- **Exposant** : 10010001101 représente l'entier naturel $2^0 + 2^2 + 2^3 + 2^7 + 2^{10} = 1165$, donc l'exposant est $1165 - 1023 = 142$

1. Institute of Electrical and Electronics Engineers

— **Mantisse** :

$$m = \underline{1,101101001000\dots 0}$$

$$= 1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-9} = \frac{873}{2^9}$$

— Le nombre représenté est donc : $-\frac{873}{2^9} \times 2^{142} = -873.2^{133} \approx -9,506.10^{42}$

Exemple

Quel est le nombre flottant représenté par

0 **00010001101** **0011010010000** **...** **0**

— **Signe** : 0, donc nombre positif.

— **Exposant** : **00010001101** représente l'entier naturel $2^0 + 2^2 + 2^3 + 2^7 = 141$, donc l'exposant est $141 - 1023 = -882$

— **Mantisse** :

$$m = \underline{1,0011010010000\dots 0}$$

$$= 1 + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-9} = \frac{617}{2^9}$$

— Le nombre représenté est donc : $\frac{617}{2^9} \times 2^{-882} = 617.2^{-891} \approx 3,737.10^{-266}$

3) Cas particuliers pour l'exposant

La représentation précédente ne permet pas de représenter zéro. Par convention, zéro est représenté avec tous les bits de l'exposant et tous les bits de la mantisse à 0 (deux signes possibles : un zéro positif et un zéro négatif!)

Par extension, si l'exposant n'est composé que de 0, on considère que le bit caché est 0 et que l'exposant vaut -1022 . Cela permet de représenter des nombres de valeur absolue plus petite, et donc d'avoir une répartition plus uniforme des valeurs infinitésimales représentables.

Enfin, on a trois valeurs exceptionnelles pour lesquelles les 11 bits de l'exposant valent tous 1.

	Signe	Exposant	Mantisse
$+\infty$	0	111...1	000...0
$-\infty$	1	111...1	000...0
NaN (Not a Number)	0 ou 1	111...1	Autre que 000...0

II. Conséquences

1) Dépassement de capacité

On ne peut pas représenter un nombre :

- dont l'exposant est supérieur à 1023
- dont l'exposant est 1023 mais la mantisse est supérieure à la plus grande mantisse représentable : un 1 (bit caché) suivi de 52 bits à 1 après la virgule.

Lorsqu'un calcul avec les flottants donne une valeur trop grande en valeur absolue pour être représentable, on parle alors de dépassement arithmétique ou *overflow*.

On utilise dans ce cas les valeurs $+\infty$ et $-\infty$ (avec les règles usuelles de calcul, en utilisant NaN en cas d'ambiguïté)

Exemple

```
>>> k = 2.**1023
>>> k
8.98846567431158e+307
>>> 2*k
inf
>>> -3*k
-inf
>>> 2*k - 3*k
nan
```

```
>>> 2.**1024
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    2.**1024
OverflowError: (34, 'Result too large')
```

De même, on ne peut pas représenter un nombre de valeur absolue plus petite que le plus petit nombre dénormalisé positif. Lorsqu'un calcul avec les flottants donne une valeur trop petite en valeur absolue pour être représentable, on parle de dépassement par valeurs inférieures ou de sous-dépassement arithmétique (ou encore *underflow*).

Dans ce cas, soit une erreur se produit, soit la valeur est arrondie à zéro (en conservant le signe)

Exemple

```
>>> e = 2.**(-1074)
>>> e
5e-324
>>> e/2
0.0
>>> (-e)/2
-0.0
>>> 1/e
inf
>>> 1/(1/e)
0.0
>>> 1/(e/2)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    1/(e/2)
ZeroDivisionError: float division by zero
```

2) Erreurs d'arrondi

Les nombres décimaux ne sont pour la plupart pas représentables de manière exacte sous forme de flottants.

Par exemple,

$$0,3 = 0,010011001100110011001\dots$$

La représentation en virgule flottante sera donc une valeur approchée (la valeur représentable la plus proche). Ceci peut générer des comportements dus à ces arrondis.

Exemple

```
>>> 3 * 0.1
0.30000000000000004
>>> 1/49 * 49
0.9999999999999999
```

Enfin, le résultat d'une opération sur deux flottants n'est pas forcément représentable de manière exacte.

Par exemple, 1 et 2^{-53} sont représentables de manière exacte, mais pas $1 + 2^{-53}$.

```
>>> p = 1.
>>> q = 2.**(-53)
>>> p, q
(1.0, 1.1102230246251565e-16)
>>> p + q
1.0
>>> p + q - p
0.0
>>> p - p + q
1.1102230246251565e-16
```

3) Tests d'égalité

Lorsqu'on utilise des flottants, il faut donc manipuler les tests d'égalité avec précaution...

```
>>> 0.1*3 == 0.3  
False
```

On remplacera alors de tels tests `a == b` par des tests du type $\text{abs}(a - b) < \varepsilon$ avec ε une valeur proche de zéro choisie en fonction de l'ordre de grandeur des erreurs probables sur a et b .