REPRÉSENTATION DES NOMBRES

B. Landelle

Table des matières

I Introduction

1 Système binaire

C'est la base 2 encore appelé *système binaire* qui structure l'ensemble de l'information stockée ou transmise en informatique.

Définition 1. *L'unité d'information élémentaire valant 0 ou 1 s'appelle le* bit *(contraction de binary digit). Une séquence de* 8 *bits s'appelle un* octet *(byte en anglais).*

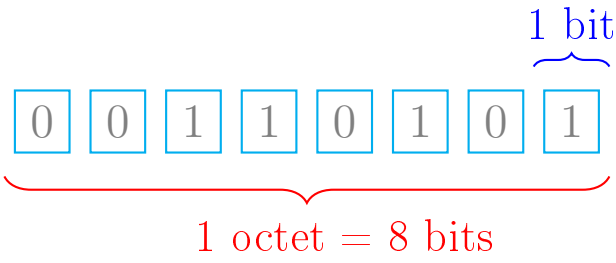


FIGURE 1 – Un octet constitué de 8 bits

La numération qui nous est familière est la numération dite *décimale positionnelle*. On décompose par exemple

$$123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

On peut procéder à la même décomposition en base 2. Les chiffres ne sont plus $0, 1 \ldots, 9$ mais 0 et 1 puisqu'on regroupe par paquets de 2. Ceci donne par exemple

$$123 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Cette décomposition s'obtient simplement par divisions euclidiennes successives par 2. Dans le schéma qui suit, l'écriture binaire se lit *de droite à gauche*.

123	2					
122	61	2				
1	60	30	2			
	1	30	15	2		
		0	14	7	2	
			1	6	3	2
				1	2	1
					1	

$\leftarrow$

$$123 = \langle 1, 1, 1, 1, 0, 1, 1 \rangle$$

Expérimentation :

```
>>> bin(123)
'0b1111011'
```

Plus généralement, on a le résultat suivant :

Théorème 1. *Soit n un entier. Alors il existe un entier p non nul et un unique p-uplet $(d_0, \dots, d_{p-1}) \in \{0,1\}^p$ tel que*

$$n = \sum_{i=0}^{p-1} d_i \, 2^i$$

Notation : On notera $\langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle$ cette écriture binaire de n.

Exemple : On a $123 = \langle 1,1,1,1,0,1,1 \rangle$

Sans autre règle, le choix de p n'est pas unique :

$$123 = \langle 1,1,1,1,0,1,1 \rangle = \langle 0,0,1,1,1,1,0,1,1 \rangle$$

Définition 2. *Soit n un entier dont une écriture binaire donnée est $n = \langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle$. le chiffre d_{p-1} est appelé bit de* poids fort *tandis que d_0 est appelé bit de* poids faible.

Excepté zéro, un entier contient nécessairement un bit égal à 1 dans son écriture binaire. On peut, comme on le fait pour l'écriture décimale, considérer l'écriture binaire de n sans zéros à gauche, autrement dit avec le bit de poids fort égal à 1. Avec ce formalisme, on a le résultat plus précis suivant :

Théorème 2. *Soit n un entier non nul. Alors il existe un unique entier p non nul et un unique p-uplet $(d_0, \dots, d_{p-1}) \in \{0,1\}^{p-1} \times \{1\}$ tel que*

$$n = \sum_{i=0}^{p-1} d_i \, 2^i$$

Autrement dit, pour un entier non nul, en imposant le bit de poids fort égal à 1, son écriture binaire est unique.

Remarque : Quelques puissances de 2

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

Exercice : Déterminer l'écriture binaire des nombres suivants : 6, 27.

Corrigé :

6	2	
6	3	2
0	2	1
	1	

←

$6 = \langle 1,1,0 \rangle$

27	2		
26	13	2	
1	12	6	2
	1	6	3	2
		0	2	1
			1	

←

$27 = \langle 1,1,0,1,1 \rangle$

Une autre base est fréquemment utilisée en informatique même si elle ne sera pas spécialement utilisée dans le cadre de ce cours : la base 16.

Définition 3. *La base* 16 *est appelée* système hexadécimal. *Les chiffres de cette base sont :* 0, 1, …, 9, A, B, …, F.

On peut donc coder un octet avec deux chiffres hexadécimaux.

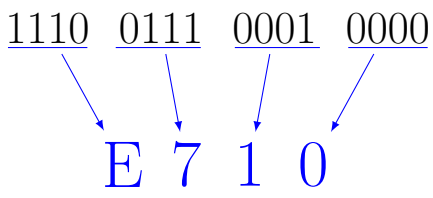


FIGURE 2 – Du binaire vers l'hexadécimal

2 Aspects quantitatifs de l'écriture binaire

Dans cette partie, on retient la convention d'un bit de poids fort égal à 1 dans toute écriture binaire.

Proposition 1. *Le nombre de chiffres nécessaires à l'écriture binaire d'un entier non nul n'est pas borné par une constante indépendante de l'entier en question.*

Démonstration. Pour p entier, on a

$$2^p = \langle 1, \underbrace{0, \ldots, 0}_{p \text{ zéros}} \rangle$$

dont l'écriture binaire requiert $p+1$ chiffres avec p qui peut être choisi arbitrairement grand. □

Dans l'écriture binaire d'un entier non nul $n = \langle d_{p-1}, \ldots, , d_0 \rangle$, chaque chiffre binaire d_i va donc consommer un bit de mémoire pour son stockage. La mémoire physique de l'ordinateur étant finie par nature, il est raisonnable que la représentation des entiers en machine soit également finie. On peut donc borner $p \leqslant p_{\max}$ dans l'écriture précédente. L'ordinateur sera alors en mesure de traiter les entiers dans la plage de valeurs $[\![0 ; 2^{p_{\max}} - 1]\!]$ puisque

$$0 \leqslant \sum_{i=0}^{p-1} d_i 2^i \leqslant \sum_{i=0}^{p_{\max}-1} 2^i = 2^{p_{\max}} - 1$$

Exercice : Avec un codage sur un octet, quelle plage de valeurs peut-on coder en binaire ? Même question avec un codage sur 4 octets ?

Corrigé : Un octet vaut 8 bits ce qui permet de coder la plage $[\![0 ; 2^8 - 1]\!] = [\![0 ; 255]\!]$. Avec 4 octets (32 bits), on code de la plage $[\![0 ; 2^{32} - 1]\!] = [\![0 ; 4294967295]\!]$.

On peut aussi envisager le fait que la taille mémoire dédiée à l'écriture d'un nombre ne soit pas bornée *a priori* mais fonction des ressources disponibles de la machine. Sous cette approche, l'aspect inverse est intéressant : pour un entier n, combien de chiffres sont nécessaires à son écriture en base 2.

Définition 4. *La* partie entière *d'un réel x est l'unique entier relatif noté $\lfloor x \rfloor$ tel que*

$$\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1$$

Remarque : On a les encadrements suivants (utiles en pratique) :

$$\forall x \in \mathbb{R} \qquad \lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1 \iff x - 1 < \lfloor x \rfloor \leqslant x$$

Exemple : $\lfloor 2.3 \rfloor = 2$, $\lfloor \pi \rfloor = 3$, $\lfloor -\pi \rfloor = -4$.

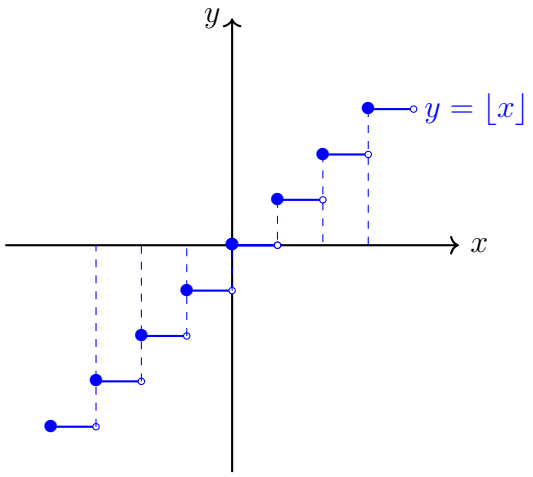


FIGURE 3 – Graphe de la partie entière

Définition 5. *La fonction* logarithme en base 2 *notée* $\log_2$ *est définie sur* $]0;+\infty[$ *par*

$$\forall x>0 \qquad \log_2(x)=\frac{\log(x)}{\log(2)}, \qquad \log \text{ désignant le logarithme népérien}$$

Remarque : La fonction $\log_2$ hérite de la propriété fondamentale du logarithme usuel (népérien) à savoir

$$\forall (x,y)\in]0;+\infty[^2 \qquad \log_2(x\times y)=\log_2(x)+\log_2(y)$$

et de ses conséquences

$$\forall (x,y,\alpha)\in]0;+\infty[^2\times\mathbb{R} \qquad \log_2\left(\frac{x}{y}\right)=\log_2(x)-\log_2(y) \quad \text{et} \quad \log_2(x^\alpha)=\alpha\log_2(x)$$

Proposition 2. *Soit n entier non nul. Il faut $p=\lfloor \log_2(n)+1\rfloor$ chiffres pour son écriture binaire.*

Démonstration. Si p est le nombre de chiffres de l'écriture binaire de n, cela signifie que

$$n=2^{p-1}+\sum_{i=0}^{p-2} d_i 2^i$$

avec les $d_i\in\{0,1\}$. Par convention, la somme vaut zéro si $p\leqslant 1$. Ainsi, on a l'encadrement

$$2^{p-1}\leqslant n\leqslant \sum_{i=0}^{p-1}2^i=\frac{2^p-1}{2-1}=2^p-1<2^p$$

Passant au logarithme, fonction strictement croissante, on obtient

$$(p-1)\log(2)\leqslant \log(n)<p\log(2)\Longrightarrow p\leqslant \log_2(n)+1<p+1$$

Le résultat en découle. □

Remarque : On retrouve notamment le résultat de la proposition 1 puisque $\lfloor \log_2(n)+1\rfloor\to+\infty$ pour $n\to+\infty$.

Expérimentation : La méthode `bit_length` renvoie la taille de l'écriture binaire d'un nombre

```
>>> a=123
>>> bin(a)
'0b1111011'
>>> a.bit_length()
7
>>> import numpy as np
>>> int(np.log2(a)+1)
7
```

Pour un nombre positif, la conversion d'un flottant en entier renvoie sa partie entière (c'est faux pour un nombre négatif).

Exercice : Soit x un réel et n un entier non nul d'écriture binaire $n = \langle d_{p-1}, \dots, d_0 \rangle$. Écrire x^n en fonction des x^{2^i} pour $i \in [\![0\,;\, p-1]\!]$.

Corrigé : On a $n = \sum_{i=0}^{p-1} d_i 2^i$ d'où

$$x^n = x^{\left(\sum_{i=0}^{p-1} d_i 2^i\right)} = \prod_{i=0}^{p-1} \left(x^{d_i 2^i}\right) = \prod_{i=0}^{p-1} \left(x^{2^i}\right)^{d_i}$$

Cette écriture permet d'envisager un algorithme performant pour le calcul de x^n, algorithme dit d'*exponentiation rapide*.

3 Opérations en représentation binaire

Dans cette partie, on retient la convention d'un bit de poids fort égal à 1 dans toute écriture binaire.

Théorème 3 (Division euclidienne). *Soit $(a, b) \in \mathbb{N} \times \mathbb{N}^*$. Il existe un unique couple $(q, r) \in \mathbb{N} \times [\![0\,;\, b-1]\!]$ tel que $a = b \times q + r$. Le terme q est appelé* quotient *et le terme r est appelé* reste.

Remarque : C'est la division usuelle pratiquée depuis les petites classes.

L'algorithme d'addition de deux entiers binaires est identique à celui employé en écriture décimale. Concernant la multiplication et la division par 2, on dispose du résultat suivant :

Proposition 3. *Soit n un entier non nul avec $n = \langle d_{p-1}, \dots, d_1, d_0 \rangle$. On a*

$$2 \times n = \langle d_{p-1}, \dots, d_0, 0 \rangle \quad \text{et} \quad n//2 = \langle d_{p-1}, \dots, d_1 \rangle$$

Démonstration. On a

$$2 \times \langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle = 2 \times \left(\sum_{i=0}^{p-1} d_i 2^i \right) = \sum_{i=1}^{p} d_{i-1} 2^i = \langle d_{p-1}, \dots, d_0, 0 \rangle$$

et

$$\langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle = \sum_{i=0}^{p-1} d_i 2^i = 2 \times \left(\sum_{i=0}^{p-2} d_{i+1} 2^i \right) + d_0$$

Ainsi, d'après le théorème de la division euclidienne

$$\langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle // 2 = \langle d_{p-1}, \dots, d_1 \rangle$$

□

Remarque : La multiplication par 2 consiste donc en un décalage de l'écriture binaire de 1 bit vers la gauche et la division par 2 consiste donc en un décalage de l'écriture binaire de 1 bit vers la droite, avec perte du bit de poids faible.

Exercice : Quelle opération mathématique transforme l'entier binaire $\langle d_{p-1}, \dots, d_0 \rangle$ en :

1. $\langle d_{p-1}, \dots, d_0, 0, 0, 0 \rangle$?
2. $\langle d_{p-1}, \dots, d_2 \rangle$?

Corrigé : D'après la proposition 3, le décalage de 3 bits à gauche correspond à une multiplication par $2\times2\times2 = 2^3$ et le décalage de 2 bits à droite correspond à une division par $2\times2 = 2^2$.

Exercice : Établir la relation

$$\langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle \times \langle e_{q-1}, e_{q-2}, \dots, e_1, e_0 \rangle = \sum_{i=0}^{q-1} e_i \times \langle d_{p-1}, d_{p-2}, \dots, d_1, d_0, \underbrace{0, \dots, 0}_{i \text{ zéros}} \rangle$$

Corrigé : On distribue le produit simplement.

$$\begin{aligned}\langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle \times \langle e_{q-1}, e_{q-2}, \dots, e_1, e_0 \rangle &= \sum_{i=0}^{q-1} e_i 2^i \times \langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle \\ &= \sum_{i=0}^{q-1} e_i \times \langle d_{p-1}, d_{p-2}, \dots, d_1, d_0, \underbrace{0, \dots, 0}_{i \text{ zéros}} \rangle\end{aligned}$$

4 Exponentiation rapide

Soit x un réel et n un entier non nul d'écriture binaire $n = \langle d_{p-1}, \dots, d_0 \rangle$. On a

$$x^n = x^{\left(\sum_{i=0}^{p-1} d_i 2^i\right)} = \prod_{i=0}^{p-1} \left(x^{d_i 2^i}\right) = \prod_{i=0}^{p-1} \left(x^{2^i}\right)^{d_i}$$

Dans le produit, à i fixé dans $[\![0 ; p-2]\!]$, on passe du terme x^{2^i} au suivant $x^{2^{i+1}}$ en élevant au carré :

$$x^{2^{i+1}} = x^{2\times 2^i} = \left(x^{2^i}\right)^2$$

La contribution de x^{2^i} dans le produit est déterminée par la valeur de d_i : si $d_i = 0$, le terme n'apparaît pas dans le produit et sinon il apparaît. Cette écriture permet d'envisager un algorithme performant pour le calcul de x^n, algorithme dit d'*exponentiation rapide* :

```
def expo(x,n):
    """expo(x:int or float,n:int)->int or float
    Calcul de x**n par exponentiation rapide"""
    a,r,e=x,1,n
    while e>0:
        if e%2==1:
            r*=a
        a*=a
        e//=2
    return r
```

La variable `r` sert au calcul du résultat final.

La variable `a` sert au calcul des termes x^{2^i} successifs. Elle est initialisée à x et est élevée au carré lors de chaque passage dans la boucle.

Enfin, on utilise la variable `e` pour connaître la valeur de d_i au cours du produit.

La variable `e` est initialisée à $n = \langle d_{p-1}, \dots, d_0 \rangle$ et l'instruction `e%2` renvoie d_0, son bit de poids faible.

Ensuite, la variable `e` reçoit son quotient par 2 ce qui fait qu'elle reçoit $\langle d_{p-1}, \dots, d_1 \rangle$. Son bit de poids faible renvoyé par `e%2` est d_1.

Ainsi, au cours de la boucle, l'instruction `e%2` renvoie successivement d_0, d_1, ... Si $d_i = 1$, la variable `r` reçoit son produit par `a` et sinon elle n'est pas modifiée.

On répète ce procédé tant que `e>0`. La boucle s'arrête puisque la taille de l'écriture binaire de `e` décroît strictement.

On peut écrire une version itérative de ce code mais ceci impose de calculer la taille de l'écriture binaire de n.

```
def expo(x,n):
    a,r=x,1
    if n>0:
        p=int(np.log2(n)+1);
        e=n
        for k in range(p):
            if e%2==1:
                r*=a
            a*=a;e//=2
    return r
```

II Les entiers dans python

1 Les entiers signés de taille fixe

La plupart des langages de programmation permettent de travailler sur des entiers relatifs (on parle d'entiers *signés*) dans une plage de valeurs données. Selon le langage ou l'architecture de la machine, le codage de ces entiers se fait 32 bits, 64 bits ,...

Pour réaliser le codage de nombres négatifs, le bit dit *de poids fort* (celui le plus à gauche) sert à coder les nombres négatifs par soustraction à un entier fixe. Cette méthode est dite *méthode de complément à* 2.

Définition 6. *En* méthode de complément à 2, *le codage d'entiers relatifs est défini par l'écriture binaire*

$$\langle d_{p-1}, d_{p_2}, \dots, d_0 \rangle_{\texttt{CPL2}} = -d_{p-1}2^{p-1} + \sum_{i=0}^{p-2} d_i 2^i$$

$-$	$+$	$\dots$	$\dots$	$+$	$+$
d_{p-1}	d_{p-2}	$\dots$	$\dots$	d_1	d_0

SCHÉMA - Codage en complément à 2

Remarque : On aurait pu imaginer que le codage du signe soit réalisé par un bit, le bit de poids fort par exemple.

$$\langle d_{p-1}, d_{p-2}, \dots, d_1, d_0 \rangle_? = (-1)^{d_{p-1}} \times \sum_{i=0}^{p-2} d_i 2^i$$

Avec ce codage, on voit que zéro peut être codé de deux manières distinctes ce qui n'est pas satisfaisant si l'on doit tester la nullité d'une variable. Mais surtout, cette écriture n'est pas compatible avec l'algorithme d'addition tandis que la méthode de complément à 2 l'est.

Exemple : Avec une écriture binaire à 4 bits, on a

$$\langle 0,0,0,0 \rangle_? = \langle 1,0,0,0 \rangle_? = 0$$

puis $$\langle 0,0,0,1 \rangle_? = 1, \qquad \langle 1,0,0,1 \rangle_? = -1$$

et, en appliquant naïvement l'algorithme d'addition sur ces écritures binaires, on obtient

$$\begin{aligned} &\langle 0,0,0,1 \rangle_? \\ +\ &\langle 1,0,0,1 \rangle_? \\ =\ &\langle 1,0,1,0 \rangle_? \neq \langle 0,0,0,0 \rangle_? \end{aligned}$$

Ainsi, il faudrait écrire un nouvel algorithme d'addition pour gérer les entiers signés.

Avec la méthode de complément à deux, on a

$$\langle 0,0,0,1 \rangle_{\texttt{CPL2}} = 1, \qquad \langle 1,1,1,1 \rangle_{\texttt{CPL2}} = -2^3 + 1 + 2 + 2^2 = -2^3 + 2^2 - 1 = -1$$

et avec l'algorithme usuel d'addition (le bit de poids fort étant perdu en cas de dépassement)

$$\begin{aligned} &\langle 0,0,0,1 \rangle_{\texttt{CPL2}} \\ +\ &\langle 1,1,1,1 \rangle_{\texttt{CPL2}} \\ =\ &\langle 0,0,0,0 \rangle_{\texttt{CPL2}} = 0 \end{aligned}$$

ce qui illustre la compatibilité de cette écriture avec l'addition de nombres binaires.

Proposition 4. *La méthode de complément à 2 sur p bits permet de coder les entiers relatifs de la plage de valeurs* $[\![-2^{p-1} ; 2^{p-1} - 1]\!]$.

Démonstration. On a

$$-1 \times 2^{p-1} + \sum_{i=0}^{p-2} 0 \times 2^i \leqslant -d_{p-1}2^{p-1} + \sum_{i=0}^{p-2} d_i 2^i \leqslant 0 \times 2^{p-1} + \sum_{i=0}^{p-2} 1 \times 2^i = 2^{p-1} - 1$$

□

Exercice : Soit $n = \langle d_{31}, \dots, d_0 \rangle_{\texttt{CPL2}} \in [\![-2^{31} ; 2^{31} - 1]\!]$.

1. Calculer $1 + \sum_{i=0}^{30} 2^i$.
2. En déduire l'écriture binaire en méthode de complément à 2 de $-n-1$.
   Indication : on pourra distinguer le cas $n \geqslant 0$ du cas $n < 0$.

Corrigé : 1. On a $1 + \sum_{i=0}^{30} 2^i = 2^{31}$ (somme géométrique).

2. Supposons $n \geqslant 0$. Ainsi, $d_{31} = 0$ et on a

$$n = \sum_{i=0}^{30} d_i 2^i \Longrightarrow -n = -\sum_{i=0}^{31} d_i 2^i$$

Cette dernière écriture n'étant pas adaptée au codage binaire, on décale $-n$ de 2^{31} :

$$2^{31} - n = 1 + \sum_{i=0}^{30} \underbrace{(1 - d_i)}_{\in\{0,1\}} 2^i \Longrightarrow -n - 1 = \langle 1, 1 - d_{30}, \dots, 1 - d_0 \rangle_{\texttt{CPL2}}$$

Supposons ensuite $n < 0$. On a $d_{31} = 1$ d'où

$$n = -1 \times 2^{31} + \sum_{i=0}^{30} d_i 2^i \Longrightarrow -n = 2^{31} - \sum_{i=0}^{30} d_i 2^i$$

En utilisant l'égalité la question 1, il vient

$$-n = 1 + \sum_{i=0}^{30} \underbrace{(1 - d_i)}_{\in\{0,1\}} 2^i \Longrightarrow -n - 1 = \langle 0, 1 - d_{30}, \dots, 1 - d_0 \rangle_{\texttt{CPL2}}$$

Dans tous les cas, on a démontré que pour $n = \langle d_{31}, d_{30}, \dots, d_1, d_0 \rangle_{\texttt{CPL2}}$

$$-1 - n = \langle 1 - d_{31}, 1 - d_{30}, \dots, 1 - d_1, 1 - d_0 \rangle_{\texttt{CPL2}}$$

2 Le type `int`

Définition 7. *Un entier de type* `int` *sous Python est un entier relatif dont la taille de codage est a priori quelconque.*

Remarque : En réalité, la taille est limitée par la mémoire allouée par l'ordinateur à l'interpréteur python.

```
>>> 2
2
>>> 2**2026
770497466833853895681259076325177001010038916342627621463068614662281397315
690300302438769615242802231042813323066187369683255175363801028611377534543234899535915
>>> type(2)
<class 'int'>
>>> type(2**2020)
<class 'int'>
```

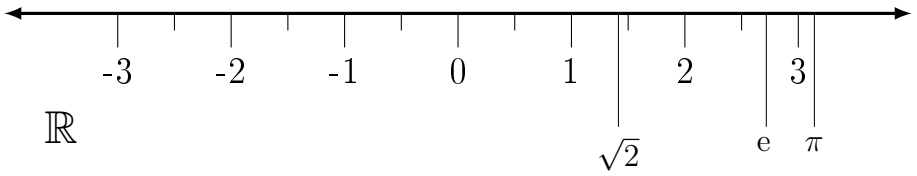
En apparence, on n'observe aucune différence de traitement entre les petits et les grands entiers. La réalité est plus complexe. Les grands entiers sont découpés en tableau d'entiers courts ce qui rend tous les calculs sur ces nombres beaucoup plus lents.

Le format optimisé dans python est celui des petits entiers codés selon la méthode de complément à 2 dans la plage -2^{30}, $2^{30}-1$. Pour l'utilisateur, le basculement hors de cette plage de valeurs est totalement transparent mais il induit un coût en terme de performance.

III Nombres à virgules dans python

1 Arithmétique flottante

Qu'est-ce qu'un réel ? La réponse n'est pas si simple si on veut donner un sens mathématique précis à cette notion.



On se contentera d'admettre leur existence et surtout leur représentation décimale. Ainsi, un réel x peut s'écrire en base 10

$$x = \pm\, e_p e_{p-1} \dots e_0, d_1 d_2 d_3 \dots$$

où les e_i et les d_i désignent des entiers entre 0 et 9. L'entier $e_p \dots e_0$ est la partie entière de $|x|$. Les entiers d_i sont appelées les décimales de x. La « plupart » des nombres réels possèdent un nombre infini de décimales. La mémoire d'un ordinateur étant finie par nature, on ne peut pas coder fidèlement des nombres réels sur une machine.

Expérimentation :

```
>>> import numpy as np
>>> 1/3
0.3333333333333333
>>> np.sqrt(2)
1.4142135623730951
>>> np.pi
3.141592653589793
```

Le calcul des décimales de π est un thème de recherche toujours très actif (202 billions de décimales en juin 2024).

Définition 8. *Un nombre* décimal *est un réel x tel qu'il existe n entier naturel avec $x \times 10^n$ entier relatif.*

Remarque : Un nombre décimal est donc un nombre dont l'écriture décimale est finie. Par exemple, on a

$$1.234 = 1234 \times 10^{-3} \iff 1.234 \times 10^3 = 1234$$

Cette écriture sous forme de produit est intéressante : le nombre 1234×10^{10} est aussi facile à écrire que 1.234 alors que son écriture décimale complète est très grande. Ainsi, en séparant la contribution puissance de 10 du reste du nombre, on obtient une représentation concise pour des nombres très grands ou très petits. Cette démarche a donc inspiré la représentation décrite par le théorème suivant :

Théorème 4. *Soit x réel non nul. Il existe un unique triplet $(\varepsilon, m, e) \in \{-1,1\} \times [\,1\,;2\,[\, \times \mathbb{Z}$ tel que*

$$x = \varepsilon \times m \times 2^e$$

Exemple : Codage de $x = 11.8$ selon le théorème 4.
On a clairement $\varepsilon = 1$. Puis, il vient

$$|x| = m \times 2^e \Longrightarrow e = \log_2(|x|) - \log_2(m)$$

Comme e est un entier, il s'ensuit que $e = \lfloor \log_2(|x|) \rfloor$ donc ici $e = 3$ et m s'en déduit avec $m = \dfrac{x}{2^e} = \dfrac{11.8}{2^3} = 1.475$. On a obtenu

$$11.8 = 1 \times 1.475 \times 2^3$$

Expérimentation :

```
>>> x=11.8
>>> e=np.floor(np.log2(x))
>>> m=x/(2**3)
>>> m,e
(1.475, 3.0)
>>> 1.475*2**3
11.8
```

2 La norme IEEE 754

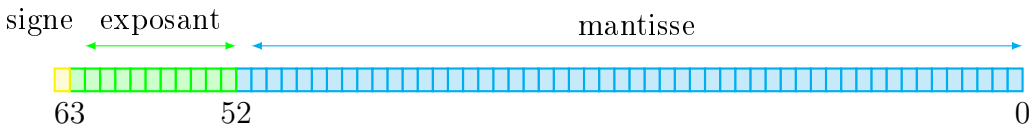
La norme IEEE[1] 754 définit le format de nombres en *virgule flottante* en s'appuyant sur le théorème 4. Ainsi, pour coder un réel en machine, on va coder le triplet (ε, m, e) et obtenir le format *virgule flottante*.

Définition 9. *La norme* IEEE 754 *définit le format de nombres en* virgule flottante *en base 2 de la forme*

$$(-1)^s \times \mathrm{M} \times 2^{\mathrm{E}-1023}$$

où $s \in \{0,1\}$ code le signe sur 1 bit, E *est un entier non nul appelé* exposant *codé sur* 11 *bits et* M *est un nombre binaire à virgule dans* $[\,1\,;2\,[$ *appelé* mantisse *codé sur* 52 *bits.*

Remarque : Ce codage en virgule flottante dit à *double précision* nécessite $1 + 52 + 11 = 64$ bits.



Expérimentation :

```
>>> type(1/3)
<class 'float'>
>>> type(np.pi)
<class 'float'>
```

1. Institute of Electrical and Electronical Engineers

```
>>> type(1.1)
<class 'float'>
>>> type(1.)
<class 'float'>
```

Notons $m_1, \dots, m_{52}$ les bits codant la mantisse. Comme $\mathrm{M} \in [\,1\,;2\,[$, le chiffre des unités de M est nécessairement 1 et ne nécessite donc pas un bit de stockage. Seules les puissances négatives de 2 sont codées avec

$$\mathrm{M} = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i}$$

L'exposant est un entier binaire codé conformément à la description qui en a été faite précédemment à savoir

$$\mathrm{E} = \langle e_{10}, \dots, e_0 \rangle = \sum_{i=0}^{10} e_i 2^i$$

Vocabulaire : L'appellation *virgule flottante* se comprend par opposition à *virgule fixe* : comme l'exposant n'est pas fixé, la virgule n'est pas à une position fixe (contrairement à celle de la mantisse), elle « flotte ».

Proposition 5. *Un nombre en virgule flottante suivant la norme IEEE 754 s'écrit*

$$(-1)^s \times \mathrm{M} \times 2^{\mathrm{E}-1023}$$

avec $s \in \{0,1\}$, $\mathrm{M} = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i}$ *où les* $m_i \in \{0,1\}$ *et* $\mathrm{E} = \langle e_{10}, \dots, e_0 \rangle = \sum_{i=0}^{10} e_i 2^i$ *où les* $e_i \in \{0,1\}$ *sont non tous nuls.*

s	e_{10}	e_9	$\dots$	$\dots$	e_1	e_0	m_{52}	m_{51}	$\dots$	$\dots$	m_2	m_1

SCHÉMA - Codage du type `float`

Exceptions

Si la mantisse commence toujours par 1, alors on ne peut en principe pas coder zéro (ennuyeux ...) On fixe par convention qu'un nombre vaut zéro si tous les bits de l'exposant et de sa mantisse sont à zéro (le bit de signe n'est pas contraint, d'où deux codages de zéro selon cette convention). On dit que zéro est un nombre *dénormalisé*, il ne suit pas la norme. Il existe d'autres nombres dénormalisés recensés précisément.

Exercice : Combien de nombres dans $[\,1\,;2\,[$ peuvent être codés fidèlement au format flottant ?

Corrigé : Pour un nombre flottant dans la plage $[\,1\,;2\,[$, on a $s = 0$, $\mathrm{E} = 1023$ et $\mathrm{M} = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i}$ avec $(m_i)_{i \in [\![1\,;52]\!]} \in \{0,1\}^{52}$. On peut donc coder fidèlement $\mathrm{Card}\,\{0,1\}^{52} = 2^{52}$ nombres dans $[\,1\,;2\,[$ ce qui est beaucoup en pratique, mais peu au regard de l'infinité de valeurs de $[\,1\,;2\,[$.

3 Limites de codage

Les nombres à virgule flottante étant codés sur un nombre fini de bits, ils sont nécessairement en nombre fini. Il est donc clair qu'on va se heurter à des limitations numériques intrinsèques au codage. Typiquement, si l'exposant est trop grand ou trop petit, il sera impossible à coder avec le nombre de bit prévu à cet effet. On parle d'*overflow* quand l'exposant est trop grand et d'*underflow* quand il est trop petit.

Si l'exposant peut être correctement codé, il n'en demeure pas moins une limitation majeure. Le codage de la mantisse

$$\mathrm{M} = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i} = 1 + \frac{\sum_{i=1}^{52} m_i 2^{52-i}}{2^{52}} \quad \text{avec} \quad \text{les } m_i \in \{0,1\}$$

permet de coder les éléments de la forme $1 + \dfrac{k}{2^{52}}$ avec $k \in [\![0\,;\, 2^{52}-1]\!]$ mais ne permet donc pas un codage de n'importe quel nombre réel de l'intervalle $[\,1\,;2\,[$.

Expérimentation :

```
>>> 0.1
0.1
>>> 0.2
0.2
```

En apparence, tout va bien : les nombres sont correctement affichés par l'interpréteur.

```
>>> 0.1+0.2
0.30000000000000004
```

Tout se gâte. Python se « trompe » sur une opération très simple. On est exactement dans la situation où le codage de la mantisse sur un nombre fini de bits entraîne une troncature : il faudrait un nombre infini de bits pour coder les mantisses de 0.1 et de 0.2 d'où une approximation stockée en machine et l'erreur de calcul grossière qui en découle.

```
>>> print(format(.1,".20f"))
0.10000000000000000555
>>> print(format(.2,".20f"))
0.20000000000000001110
```

En exigeant plus de décimales dans l'affichage des flottants 0.1 et 0.2, on constate clairement qu'ils ne sont pas codés fidèlement à leur écriture décimale. Ceci est inévitable et structurel : la mantisse est codée en puissances négatives de 2 et non de 10. La fraction binaire du nombre 0.1 s'écrit

$$0.1_{\text{dec}} = 0.0001100110011001100110011001100110011001100110011001100110011001100\ldots_{\text{bin}}$$

Cette égalité signifie

$$0.1 = 0 + \frac{0}{2} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \ldots$$

Par conséquent, dès la saisie des flottants, le « mal » est fait. Ils ne sont qu'imparfaitement codés en machine même si l'affichage (partiel) laisse croire le contraire.

En revanche, des opérations sur des flottants de la forme $1 + \dfrac{k}{2^{52}}$ avec $k \in [\![0\,;\, 2^{52}-1]\!]$ se déroulent impeccablement.

Expérimentation :

```
>>> 0.5-0.25
0.25
>>> 0.25-0.125
0.125
>>> 1/2**10-1/2**7
-0.0068359375
```

Sans mettre en défaut le codage de la mantisse, on peut détecter si l'on est dans ou hors du champ de précision du codage flottant. On parle d'erreur par *absorption*.

```
>>> 1+2**(-10)-1==2**(-10)
True
>>> 1+2**(-100)-1==2**(-100)
False
```

Exercice : Soient `a` et `b` des entiers. Comparer les opérations `a//b` et `int(a/b)`.

Corrigé : L'opération `a//b` est une opération exacte effectuée sur des entiers. En revanche, l'opération `a/b` est à valeurs dans les flottants et peut donc donner lieu à des erreurs d'exécution en cas de dépassement ou pire, donner lieu à des erreurs de calculs, même après conversion en entier. Ainsi, on observe

```
>>> (2**54+2)//2
9007199254740993
>>> int((2**54+2)/2)
9007199254740992
```

avec un second résultat faux du fait d'un débordement dans la mantisse pour le codage en flottant de $(2^{54}+2)/2$.

La représentation flottante doit donc influer sur notre façon de considérer ces nombres. Ainsi, tester $x = 0$ avec x à virgule flottante n'a, en général, pas de sens du fait de possibles erreurs d'arrondi. Il est beaucoup plus pertinent d'envisager de tester $|x| < \varepsilon$ où ε est un seuil de précision fixé par l'utilisateur en fonction des données du problème.

Expérimentation :

Considérons le polynôme $\mathrm{P} = (\mathrm{X} - 0.1)^2 = \mathrm{X}^2 - 0.2\mathrm{X} + 0.01$

Testons la nullité du discriminant.

```
>>> a=1
>>> b=-.2
>>> c=.01
>>> delta=b**2-4*a*c
>>> delta==0
False
>>> delta
6.938893903907228e-18
```

Le phénomène de *cancellation* (ou élimination) se produit lors de la soustraction de deux nombres très proches. Calculons un taux d'accroissement de la fonction $t \mapsto t$ en 1.

```
>>> h=1e-15
>>> (1+h-1)/h
1.1102230246251565
```

Le calcul du taux d'accroissement est médiocre. La soustraction de deux nombres proches révèle les approximations de leurs représentations flottantes. Le nombre $1+h$ n'est pas codé fidèlement et l'imprécision dans la différence $1+h-1$ est exacerbée quand on la divise par h qui est petit.

On rencontre parfois de bonnes surprises avec le format flottant :

```
>>> (1/3)*3
1.0
>>> (1/3)*3==1
True
```

et de moins bonnes :

```
>>> 10*(1/3)-1/3
2.9999999999999996
>>> np.sqrt(10)**2
10.000000000000002
```

Considérons un réel x et son codage flottant x' sous Python. Le signe et l'exposant de x et x' sont égaux puisque leurs codages ne requièrent aucune troncature. L'erreur de codage entre x et x' vient donc de la mantisse. On trouve :

$$|x - x'| = 2^{\mathrm{E}-1023} \times |\mathrm{M} - \mathrm{M}'|$$

où M et M′ désignent les mantisses respectives de x et x'. On a

$$\mathrm{M}' = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i} \quad \text{et} \quad \mathrm{M} = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i} + \frac{m_{53}}{2^{53}} + \frac{m_{54}}{2^{54}} + \dots$$

d'où
$$|\mathrm{M} - \mathrm{M}'| \leqslant \frac{1}{2^{53}} + \frac{1}{2^{54}} + \dots = \lim_{n\to+\infty} \frac{1}{2^{53}} \left(1 + \frac{1}{2} + \dots + \frac{1}{2^n}\right) = \frac{1}{2^{52}}$$

et donc
$$|x - x'| \leqslant 2^{\mathrm{E}-1023-52}$$

Exercice : Écrire un programme en langage python qui détecte la plus petite puissance de 2 pouvant s'écrire en format flottant (il s'agit d'un nombre dénormalisé).

Corrigé : On exécute le code suivant :

```
a=1
n=0
while a>0:
    n+=1
    a/=2
```

et on trouve

```
>>> n
1075
```

Ainsi, la plus petite puissance de 2 accessible en format flottant est 2^{-1074}, soit une puissance beaucoup plus petite que celle prévue par la norme IEEE qui devrait être 2^{-1023}. En fait, le 1 de la mantisse interdit toute représentation flottante entre 0 et 2^{-1023}. Pour accéder à de plus petites puissances, on a dénormalisé l'écriture avec

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

d'où un minimum de $2^{-52+1-1023} = 2^{-1074}$ que l'on peut vérifier expérimentalement.

```
>>> 2**(-1074)
5e-324
>>> 2**(-1075)
5e-324
```