

Représentation des nombres

Dans ce chapitre, nous allons nous intéresser à la façon dont un nombre (entier ou réel) peut être représenté à l'intérieur d'un ordinateur. Nous savons déjà¹ que la mémoire des ordinateurs est découpée en blocs de 8 bits (un octet) et qu'un processeur 64 bits va travailler avec des paquets de 8 octets (contre 4 pour les processeurs 32 bits). Nous disposons de 64 bits pour représenter un nombre ; il est donc naturel de faire intervenir la décomposition de ce dernier en base 2 pour le représenter en mémoire.

1. Représentation dans une base

Nous sommes habitués depuis l'enfance à utiliser l'écriture en base 10 des entiers : par exemple, 2985 représente le nombre $2 \times 10^3 + 9 \times 10^2 + 8 \times 10 + 5 \times 10^0$. Mais plus généralement, pour tout entier $b \geq 2$ on peut définir la représentation en base b d'un entier en convenant que l'écriture $(a_p a_{p-1} \dots a_0)_b$ représente le nombre :

$$a_p \times b^p + a_{p-1} \times b^{p-1} + \dots + a_1 \times b + a_0.$$

Pour s'assurer de l'unicité de l'écriture d'un entier dans une base donnée, il est nécessaire en outre d'imposer : $\forall k \in \llbracket 0, p \rrbracket, a_k \in \llbracket 0, b-1 \rrbracket$ et $a_p \neq 0$. Ainsi, en base 3 par exemple, seuls les chiffres 0, 1 et 2 seront utilisés, et le nombre $(210122)_3$ représente l'entier $2 \times 3^5 + 3^4 + 3^2 + 2 \times 3 + 2$, c'est à dire 584^2 .

Écriture en base 2

Dans le cas particulier de la base 2, seuls les chiffres 0 et 1 sont donc utilisés, ce qui rend les opérations usuelles particulièrement simples à réaliser. En effet, les algorithmes de calcul appris à l'école primaire en base 10 (addition, soustraction, multiplication, division) se généralisent en base 2.

$$\begin{array}{r}
 1011011 \\
 + 101001 \\
 \hline
 10000100
 \end{array}
 \qquad
 \begin{array}{r}
 1010 \\
 \times 101 \\
 \hline
 1010 \\
 10100 \\
 \hline
 110010
 \end{array}$$

FIGURE 1 – Un exemple d'addition et de multiplication en base 2

Exercice 1 Réaliser les opérations suivantes en base 2, sans passer par la base 10, à l'aide des algorithmes appris à l'école primaire :

$$(101010)_2 + (11000)_2$$

$$(110101)_2 - (11001)_2$$

$$(11101)_2 \times (1011)_2$$

$$(1100101)_2 \div (1011)_2$$

Changement de base

Pour convertir le nombre $x = (a_p a_{p-1} \dots a_1 a_0)_b$ en base 10, il suffit d'appliquer la formule $x = \sum_{k=0}^p a_k b^k$. Ceci peut être aisément effectué à l'aide du script PYTHON suivant :

```
def base10(x, b=2):
    s = 0
    k = len(x) - 1
    for a in x:
        s += int(a) * b ** k
        k -= 1
    return s
```

1. voir le chapitre 1.

2. on conviendra que par défaut les nombres sont écrits en base 10 ; ainsi on écrira 584 en lieu et place de $(584)_{10}$.

Attention, pour définir un nombre dans une base *non décimale* on ne peut utiliser le type `int` car tout nombre entré au clavier est implicitement supposé écrit en base 10. C'est la raison pour laquelle, dans cette fonction, les nombres que l'on souhaite convertir sont représentés par une chaîne de caractères, et la fonction `int` permet de les convertir en leur équivalent numérique. Cette fonction ne permet donc pas d'utiliser des bases supérieures à 10 puisque seuls les caractères de '0' à '9' peuvent être convertis en leur équivalent numérique.

Par exemple :

```
In [1]: base10('210122', b=3)
Out[1]: 584

In [2]: base10('1011011')
Out[2]: 91
```

On peut observer que la réponse numérique obtenue, *indépendamment de sa représentation machine dont on parlera plus tard*, est implicitement représentée en base 10 dans la console. Pour des raisons évidentes, l'interaction numérique entre la machine et l'utilisateur se fait en base 10.

Schéma de HORNER

Il est possible d'effectuer ce calcul sans calcul de puissance en appliquant la méthode de HORNER. Cette dernière consiste à itérer la suite finie (u_0, u_1, \dots, u_p) définie par :

$$u_0 = (a_p)_b, \quad u_1 = (a_p a_{p-1})_b, \quad \dots \quad u_k = (a_p a_{p-1} \dots a_{p-k})_b, \quad \dots \quad u_p = (a_p a_{p-1} \dots a_0)_b = x.$$

Les termes de cette suite sont liés par la récurrence $u_k = b u_{k-1} + a_{p-k}$, ce qui conduit à la définition alternative (et préférable) suivante :

```
def base10(x, b=2):
    u = 0
    for a in x:
        u = b * u + int(a)
    return u
```

Conversion réciproque

À l'inverse, pour convertir un entier écrit en base 10 en une base quelconque, il faut observer que si $x = (a_p a_{p-1} \dots a_1 a_0)_b$ alors le quotient de la division euclidienne de x par b est égal à $q = (a_p a_{p-1} \dots a_1)_b$ et le reste à $r = a_0$ puisque $x = bq + r$ avec $0 \leq r \leq b - 1$. Ceci conduit à la fonction suivante :

```
def baseb(x, b=2):
    s = ''
    y = x
    while y > 0:
        s = str(y % b) + s
        y //= b
    return s
```

Par exemple :

```
In [3]: baseb(584, b=3)
Out[3]: '210122'

In [4]: baseb(91)
Out[4]: '1011011'
```

Le cas particulier de la base 16

Il a été dit plus haut que l'interaction homme/machine se fait implicitement en base 10. Il y a néanmoins deux exceptions. Il est possible d'introduire directement au clavier un nombre écrit en base 2 ; il suffit de le faire précéder des caractères `0b` :

```
In [5]: 0b1011011
Out[5]: 91
```

La seconde exception est la base 16, en faisant précéder le nombre des caractères 0x :

```
In [6]: 0xa27c
Out[6]: 41596
```

En effet, $10 \times 16^3 + 2 \times 16^2 + 7 \times 16 + 12 = 41596$.

La raison du rôle particulier que joue la base 16 en informatique provient du fait que l'écriture binaire d'un nombre présente l'inconvénient d'être très longue à écrire, ce qui a incité les informaticiens à écrire ces nombres dans une base plus élevée. Le choix de la base 10 pourrait paraître naturel, mais malheureusement convertir un nombre de la base 10 à la base 2 ou inversement n'est pas chose facile. En revanche, nous allons voir que passer de la base 2 à la base 16 est très simple à réaliser.

Pour écrire un nombre en base 16, nous avons besoin d'un caractère pour chacun des entiers de 0 à 15 ; on complète donc les chiffres de 0 à 9 par les lettres a, b, c, d, e et f. Ainsi, $(a)_{16} = 10$, $(b)_{16} = 11$, $(c)_{16} = 12$, $(d)_{16} = 13$, $(e)_{16} = 14$, $(f)_{16} = 15$.

Sachant que $2^4 = 16$, tout nombre écrit en base 2 à l'aide de 4 chiffres s'écrit en base 16 à l'aide d'un seul chiffre :

$(0000)_2 = (0)_{16}$	$(0100)_2 = (4)_{16}$	$(1000)_2 = (8)_{16}$	$(1100)_2 = (c)_{16}$
$(0001)_2 = (1)_{16}$	$(0101)_2 = (5)_{16}$	$(1001)_2 = (9)_{16}$	$(1101)_2 = (d)_{16}$
$(0010)_2 = (2)_{16}$	$(0110)_2 = (6)_{16}$	$(1010)_2 = (a)_{16}$	$(1110)_2 = (e)_{16}$
$(0011)_2 = (3)_{16}$	$(0111)_2 = (7)_{16}$	$(1011)_2 = (b)_{16}$	$(1111)_2 = (f)_{16}$

Aussi, pour convertir un nombre quelconque de la base 2 à la base 16, il suffit de regrouper les chiffres qui le composent par paquet de 4 et convertir chacun de ces paquets en un chiffre en base 16. Par exemple, $(1011\ 0110\ 1110\ 1001)_2 = (b6e9)_{16}$.

L'écriture hexadécimale (c'est-à-dire en base 16) est fréquemment utilisée en informatique car un octet (qui rappelons le vaut 8 bits) sera toujours représenté par deux caractères hexadécimaux. Par exemple, une couleur d'une page web est définie par trois octets représentant ses composantes RVB³. Ainsi, un navigateur web va interpréter le code couleur $(ffa500)_{16}$ comme du orange, $(00ff00)_{16}$ comme du vert, ou encore $(ee82ee)_{16}$ comme du violet. Potentiellement, $256^3 = 16\,777\,216$ couleurs différentes sont accessibles.

Au vu de l'importance de la base 2 et de la base 16 en informatique, il existe deux fonctions qui réalisent la conversion vers la base 2 et vers la base 16 : les fonctions `bin` et `hex`. Ces fonctions prennent en argument un objet de type `int` et retournent une chaîne de caractères (précédée des caractères 0b ou 0x).

```
In [7]: bin(41397)
Out[7]: '0b1010000110110101'

In [8]: hex(41397)
Out[8]: '0xa1b5'

In [9]: 0b1010000110110101 + 0xa1b5
Out[9]: 82794
```

2. Codification des nombres entiers

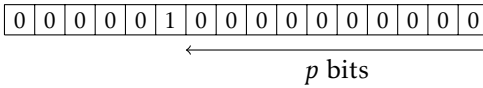
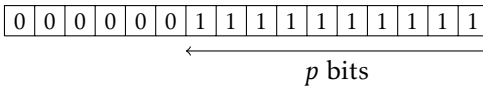
Pour pouvoir être stocké et manipulé par un ordinateur, un nombre doit être représenté par une succession de bits. Le principal problème est la limitation de la taille du codage : un nombre mathématique peut prendre des valeurs arbitrairement grandes, tandis que le codage dans l'ordinateur doit s'effectuer sur un nombre de bits fixé.

2.1 Les entiers naturels

Les entiers naturels sont essentiellement utilisés pour représenter les adresses en mémoire. Un codage sur n bits permet de représenter tous les nombres naturels compris entre 0 et $2^n - 1$. Ainsi, un octet permet de coder les entiers allant de $0 = (00)_{16} = (0000\ 0000)_2$ à $255 = (ff)_{16} = (1111\ 1111)_2$, et 64 bits (soit 8 octets) tous les nombres allant de $0 = (0000\ 0000\ 0000\ 0000)_{16}$ à $2^{64} - 1 = (ffff\ ffff\ ffff\ ffff)_{16}$.

3. les composantes Rouge Vert Bleu en synthèse additive.

Deux valeurs particulières demandent à être bien connues, la représentation des entiers de la forme 2^p et ceux de la forme $2^p - 1$:

- l'entier naturel 2^p est représenté par 
- l'entier naturel $2^p - 1$ est représenté par 

2.2 Les entiers relatifs

Pour représenter un entier relatif il est nécessaire de coder le signe de ce dernier sur un bit (0 pour les nombres positifs et 1 pour les nombres négatifs) ; ainsi le processeur va réserver le premier bit pour le signe, et les $n - 1$ autres bits pour l'entier à proprement parler. Le plus grand entier relatif positif représentable sur n bits est donc égal à $2^{n-1} - 1$. En base 2, il s'écrit : $(\underbrace{01111111 \dots 1111}_{n-1 \text{ chiffres } 1})_2$.

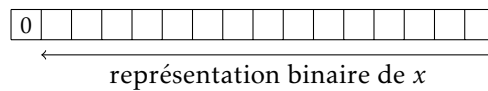


FIGURE 2 – Représentation machine d'un entier relatif positif.

Sur une architecture 64 bits le plus grand entier relatif est donc égal à $2^{63} - 1$, qu'on peut écrire en base 16 sous la forme $(\underbrace{7\text{fff}\dots\text{fff}}_{15 \text{ chiffres } f})_{16}$. En base 10, il vaut :

```
In [1]: 0x7fffffffffffffffff
Out[1]: 9223372036854775807
```

Le codage des nombres négatifs

On pourrait s'attendre à ce que la représentation d'un entier relatif négatif n soit un 1 (le bit de signe) suivi de la représentation binaire de $|n|$. *Il n'en est rien*, car cette méthode possède plusieurs inconvénients :

- Le nombre 0 posséderait deux représentations (et il est toujours préférable d'avoir unicité de la représentation d'un objet) ;
- L'algorithme d'addition ne s'applique qu'à des entiers de même signe et cette représentation le rendrait inapplicable pour additionner des entiers relatifs.

C'est pourquoi on utilise le codage particulier pour représenter les entiers négatifs, dit en *complément à deux* :

le nombre négatif x est représenté en mémoire par la représentation binaire de l'entier (positif) $2^n + x$.

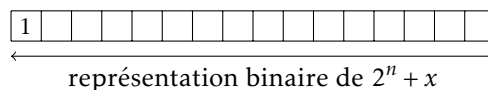


FIGURE 3 – Représentation machine d'un entier relatif négatif.

Pour que cette représentation sur n bits commence par un 1, il est donc nécessaire que $2^n + x$ soit compris entre $2^{n-1} = (\underbrace{1\ 000\ 0000 \dots 0000}_{n-1 \text{ chiffres}})_2$ et $2^n - 1 = (\underbrace{1\ 111\ 1111 \dots 1111}_{n-1 \text{ chiffres}})_2$ soit :

$$2^{n-1} \leq 2^n + x \leq 2^n - 1 \iff -2^{n-1} \leq x \leq -1$$

Le plus petit entier négatif représentable sur une architecture à n bits est donc égal à -2^{n-1} et est représenté par : $(\underbrace{1\ 000 \dots 0000}_{n-1 \text{ bits}})_2$.

Ainsi, les entiers relatifs représentables sur une architecture à n bits sont compris entre -2^{n-1} et $2^{n-1} - 1$.

Attention donc à bien distinguer un nombre de sa représentation, qui ne coïncide pas avec sa représentation binaire dans le cas des nombres négatifs.

Exemple. Pour simplifier les calculs, considérons une représentation sur 8 bits ($n = 8$).

L'entier relatif 105 est représenté par l'octet 01101001, ce qui correspond à la représentation en base 2 de l'entier $105 = 2^6 + 2^5 + 2^3 + 2^0$.

L'entier relatif -105 est représenté par l'octet 10010111, ce qui correspond à la représentation en base 2 de l'entier $256 - 105 = 151 = 2^7 + 2^4 + 2^2 + 2^1 + 2^0$.

Exercice 2 Dans une représentation en complément à deux sur 8 bits, quels sont les entiers relatifs représentés par 01101101 et par 10010010 ?

nombre	représentation
0	0000.....0000
1	0000.....0001
...
$2^{n-1} - 1$	0111.....1111
-2^{n-1}	1000.....0000
...
-1	1111.....1111

FIGURE 4 – Les entiers relatifs codés sur n bits, classés par représentation croissante.

Addition de deux entiers relatifs

La première conséquence de la représentation par complément à deux est que tout entier de l'intervalle $\llbracket -2^{n-1}, 2^{n-1} - 1 \rrbracket$ possède une *unique* représentation. Mais l'intérêt majeur n'est pas là : nous allons le constater qu'*additionner deux nombres relatifs revient à additionner leurs représentations* en ne gardant que les n bits de poids faible.

Avant de le justifier, illustrons ce résultat en calculant la somme de -91 et de 113 avec un codage sur 8 bits (on a $2^8 = 256$; les nombres représentables sont compris entre -128 et 127).

-91 est négatif donc il est représenté par son complément à deux $256 - 91 = 165 = (10100101)_2$.

113 est positif donc il est représenté par sa décomposition en base 2 : $113 = (01110001)_2$.

On additionne ces deux représentations :

$$\begin{array}{r}
 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1 \\
 +\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1 \\
 \hline
 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0
 \end{array}$$

On ne garde que les 8 derniers bits, à savoir 00010110 ; le premier bit est un 0 donc le résultat de la somme est positif ; il représente donc l'entier $(10110)_2 = 22$, qui est bien le résultat de l'addition de -91 avec 113.

Justifions maintenant cette méthode en envisageant quatre cas lorsqu'on additionne deux entiers relatifs a et b . Nous allons supposer que $a + b$ est représentable en machine, c'est-à-dire que $a + b \in \llbracket -2^{n-1}, 2^{n-1} - 1 \rrbracket$:

- si $a > 0$ et $b > 0$ on calcule $(a + b)$;
- si $a > 0$ et $b < 0$ on calcule $a + (2^n + b) = (a + b) + 2^n$;
 - si $a + b \geq 0$ ce nombre s'écrit sur $n + 1$ bit donc est tronqué : on obtient la représentation de $a + b$;
 - si $a + b < 0$ ce nombre n'est pas tronqué : on obtient la représentation de $a + b$;
- si $a < 0$ et $b > 0$ la situation est identique au cas précédent ;
- si $a < 0$ et $b < 0$ on calcule $(2^n + a) + (2^n + b) = 2^n + (2^n + a + b)$; ce nombre s'écrit sur $n + 1$ bits donc est tronqué : il reste $2^n + a + b$ qui est la représentation de $a + b$.

Soustraction de deux entiers relatifs

Calculer la soustraction $a - b$ ça n'est jamais qu'additionner a et $-b$; s'il est possible de calculer facilement l'opposé avec la représentation en complément à deux nous pourrons ramener toute soustraction à une addition. Considérons donc un entier x et considérons une fois de plus deux cas :

- si $x \geq 0$ il est représenté par l'entier naturel x et son opposé $-x$ est représenté par l'entier naturel $2^n - x$.
Notons $x = (0x_{n-2} \cdots x_1 x_0)_2$ et posons $y = (1y_{n-2} \cdots y_1 y_0)_2$ en convenant que $y_i = 1 - x_i$. Alors $x + y = (11 \cdots 11)_2 = 2^n - 1$ donc $2^n - x = y + 1$.
- si $x < 0$ il est représenté par l'entier naturel $2^n + x$ et son opposé $-x$ est représenté par l'entier naturel $-x$.
Notons $2^n + x = (1x_{n-2} \cdots x_1 x_0)_2$ et posons $y = (0y_{n-2} \cdots y_1 y_0)_2$ en convenant que $y_i = 1 - x_i$. Alors $2^n + x + y = (11 \cdots 11)_2 = 2^n - 1$ donc $-x = y + 1$.

Dans les deux cas, nous constatons que pour obtenir la représentation de l'opposé de x il suffit de procéder ainsi :

- (i) considérer la représentation de x et remplacer tous les bits égaux à 0 par des 1 et réciproquement ;
- (ii) additionner 1 au résultat obtenu.

Exemples. Illustrons cette démarche avec un codage sur 8 bits.

$x = 113$ est représenté par 01110001 ; on a donc $y = 10001110$ et $-x$ est représenté par $y + 1 = 10001111$.
En effet, $256 - 113 = 143 = (10001111)_2$.

$x = -91$ est représenté par 10100101 ; on a donc $y = 01011010$ et $-x$ est représenté par $y + 1 = 01011011$.
En effet, $91 = (01011011)_2$.

Dépassement de capacité

Nous venons de voir que si un processeur alloue n bits à la représentation des entiers relatifs, les entiers représentables appartiennent à l'intervalle $\llbracket -2^{n-1}, 2^{n-1} - 1 \rrbracket$. Or si on effectue des opérations sur ces entiers il est fort possible que le résultat du calcul n'appartienne plus à cet intervalle.

Observons ce qui se passe lorsqu'une addition dépasse la borne supérieure en considérant l'addition de 72 et de 55 avec un codage sur 8 bits :

$72 = (1001000)_2$ est représenté par 01001000 . $59 = (111011)_2$ est représenté par 00111011 .

Effectuons l'addition de ces deux représentations :

$$\begin{array}{r} 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0 \\ +\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \end{array}$$

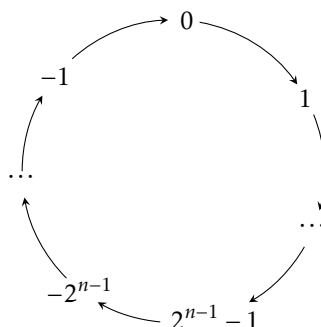
On obtient le nombre négatif représenté par 10000011 c'est à dire -125 , alors que $72 + 59 = 131$. Ce n'est pas anormal puisque sur 8 bits les seuls entiers relatifs représentables sont compris entre -128 et $+127$, ce qui n'est pas le cas de 131.

On peut observer que le résultat obtenu, -125 , est égal à $131 - 256$; ce n'est pas un hasard et ceci résulte de la propriété suivante :

THÉORÈME. — Dans une représentation en complément à deux sur n bits, le successeur de $2^{n-1} - 1$ est égal à -2^{n-1} .

Preuve. Ceci est immédiat si on se souvient que $2^{n-1} - 1$ est représenté par $0111 \dots 1111$ et -2^{n-1} par $1000 \dots 0000$. \square

Autrement dit, les entiers relatifs représentés par complémentation à deux sont pourvus d'une structure cyclique, que l'on peut représenter par le schéma ci-dessous, les flèches désignant le successeur d'un entier :



Certains langages de programmation déterminent un éventuel dépassement de capacité mais d'autres, pour des raisons d'efficacité⁴, ne le font pas. C'est par exemple les cas du langage CAML utilisé par l'option informatique :

```
# max_int ;;
- : int = 4611686018427387903

# min_int ;;
- : int = -4611686018427387904

# max_int + 1 ;;
- : int = -4611686018427387904
```

En CAML les entiers sont codés en complément à deux sur 63 bits. Nous avons :

$$2^{62} - 1 = 4611686018427387903 \quad \text{et} \quad -2^{62} = -4611686018427387904.$$

Entiers de taille arbitraire

Quant est-il de PYTHON ? Pour y répondre, rien de plus simple, il suffit de regarder quel est le successeur du plus grand entier représentable en complément à deux sur 64 bits :

```
In [2]: x = 0x7fffffffffffffff

In [3]: x
Out[3]: 9223372036854775807

In [4]: x + 1
Out[4]: 9223372036854775808
```

Que s'est-il passé ? Contrairement au langage CAML, l'interprète PYTHON détecte le débordement de capacité : lorsque le résultat d'un calcul dépasse le plus grand entier représentable en complément à deux (à savoir $2^{63} - 1$) la représentation des entiers change pour adopter la représentation sous forme dite des *entiers longs*.

Contrairement au complément à deux, cette représentation n'est pas limitée en taille. Cela présente bien évidemment des avantages, mais aussi des inconvénients : les opérations sur les entiers longs prennent plus de temps que sur les entiers classiques. Nous n'aborderons pas le problème de la représentation interne des entiers longs, plus complexe que la représentation par complément à deux.

3. Codification des nombres décimaux

3.1 Les nombres flottants

Rappelons qu'un nombre décimal est un rationnel qui peut s'écrire sous la forme $\frac{x}{10^n}$ où x est un entier relatif.

Si on considère la représentation en base 10 de l'entier $x = \pm(a_p a_{p-1} \dots a_1 a_0)_{10}$, on obtient l'écriture décimale de $\frac{x}{10^n}$ en plaçant une virgule séparant les n chiffres les plus à droite des autres : $\frac{x}{10^n} = \pm(a_p \dots a_n, a_{n-1} \dots a_0)_{10}$.

Par exemple, $\frac{25}{4}$ est un nombre décimal car il peut aussi s'écrire $\frac{625}{100}$, et son écriture décimale est 6,25.

De la même façon, un nombre *dyadique* est un rationnel qui peut s'écrire sous la forme $\frac{x}{2^n}$ où x est un entier relatif, et son développement dyadique s'obtient en plaçant une virgule séparant les n chiffres les plus à droite des autres.

Par exemple, $\frac{25}{4}$ est un nombre dyadique et son développement dyadique est $(110,01)_2$. En effet,

$$25 = (11001)_2 \quad \text{et} \quad \frac{25}{4} (110,01)_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2} = 6,25.$$

On s'en doute, contrairement aux humains les ordinateurs ne vont pas manipuler des nombres décimaux mais des nombres dyadiques. Ce n'est pas un problème quand il s'agit d'entiers puisque dans ce cas la conversion

4. C'est le cas en général des langages compilés.

binaire/décimal est exacte, mais cela pose un problème pour les nombres décimaux car *le développement dyadique d'un nombre décimal peut être infini*⁵.

Par exemple, le nombre 0,1 a une représentation décimale finie et une représentation dyadique infinie :

$$0,1 = (0,00011001100110011001100110011001100110011001100110011\cdots)_2$$

Ainsi, sa représentation machine sera nécessairement tronquée et ne correspondra pas exactement au nombre 0,1 (mais en sera néanmoins très proche).

La conversion d'un nombre décimal en nombre dyadique va donc souvent provoquer une *approximation* qui dans certains cas conduit à des résultats qui peuvent paraître étranges à un utilisateur non averti. Par exemple :

```
In [1]: 0.1 + 0.2
Out[1]: 0.30000000000000004

In [2]: (0.1 + 0.2) - 0.3
Out[2]: 5.551115123125783e-17
```

De ceci il faudra retenir qu'un calcul sur des nombres décimaux sera toujours entaché d'une certaine marge d'erreur dont il faudra tenir compte, avec une conséquence importante :

L'égalité entre nombres flottants n'a pour ainsi dire aucun sens.

```
In [3]: 0.1 + 0.2 == 0.3
Out[3]: False
```

Quand on utilise le type *float*, il est rarissime (hormis dans un but pédagogique) d'utiliser l'égalité de valeur `==` ; on fera toujours intervenir une marge d'erreur (absolue ou relative).

```
In [4]: abs(0.1 + 0.2 - 0.3) <= 1e-10
Out[4]: True
```

(On a choisi ici une marge d'erreur absolue en considérant que toute quantité inférieure ou égale à 10^{-10} est nulle.)

La norme IEEE-754

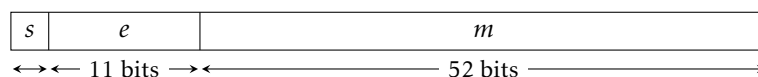
Cette norme est actuellement le standard pour la représentation des nombres à virgule flottante en binaire. Nous allons en donner une description (incomplète) pour une architecture 64 bits.

Un nombre dyadique non nul possède une représentation normalisée de la forme $\pm(1, b_1 \cdots b_k)_2 \times 2^e$, où e est un entier relatif. Par exemple, $6,25 = (110,01)_2$ a pour représentation normalisée $(1,1001)_2 \times 2^2$ et $-0,375 = -(0,011)_2$ la représentation normalisée $-(1,1)_2 \times 2^{-2}$.

La suite de bits $b_1 \cdots b_k$ est appelée la *mantisse* du nombre, et la puissance de 2, l'*exposant*.

Dans cette norme, les nombres dyadiques sont codés sur 64 bits en réservant :

- 1 bit pour le signe ;
- 11 bits pour l'exposant ;
- 52 bits pour la mantisse.



L'exposant est un entier relatif, mais pour permettre une comparaison plus aisée des nombres flottants, il n'est pas codé suivant la technique de complément à deux mais suivant la technique du décalage : l'exposant e est représenté en machine par l'entier positif $e' = e + 2^{10} - 1$.

Un entier naturel codé sur 11 bits est compris entre 0 et $2^{11} - 1$ donc *a priori* :

$$0 \leq e' \leq 2^{11} - 1 \iff 1 - 2^{10} \leq e \leq 2^{10} \quad \text{soit} \quad -1023 \leq e \leq 1024.$$

Cependant, les valeurs extrêmes ($e' = 0$ et $e' = 2^{11} - 1$) sont réservées à la représentation de certaines valeurs particulières, comme nous allons le voir maintenant.

5. En revanche, la conversion réciproque ne pose pas de problème, puisque tout nombre dyadique est aussi décimal.

3.2 Calculs sur les nombres flottants

La représentation interne des nombres flottants a une incidence sur les opérations que l'on est susceptible de demander à l'ordinateur, et provoque quelques désagréments dont il faut avoir conscience. Sans prétendre à l'exhaustivité, nous allons passer en revue quelques-unes des conséquences de cette représentation.

Erreurs d'arrondis dues aux changement de bases

Nous l'avons déjà constaté, la conversion d'un nombre décimal en nombre dyadique et réciproquement provoque des erreurs d'arrondis qui ne sont pas sans conséquences :

```
In [5]: (0.1 + 0.2) - 0.3
Out[5]: 5.551115123125783e-17
```

Aucun des trois nombres 0,1, 0,2 et 0,3 n'est un nombre dyadique, donc leurs représentations machine n'est qu'approximative.

Évidemment, un calcul isolé ne produira pas d'erreur importante, mais il peut en être autrement lorsqu'une longue suite de calculs provoque un cumul des erreurs d'arrondi (le fameux « effet papillon » dans l'étude de phénomènes chaotiques).

Avant de poursuivre, nous allons tenter de décortiquer le calcul ci-dessus.

Lors de la conversion décimal \rightarrow dyadique la règle est d'arrondir au plus proche⁶. Par exemple, la représentation dyadique de 0,1 est infinie : $0,1 = (0,0001\overline{1001}\dots)_2$ (le motif $\overline{1001}$ se répète infiniment) donc la représentation machine de 0,1 est :

$$1, \underbrace{1001\ 1001 \dots 1001}_{48 \text{ bits}} 1010 \times 2^{-4}$$

Sans surprise, la représentation machine de 0,2 est le double de celle-ci, à savoir :

$$1, \underbrace{1001\ 1001 \dots 1001}_{48 \text{ bits}} 1010 \times 2^{-3}$$

Pour additionner ces deux quantités, la plus petite des deux voit ses bits décalés d'un cran : 0,1 est provisoirement représenté par $0,1\ 1001\ 1001 \dots 1001\ 101 \times 2^{-3} = 0, \underbrace{11001100 \dots 1100}_{48 \text{ bits}} 1101 \times 2^{-3}$.

On réalise l'addition de ces deux nombres dyadiques pour obtenir :

$$10, \underbrace{01100110 \dots 0110}_{48 \text{ bits}} 0111 \times 2^{-3}$$

Ce résultat est enfin normalisé (donc arrondi au plus proche), ce qui donne : $1, \underbrace{0011\ 0011 \dots 0011}_{48 \text{ bits}} 0100 \times 2^{-2}$.

Sachant que 0,3 possède la représentation dyadique infinie : $(0,01\overline{0011}\dots)_2$ sa représentation machine est $1, \underbrace{0011\ 0011 \dots 0011}_{48 \text{ bits}} 0011 \times 2^{-2}$, ce qui explique que le calcul $0.1 + 0.2 - 0.3$ ne donne pas 0 mais le nombre

$$0, \underbrace{0000\ 0000 \dots 0000}_{48 \text{ bits}} 0001 \times 2^{-2} = 2^{-54}. \text{ Vérifions-le :}$$

```
In [6]: 2**(-54)
Out[6]: 5.551115123125783e-17
```

C'est bien cela !

Absorption

En mathématique, l'addition est associative : $(x + y) + z = x + (y + z)$; ce n'est pas le cas pour l'addition entre nombres flottants :

```
In [7]: (1. + 2.**53) - 2.**53
Out[7]: 0.0
```

6. Plus précisément, la règle est : *round to nearest, ties to even*. En cas d'égalité, le nombre pair le plus proche est choisi.

Le résultat de ce calcul est facile à comprendre. L'écriture normalisée de $1 + 2^{53}$ est égale à :

$$(1, \underbrace{0000 \dots 0000}_{{52 \text{ fois } 0}} 1)_2 \times 2^{53}$$

mais comme la mantisse n'occupe que 52 bits, le résultat de cette addition n'est pas représentable en machine et sera approché par :

$$(1, \underbrace{0000 \dots 0000}_{{52 \text{ fois } 0}})_2 \times 2^{53}$$

Autrement dit, en arithmétique flottante, $1 + 2^{53} = 2^{53}$. Ce mécanisme porte le nom d'absorption : *l'addition de deux quantités dont l'écart relatif est très important entraîne l'absorption de la plus petite de ces deux quantités par la plus grande.*

En conséquence, il est parfois nécessaire de réorganiser les calculs pour obtenir un résultat plus conforme à nos attentes :

```
In [8]: 1. + (2.**53 - 2.**53)
Out[8]: 1.0
```

Cancellation

Un autre problème se présente lors de la soustraction de deux quantités très proches. Pour fixer les idées, supposons que l'on connaisse deux quantités voisines x et y avec une précision de 20 chiffres significatifs après la virgule :

```
x = 1,10010010000111111011
y = 1,10010010000111100110
x-y = 0,0000000000000000101
```

le résultat $x - y$ sera normalisé pour être représenté en machine par $(1,0101)_2 \times 2^{-16}$; autrement dit, la précision ne sera plus que de 4 chiffres après la virgule. Cette perte drastique de précision porte le nom de cancellation, car *la différence de deux quantités très voisines fait littéralement s'évanouir les chiffres significatifs.*

Prendre conscience de ce phénomène permet de conditionner un calcul pour le rendre moins sensible à l'évanescence des chiffres significatifs. Par exemple, pour un calcul numérique il est préférable de calculer $\frac{1}{x(x+1)}$ plutôt que $\frac{1}{x} - \frac{1}{x+1}$ car, bien que ces deux quantités soient mathématiquement égales, la seconde est beaucoup plus sensible au phénomène de cancellation.

En règle générale, lors d'un calcul numérique il faut suivre les recommandations suivantes :

- on évite d'additionner deux quantités dont l'écart relatif est très grand ;
- on évite de soustraire deux quantités très voisines.

Et pour finir...

Il ne faudrait surtout pas croire que ces petites erreurs de calcul et d'arrondi soient négligeables, et l'anecdote suivante devrait vous convaincre de l'importance qu'il y a à en prendre conscience.

Le 25 février 1991, à Dharan en Arabie Saoudite, un missile Patriot américain a raté l'interception d'un missile Scud irakien, ce dernier provoquant la mort de 28 personnes. La commission d'enquête chargée de comprendre la raison de cet échec a mis en évidence le défaut suivant :

L'horloge interne du missile Patriot mesure le temps en 1/10s. Pour obtenir le temps en seconde, le système multiplie ce nombre par 10 en utilisant un registre de 24 bits en virgule fixe. Or 1/10 n'est pas un nombre dyadique donc a été arrondi : le registre de 24 bits contient $(0,00011001100110011001100)_2$ et induit une erreur binaire de $(0,0000000000000000000000011001100\dots)_2$, soit approximativement 0,000000095 en notation décimale.

En multipliant cette quantité par le nombre de 1/10s pendant 100h (le temps écoulé entre la mise en marche du système et le lancement du missile Patriot), on obtient le décalage entre l'horloge interne de missile et le temps réel, soit :

$$0,000000095 \times 100 \times 3600 \times 10 \approx 0,34s.$$

Or un missile Scud vole à la vitesse approximative de 1,676m/s donc parcourt plus de 500m en 0,34s, ce qui le fait largement sortir de la zone d'acquisition de sa cible par le missile d'interception ⁷.

7. Référence : <http://ta.twi.tudelft.nl/nw/users/vuik/wi211/disasters.html>.