Chapitre $\underline{1000}_2$
Représentation des nombres

5 décembre 2017

1 Base de numération

1.1 Rappel de CP : la base dix

Chiffre : symbole utilisé pour représenter certains entiers.
Les chiffres «usuels» : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Le nombre dix joue un rôle particulier.

— C'est le plus petit entier naturel non représentable uniquement par un chiffre.

— Pour compter des objets en grand nombre, on les regroupe par paquets de dix.

Exemple : Pour compter ||||||||||||||||||||||||, on obtient

|||||||||| |||||||||| ||||

Deux paquets (deux dizaines), reste quatre unités.

Quand il y a trop de dizaines, on regroupe les dizaines par paquets de dix (centaines), les centaines par paquets de dix (milliers), etc.

1.2 Numération de position en base 10

On décompose un entier en dizaines, centaines, milliers, etc. L'essentiel est alors qu'il y ait strictement moins de dix éléments dans chaque type de paquet. Ce nombre d'éléments peut être représenté par un chiffre. On écrit alors tous les chiffres à la suite. À gauche, on place les *chiffres de poids fort* (gros paquets). À droite, les *chiffres de poids faible*.

Ainsi 2735 représente deux milliers plus sept centaines plus trois dizaines plus cinq unités.

De manière générale, avec $B = 10$ et $n \in \mathbb{N}$,

$$\underline{a_n a_{n-1} \dots a_1 a_0}_{\ B} = \sum_{k=0}^{n} a_k B^k, \quad \text{et} \quad \forall k \in [\![0; n]\!],\ a_k \in [\![0; B[\![.$$

1.3 Pourquoi dix ?

Pourquoi regrouper par dix pas plutôt par deux ? ou trois ? ou six ? ou huit ?

— Raison anthropomorphique (dix doigts) et poids de l'histoire.

— À peu près aucune raison mathématique.

On peut choisir une autre base.

— Deux : Amérique du Sud et Océanie.

— Cinq : Afrique, Romains et Maya (partiellement).

— Six : Papouasie Nouvelle-Guinée.

— Huit : certains dialectes amérindiens (Pame, Mexique ; Yuki, Californie), proposition de Charles XII de Suède.

— Douze : Népal, Europe.

— Vingt : Bhoutan, Aztèques, Maya, Gaulois (?), Basques (?).

— Soixante : Babyloniens, Indiens et Arabes (trigo)

(source : Wikipédia, article *Numération*)

À chaque fois, le principe est identique : on change juste B dans l'écriture précédente.

1.4 Un premier exemple : la base huit (octale)

En pratique, on ne l'utilisera pas ...

On a besoin de huit symboles pour représenter les huit chiffres (représentant les nombres de zéro inclus à huit exclu). On prendra naturellement $0, \ldots, 7$. Pour compter, on regroupe les unités par huitaine puis par huitaines de huitaines, etc. On note les nombres sur le même principe que l'écriture décimale : $\underline{2735}_8$ représente cinq unités plus trois huitaines plus sept huitaines de huitaines, plus deux huitaines de huitaines de huitaines (soit 1501, exprimé ici en base 10 !).

De manière générale, avec $B = 8$ et $n \in \mathbb{N}$,

$$\underline{a_n a_{n-1} \ldots a_1 a_0}_{\ B} = \sum_{k=0}^{n} a_k B^k, \quad \text{et} \quad \forall k \in [\![0; n]\!],\ a_k \in [\![0; B[\![.$$

Vous remarquerez la forte similitude avec la formule précédente ...

Comptons en octal :

$\underline{0}_8 = 0$	$\underline{10}_8 = 8$	$\underline{20}_8 = 16$	$\underline{30}_8 = 24$	$\underline{100}_8 = 64$
$\underline{1}_8 = 1$	$\underline{11}_8 = 9$	$\underline{21}_8 = 17$	$\underline{40}_8 = 32$	$\underline{200}_8 = 128$
$\underline{2}_8 = 2$	$\underline{12}_8 = 10$	$\underline{22}_8 = 18$	$\underline{50}_8 = 40$	$\underline{1\,000}_8 = 512$
$\underline{3}_8 = 3$	$\underline{13}_8 = 11$	$\underline{23}_8 = 19$	$\underline{60}_8 = 48$	$\underline{10\,000}_8 = 4096$
$\underline{4}_8 = 4$	$\underline{14}_8 = 12$	$\underline{24}_8 = 20$	$\underline{70}_8 = 56$	$\underline{100\,000}_8 = 32768$
$\underline{5}_8 = 5$	$\underline{15}_8 = 13$	$\underline{25}_8 = 21$		
$\underline{6}_8 = 6$	$\underline{16}_8 = 14$	$\underline{26}_8 = 22$		
$\underline{7}_8 = 7$	$\underline{17}_8 = 15$	$\underline{27}_8 = 23$		

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

Table d'addition en octal

*	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

Table de multiplication en octal

Comment faire une addition de nombres à plusieurs chiffres ?
Exactement comme en base 10, mais en utilisant la table d'addition de la base 8.
Pour la multiplication : aussi.

1.5 Une base (plus) utile : la base seize (hexadécimale)

On reprend le même principe que précédemment, avec $B = 16$ (on forme des paquets de 16 etc.). Mais cette fois on manque de chiffres pour représenter les nombres de zéro

inclus à seize exclu (il en manque six).

On rajoute de nouveaux « chiffres » :

a dix

b onze

c douze

d treize

e quatorze

f quinze

On peut alors se mettre à compter en hexadécimal !

$$\underline{0}_{16} = 0 \quad \underline{10}_{16} = 16 \quad \underline{20}_{16} = 32 \quad \underline{30}_{16} = 48 \quad \underline{100}_{16} = 256$$
$$\underline{1}_{16} = 1 \quad \underline{11}_{16} = 17 \quad \underline{21}_{16} = 33 \quad \underline{40}_{16} = 64 \quad \underline{200}_{16} = 512$$
$$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \underline{1\,000}_{16} = 4096$$
$$\underline{9}_{16} = 9 \quad \underline{19}_{16} = 25 \quad \underline{29}_{16} = 41 \quad \underline{90}_{16} = 144 \quad \underline{10\,000}_{16} = 65536$$
$$\underline{a}_{16} = 10 \quad \underline{1a}_{16} = 26 \quad \underline{2a}_{16} = 42 \quad \underline{a0}_{16} = 160$$
$$\underline{b}_{16} = 11 \quad \underline{1b}_{16} = 27 \quad \underline{2b}_{16} = 43 \quad \underline{b0}_{16} = 176$$
$$\underline{c}_{16} = 12 \quad \underline{1c}_{16} = 28 \quad \underline{2c}_{16} = 44 \quad \underline{c0}_{16} = 192$$
$$\underline{d}_{16} = 13 \quad \underline{1d}_{16} = 29 \quad \underline{2d}_{16} = 45 \quad \underline{d0}_{16} = 208$$
$$\underline{e}_{16} = 14 \quad \underline{1e}_{16} = 30 \quad \underline{2e}_{16} = 46 \quad \underline{e0}_{16} = 224$$
$$\underline{f}_{16} = 15 \quad \underline{1f}_{16} = 31 \quad \underline{2f}_{16} = 47 \quad \underline{f0}_{16} = 240$$

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Table d'addition en hexadécimal

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Table de multiplication en hexadécimal

1.6 Enfin : la base deux !

C'est toujours le même principe, mais avec $B = 2$; on représente les nombres « par paquets de deux ». Pas de problème pour les chiffres, nous n'en avons besoin que de deux. Par convention : 0 et 1.

$\underline{0}_2 = 0$ $\quad$ $\underline{10}_2 = 2$ $\quad$ $\underline{100}_2 = 4$ $\quad$ $\underline{1000}_2 = 8$

$\underline{1}_2 = 1$ $\quad$ $\underline{11}_2 = 3$ $\quad$ $\underline{101}_2 = 5$ $\quad$ $\underline{1\,0000}_2 = 16$

$\underline{110}_2 = 6$ $\quad$ $\underline{10\,0000}_2 = 32$

$\underline{111}_2 = 7$ $\quad$ $\underline{100\,0000}_2 = 64$

$\underline{1000\,0000}_2 = 128$

$\underline{1\,0000\,0000}_2 = 256$

+	0	1
0	0	1
1	1	10

Table d'addition en binaire

*	0	1
0	0	0
1	0	1

Table de multiplication en binaire

1.6.1 Intérêts du binaire

Les nombres binaires sont facilement représentables par un dispositif mécanique/-électrique/électronique/optique/électromagnétique etc. De plus, les tables d'opérations très simples et sont facilement calculables par un dispositif mécanique/électrique/électronique etc.

C'est le système utilisé pour représenter les nombres en interne dans un ordinateur.

1.6.2 Écriture d'un naturel p en binaire

Donnons d'abord un algorithme par divisions successives. Si $p = \underline{a_n \dots a_0}\,_2$, alors

$$p = \sum_{k=0}^{n} a_k 2^k = 2 \sum_{k=1}^{n} a_k 2^{k-1} + a_0.$$

Ainsi, a_0 est le reste de la division euclidienne de p par 2 et, si $p \neq a_0$, $\underline{a_n \dots a_1}\,_2$ est le quotient de la division euclidienne de p par 2.

On peut donc écrire la fonction suivante.

```
def conv_b2(p):
    """Convertit l'entier p en base 2 (renvoie une chaîne)"""
    x = p # On copie p
    s = ""
    while x > 1 :
```

```
        s = str(x%2) + s
        x = x // 2
    return str(x)+s

print('0='+conv_b2(0)+' et 1='+conv_b2(1)+' et 42='+conv_b2(42))
```

Cela renvoie alors :

```
0=0 et 1=1 et 42=101010
```

Voici une autre idée : calculer 2^k pour $k = 0, \ldots$ jusqu'à avoir $2^k > p$. Alors, p s'écrit sur k bits et le bit de poids fort est 1. Le reste des bits est donné par la représentation en binaire de $p - 2^{k-1}$.

1.6.3 Calcul d'entier représenté en binaire

On veut calculer l'entier p, représenté par une suite de bits $\underline{a_n a_{n-1} \ldots a_1 a_0}_2$, *i.e.* $\sum_{k=0}^{n} a_k 2^k$.

On peut effectuer le calcul naïvement, en pensant bien à calculer les puissances de proche en proche.

```
def calc_b2_naif(s):
    """Renvoie l'entier p représenté en binaire par s"""
    p = 0
    x = 1 ## 2**0
    for i in range(len(s)):
        # Invariant : p = s[len(s)-i:]
        p = p+int(s[len(s)-i-1])*x
        x = 2*x
    return p

print(0==calc_b2_naif("0"))
print(1==calc_b2_naif("1"))
print(42==calc_b2_naif("101010"))
```

Cela renvoie alors :

```
True
True
True
```

On peut faire mieux en mettant en œuvre l'algorithme de Horner. Il suffit de remarquer que

$$p = \sum_{k=0}^{n} a_k 2^k = a_0 + 2\left(\sum_{k=1}^{n} a_k 2^{k-1}\right) = a_0 + 2\left(a_1 + 2\left(a_2 + 2\left(\ldots + 2a_n\right)\right)\right)$$

```python
def calc_b2_horner(s):
    """Renvoie l'entier p représenté en binaire par s"""
    p = int(s[0])
    for i in range(1,len(s)):
        p = int(s[i])+2*p
    return p

print(0==calc_b2_naif("0"))
print(1==calc_b2_naif("1"))
print(42==calc_b2_naif("101010"))
```

Cela renvoie alors :

```
True
True
True
```

1.6.4 Remarques

1. S'ils sont bien mis en œuvre, ces algorithmes de conversion demandent un temps de calcul de l'ordre de n opérations pour un nombre de n chiffres (binaires ou décimaux), soit de l'ordre de $logp$ opérations.
2. Il existe des algorithmes plus efficaces. Meilleure complexité connue : complexité d'une multiplication de nombres de n chiffres, soit $O(n \log n \log\log n)$ [Knuth].
3. Peu importe la base dans laquelle vous faites vos calculs, ces algorithmes permettent de convertir entre la base 2 et votre base habituelle.

2 Représentation des entiers sur ordinateur

2.1 Cadre

Sur un ordinateur récent :

- on travaille sur des mots-machine de 64 bits (8 octets) ;
- les opérations d'addition et de multiplication d'entiers internes au processeur se font sur 64 bits.

De manière générale, on s'intéressera au fonctionnement sur des ordinateurs travaillant sur des mots de n bits ($n \geq 2$), mais pour les exemples, on prendra systématiquement $n = 16$.

2.2 Somme d'entiers naturels

Sur un processeur n bits, un registre du processeur a n bits et peut représenter tout entier (naturel) de $[\![0, 2^n[\![$.

Lorsqu'on effectue l'addition de deux registres r_1 et r_2 pour stocker le résultat dans r_3, le registre fait n bits : s'il y a une retenue, elle est perdue. [1]

Exemple 2.2.1. Après addition de $\underline{1111\ 0000\ 1111\ 0000}_2$ et $\underline{0011\ 0011\ 0011\ 0011}_2$ sur 16 bits, le registre résultat contient : $\underline{0010\ 0100\ 0010\ 0011}_2$.

2.3 Entiers relatifs

On veut maintenant pouvoir travailler avec des entiers relatifs et notamment les additionner et les soustraire !

2.3.1 Avec signe et valeur absolue

Première possibilité de codage d'un entier relatif sur n bits : on utilise $n-1$ bits pour la valeur absolue et 1 bit pour le signe.

Exemple 2.3.1.
- On représente -4 par $\underline{1000\ 0000\ 0000\ 0100}_2$.
- On représente 4 par $\underline{0000\ 0000\ 0000\ 0100}_2$.

Mais cela a deux inconvénients majeurs.

1. On a deux représentations pour zéro ($\underline{1000\ 0000\ 0000\ 0000}_2$ et $\underline{0000\ 0000\ 0000\ 0000}_2$).
2. L'addition est compliquée à mettre en œuvre, notamment par rapport à celle définie par les entiers naturels.

Ainsi, cette représentation n'est quasiment jamais utilisée pour les nombres *entiers* d'un processeur.

2.3.2 Complément à deux

On va utilise l'idée suivante. Remarquons que l'addition d'entiers naturels sur le processeur n'est pas correcte mais l'est modulo 2^n. De plus, pour tout $p \in \mathbb{Z}$, $p \,\%\, 2^n \in [\![0, 2^n[\![$. Ainsi, $p \,\%\, 2^n$ est représentable sur n bits.

C'est donc la *représentation en complément à deux* qui est le plus souvent utilisée : un entier relatif p est représenté sur n bits comme l'entier naturel $p \,\%\, 2^n$.

Remarque 2.3.2. L'addition d'entiers relatifs (on dit aussi *signés*) sera correcte modulo 2^n et utilisera les mêmes circuits que l'addition d'entiers naturels.

Exemple 2.3.3.
- -5 est codé par $-5 \,\%\, 2^{16} = 65531 = \underline{1111\ 1111\ 1111\ 1011}_2$.
- 3 est codé par $3 \,\%\, 2^{16} = 3 = \underline{0000\ 0000\ 0000\ 0011}_2$.
- La somme obtenue par le processeur est

$$\underline{1111\ 1111\ 1111\ 1110}_2 = 65534 = 2^{16} - 2 = -2 \,\%\, 65536$$

1. En fait une trace en est généralement gardée dans un autre registre du processeur.

Exemple 2.3.4.
- -4 est codé par $-4 \,\%\, 2^{16} = 65532 = \underline{1111\,1111\,1111\,1100}_2$.
- 6 est codé par $6 \,\%\, 2^{16} = 6 = \underline{0000\,0000\,0000\,0110}_2$.
- La somme obtenue par le processeur est

$$\underline{0000\,0000\,0000\,0010}_2 = 2 = 2 \,\%\, 65536$$

2.3.3 Soustraction d'entiers relatifs

Elle peut se faire relativement facilement (voir annexe).

2.4 Dans les langages de programmation

Dans de nombreux langages (C, Java, ...) :

$$\text{Entiers du langage} = \text{Entiers sur } n \text{ bits}$$

Dans ces langages, sur une machine 64 bits, `4 * 2**62` vaut `0`.

Dans d'autres langages, les entiers ne sont pas les entiers machines. Plusieurs représentation sont possibles. Parmis celles classiques : on utilise un tableau dont les éléments sont des octets/mots machines/chiffres dans une base B (avec B puissance de 2 ou 10). Des fonctions internes au langage prennent alors soin d'effectuer les opérations correctement (en utilisant le fait que le processeur sait calculer sur n bits). Python est dans ce cas.

3 Représentation des réels

L'essentiel à savoir :
- Le principe de la représentation des nombres *normalisés*.
- Les origines des problèmes de précision.
- Les conséquences de ces problèmes.

3.1 Généralités

Mathématiquement, il y a de nombreuses façons de voir les réels.

Une façon particulière : c'est la donnée d'un entier relatif, donnant la partie entière, et d'une suite (infinie) de chiffres, donnant la partie fractionnaire.

Peut-on représenter une suite de chiffres infinies ? Oui, par un algorithme.

Peut-on représenter toutes les suites de chiffres infinies ? Non (cela découle des travaux de Cantor et de Turing).

Pour des besoins de calcul scientifique, nous n'avons pas besoin de représenter tous les réels. On travaille avec des approximations des réels : ici, les nombres décimaux.

Remarque 3.1.1. Qui dit approximation dit *erreur*.

Définition 3.1.2. Soit $(a, x) \in \mathbb{R}^2$. On distingue deux notions d'erreurs dans l'approximation de x par a.

Erreur absolue : $|x - a|$

Erreur relative : $\dfrac{|x-a|}{|x|}$ (non définie si $x = 0$)

On a aussi besoin d'avoir une représentation des nombres de taille réduite :

- pour prendre une place réduite (en mémoire, sur disque, sur le réseau) ;
- pour calculer vite.

3.2 Virgule fixe

On représente tous les nombres décimaux avec un nombre n fixé de chiffres après la virgule.

Avantage : on comprend bien comment ça marche.

Inconvénient :

- On a parfois besoin de beaucoup de chiffres après la virgule (masse de l'électron : 9×10^{-31}kg, $h \approx 6 \times 10^{-34}$J.s).
- Garder 30 chiffres après la virgule est parfois inutile pour manipuler un grand nombre (durée de vie moyenne de l'électron : 10^{34}s).

Dans cette représentation, l'erreur absolue est au plus 10^{-n}. Mais l'important est souvent l'erreur *relative*.

3.3 Virgule flottante

On utilise plutôt l'idée de la notation scientifique des nombres. Un nombre est représenté sous la forme $s \times m \times 10^e$, avec (s, m, e) définis comme suit.

- $s \in \{-1; +1\}$ est le *signe*.
- $m \in [1, 10[$ est un nombre décimal, avec n chiffres après la virgule (n fixé). C'est la *mantisse*.
- e : entier (relatif) appartenant à une plage de valeurs fixée. C'est *l'exposant*.

Exemple 3.3.1. Sur une calculatrice HP48SX (d'après tests personnels) :

- la mantisse m a 11 chiffres après la virgule,
- l'exposant $e \in [\![-499, 500[\![$.

Cela permet de représenter :

- de très grands nombres : jusqu'à $9,999\,999\,999\,99 \times 10^{499}$;
- de très petits (en valeur absolue) : jusqu'à 10^{-499} ;
- et leurs opposés : $-9,999\,999\,999\,99 \times 10^{499}$ et -10^{-499} ;
- avec une erreur relative inférieure à 10^{-11}.

avec seulement 12 chiffres décimaux, un signe et trois chiffres pour l'exposant.

3.4 Virgule flottante en binaire

La notation scientifique présentée plus haut utilise la base 10. C'est souvent cohérent, mais pas toujours en informatique où l'on préférera utiliser la base 2. On a alors l'équivalent de la notion de nombre décimal, dans la base 2.

Définition 3.4.1. Un nombre (ou fraction) décimal est un nombre de la forme $\frac{n}{10^k}$, avec $n \in \mathbb{Z}$ et $k \in \mathbb{N}$.

Définition 3.4.2. Un nombre (ou fraction) dyadique est un nombre de la forme $\frac{n}{2^k}$, avec $n \in \mathbb{Z}$ et $k \in \mathbb{N}$.

Exemple 3.4.3. En décimal le nombre $12345/10^3$ s'écrit $12,345$.

Exemple 3.4.4. En binaire, le nombre $\frac{\underline{10101011}_2}{\underline{10}_2^{\underline{101}_2}}$ s'écrit $\underline{101,01011}_2$. Il vaut $\frac{171}{2^5} = 5,34375$.

Autre façon de calculer :

$$\underline{101,01011}_2 = 2^2 + 0 \times 2^1 + 2^0 + \frac{0}{2} + \frac{1}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} = 5 + \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^5}$$

Un nombre sera donc représenté en virgule flottante en base 2 sous la forme $s \times m \times 2^e$, avec (s, m, e) comme suit.

- $s \in \{-1; +1\}$ est le *signe*.
- $m \in [1, 2[$ est un nombre dyadique, avec n chiffres après la virgule (n fixé). C'est la *mantisse*.
- e : entier (relatif) appartenant à une plage de valeurs fixée. C'est *l'exposant*.

On commet au plus une erreur relative de 2^{-n} en représentant un réel ainsi.

3.5 Norme IEEE 754

La norme IEEE 754 est utilisée dans tous les ordinateurs pour les nombres à virgule flottante. Elle existe en plusieurs versions (simple précision, double précision, double précision étendue). On ne parlera ici que de la double précision (la plus répandue).

Les nombres réels seront donc représentés en virgule flottante avec double précision. Chaque nombre est représenté sur 64 bits, utilisés comme suit.

- 1 bit pour le signe (0 pour $+$, 1 pour $-$).
- 11 bits pour l'exposant décalé e (exposant plus 1023).
- 52 bits pour les 52 chiffres après la virgule de la mantisse (inutile de garder le premier bit de m : c'est 1).

On interprète donc la suit de bits $se_{10} \dots e_0 m_1 \dots m_{52}$ comme le nombre x défini comme suit.

Notons $e = \underline{e_{10} \dots e_0} = \sum_{k=0}^{10} e_k 2^k$.

— Si $e \in [\![1, 2047[\![$, x est le nombre *normalisé* :

$$x = s \times \underline{1, m_1 \dots m_{52}} \times 2^{\left(-1023+\underline{e_{10}\dots e_0}\right)} = s \times \left(1 + \sum_{k=1}^{52} \frac{m_k}{2^k}\right) \times 2^{\left(-1023+\sum_{k=0}^{10} e_k 2^k\right)}$$

— Si $e = 0$ et $m_1 = \dots = m_{52} = 0$: $x = 0$ (deux versions : $+0$ et -0).

— Si $e = 0$ et $m_1, \dots, m_{52}$ non tous nuls, x est le nombre *dénormalisé* :

$$x = s \times \underline{0, m_1 \dots m_{52}} \times 2^{-1022} = s \times \left(\sum_{k=1}^{52} \frac{m_k}{2^k}\right) \times 2^{-1022}$$

— Si $e = 2047$ et $m_1 = \dots = m_{52} = 0$: $x = s\infty$ ($+\infty$ ou $-\infty$).

— Si $e = 2047$ et $m_1, \dots, m_{52}$ non tous nuls : $x = NaN$.

Remarque 3.5.1. On ne rentrera pas dans le détail des signification de $+\infty$, $-\infty$ et de NaN.

Les nombres normalisés permettent de représenter de façon précise les réels de $[-M, -m] \cup [m, M]$ avec

$$m \approx 2^{-1022} \approx 2 \times 10^{-308}$$
$$\text{et } M \approx 2^{1024} \approx 1,8 \times 10^{308}$$

Les nombres dénormalisés ne respectent pas la convention de la notation scientifique standard, mais permettent de représenter des nombres plus petits que les nombres normalisés ne peuvent.

3.6 En Python

Avec Python, on peut accéder à la représentation d'un nombre flottant par la méthode `.hex()`. Attention, le nombre est écrit en hexadécimal.

Exemple 3.6.1. Avec `5.5`.

```
>>> 5.5.hex()
'0x1.6000000000000p+2'
```

En effet, on a

$$5,5 = \frac{11}{8} \times 4 = \left(1 + \frac{1}{4} + \frac{1}{8}\right) \times 2^2.$$

En binaire, on écrit $1,375 = 1 + \frac{1}{4} + \frac{1}{8}$ comme

$$1, \underbrace{0110}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}\ \underbrace{0000}.$$

Le regroupement indiqué par les accolades donne l'écriture hexadécimale

$$1,6000000000000.$$

3.7 Problèmes de précision

On rencontre différents types de problèmes de précision.

1. Les problèmes liés aux arrondis des calculs.
2. Les problèmes liés au passage à la représentation binaire.

3.7.1 Problèmes liés aux arrondis

Supposons que l'on veuille effectuer des calculs avec des chiffres décimaux n'ayant que deux chiffres après la virgule.

Exemple 3.7.1. Pour la multiplication : $1,23 \times 1,56 = 1,9188$

Exemple 3.7.2. Pour l'addition : $1,23 \times 10^3 + 4,56 \times 10^0 = 1,23456 \times 10^3$

Pour garder deux chiffres après la virgule, on arrondit le résultat et l'on introduit donc une erreur d'approximation.
Ce problème se pose en décimal, comme en binaire !

3.7.2 Problèmes liés au passage à la représentation binaire

Attention : Les représentations binaires et décimales partagent les *mêmes* problèmes d'arrondis. Cependant, on crée des erreurs d'arrondis lors du *passage* d'une représentation à l'autre.

Exemple 3.7.3. En Python, on rentrera dans la console des nombres en écriture décimale mais le calcul interne se fera en binaire. Cela donne la chose suivante.

```
>>> 0.1+0.2 == 0.3
False
>>> 0.1+0.2
0.30000000000000004
>>> 0.1+0.2-0.3
5.551115123125783e-17
```

Que se passe-t-il ? ? ?

3.7.3 Origine de ce problème

Théorème 3.7.4. Soit p/q un nombre rationnel écrit sous forme irréductible, c'est-à-dire avec p et q entiers, premiers entre eux et $q > 0$. Alors :

1. p/q est un nombre décimal si et seulement si q est de la forme $2^\alpha 5^\beta$ où $(\alpha, \beta) \in \mathbb{N} \times \mathbb{N}$;
2. p/q est un nombre dyadique si et seulement si q est de la forme 2^α où $\alpha \in \mathbb{N}$.

Ainsi, $\frac{1}{10}$ n'est pas un nombre dyadique. En écriture décimale,

$$1/10 = 0,1$$

alors qu'en écriture binaire,

$$1/10 = \underline{0,0001100110011001100110011001\ldots}_2.$$

Le flottant [2] (arrondi par défaut) x représentant $\frac{1}{10}$ est donc

$$\underline{1,10011001100110011001100110011001100110011001100110011001}_2 \times 2^{-4}.$$

L'approximation dyadique au plus près de $0,1$ vaut donc

$$\underline{1,10011001100110011001100110011001100110011001100110011010}_2 \times 2^{-4},$$

qui est la représentation exacte de

$$0,1000000000000000055511151231257827021181583404541015625$$

De même, le flottant y (arrondi au plus proche) représentant $\frac{2}{10}$ est

$$\underline{1,10011001100110011001100110011001100110011001100110011010}_2 \times 2^{-3}.$$

Ainsi, si on effectue le calcul de $x + y$, on obtient :

$$\begin{aligned} &\underline{0}_2, \underline{1100110011001100110011001100110011001100110011001100110011010}_2.2^{-3} \\ +\ &\underline{1}_2, \underline{1001100110011001100110011001100110011001100110011001100110011010}_2.2^{-3} \\ =&\underline{10}_2, \underline{0110011001100110011001100110011001100110011001100110011001110}_2.2^{-3} \end{aligned}$$

Arrondi au flottant le plus proche, cela donne :

$$\underline{1}_2, \underline{0011001100110011001100110011001100110011001100110011001100110100}_2.2^{-2},$$

soit

$$0,30000000000000004440892098500626161694526672363281 25.$$

Cela explique bien ce que donne Python.

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

Quand on effectue le calcul `0.1 + 0.2 - 0.3`, on crée de nouvelles erreurs d'arrondi. Ces dernières sont «négligeables» devant $0,1$, mais pas négligable 4×10^{-17} ! D'où le résultat final de l'ordre de 6×10^{-17} :

```
>>> 0.1+0.2-0.3
5.551115123125783e-17
```

2. On comprendra : représentation normalisée à virgule flottante en double précision.

3.8 Erreurs d'arrondis : conséquences

UN TEST DE LA FORME `x == 0` OU `x == y` POUR DES FLOTTANTS N'A AUCUN SENS !

La seule possibilité parfois raisonnable est le « test de petitesse ».

Exemple 3.8.1. On peut tester `abs(x)< epsilon` avec `epsilon = 1e-6`.

La question qui se pose alors est : quelle valeur de `epsilon` choisir ? Il n'y a pas de réponse universelle, cela dépend du problème étudié ...

De même, on se méfiera des tests du type `x<y` ou `x<=y`.

Exemple 3.8.2.

Construisons une équation du second degré.

```
>>> r1 = 1 + 1.2e-16
>>> r1
1.0000000000000002
>>> r2 = 1
>>> a, b, c = 1, -(r1+r2), r1 * r2
```

Normalement `r1` et `r2` sont les deux racines réelles distinctes de $\mathtt{a}X^2 + \mathtt{b}X + \mathtt{c}$, qui a donc un discriminant strictement positif. Vérifions cela.

```
>>> Delta = b**2 - 4*a*c
>>> Delta
-8.881784197001252e-16
```

Oups...

```
>>> a*r1**2 + b*r1 + c
2.220446049250313e-16
>>> a*r2**2 + b*r2 + c
2.220446049250313e-16
```

On peut aussi trouver des cas où `Delta` est nul avec le polynôme qui s'annule au moins sur deux flottants, dont l'un n'est pas supposé être une racine...

4 Annexe : représentation détaillé des entiers

4.1 Conversion d'un entier en base 2

On s'intéresse à la démonstration de l'algorithme de converion par divisions successives. Si $p = \underline{a_n \dots a_0}_{\,2}$, alors

$$p = \sum_{k=0}^{n} a_k 2^k = 2 \sum_{k=1}^{n} a_k 2^{k-1} + a_0.$$

Ainsi, a_0 est le reste de la division euclidienne de p par 2 et, si $p \neq a_0$, $\underline{a_n \dots a_1}_{\ 2}$ est le quotient de la division euclidienne de p par 2.

On peut donc écrire la fonction suivante.

```
def conv_b2(p):
    """Convertit l'entier p en base 2 (renvoie une chaîne)"""
    x = p # On copie p
    s = ""
    i=0
    while x > 1 :
        s = str(x%2) + s
        x = x // 2
        i=i+1
    return str(x)+s

print('0='+conv_b2(0)+' et 1='+conv_b2(1)+' et 42='+conv_b2(42))
```

— **Invariant :** $p = x + s$, avec :

  — **s :** $s = \underline{a_{i-1} \dots a_1 a_0}_2$

  — **x :** $x = \sum_{k=i}^{n} a_k 2^{k-i}$

— **Initialisation :** $i = 0$

  — **s :** s est vide

  — **x :** $x = \sum_{k=i=0}^{n} a_k 2^{k-0} = p$

— On suppose l'hypothèse vrai au rang i, montrons qu'elle est vrai au rang $i+1$:

— $x = \sum_{k=i}^{n} a_k 2^{k-i} = a_i + \sum_{k=i+1}^{n} a_k 2^{k-i} = a_i + 2 \cdot \sum_{k=i+1}^{n} a_k 2^{k-(i+1)}$

  — $x\%2 \to a_i$

  — $x//2 \to \sum_{k=i+1}^{n} a_k 2^{k-(i+1)}$

— **Terminaison :** à la sortie de la boucle $x \leq 1$ ce qui donne :

$$x = \sum_{k=i+1}^{n} a_k 2^{k-(i+1)} \leq 1$$

obtenue pour $i = n$ donc à la fin de la boucle $x = a_n$ et $s = \underline{a_{n-1} \dots a_1 a_0}_2$, il faut donc bien ajouter a_n à s.

4.2 Somme de naturels

Dans un processeur n bits :

1. Un registre du processeur a n bits et peut représenter tout entier de $[\![0, 2^n[\![$.
2. Lorsqu'on effectue l'addition de deux registres r_1 et r_2 pour stocker le résultat dans r_3, le registre fait n bits : s'il y a une retenue, elle est perdue. [3]
   Exemple : après addition de $\underline{1111\,0000\,1111\,0000}_2$ et $\underline{0011\,0011\,0011\,0011}_2$ sur 16 bits, registre résultat : $\underline{0010\,0100\,0010\,0011}_2$.

Troncature d'un entier p à ses n bits de poids faibles : valeur du reste de la division de p par 2^n (noté $p \,\%\, 2^n$).

Définitions :

1. Soit $p \in \mathbb{N}$. *p représentable comme entier non signé sur n bits* si $p \in [\![0, 2^n[\![$.
2. *Représentation de p comme entier non signé sur n bits* : suite des n chiffres de son écriture en binaire.
3. Abus de notation : on identifie $[\![0, 2^n[\![$ et les représentations sur n bits.
4. Soit $(p, q) \in [\![0, 2^n[\![^2$. *somme (non signée) de p et q sur n bits* : $(p + q) \,\%\, 2^n$, notée $p +_n q$ (notation non canonique).

Remarques pour $(p, q) \in [\![0, 2^n[\![^2$:

1. $p +_m q \equiv p + q \ [2^n]$.
2. $p +_n q = p + q$ si $p + q < 2^n$.
3. $p +_n q = p + q - 2^n$ si $p + q \geq 2^n$.

4.3 Entiers relatifs

4.3.1 Avec signe et valeur absolue

Première possibilité : $n - 1$ bits pour la valeur absolue et 1 bit pour le signe, par ex. :

— on représente -4 par $\underline{1000\,0000\,0000\,0100}_2$;

— on représente 4 par $\underline{0000\,0000\,0000\,0100}_2$.

Inconvénients :

1. Deux représentations pour zéro ($\underline{1000\,0000\,0000\,0000}_2$ et $\underline{0000\,0000\,0000\,0000}_2$).
2. Plus compliqué à additionner que les entiers naturels.

Représentation quasiment jamais utilisée pour les nombres *entiers* d'un processeur.

3. En fait une trace en est généralement gardée dans un autre registre du processeur.

4.3.2 Complément à deux

Idée géniale :

1. L'addition d'entiers naturels sur le processeur n'est pas correcte mais l'est modulo 2^n ;
2. pour tout $p \in \mathbb{Z}$, $p \,\%\, 2^n \in [\![0, 2^n[\![$;
3. donc $p \,\%\, 2^n$ représentable sur n bits ;
4. l'addition de ces entiers relatifs sera correcte modulo 2^n.

Exemples :

1. -5 est codé par $-5 \,\%\, 2^{16} = 65531 = \underline{1111\ 1111\ 1111\ 1011}_2$.
2. 3 est codé par $3 \,\%\, 2^{16} = 3 = \underline{0000\ 0000\ 0000\ 0011}_2$.
3. La somme obtenue par le processeur est

$$\underline{1111\ 1111\ 1111\ 1110}_2 = 65534 = 2^{16} - 2 = -2 \,\%\, 65536$$

4. -4 est codé par $-4 \,\%\, 2^{16} = 65532 = \underline{1111\ 1111\ 1111\ 1100}_2$.
5. 6 est codé par $6 \,\%\, 2^{16} = 6 = \underline{0000\ 0000\ 0000\ 0110}_2$.
6. La somme obtenue par le processeur est

$$\underline{0000\ 0000\ 0000\ 0010}_2 = 2 = 2 \,\%\, 65536$$

Définitions :

1. Soit $p \in [\![0, 2^n[\![$. *Complément à deux de p sur n bits* : $(-p) \,\%\, 2^n$, noté $c_n(p)$ (non canonique).
2. Soit $p \in \mathbb{Z}$. p est *représentable comme entier signé sur n bits* si $p \in [\![-2^{n-1}, 2^{n-1}[\![$.
3. Soit $p \in [\![-2^{n-1}, 2^{n-1}[\![$. *Représentation en complément à deux de p : $p \,\%\, 2^n$,* notée $r_n(p)$ (non canonique).

Remarques :

1. c_n involution [4] de $[\![0, 2^n[\![$.
2. Soit $p \in [\![0, 2^n[\![$. $c_n(p) = 2^n - p$ si $p \neq 0$, 0 si $p = 0$.
3. r_n bijection de $[\![-2^{n-1}, 2^{n-1}[\![$ sur $[\![0, 2^n[\![$.
4. $\forall p \in [\![-2^{n-1} + 1, 2^{n-1}[\![\quad r_n(-p) = c_n(r_n(p))$.
5. Pour tout $(p, q) \in [\![-2^{n-1}, 2^{n-1}[\![^2$ on a $r_n^{-1}\left(r_n(p) +_n r_n(q)\right) \equiv p + q \ [2^n]$.
6. On a l'égalité si $p + q \in [\![-2^{n-1}, 2^{n-1}[\![$ (en particulier si p et q de signes différents).
7. Le bit de poids fort de $r_n(p)$ vaut 1 si et seulement si $p < 0$.

Intérêt du complément à deux : Pour calculer une addition, on utilise exactement les mêmes circuits électroniques que pour des entiers non signés !

4. Bijection d'un ensemble sur lui-même qui est sa propre bijection réciproque.

4.3.3 Calcul de la représentation en complément à deux

Définition *Complément à un sur n bits de $\underline{a_{n-1}\dots a_0}_2$:* $\underline{b_{n-1}\dots b_0}_2$ où $b_k = 1 - a_k$ pour $k \in [\![0, n[\![$. On note $c'_n(p)$ cette valeur.

Proposition Pour tout $p \in [\![0, 2^n[\![$,

$$p + c'_n(p) = \sum_{k \in [\![0,n[\![} 2^k = 2^n - 1$$

Conséquence Pour tout p entier non signé sur n bits,

$$c_n(p) = c'_n(p) +_n 1$$

Conséquence (bis) Pour tout $p \in [\![-2^{n-1} + 1, 2^{n-1}[\![$

$$r_n(-p) = c'_n(r_n(p)) +_n 1$$

Exemple (sur 16 bits) :

1. Représentation de 5 en tant qu'entier non signé ?
2. Complément à un de 5 ?
3. Complément à deux de 5 ?
4. Représentation de -5 sur 16 bits ?
5. Calcul de l'opposé de l'entier signé $\underline{1111\,1111\,1111\,1000}$?
6. Que vaut l'entier signé $\underline{1111\,1111\,1111\,1000}$?

4.3.4 Soustraction

Soit $(p, q) \in [\![2^{n-1} - 1, 2^{n-1}[\![$.

Définition *Différence sur n bits de p et q :* $(p - q) \,\%\, 2^n$, notée $p -_n q$ (non canonique).

Proposition $p -_n q = p +_n c_n(q)$.

1. Complément à deux facile à calculer.
2. Addition/Soustraction : utilisation du même circuit !

5 Annexe : Octal et hexadécimal en informatique

1. Représenter des données par une succession de chiffres binaires est bien adapté à l'utilisation de l'électronique pour construire des ordinateurs.
2. C'est également bien adapté au calcul sur les entiers.

Exemple de représentation par du binaire :

— Une adresse IPv6 est une suite de 128 bits. Exemple d'adresse :

$$00100000000000010000011001111100$$
$$00000010111010000000000000100010$$
$$00000000000000000000000000000000$$
$$11000001000000000000011010001011$$

— Une adresse IPv4 est une suite de 32 bits. Exemple :

$$10000001101011110000111100001011$$

Ce n'est pas très pratique à manipuler...
Comment trouver une représentation plus simple à manipuler ?
Deux possibilités :

1. Considérer ces suites de bits comme des entiers binaires et convertir en décimal (difficile à faire de tête)
2. Grouper ces suites de bits, par exemple par octet, remplacer chaque octet par sa représentation décimale.

Adresses IPv4 : 2e solution. Sur l'adresse précédente : $129.175.15.11$
2e solution : pas pratique si la suite de bits représente un entier (difficile ensuite de calculer sur la représentation).
Autre solution :

1. Grouper les chiffres binaires par 4 (en partant de la droite)
2. Remplacer chaque groupement par le chiffre hexadécimal correspondant.

Avantages :

1. Facile à faire à la main et même de tête.
2. A du sens mathématiquement : le nombre hexadécimal obtenu a la même valeur que le nombre binaire de départ.

Solution prise pour les adresses IPv6.
Exemples :

— traduire en hexadécimal le nombre $\underline{\texttt{11 0101 0010 1010 1110}}_2$

— traduire en binaire le nombre $\underline{\texttt{90f5e56712}}_{16}$

Pour l'octal : même principe mais en groupant par 3.

1. Binaire, octal et hexadécimal sont très utilisés en informatique.
2. En Python, notations autorisées :

```
>>> 0b101010
42
>>> 0o52
42
>>> 0x2a
42
```

3. Écriture d'un entier `n` en décimal, binaire, octal ou hexadécimal : `str(n)`, `bin(n)`,`oct(n)`, `hex(n)`.

Danger, ancienne notation encore autorisée en Python 2 mais à ne plus utiliser :

```
>>> 052
42
```

En python 3 :

```
>>> 052
  File "<stdin>", line 1
    052
      ^
SyntaxError: invalid token
```