Chapitre 1 : Représentation des nombres

Thomas MEGARBANE

PCSI

Plan

Plan

Objectifs

Objectif (Définir une structure)

Se donner un cadre de travail avec :

- *l'**ensemble** des nombres que l'on va chercher à manipuler*
- *les **opérations** que l'on a le droit ou pas de faire*

Objectif (Choisir une représentation)

Transformer les nombres que l'on veut manipuler en données, suivant deux critères :

- *être **fidèle** : elle représente le plus de nombres possible, le plus précisément possible, et deux nombres (très) différents ont des représentations (très) différentes*
- *être **manipulable** : les opérations se font efficacement*

Remarque

*On cherche en fait à **modéliser** les nombres par un ordinateur. À la manière des modèles (mathématiques ou physiques) qui doivent être en accord avec une réalité, la représentation des nombres doit être conforme à :*

- *l'usage qu'on en fait (aspect manipulable)*
- *l'idée qu'on s'en fait (aspect fidèle)*

Plan

Numération suivant une base

Théorème-Définition

Étant donné $b \in \mathbb{N}$ avec $b > 1$, tout entier $n \in \mathbb{N}$ peut s'écrire de manière unique sous la forme :

$$n = \sum_{k=0}^{+\infty} n_k b^k$$

où pour tout $k \in \mathbb{N}$ on a : $n_k \in \{0, \ldots, b-1\}$.
*Les n_k ainsi définis forment une suite nulle à partir d'un certain rang, que l'on note $p \in \mathbb{N}^*$, et constituent **l'écriture de** n **en base** b, ce que l'on note comme :*

$$n_{(b)} = n_{p-1} n_{p-2} \ldots n_0.$$

Démonstration.

Par divisions euclidiennes successives. □

Exemples

On a les choix de b suivants qui reviennent souvent :

- *$b = 2$: le système **binaire**, très fréquent en informatique*
- *$b = 10$: le système **décimal**, qu'on utilise tous les jours*
- *$b = 16$: le système **hexadécimal**, très utilisé aussi en informatique, où l'on note les chiffres $0, 1, \ldots, 9, A, B, C, D, E, F$ (pour avoir en un seul caractère les valeurs de 0 à $b - 1 = 15$).*

Les Babyloniens comptaient en base 60, et les Gaulois en base 20.

Numération suivant une base

Proposition

Avec les notations précédentes, pour tout $k \in \mathbb{N}$, le nombre n_k est le reste de la division euclidienne de q_k par b, où q_k est le quotient de la division euclidienne de n par b^k.
De plus, les q_k sont donnés récursivement par :

$$q_k = \frac{n - \sum_{i=0}^{k-1} n_i b^i}{b^k} = \frac{q_{k-1} - n_{k-1}}{b}.$$

Remarque

On a donc deux approches pour déterminer l'écriture en base b :

- *soit en calculant les restes successifs, et en les retranchant, ce qui donne une approche récursive ;*
- *soit de manière plus directe, en calculant directement le chiffre qui nous intéresse.*

La première méthode calcule les chiffres par indices croissants.
La seconde méthode est lourde en calculs en général : elle est plus efficace en déterminant d'abord p, puis les chiffres par indices décroissants.

Proposition

Avec les mêmes notations : $p = \lfloor \ln_b(n) \rfloor + 1 = \left\lfloor \frac{\ln(n)}{\ln(b)} \right\rfloor + 1.$

Numération suivant une base

Exemple

Donner les entiers suivants :

- $1011001_{(2)} = 1 + 0 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 0 \cdot 32 + 1 \cdot 64 = 89$
- $37_{(16)} = 3 \cdot 16 + 7 = 55$
- $457_{(100)} = 4 \cdot 100^2 + 5 \cdot 100 + 7 = 40507$

Remarque

On peut avoir une ambiguïté sur les chiffres utilisés. Par exemple, comme 45 *ou* 57 *sont des chiffres en base* 100 *(des éléments de* $[\![0; 99]\!]$*, on pourrait avoir :*

$$457_{(100)} = 4 \cdot 100 + 57 = 457 \text{ ou } 457_{(100)} = 45 \cdot 100 + 7 = 4507.$$

Pour éviter ce problème, on travaille avec des ***listes*** *de chiffres plutôt qu'avec des chaînes de caractères. L'ordre est toujours le même : si* $n_{(b)} = n_{p-1}n_{p-2}\ldots n_0$*, alors la liste associée est* $[n_{p-1}, n_{p-2}, \ldots, n_0]$*.*

Numération suivant une base

Exemple

Les deux algorithmes suivants convertissent directement une liste de chiffre (en base b) en le nombre correspondant, en utilisant que : $n = \sum_{k=0}^{p-1} n_k b^k$.

```
1 def base2dec(L:list,b:int)->int:
2     s=0
3     p=len(L)
4     for i in range(p) :
5         s+=L[p-1-i]*(b**i)
6     return s
```

```
1 def base2dec(L:list,b:int)->int:
2     s=0
3     B=1
4     p=len(L)
5     for i in range(p) :
6         s+=L[p-1-i]*B
7         B=B*b
8     return s
```

*L'algorithme ci-dessus est plus efficace et repose sur la **méthode de Horner**, qui sera expliquée au prochain chapitre :*

```
1 def base2dec(L:list,b:int)->int:
2     s=0
3     for l in L :
4         s*=b
5         s+=l
6     return s
```

Numération suivant une base

Exemple

Coder le nombre $n = 37$ en base 2 : on fait par divisions euclidiennes :

- *par indices croissants :*

$$\begin{aligned} 37 &= 1 + 2 \cdot 18 \\ 18 &= 0 + 2 \cdot 9 \\ 9 &= 1 + 2 \cdot 4 \\ 4 &= 0 + 2 \cdot 2 \\ 2 &= 0 + 2 \cdot 1 \\ 1 &= 1 + 2 \cdot 0 \end{aligned}$$

37	2				
1	18	2			
	0	9	2		
		1	4	2	
			0	2	2
				0	1	2
					1	0

k	n_k	q_k
0	1	18
1	0	9
2	1	4
3	0	2
4	0	1
5	1	0

- *par indices décroissants : comme* $37 \in [32; 64[= [2^5; 2^6[$*, alors* $p = 6$ *:*

$$\begin{aligned} 37 &= 1 \cdot 32 + 5 = 1 \cdot 2^5 + 5 \\ 5 &= 0 \cdot 16 + 5 = 0 \cdot 2^4 + 5 \\ 5 &= 0 \cdot 8 + 5 = 0 \cdot 2^3 + 5 \\ 5 &= 1 \cdot 4 + 1 = 1 \cdot 2^2 + 1 \\ 1 &= 0 \cdot 2 + 1 = 0 \cdot 2^1 + 1 \\ 1 &= 1 \cdot 1 + 0 = 1 \cdot 2^0 + 0 \end{aligned}$$

k	n_k
5	1
4	0
3	0
2	1
1	0
0	1

Numération suivant une base

Exemple

Coder le nombre 243 *en base* 2, 4, 8, 16 *:*

- *base* 2 *:*

$$\begin{aligned} 243 &= 1 + 2 \cdot 121 \\ 121 &= 1 + 2 \cdot 60 \\ 60 &= 0 + 2 \cdot 30 \\ 30 &= 0 + 2 \cdot 15 \\ 15 &= 1 + 2 \cdot 7 \\ 7 &= 1 + 2 \cdot 3 \\ 3 &= 1 + 2 \cdot 1 \\ 1 &= 1 + 2 \cdot 0 \end{aligned}$$

- *base* 4 *:*

$$\begin{aligned} 243 &= 3 + 4 \cdot 60 \\ 60 &= 0 + 4 \cdot 15 \\ 15 &= 3 + 4 \cdot 3 \\ 3 &= 3 + 4 \cdot 0 \end{aligned}$$

- *base* 8 *:*

$$\begin{aligned} 243 &= 3 + 8 \cdot 30 \\ 30 &= 6 + 8 \cdot 3 \\ 3 &= 3 + 8 \cdot 0 \end{aligned}$$

- *base* 16 *:*

$$\begin{aligned} 243 &= 3 + 16 \cdot 15 \\ 15 &= \underbrace{15}_{=F} + 16 \cdot 0 \end{aligned}$$

Donc : $243 = 11110011_{(2)} = 3303_{(4)} = 363_{(8)} = F3_{(16)}$

Numération suivant une base

Exemple

Pour les chiffres par indices croissants, on a les codes suivants :

```
def dec2base(n:int,b:int)->list:
    if n<b :
        return [n]
    N,L=n,[]
    while N!=0 :
        r,N=N%b,N//b
        L=[r]+L
    return L
```

```
def dec2base(n:int,b:int)->list:
    if n<b :
        return [n]
    L=dec2base(n//b,b)
    return L+[n%b]
```

Et pour les chiffres par indices décroissants :

```
def dec2base(n:int,b:int)->list:
    B=b
    L=[]
    N=n
    while n>=B :
        B*=b
    while B!=1 :
        B=B//b
        q=N//B
        L.append(q)
        N-=q*B
    return L
```

Manipulation dans une base donnée

Méthode

Les opérations entre nombres se déduisent des opérations entre chiffres suivant les ***tables*** *(comme pour la base* 10*). Il suffit alors de poser les opérations.*

Remarque

La multiplication (resp. la division) par b revient simplement à décaler les chiffres vers la gauche (resp. la droite). Les autres divisions sont plus compliquées à poser.

Proposition

En base 2*, on a les tables suivantes :*

$(+)$	0	1
0	0	1
1	1	10

et

$(\times)$	0	1
0	0	0
1	0	1

Manipulation dans une base donnée

Exemples

Effectuer les opérations suivantes :

1. $11001_{(2)} + 10011_{(2)}$ *:*

$$\begin{array}{cccccc} & 1 & 1 & 0 & 0 & 1 \\ + & 1 & 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \end{array}$$

2. $11001_{(2)} \times 11_{(2)}$ *:*

$$\begin{array}{cccccccc} & & 1 & 1 & 0 & 0 & 1 \\ \times & & & & & 1 & 1 \\ \hline & 0 & 1 & 1 & 0 & 0 & 1 \\ & 1 & 1 & 0 & 0 & 1 & . \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}$$

3. $227_{(8)} + 176_{(8)}$ *:*

$$\begin{array}{cccc} & 2 & 2 & 7 \\ + & 1 & 7 & 6 \\ \hline & 4 & 2 & 5 \end{array}$$

4. $227_{(8)} \times 76_{(8)}$ *:*

$$\begin{array}{ccccc} & & 2 & 2 & 7 \\ & \times & & 7 & 6 \\ \hline & 1 & 6 & 1 & 2 \\ 2 & 0 & 4 & 1 & . \\ \hline 2 & 2 & 2 & 2 & 2 \end{array}$$

Remarque

Il suffit en fait de toujours raisonner avec des chiffres, et donc transformer tout élément qui n'est pas dans $[\![0; b-1]\!]$ pour s'y ramener : tous les calculs et tous les nombres doivent être alors écrits en base b.

Représentation des entiers sur mots de taille fixe

Méthode

On fixe $b > 1$ une base et $n \in \mathbb{N}^*$ un nombre de chiffres.

Pour $a_0, \dots, a_{n-1} \in [\![0; b-1]\!]$, on représente par le n-uplet $(a_{n-1}, \dots, a_0)$ l'entier $\sum_{k=0}^{n-1} a_k b^k$.

Proposition

Avec les notations précédentes, on peut représenter b^n nombres : tous les entiers entre 0 et $b^n - 1$.

Définition

*Si en additionnant ou en multipliant des entiers on dépasse $b^n - 1$, on a un problème d'**overflow** (ou **dépassement**) : les calculs se feront, mais seront erronés.*

Représentation des entiers sur mots de taille fixe

Exemples

- *En base 2 sur 4 chiffres :*

$$\begin{array}{cccccc} & 1 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 1 \\ \hline \not{1} & 0 & 0 & 0 & 0 \end{array}$$

- *Plus généralement, on trouve toujours* $(b^n - 1) + 1 = 0$.

Remarque

En Python, les entiers sont représentés de base par une structure dynamique : ils sont représentés sur des mots de taille fixe, mais cette taille est incrémentée dès qu'un dépassement est détecté (ce qui n'est pas toujours le cas...).

Exemple

En binaire, on rencontre souvent les conventions suivantes, utilisées par Python :

- *int32* $(n = 32)$ *: entiers de* 0 *à* $2^{32} - 1 = 4\,294\,967\,295$ *;*
- *int64* $(n = 64)$ *: entiers de* 0 *à* $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$.

Représentation des entiers sur mots de taille fixe

Méthode

On fixe $b > 1$ une base et $n \in \mathbb{N}^*$ avec $n \geq 2$ et on représente des entiers relatifs sur n chiffres. Pour $s \in \{0, 1\}$ et $a_0, \ldots, a_{n-2} \in [\![0; b-1]\!]$, on représente par le n-uplet $(s, a_{n-2}, \ldots, a_0)$ l'entier $(-1)^s \sum_{k=0}^{n-2} a_k b^k$.
*On parle alors d'**entiers signés**.*

Proposition

Avec les notations précédentes, on peut représenter $2 \cdot b^n - 1$ nombres : tous les entiers entre $-(b^{n-1} - 1)$ et $b^{n-1} - 1$.

Remarque

Cette convention présente les inconvénients suivants :

- 0 *possède deux représentations (*$10\ldots00$ *et* $00\ldots00$*);*
- *problèmes d'overflow ;*
- *signe d'un type différent*
- *addition et soustraction à coder séparément, et des additions différentes pour les nombres positifs et négatifs*
- *addition de deux nombres a, b nécessite de comparer $|a|$ et $|b|$ ce qui est coûteux*

Représentation des entiers sur mots de taille fixe

Exemple

Avec $b = 2$ et $n = 4$, on a :

$$(+3) = (0011) \text{ et } (-3) = 1011$$

mais l'addition ne fonctionne pas bien :

$$\begin{array}{cccccc} & & 0 & 0 & 1 & 1 \\ + & & 1 & 0 & 1 & 1 \\ \hline & & 1 & 1 & 1 & 0 \end{array}$$

et on aurait $(+3) + (-3) = (1110) = -6$.
Pas super...
Et les soustractions ne sont pas mieux :

$$\begin{array}{cccccc} & & 1 & 0 & 1 & 1 \\ - & & 0 & 0 & 1 & 1 \\ \hline & & 1 & 0 & 0 & 0 \end{array}$$

et on aurait $(-3) - (+3) = (1000) = 0$.
Pas tellement mieux...

Remarque

Les opérations (additions, soustractions, multiplications) fonctionnent bien avec les nombres positifs, suivant les règles du calcul en base b, à condition de ne pas avoir d'overflow.

Représentation des entiers sur mots de taille fixe

Méthode (Complément à 2)

On fixe $n \in \mathbb{N}^*$ un nombre de chiffres.
L'entier $m \in [\![-2^{n-1}; 2^{n-1}-1]\!]$ est représenté par :

1. *si $m \geq 0$: sa représentation sur n chiffres (non signée), de la forme $0\,a_{n-2}\,a_{n-3}\ldots a_1\,a_0$*
2. *si $m < 0$: la représentation sur n chiffres (non signée) de $2^n + m \in [\![2^{n-1}; 2^n-1]\!]$, de la forme $1\,a_{n-2}\,a_{n-3}\ldots a_1\,a_0$*

avec dans les deux cas $a_{n-2}, a_{n-3}, \ldots, a_1, a_0 \in \{0,1\}$.

Remarque

Cela revient à travailler dans $[\![0; 2^n-1]\!]$ en binaire modulo 2^n.

Exemple

En complément à 2 avec $n = 4$, on a les représentations :

Positifs			*Négatifs*		
s	*code*	*N*	*s*	*code*	*N*
0	*0 0 0*	*0*	*1*	*0 0 0*	*-8*
0	*0 0 1*	*1*	*1*	*0 0 1*	*-7*
0	*0 1 0*	*2*	*1*	*0 1 0*	*-6*
0	*0 1 1*	*3*	*1*	*0 1 1*	*-5*
0	*1 0 0*	*4*	*1*	*1 0 0*	*-4*
0	*1 0 1*	*5*	*1*	*1 0 1*	*-3*
0	*1 1 0*	*6*	*1*	*1 1 0*	*-2*
0	*1 1 1*	*7*	*1*	*1 1 1*	*-1*

Représentation des entiers sur mots de taille fixe

Exemple

On a donc :

$$(+3) = (0011) \text{ et } (-3) = 1101$$

et on a cette fois-ci :

$$\begin{array}{cccccc} & & 0 & 0 & 1 & 1 \\ & + & 1 & 1 & 0 & 1 \\ \hline \not{1} & & 0 & 0 & 0 & 0 \end{array}$$

et on a bien $(+3) + (-3) = (0000) = 0$.
Super !

Proposition

Pour coder l'opposé d'un nombre codé par complément à 2 (peu importe son signe), il suffit de changer tous ses chiffres $0 \leftrightarrow 1$ *puis de lui ajouter* 1.

Exemple

C'est ce qu'on avait avec 3 *et* -3 *pour* $n = 4$ *:*

$$(3) = (0011) \overset{0\leftrightarrow 1}{\Longrightarrow} (1100) \overset{+1}{\Longrightarrow} (1101) = (-3)$$

$$(-3) = (1101) \overset{0\leftrightarrow 1}{\Longrightarrow} (0010) \overset{+1}{\Longrightarrow} (0011) = (3)$$

Représentation des entiers sur mots de taille fixe

Démonstration.

Si on code sur n chiffres, alors on raisonne modulo 2^n. Et on a pour tout entier a :

$$-a = 2^n - a = \left[(2^n - 1) - a\right] + 1$$

- $2^n - 1$ est représenté par uniquement des 1 ;
- $(2^n - 1) - a$ est donc obtenu en changeant tous les chiffres de a
- le “$+1$” conclut la formule

□

Remarque

On ne va plus faire de soustractions : pour faire l'opération $a - b$, on transforme b en b et on calcule $a + (-b)$.

Représentation des entiers sur mots de taille fixe

Exemple

Donner l'écriture décimale de $A = (1\ 1010\ 1010\ 1101)$ *:*

- *par définition du complément à 2 :* $A < 0$*, et on va coder* $-A$ *en base 2 :*

$$(A) = (1\ 1010\ 1010\ 1101) \overset{0\leftrightarrow 1}{\Longrightarrow} (0\ 0101\ 0101\ 0010) \overset{+1}{\Longrightarrow} (0\ 0101\ 0101\ 0011) = (-A)$$

et par calcul en base 2 :

$$(-A) = 1 + 2 + 16 + 64 + 256 + 1024 = 1363$$

ou plus simplement en base $16 = 2^4$ *:*

$$(-A) = 3 + 5 \cdot 16 + 5 \cdot 256 = 1363$$

donc $A = -1363$*.*

- *en raisonnant modulo* $2^{13} = 8192$ *:*

$$A \equiv 1 + 4 + 8 + 32 + 128 + 512 + 2048 + 4096 = 6829$$

donc : $A = 6829 - 8192 = -1363$*.*

Représentation des entiers sur mots de taille fixe

Exemple

La transformation de a en $-a$ suivant la représentation de complément à 2 peut se coder ainsi :

```
def plusun(L:list)->list:
    n=len(L)
    r=1
    i=n-1
    while r==1 and i>=0 :
        if L[i]== 0 :
            L[i]=1
            r=0
        else :
            L[i]=0
        i-=1
    return L

def negint(L:list)->list:
    for k in range(len(L)) :
        L[k]=1-L[k]
    return plusun(L)
```

Plan

Extension aux non-entiers

Théorème

Étant donné $x \in \mathbb{R}_+$ et $b \in \mathbb{N}^*$ avec $b \geq 2$, alors x s'écrit de manière unique sous la forme :

$$x = \underbrace{\sum_{i=0}^{m-1} a_i b^i}_{=\lfloor x \rfloor} + \underbrace{\sum_{j=1}^{+\infty} a_{-j} b^{-j}}_{=x-\lfloor x \rfloor} = a_{m-1} a_{m-2} \dots a_1 a_0, a_{-1} a_{-2} \dots$$

où $m \in \mathbb{N}$ et les a_k pour $k \in]\!]-\infty; m-1]\!]$ sont des éléments de $[\![0; b-1]\!]$.

Remarques

- *C'est ce qu'on a avec les réels en base 10 : tout réel s'écrit de manière unique sous forme décimale (avec une partie décimale éventuellement infinie).*
- *Les rationnels sont les seuls dont l'écriture est périodique à partir d'un rang (et ce peu importe la valeur de b).*
- *Un nombre admet une écriture finie si, et seulement si, il est de la forme $\frac{p}{q}$ où q divise une puissance de b (de manière équivalente : les nombres premiers qui apparaissent dans la factorisation de q apparaissent tous dans celle de b). Par exemple, si $b = 10$, on retrouve :*

$$\mathbb{D} = \{\frac{p}{10^n} \mid n \in \mathbb{N}\} = \{\frac{p}{2^n 5^m} \mid n, m \in \mathbb{N}\}.$$

Extension aux non-entiers

Méthode

On représente des réels de manière approchée sous la forme :

$$x \simeq \pm 1 \left(\sum_{i=0}^{m-1} a_i b^i + \sum_{j=1}^{f} a_{-j} b^{-j} \right) = \pm a_{m-1} a_{m-2} \dots a_1 a_0 , a_{-1} a_{-2} \dots a_{-f}$$

où $m, f \in \mathbb{N}$ sont fixés, ce qui revient à :

- *un caractère pour le signe*
- *m caractères pour la partie entière*
- *f caractères pour la partie fractionnaire*

Proposition

Avec les notations précédentes, on représente :

- *$2 \cdot b^{m+f}$ nombres*
- *de $]-b^m; b^m[$*
- *avec précision de b^{-f} près.*

Extension aux non-entiers

Exemple

Avec $b = 2$, $m = 32$ et $f = 31$, on représente sur 64 caractère $2^{64} \simeq 10^{19}$ nombres à $2^{-31} \simeq 5 \cdot 10^{-10}$ près.
Les nombres représentés sont de valeur absolue au plus $2^{32} \simeq 4 \cdot 10^{9}$.

Remarque

On est toujours en erreur absolue, et les erreurs relatives deviennent considérables (pour les tout petits nombres) ou inutilement petites (pour les très grands nombres).
Et les très grands nombres ($> 2^{32}$) et les très petits nombres ($< 2^{-31}$) ne peuvent être représentés.

Extension aux non-entiers

Exemple

Codons le nombre $a = 37.35$ *avec* $b = 2$, $m = 16$ *et* $f = 15$.

- *signe :* $a > 0$ *donc* $s = 0$
- *partie entière :* $\lfloor a \rfloor = 37 = 1 + 36 = 1 + 9 \cdot 4 = 1 + 4 + 32 = (0000\ 0010\ 0101)_2$
- *partie fractionnaire :* $b = a - \lfloor a \rfloor = 0.35$ *:*
  - *de proche en proche :*

$$
\begin{array}{lcl}
2 \cdot b = 0.7 = 0 + 0.7 & \Rightarrow & a_{-1} = 0 \\
2 \cdot 0.7 = 1.4 = 1 + 0.4 & \Rightarrow & a_{-2} = 1 \\
2 \cdot 0.4 = 0.8 = 0 + 0.8 & \Rightarrow & a_{-3} = 0 \\
2 \cdot 0.8 = 1.6 = 1 + 0.6 & \Rightarrow & a_{-4} = 1 \\
2 \cdot 0.6 = 1.2 = 1 + 0.2 & \Rightarrow & a_{-5} = 1 \\
2 \cdot 0.2 = 0.4 = 0 + 0.4 & \Rightarrow & a_{-6} = 0 \\
2 \cdot 0.4 = 0.8 = 0 + 0.8 & \Rightarrow & a_{-7} = 0 \\
2 \cdot 0.8 = 1.6 = 1 + 0.6 & \Rightarrow & a_{-8} = 1 \\
2 \cdot 0.6 = 1.2 = 1 + 0.2 & \Rightarrow & a_{-9} = 1 \\
2 \cdot 0.2 = 0.4 = 0 + 0.4 & \Rightarrow & a_{-10} = 0 \\
2 \cdot 0.4 = 0.8 = 0 + 0.8 & \Rightarrow & a_{-11} = 0 \\
2 \cdot 0.8 = 1.6 = 1 + 0.6 & \Rightarrow & a_{-12} = 1 \\
2 \cdot 0.6 = 1.2 = 1 + 0.2 & \Rightarrow & a_{-13} = 1 \\
2 \cdot 0.2 = 0.4 = 0 + 0.4 & \Rightarrow & a_{-14} = 0 \\
\vdots & & \vdots
\end{array}
$$

  - *par division euclidienne par* 1 *:*

0.35	1
0.7	
1.4	0.01011001100110011 . . .
0.8	
1.6	
1.2	
0.4	
0.8	
1.6	
1.2	
0.4	
0.8	
1.6	
1.2	
0.4	
⋮	

Donc $a = (+\ 0000\ 0010\ 0101, 0101\ 1001\ 100)_2$

Extension aux non-entiers

Remarque

Autre méthode : on peut multiplier par $2^f = 2^{15} = 32768$ *et prendre la partie entière, qu'on écrit en base 2.*
Dans l'exemple précédent :

$$a \cdot 2^{15} = 1223884.8$$

et on a :

$$\begin{aligned}
1223884 &= 0 + 2 \cdot 611942 \\
611942 &= 0 + 2 \cdot 305971 \\
305971 &= 1 + 2 \cdot 152985 \\
152985 &= 1 + 2 \cdot 76492 \\
76492 &= 0 + 2 \cdot 38246 \\
38246 &= 0 + 2 \cdot 19123 \\
19123 &= 1 + 2 \cdot 9561 \\
9561 &= 1 + 2 \cdot 4780 \\
4780 &= 0 + 2 \cdot 2390 \\
2390 &= 0 + 2 \cdot 1195 \\
1195 &= 1 + 2 \cdot 597
\end{aligned}$$

$$\begin{aligned}
597 &= 1 + 2 \cdot 298 \\
298 &= 0 + 2 \cdot 149 \\
149 &= 1 + 2 \cdot 74 \\
74 &= 0 + 2 \cdot 37 \\
37 &= 1 + 2 \cdot 18 \\
18 &= 0 + 2 \cdot 9 \\
9 &= 1 + 2 \cdot 4 \\
4 &= 0 + 2 \cdot 2 \\
2 &= 0 + 2 \cdot 1 \\
1 &= 1 + 2 \cdot 0
\end{aligned}$$

ce qui donne $(1223884) = (10010101011001100)_2$*, et on retrouve le résultat précédent en replaçant la virgule en redivisant par* 2^{15}*.*

Les flottants, ou nombres à virgule flottante

Théorème-Définition

Étant donné $b \in \mathbb{N}$ avec $b \geq 2$ et $x \in \mathbb{R}^*$, alors il existe une unique écriture de la forme :

$$x = s \times m \times b^e$$

où :

- *s est le **signe** (± 1)*
- *m est la **mantisse** (un élément de $[1; b[$, ou parfois de $[1/b; 1[$ selon les conventions)*
- *e est l'**exposant** (un élément de $\mathbb{Z}$)*

*Une telle écriture s'appelle **nombre flottant** (ou nombre à **virgule flottante**).*

Remarque

*Pour $b = 10$ et une normalisation de la mantisse dans $[1; 10[$, on retrouve la **notation scientifique**.*

Les flottants, ou nombres à virgule flottante

Exemple

Pour $b = 10$, selon les normalisations, la mantisse s'écrit :

$$m = \underbrace{a_0}_{\neq 0}, a_{-1}a_{-2}\dots \quad ou \; 0, \underbrace{a_{-1}}_{\neq 0} a_{-2}a_{-3}\dots$$

- $135435 = 1,35435 \times 10^5$
- $7568,78 = 7,56878 \times 10^3$
- $0,005487 = 5,487 \times 10^{-3}$
- $1,05418 = 1,05418 \times 10^0$

- $135435 = 0,135435 \times 10^6$
- $7568,78 = 0,756878 \times 10^4$
- $0,005487 = 0,5487 \times 10^{-2}$
- $1,05418 = 0,105418 \times 10^1$

Remarque

*Les représentations des nombres se fait toujours avec la même **erreur relative** (déterminée par la mantisse) et les nombres représentés peuvent être aussi bien très petits que très grands (selon l'exposant).*

Les flottants, ou nombres à virgule flottante

Exemple

Si $b = 2$, que l'on représente la mantisse sur M chiffres et l'exposant sur E chiffres, alors les nombres représentés sont :

- *connus à un écart relatif de 2^{-M} près*
- *de l'ordre de 2^e pour $e \in [-2^{E-1}; 2^{E-1}]$.*

Par exemple, si on représente m sur 23 chiffres (par écriture en base 2 des nombres à virgule) et e sur 8 chiffres (par complément à 2), on connaît :

- *à $2^{-23} \simeq 10^{-7}$ près (donc 7 chiffres significatifs en base 10)*
- *de l'ordre de $2^{2^7} \simeq 3 \cdot 10^{38}$ (plus grand) à $2^{-2^7} \simeq 3 \cdot 10^{-39}$ (plus petit).*

Remarque

Un des intérêts de travailler avec $b = 2$ est qu'il n'y a qu'un seul nombre non nul en base 2 : 1 !
Et donc, selon la convention choisie, la mantisse est de la forme
$m = 1, a_{-1}a_{-2}\ldots$ ou $0, 1a_{-2}a_{-3}\ldots$. On gagne donc un chiffre.

Les flottants, ou nombres à virgule flottante

Exemple

En norme IEEE-754, on travaille sur 32 ou 64 caractères, avec les choix :

Nombre de caractères	*Signe*	*Exposant*	*Mantisse*
32	*1*	*8*	*23*
64	*1*	*11*	*52*

qui permettent d'avoir un bon compromis entre nombres représentés et erreur relative.

Exemple

Pour représenter un nombre en flottant, on cherche successivement et dans cet ordre le signe, l'exposant et la mantisse. Avec les conventions précédentes, pour représenter $a = 37,35$*, on a :*

- $s = 1$
- $37,35 \in [32;64[= [2^5;2^6[$ *donc* $e = 5$ *avec la convention* $m \in [1;2[$ *(on aurait* $e = 6$ *pour la convention* $m \in [1/2;1[$*)*
- $m = \dfrac{37,35}{32} = 1.1671875 = \dfrac{37,35}{2^5}$

On code s par 0
On code e par complément à 2 *sur* 8 *chiffres :* $e = (00000101)$.
On code m sur 23 *chiffres* ***après la virgule*** *:* $m = 1,00101010110011001100110$.
Et finalement : $a = (\underbrace{0}_{s}\ \underbrace{00000101}_{e}\ \underbrace{00101010110011001100110}_{m})$.

Avantages et inconvénients de flottants

Remarque

Avantages :

- *comparaisons rapides de nombres*
- *beaucoup de nombres représentés (très petits et très grands)*
- *précision relative, donc toujours probante*
- *calculs rapide à faire*

Inconvénients :

- *“trous” dans la représentation*
- *amplitude limitée des nombres représentés (underflow et overflow)*
- *jamais de calculs exacts*
- *erreurs de calculs*

Définition (Seuil d’overflow/underflow)

Un seuil d’overflow est un nombre M tel que tout flottant a tel que $|a| \geq M$ sera traité comme $\pm\infty$.
Un seuil d’underflow est un nombre ε tel que tout flottant a tel que $|a| \leq \varepsilon$ sera traité comme 0.

Avantages et inconvénients de flottants

Proposition

Dans les flottants :

1. *l'égalité bouléenne "==" n'a pas de sens*
2. *l'addition n'est pas associative*
3. *le théorème de Fermat est faux*

Exemple

```
>>> 0.3 + 0.3 + 0.3 == 0.9
False

>>> (0.1 + 0.2) + 0.3 == 0.1 + (0.2 + 0.3)
False

>>> 1 + 2.**100 - 2.**100 == 1
False

>>> 1 + 2.**100 - 2.**100
0.0

>>> (2.**45)**4 + (2.**59 - 1)**4 == (2.**59+1)**4
True
```

Remarque

Pour l'égalité, on utilise le seuil d'underflow : on remplace le test `a==b` *par* `abs(a-b)<e` *avec* `e` *le seuil.*

Avantages et inconvénients de flottants

Remarque

Les Simpson avaient déjà donné des contre-exemples au théorème de Fermat :