IF1b – Représentation des flottants

Nous savons représenter en machine un nombre entier si ce nombre est compris entre deux bornes qui dépendent du nombre d'octets utilisées. Il n'y en a qu'un nombre fini. Pour les nombres réels, c'est plus compliqué. Il y a en a une infinité entre deux bornes quelconques. De plus avec des mots de taille fixe, nous ne pouvons stocker qu'un nombre fini de décimales. Donc, même entre deux bornes comme 1 et 2, nous ne pouvons représenter qu'un nombre fini de réels de manière exacte.

Objectifs :

- Distinction entre nombres réels, décimaux et flottants.
- Représentation des flottants sur des mots de taille fixe.
- Précision des calculs en flottant.

I – Codification des nombres décimaux

I-1) Les nombres flottants

Rappelons qu'un nombre décimal est un rationnel qui peut s'écrire sous la forme $d = \frac{x}{10^i}$ où x est un entier relatif. Si on considère la représentation en base 10 de l'entier $x = \pm c_{n-1} \dots c_i \dots c_0$, on obtient l'écriture décimale de d en plaçant une virgule séparant les i chiffres les plus à droite des autres : $\frac{x}{10^i} = \pm c_{n-1} \dots c_i \, , \; c_{i-1} \dots c_0$. Par exemple, $\frac{25}{4}$ est un nombre décimal car il peut aussi s'écrire $\frac{625}{10^2}$, et son écriture décimale est 6,25.

De la même façon, en binaire, un nombre dyadique est un

rationnel qui peut s'écrire sous la forme $b = \frac{x}{2^i}$ où x est un entier relatif, et son développement dyadique s'obtient en plaçant une virgule séparant les i chiffres les plus à droite des autres. Par exemple, $\frac{25}{4}$ est un nombre dyadique $\frac{25}{2^2}$ et son développement dyadique est $110{,}01$ en base 2 que l'on note $(110{,}01)_2$. En effet,

$$\begin{cases} 25 = (11001)_2 \\ \frac{25}{2^2} = (110{,}01)_2 = 1\times 2^2 + 1\times 2^1 + 0\times 2^0 + 0\times 2^{-1} + \overbrace{1\times 2^{-2}}^{0{,}25} \end{cases}$$

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 binaire/décimal est exacte, mais cela pose un problème pour les nombres décimaux car le développement dyadique d'un nombre décimal peut être infini. Par exemple, le nombre 0,1 a une représentation décimale finie et une représentation dyadique infinie :

$$\begin{cases} 0{,}1 = \frac{1}{10^1} = 1\times 10^{-1} \\ 0{,}1 = 0\times 2^0 + 0\times 2^{-1} + 0\times 2^{-2} + 0\times \overbrace{2^{-3}}^{0{,}125} + 1\times \overbrace{2^{-4}}^{0{,}0625} + 1\times \overbrace{2^{-5}}^{0{,}03125} \ldots \end{cases}$$

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 :

```
print(0.1+0.2)
0.1+0.2==0.3
```

```
0.30000000000000004
False
```

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.

I-2) La norme IEEE-754

a) Ecriture scientifique

L'encodage des nombres flottants est inspiré de l'écriture scientifique des nombres décimaux qui se compose :

- D'un signe « + » ou « - ».
- D'un nombre décimal m appelé mantisse compris dans l'intervalle **[1,10[**
- D'un entier relatif n appelé exposant.

Ainsi :

Nombre	Ecriture scientifique
2156	$+2{,}156 \times 10^3$
$-0{,}142$	$-1{,}42 \times 10^{-1}$
Cas général	$\pm m \times 10^n$

On remarquera que le nombre zéro ne peut pas être représenté avec cette écriture.

b) Principe

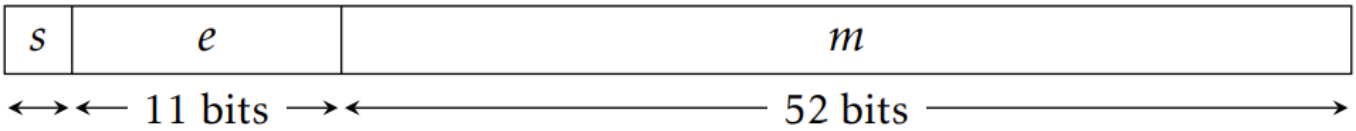
Cette norme est actuellement le standard pour la représentation des nombres à virgule flottante en binaire. Nous allons en donner une description pour une architecture 64 bits. Un nombre non nul

possède une représentation normalisée de la forme :

$$(-1)^s \times m \times 2^{n-d}$$

En 64 bits :

- 1 bit (n° 63) est réservé pour le signe s (0 pour le signe « + »)
- 11 bits (n° 52 à 62) pour l'exposant e ;
- 52 bits (n° 0 à 51) pour la mantisse tel que $m \in [1,2[$



En 32 bits : 1bit pour le signe, 23 pour la mantisse et 8 pour l'exposant.

- Exposant :

Afin de représenter des exposants positifs et négatifs, la norme n'utilise pas l'encodage par complément à 2, mais une technique qui consiste à stocker l'exposant de manière décalée sous la forme d'un nombre non signé. Ainsi l'exposant décalé n est un entier sur 11 bits qui représente les entiers de 0 à 2047 qui est décalé de d=1023. Ce qui permet de représenter des entiers de $[-1023,1024]$. Cependant les valeurs 0 et 2047 sont réservées en 64 bits donc les exposants sont compris dans l'intervalle :

$$e = \begin{cases} [-126,127] \; en \; 32 \; bits \\ [-1022,1023] \; en \; 64 \; bits \end{cases}$$

- Mantisse :

La mantisse étant toujours comprise dans l'intervalle $[1,2[$, elle représente un nombre commençant par le chiffre 1. Par conséquent ; pour gagner 1 bit de précision, les 52 bits dédiés à la mantisse représentent les chiffres après la virgule, qu'on appelle la fraction.

Ainsi en 32 bits ou 64 bits on peut représenter l'écriture d'un

nombre par :

	Exposant (e)	Fraction (f)	Valeur
32 bits	8 bits	23 bits	$(-1)^s \times (1,f) \times 2^{e-127}$
64 bits	11 bits	52 bits	$(-1)^s \times (1,f) \times 2^{e-1023}$

Ainsi en précision 32 bits on pourra représenter un nombre dans l'intervalle $[10^{-38}, 10^{38}]$ et en 64 bits dans l'intervalle $[10^{-308}, 10^{308}]$ approximativement.

Prenons l'exemple suivant en 32 bits :

$$\overbrace{1}^{signe=1}\ \overbrace{10000110}^{exposant=134}\ \overbrace{10101101100000000000000}^{fraction}$$

Dont la valeur est :

$$(-1)^1 \times (1,f) \times 2^{134-127}$$

$$O\grave{u}\ f = 2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-8} + 2^{-9} = 0{,}677734375$$

$$\Rightarrow N = -1{,}677734375 \times 2^7 = -214{,}75$$

c) Valeurs spéciales

Le format des nombres flottants ne permet pas de représenter le nombre zéro. Pour remédier à ce problème, la norme IEEE 754 utilise les valeurs de l'exposant inutilisées (0 et 255 en 32 bits).

Les valeurs spéciales sont regroupées dans le tableau suivant :

Valeur spéciale	Signe s	Exposant e	Fraction f
$+0$	0	0	0
-0	1	0	0
$+\infty\ (inf)$	0	255 (2047)	0
$-\infty\ (-inf)$	1	255 (2047)	0
Not a Number (NaN)	0	255 (2047)	$\neq 0$

Vu que 0 et 255 sont réservés pour l'exposant, le plus petit nombre

décimal positif représentable sera $(-1)^0 \times (1{,}0\ldots) \times 2^{1-127}$ c'est-à-dire 2^{-126}. Cependant, pour rééquilibrer la norme autour de zéro, on préfère utiliser les nombres dénormalisés qui s'écrivent :

$$(-1)^s \times (0, f) \times 2^{e-126} \text{ où } e = 00000000$$

Ainsi si $f = 00000000000000000000$, le plus petit chiffre dans cette norme devient : $x = 2^{-23} \times 2^{-126} = 2^{-149}$

II – Précision des calculs en flottants

II-1) Calculs approchés

La règle pour calculer avec des flottants est la prudence. Les calculs effectués avec des flottants sont des calculs approchés. Par exemple le nombre décimal 1,6 ne peut pas être représenté exactement avec cet encodage. Pour cela la norme IEEE 754 va arrondir cette valeur au plus près : on choisit le flottant le plus proche de la valeur exacte (on privilégie le flottant pair c'est-à-dire une mantisse se finissant par zéro).

Quelques conséquences de ces valeurs approchées :

- L'addition n'est pas associative compte tenu des nombres flottants existants. Par exemple :

```
a=(1.+2.**53)-2.**53
print(a)
```

```
0.0
```

En effet 2^{53} va avoir $m = (1, \overbrace{000\ldots000}^{52\ fois})$. L'ajout de « 1 » se ferait sur la 53ème case de la mantisse, ce qui n'est pas possible.

- La multiplication non plus, par exemple l'égalité suivante n'est pas vérifiée.

```
3.0*10**29==(3.0*10**30)*(10**(-1))
```

```
False
```

II-2) Comparaisons

La précision étant limitée, pour comparer deux nombres flottants, on ne teste pas une égalité entre les deux nombres mais on vérifie s'ils sont suffisamment proches.

Ainsi il est préférable d'écrire un test d'inégalité :

$$|a - b| < \epsilon \text{ où } \epsilon = \textit{borne de précision}$$

Par exemple :

```
a=0.1+0.2
b=0.3
print(a==b)
print(abs(a-b)<1e-12)
```

```
False
True
```

Conclusion :

- Deux réels distincts peuvent être égaux pour la machine.
- Deux expressions qui produisent le même résultat en mathématiques peuvent produire des résultats distincts en machine.
- ...

La grande prudence est donc recommandée en informatique sur la manipulation des flottants.