Représentation approximative des nombres réels : nombres flottants

En informatique, les nombres réels (comme 3.14, 0.1 ou 1/3) ne peuvent pas être représentés de manière exacte dans la plupart des cas. On utilise une représentation approximative appelée **nombre flottant** (ou *float* en anglais[1]). Cette limitation a des conséquences importantes en programmation, notamment lorsqu'on effectue des calculs ou des comparaisons.

Écriture binaire d'un nombre flottant

On a vu dans un chapitre précédant comment représenter un nombre entier en binaire, par exemple 13_{10} s'écrit en binaire 1101_2. Mais comment représenter les nombres réels, et à quoi correspondent des bits écrits après une virgule ?

En maths, quel que soit le nombre n, on a la formule : $2^{-n} = \frac{1}{2^n}$.

Comme pour les nombres entiers, c'est la position qui indique le poids de chaque bit après la virgule, mais avec des puissances négatives de 2. Par exemple, les bits du nombre binaire $0,1011_2$ correspondent à :

bits	**1**	**0**	**1**	**1**
i	-1	-2	-3	-4
2^i	$2^{-1} = \frac{1}{2^1} = 0,5$	$2^{-2} = \frac{1}{2^2} = 0,25$	$2^{-3} = \frac{1}{2^3} = 0.125$	$2^{-4} = \frac{1}{2^4} = 0.0625$
combinaison	$1 \times 2^{-1} = 0,5$	$0 \times 2^{-2} = 0$	$1 \times 2^{-3} = 0,125$	$1 \times 2^{-4} = 0.0625$

$0,1011_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} = 0,6875_{10}$

Cours

De manière générale, un nombre $n < 1$ qui s'écrit dans le système binaire $0,b_1b_2b_3...$ (chaque b_i est un bit valant 0 ou 1) a une valeur en base 10 égale à :

$n = b_1 \times 2^{-1} + b_2 \times 2^{-2} + + b_3 \times 2^{-3} + ...$

ou encore, sans puissances négatives : $n = b_1 \times \frac{1}{2^1} + b_2 \times \frac{1}{2^2} + b_3 \times \frac{1}{2^3} + ...$

Bien sûr, on peut aussi compléter avec une partie entière comme vu précédemment :

$1101,1011_2 = 13,6875_{10}$.

Écrire un nombre binaire en décimal

La formule précédente permet d'écrire facilement un nombre binaire en décimal. Il suffit de multiplier chaque bit par la puissance de 2 correspondante et de faire la somme des valeurs obtenues. ⚠️ Attention, on commence à 2^{-1} pour le premier bit après la virgule.

Exemple :

$$101,011_2$$

$$= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-2}$$

$$= 1 \times 4 + 0 \times 2 + 1 \times 1 + 0 \times 0,5 + 1 \times 0,25 + 1 \times 0,125$$

$$= 4 + 1 + 0,25 + 0,125$$

$$= 5,375_{10}$$

Exercice corrigé

Calculer la valeur en base 10 des nombres binaires suivants :

- 0,0011
- 1,10101
- 101,1110011

Réponse

Écrire un nombre réel en binaire

Voyons maintenant comment passer de l'écriture décimale d'un nombre réel n à son écriture binaire.

Une première chose à noter est le fait que de la même manière certains nombres ne peuvent pas s'écrire avec un nombre fini de chiffres après la virgules en base 10, par exemple $1/3$ s'écrit avec une infinité de 3, tous les nombres réels ne pourront pas s'écrire avec un nombre fini de bits après la virgules en binaire. On ne pourra obtenir qu'une approximation de leur valeur en machine.

Prenons l'exemple de $13,6875_{10}$. La partie entière, 13_{10}, s'écrit en base 2 en effectuant une succession de division par 2 jusqu'à obtenir 0, comme on l'a vu avant. On trouve 1101_2. Il faut maintenant écrire la partie factionnaire, $0,6875$, en binaire.

De la même manière qu'on a utilisée précédemment pour trouver les bits d'un nombre entier par une succession de divisions entières par 2, on peut écrire une partie décimale $0,n$ sous sa forme binaire $0,b_1b_2b_3...$ en effectuant des **multiplications successives** par 2 :

Le produit de n par 2, `n * 2` en Python, peut être décomposé en deux parties :

- La partie entière égale à b_1. Cela permet d'obtenir le premier bit de l'écriture binaire de n.
- La partie décimale entière égale à $b_2b_3b_4...$. On remplace n par ce nombre pour trouver les autres bits.

Il suffit alors de répéter l'opération :

- jusqu'à ce que n soit égal à 0, on aura bien obtenu tous les bits de l'écriture binaire de n ; ou alors
- jusqu'à ce que obtenir une valeur de n déjà vue pendant les calculs, l'écriture binaire de n est alors cyclique, la même suite de bit se répète à l'infini ; ou encore
- jusqu'à obtenir un nombre de bits dans la limite de la précision que souhaite obtenir.

⚠️ Dans ces deux derniers cas, l'écriture d'un nombre flottant en machine sera une approximation de sa valeur, dans la limite du nombre de bits que l'on veut stocker, c'est souvent source de bugs !

Revenons à l'exemple de $n_{10} = 0,6875$:

$0,6875 \times 2 = 1,375$, la partie entière est 1, c'est le premier bit que l'on obtient : **1**.

```
0.687
×   2
1.375
```

```
>>> 0.6875 * 2
1.375
```

Continuons avec la partie décimale.
$0,375 \times 2 = 0,75$, la partie entière est 0, on obtient un second bit : **0**.

```
0.687
×   2
1.375
```

```
0.37
×   2
0.75
```

```
>>> 0.375 * 2
0.75
```

Puis $0,75 \times 2 = 1,5$, on obtient un autre bit: **1**.

```
0.687
×   2
1.375
```

```
0.37
×   2
0.75
```

```
0.75
×   2
1.5
```

```
>>> 0.75 * 2
1.5
```

Et $0,5 \times 2 = 1,0$, on obtient encore un bit **1**.

```
0.687
×   2
1.375
```

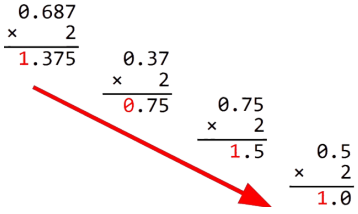
```
0.37
×   2
0.75
```

```
0.75
×   2
1.5
```

```
0.5
×   2
1.0
```

```
>>> 0.5 * 2
1.0
```

A ce stade, la partie décimale est 0, il est inutile de continuer les multiplications par 2, tous les bits ont été trouvés. On peut les lire dans l'ordre de gauche à droite, sans oublier le $0,\ldots$ au début.



L'écriture binaire de $0,675_{10}$ est donc : $0,1011_2$.

Exercice corrigé

Écrire les nombres suivants en binaire :

- $5,25$
- $10,625$

Réponse

Dans l'exemple précédant, $0,675_{10}$ peut être représenté exactement en binaire car il s'écrit comme une somme finie de puissances de 2. Mais ce n'est pas toujours le cas, prenons maintenant l'exemple de $0,1_{10}$ en binaire.

On obtient son écriture binaire en multipliant successivement par 2 :

- $0,1 \times 2 = 0,2$ → on obtient le bit 0, on continue avec $0,2$
- $0,2 \times 2 = 0,4$ → on obtient le bit 0, on continue avec $0,4$
- $0,4 \times 2 = 0,8$ → on obtient le bit 0, on continue avec $0,8$
- $0,8 \times 2 = 1,6$ → on obtient le bit 1, on continue avec $0,6$
- $0,6 \times 2 = 1,2$ → on obtient le bit 1, et on retombe sur $0,2$
- inutile de refaire les calculs, on sait que les bits suivants seront à nouveau 0, 0, 1, 1, 0, 0, 1, etc. C'est cyclique.

L'écriture binaire de $0,1_{10}$ est donc $0,0001100110011\ldots$ avec 0011 qui se répète à l'infini.

Exercice corrigé

Écrire les nombres suivants en binaire :

- $0,2$
- $\frac{1}{3}$

Réponse

Dans certains cas on ne trouvera pas de cycle, par exemple les nombres irrationnels comme π ou $\sqrt{2}$. Il faut s'arrêter quand on atteint un certain nombre de bits qui donne une approximation suffisante.

0.2 + 0.1 n'est pas égal à 0.3

L'écriture binaire de $0,1_{10}$ est donc $0,0001100110011\ldots$ avec 0011 qui se répète à l'infini.

En machine, l'ordinateur ne dispose que d'un nombre fini de bits, il doit tronquer cette représentation, ce qui crée une approximation. On peut le vérifier en Python en affichant la valeur 0.1 dans une f-string avec une précision de 20 chiffres après la virgule :

```
>>> f"{0.1:.20f}"
'0.10000000000000000555'
```

De la même façon, le nombre $0,2$ ne peut pas être représenté de façon exacte en Python :

```
>>> f"{0.2:.20f}"
'0.20000000000000001110'
```

⚠️ Puisque $0,1$ et $0,2$ ne peuvent pas être représentés exactement, leur somme ne donne pas exactement $0,3$!

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

Cours

En Python, `0.2 + 0.1` n'est pas égal à `0.3` !

Ces erreurs d'arrondi sont la cause de nombreux bugs. Il ne faut jamais tester d'égalité sur les nombres de type `float` :

```
if 0.1 + 0.2 == 0.3:
    print("Égal")
else:
    print("Différent")  # C'est ce qui s'affiche !
```

mais toujours tester avec une tolérance :

```
if abs(0.1 + 0.2 - 0.3) <= 1e-9:   # 0.000000001
    print("Les nombres sont suffisamment proches")  # C'est ce qui s'affiche !
else:
    print("Les nombres sont différents")
```

Le format des nombres flottants en machine

Les ordinateurs utilisent généralement la norme IEEE 754 pour représenter les nombres flottants. L'idée ressemble à l'écriture scientifique en mathématiques, par exemple : $1234,56 = 1,23456 \times 10^3$

En binaire, on écrit de la même façon : $101,11_2 = 1,0111_2 \times 2^2$.

Un nombre flottant sur 32 bits[2] se décompose en 3 parties :

S	*exposant*	*mantisse*
1 bit	8 bits	23 bits

où :

- S est un bit de signe, 0 pour un nombre positif, 1 pour un nombre négatif.

- L'exposant est codé en notation biaisé, en ajoutant 127 à sa valeur réelle.
- La mantisse représente les bits significatifs après la virgule. En base 2, il y a toujours un 1 avant la virgule, il n'est pas stocké, on stocke uniquement la partie après la virgule.

La valeur stockée est donc : $(-1)^S \times 1, mantisse \times 2^{exposant}$.

Prenons par exemple 5,75 codé sur 32 bits. 5.75_{10} s'écrit en binaire $101.11_2 = 1.0111 \times 2^3$. On obtient :

- Signe S = 0 (positif)
- Exposant : $3 + 127 = 130 = 10000010_2$
- Mantisse : 01110000000000000000000

En machine, 5,75 sera codé sur 32 bits en `0 10000010 01110000000000000000000`.

---

1. Le terme « virgule flottante » désigne le fait que la virgule d'un nombre peut « flotter » n'importe où à gauche, à droite ou entre les chiffres significatifs. Cette position est indiquée par l'exposant. ↩
2. Sur 64 bits (double précision, le plus courant en Python), la répartition est :
   - 1 bit de signe
   - 11 bits pour l'exposant
   - 52 bits pour la mantisse

   ↩