Représentation des données	Chap 04	Les nombres à virgule

4.1 Vers d'autres types de nombres

Tout n'est pas entier

Jusqu'ici, nous avons représenté des **entiers** : ils possèdent une écriture finie en base 10, en base 2 (et dans toutes les bases), et donc peuvent être stockés exactement.

Mais dans la vie courante, en informatique, on a besoin des **nombres à virgule** :

$$2{,}5 \qquad -1{,}75 \qquad 0{,}1 \qquad 3{,}14159\ldots$$

Ces nombres posent un **nouveau problème** car certaines écritures sont **infinies** mais un ordinateur dispose d'une **mémoire finie**. Donc comment représenter des nombres à virgule avec un nombre fini de bits ?

En base 10

Avant de répondre à comment **représenter** des nombres à virgule, regardons les un peu plus droit dans les yeux.
En base 10, certains nombres ont une écriture décimale **finie** :

$$0{,}5 \qquad 1{,}25 \qquad 2{,}75$$

D'autres ont une écriture décimale **infinie périodique** :

$$\frac{1}{3} = 0{,}333333\cdots = 0{,}\overline{3}\ldots \qquad \frac{34}{99} = 0{,}3\overline{434}\ldots$$

Enfin, il existe des nombres dont l'écriture décimale est **infinie non périodique**, c'est-à-dire sans motif qui se répète :

$$\pi = 3{,}1415926535\ldots$$

Dans tous les cas, une écriture infinie ne peut pas être stockée entièrement en mémoire : il faudra donc, à un moment, **s'arrêter**.

Pour le fun

Les nombres dont l'écriture décimale est finie ou périodique sont appelés **rationnels** et ceux dont l'écriture décimale est infinie apériodique sont appelés **irrationnels**

Conjecture

Les mathématiciens pensent que π serait peut-être même un nombre **univers**, (avec toutes les séquences possibles de nombre après la virgule) mais contrairement à ce qu'on peut voir parfois, ce n'est pas prouvé ! On ne sait même pas s'il y a une infinité de chaque chiffre après la virgule ...

Un cas intéressant

L'écriture : $0{,}999999\ldots$ peut sembler **presque égale** à 1, mais pas tout à fait.
Pourtant, en mathématiques : $0{,}999999\ldots$ est véritablement **égal à** 1.
Pour s'en convaincre, posons $x = 0{,}999999\ldots$
Alors :

$$10x = 9{,}999999\ldots$$

En soustrayant :

$$10x - x = 9{,}999999\cdots - 0{,}999999\ldots$$

Tout ce qui est après la virgule s'annule, il reste :

$$9x = 9 \qquad \Rightarrow \qquad x = 1.$$

Il existe donc plusieurs manières d'écrire un seul nombre !

Convention

En fait, en Mathématiques on interdit toutes les écritures qui finissent par une infinité de 9 par convention pour imposer une écriture unique à chaque nombre.

En base 2, c'est pareil

On pourrait penser que le problème des écritures infinies vient de la base 10. Mais en réalité, il s'agit d'un phénomène **général**.

En base 2 :

- certains nombres ont une écriture binaire finie ;
- d'autres ont une écriture binaire infinie.

Par exemple :

$$0{,}5_{10} = (0{,}1)_2 \qquad 0{,}25_{10} = (0{,}01)_2$$

Mais :

$$0{,}2_{10} = (0{,}001100110011\ldots)_2$$

Ici encore, l'écriture est infinie. Donc, en pratique, l'ordinateur devra stocker :

$$0{,}2 \approx 0{,}001100110011\ldots_2$$

en **coupant** après un certain nombre de bits.

4.2 Virgule fixe : une première solution

4.2.1 Le comprendre en binaire

Fixons la position de la virgule

La méthode de la **virgule fixe** consiste à décider une fois pour toutes où se trouve la virgule dans l'écriture binaire. Ce n'est pas ce qui est utilisé en pratique mais cela permet d'aborder en douceur le concept de représentation des nombres flottants.

Le nombre stocké en mémoire est un **entier**. La virgule n'est pas stockée : elle est simplement **interprétée** lors de la lecture du nombre.

Dans toute cette section, la position de la virgule sera donc **connue à l'avance**.

Exemple de convention : 8 bits au total, dont **3 bits après la virgule**.

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
0	0	1	0	1	1	0	1

↑

virgule fixée

Dans cet exemple, la virgule est imposée entre les colonnes 2^0 et 2^{-1}.

La valeur représentée est donc :

$$00101{,}101_2 = 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3} = 4 + 1 + \frac{1}{2} + \frac{1}{8} = 5{,}625.$$

Exemple 1 — Un second exemple de lecture

Considérons le nombre :

$$(10{,}101)_2$$

On le décompose comme pour un entier, en tenant compte des puissances négatives :

$$(10{,}101)_2 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}.$$

Donc :

$$(10{,}101)_2 = 2 + \frac{1}{2} + \frac{1}{8} = 2{,}625_{10}.$$

Exercice 1 — Lire un binaire à virgule

Donner la valeur décimale des nombres suivants :

$$(1{,}01)_2 \qquad (0{,}111)_2 \qquad (11{,}001)_2$$

4.2.2 Comment passer de base 10 à base 2

Coder les deux parties

Pour coder un nombre décimal en binaire à virgule fixe, on sépare toujours :

la partie entière et **la partie fractionnaire**.

La partie entière se convertit comme d'habitude, par divisions successives par 2.

La partie fractionnaire se convertit autrement. On utilise la **méthode des multiplications successives par 2** :

On part d'un nombre x tel que $0 \leq x < 1$.

- on multiplie x par 2 ;
- le chiffre binaire suivant est la **partie entière** obtenue (0 ou 1) ;
- on recommence avec la nouvelle partie fractionnaire.

On s'arrête lorsque la partie fractionnaire devient nulle (écriture finie) ou lorsque l'on décide de couper (écriture infinie).

Exemple 2 — Coder $2{,}625$

On sépare :

$$2{,}625 = 2 + 0{,}625.$$

Partie entière

$$2_{10} = 10_2.$$

Partie fractionnaire

On applique les multiplications successives à $0{,}625$:

Fraction	$\times 2$	Bit obtenu
$0{,}625$	$1{,}25$	1
$0{,}25$	$0{,}5$	0
$0{,}5$	$1{,}0$	1
$0{,}0$	stop	

On lit les bits dans l'ordre :

$$0{,}625_{10} = (0{,}101)_2.$$

Conclusion

En assemblant partie entière et partie fractionnaire :

$$2{,}625_{10} = (10{,}101)_2.$$

Exercice 2 — Coder en binaire

Coder les nombres suivants en binaire à virgule fixe :

$$1{,}5 \qquad 3{,}75 \qquad 4{,}125 \qquad 0{,}2$$

4.2.3 Pourquoi la virgule fixe reste limitée

Un pas minimal imposé

Reprenons notre exemple :

8 bits au total, dont 3 bits après la virgule.

Avec cette convention, tous les nombres représentables sont des multiples de :

$$2^{-3} = \frac{1}{8} = 0{,}125.$$

Autrement dit, la machine ne peut représenter que les valeurs :

$$0,\ 0{,}125,\ 0{,}250,\ 0{,}375,\ 0{,}500,\ \ldots$$

Entre deux valeurs consécutives, il n'existe **aucune autre valeur possible**. Le plus petit écart entre deux nombres représentables est donc : $0{,}125$.

Exemple 3 — Conséquence immédiate

Considérons le nombre :

$$0{,}1.$$

Ce nombre est strictement compris entre :

$$0{,}000 \quad \text{et} \quad 0{,}125.$$

Il ne peut donc pas être représenté exactement avec 3 bits après la virgule.

La machine devra choisir une valeur proche, par exemple :

$$0{,}125 \quad \text{ou} \quad 0.$$

Dans les deux cas, on obtient une **approximation**.

Dur de jouer sur tous les tableaux

Augmenter le nombre de bits après la virgule permet de réduire le pas minimal et donc d'améliorer la précision.

Mais cela réduit le nombre de bits disponibles pour la partie entière, et donc la taille des nombres que l'on peut représenter.

La virgule fixe impose donc un compromis :

plus de précision ou plus de plage de valeurs, mais pas les deux.

4.3 Virgule flottante : une meilleure idée

4.3.1 Principe de la virgule flottante

Principe

La représentation en **virgule flottante** porte bien son nom puisque le principe est :

on ne fixe plus la position de la virgule.

En base 10, plutôt que d'écrire un nombre avec une virgule fixe, on peut aussi **déplacer la virgule** et compenser ce déplacement par une puissance de 10. Par exemple :

$$123\,000 = 1{,}23 \times 10^5 \qquad \text{et} \qquad 0{,}0045 = 4{,}5 \times 10^{-3}.$$

Dans ces écritures, la position de la virgule n'est plus figée : elle dépend de la puissance de 10 utilisée. C'est exactement cette idée que l'on retrouve en informatique, mais en utilisant des puissances de 2 au lieu des puissances de 10, un nombre est alors écrit sous la forme :

$$\pm m \times 2^e$$

où :

- m est la **mantisse** (elle donne la précision) ;
- e est l'**exposant** (il déplace la virgule).

Cette approche permet de représenter des nombres très grands ou très petits, tout en conservant une précision relative.

Forme normalisée

Pour éviter plusieurs écritures possibles d'un même nombre, on adopte une convention :

la mantisse commence toujours par un seul 1.

Un nombre est donc écrit sous la forme :

$$(1, \text{quelques bits})_2 \times 2^e$$

Avec 8 bits

Pour comprendre le fonctionnement interne, nous utilisons un **format simplifié sur 8 bits** :

$$s\,EEE\,MMMM$$

- s : bit de signe ;
- EEE : exposant codé avec un **biais** choisi à 3 pour décaler l'exposant ;

— $MMMM$: mantisse (sans le premier 1).

De base 2 à base 10

Pour interpréter un nombre stocké en virgule flottante :

— on lit le bit de signe ;

— on reconstruit la mantisse ;

— on reconstruit l'exposant e ;

— on calcule la valeur réelle associée $m \times 2^e$;

— on obtient ainsi un nombre décimal approché.

La valeur obtenue n'est pas forcément le nombre réel exact, mais le plus proche nombre représentable dans le format choisi.

Exemple 4 — Lecture complète

On considère le mot :

$$0\,101\,0101$$

Le signe vaut 0 : le nombre est positif.

L'exposant vaut :

$$101_2 = 5 \quad \Rightarrow \quad e = 5 - 3 = 2.$$

La mantisse est :

$$(1{,}0101)_2 = 1{,}3125.$$

La valeur représentée est donc :

$$1{,}3125 \times 2^2 = 5{,}25.$$

De base 10 à base 2

Pour coder un nombre en virgule flottante :

— on l'écrit en binaire ;

— on le met sous forme normalisée $m \times 2^e$;

— on code le signe ;

— on code l'exposant avec le biais ;

— on tronque la mantisse si nécessaire.

Lorsque l'écriture binaire est infinie, le nombre stocké est une **approximation**.

4.3.2 La norme IEEE 754

Pourquoi une norme ?

Un ordinateur stocke un nombre flottant sur un **nombre fixe de bits**. Mais sans convention, deux machines pourraient stocker **différemment** le même nombre, et obtenir des résultats différents.

La norme **IEEE 754** est un **standard international** qui définit **exactement** :

- le découpage des bits (**signe**, **exposant**, **mantisse**) ;
- le codage de l'exposant (avec un **biais**) ;
- les règles d'**arrondi** ;
- des valeurs particulières : ± 0, $\pm \infty$, et **NaN** (**Not a Number**).

Simple précision et double précision

Les ordinateurs utilisent surtout deux formats IEEE 754 :

Simple précision (**32 bits**) :

$$s \mid \text{exposant (8 bits)} \mid \text{mantisse (23 bits)}$$

- environ **7 chiffres décimaux** significatifs ;
- plage de valeurs large (exposant sur 8 bits), mais précision limitée.

Double précision (**64 bits**) :

$$s \mid \text{exposant (11 bits)} \mid \text{mantisse (52 bits)}$$

- environ **15 à 16 chiffres décimaux** significatifs ;
- plus précis, mais deux fois plus de mémoire.

4.3.3 Limites de la virgule flottante

Une représentation forcément approchée

On est tout de même confronté au fait qu'on utilise un **nombre fini de bits** donc entre deux flottants consécutifs, il n'existe **aucune autre valeur représentable**.

Conséquence : beaucoup de nombres décimaux simples, comme $0{,}1$, n'ont pas d'écriture binaire finie et ne peuvent pas être stockés exactement.

Exemple 5 — Un exemple classique

En calcul flottant, on peut observer :

$$0{,}1 + 0{,}2 \neq 0{,}3.$$

Ce n'est pas une "erreur de Python" : c'est une conséquence du stockage approché en base 2. Les valeurs $0{,}1$ et $0{,}2$ sont déjà des approximations, et l'addition est ensuite arrondie.

Arrondis et accumulation d'erreurs

Une opération sur des flottants (+, −, ∗, /) produit souvent un résultat qui n'est pas exactement représentable.

La machine doit alors **arrondir** au flottant le plus proche. Si on répète beaucoup d'opérations, ces petits arrondis peuvent **s'accumuler**.

C'est particulièrement visible dans les boucles (variables qui s'incrémentent petit à petit, sommes répétées, etc.).

Absorption : le petit disparaît

La précision d'un flottant dépend de la **taille** du nombre. Autour d'un très grand nombre, l'écart entre deux flottants consécutifs peut devenir énorme.

Ainsi, il peut arriver que :

$$x + y = x \quad \text{avec } y \neq 0,$$

si y est trop petit devant x pour modifier les bits significatifs stockés.

4.3.4 Conséquences en Python

Lien avec Python

En Python, le type `float` correspond à un flottant IEEE 754 en **double précision** (64 bits).

Donc `0.1` n'est pas le nombre mathématique $0{,}1$: c'est le **flottant le plus proche** que l'on peut stocker avec un nombre fini de bits.

Implications sur vos programmes

- **ON NE TESTE JAMAIS L'EGALITE == sur des flottants**
- Utiliser une comparaison **avec tolérance** :

$$|a - b| \leq \varepsilon$$

- Dans une boucle, éviter d'incrémenter un flottant "au pas de 0.1" pour compter : préférer un compteur entier puis convertir à la fin.

Exemple 6 — Comparer correctement en Python

```python
import math

a = 0.1 + 0.2
b = 0.3

print(a == b)                  # souvent False
print(math.abs(a-b)<10**-3)      # True (comparaison avec tolérance)
```

Exemple 7 — Bon réflexe : compter en entier

```python
# Mauvaise idée : on ajoute 0.1 en boucle (arrondis cumulés)
x = 0.0
for _ in range(10):
    x += 0.1
print(x)  # pas forcément 1.0 exactement

# Bonne idée : on compte en entier puis on convertit
x = 0
for _ in range(10):
    x += 1
print(x / 10)  # 1.0
```