I. Nombre à virgule flottante

a) Introduction

Comment représenter un **nombre à virgule** en machine ?

Comment le faire efficacement pour des nombres extrêmement proches de 0 comme
0,000000000000000000000000000000000000000000000000000123
ou
extrêmement grands comme
876867867868721638126381623861238 6,13871638716871687168713876127 8613 ?

La réponse à ce premier problème est la **notation scientifique** : **2.3772×10^{32}**

Une autre **difficulté apparaît** :

- les machines fonctionnent **en base 2**
- tandis que nous utilisons **quotidiennement la base 10**.

Cela soulève immédiatement une autre difficulté :

Les *décimaux* sont exprimés en *base 10 (2×5).*
Ils n'ont généralement pas de représentation **exacte** en machine.

Les ordinateurs *savent manipuler* les "**nombres à virgules**"

```
>>> 1.255465 * 753156.45
945561.5624992499
```

Dans les machines, on utilise **les nombres à virgule flottante**

Les nombres sont alors appelés des ***flottants*** (***floats*** en anglais)

b) Notation positionnelle

Nombres décimaux
Dans le système décimal on utilise les **puissances de 10** avant et position des chiffres par rapport à la virgule indique la puissance correspondante :
Par exemple le nombre décimal 325,47 s'écrit

Position	100 (10^2)	10 (10^1)	1 (10^0)	virgule	1/10 (10^{-1})	1/100... (10^{-2})
chiffres	3	2	5	.	4	7

Nombres dyadiques
Dans la machine on utilise le même principe mais avec des puissances de 2.
On parle de ***nombres dyadiques***

Par exemple : 4+2+1+1/2+1/84+2+1+1/2+1/8 et s'écrit en dyadique :

Position	4 (2^2)	2 (2^1)	1 (2^0)	virgule	1/2 (2^{-1})	1/4 (2^{-2})	1/8 (2^{-3})
chiffres	1	1	0	.	1	0	1

$$4 + 2 + 1 + 1/2 + 1/8 = 7.625$$

Remarque :
0,1 a une notation ***décimale finie*** (ce sont des ***décimaux***)
Sa notation *dyadique* n'est *pas finie* !

$$0{,}1=(0{,}0001100110011001100110011001100110011001100110011\cdots)_2$$

En machine **elle est tronquée** (mais sera très proche de 0,1)
Ce n'est *généralement* pas gênant : on n'a généralement pas besoin d'une telle précision.
Cette approche est intéressante et naïvement, on pourrait penser que la machine stocke ainsi ses nombres.

c) Méthode de conversion

On peut par **analogie avec les nombres décimaux**, par exemple 3,3125 , écrire un nombre à virgule en **notation binaire** en utilisant les **puissances négatives de 2.**

$$\begin{aligned}(11{,}0101)_2 &= 1\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4}\\ &= 2 + 1 + 0 + 0{,}25 + 0 + 0{,}0625\\ &= 3{,}3125\end{aligned}$$

Mais comment faire pour trouver cette écriture binaire ?

Attention

Cela n'est faisable le plus souvent que de **manière approchée**, il faudra donc donner la **précision voulue**.

- Pour la partie entière, on fait comme pour les entiers
- Pour la partie décimale :
  - On **multiplie** la partie décimale par **2**
  - On **note la partie** entière obtenue
  - On **recommence** avec la partie **décimale restante**
  - On **s'arrête** quand la partie **décimale est nulle** ou quand la **précision souhaitée est atteinte**

  La **partie décimale** est la **concaténation des parties entières obtenues dans l'ordre de leur calcul**.

Pour notre exemple :

- Conversion de 3 : $(11)_2$
- Conversion de 0,3125 :
  - $0{,}3125 \times 2 = 0{,}625 \quad = \underline{\mathbf{0}} + 0{,}625$
  - $0{,}625 \times 2 = 1{,}25 \quad = \underline{\mathbf{1}} + 0{,}25$
  - $0{,}25 \times 2 = 0{,}5 \quad = \underline{\mathbf{0}} + 0{,}5$
  - $0{,}5 \times 2 = 1{,}0 \quad = \underline{\mathbf{1}} + 0{,}0$

  On s'arrête, la partie décimale est nulle, donc $0{,}3125 = (0{,}0101)_2$
- On a donc $3{,}3125 = (11{,}0101)_2$

Il devient **vite compliqué** d'utiliser cette méthode pour les **très petits** ou **très grands nombres.**

Pour représenter les **nombres à virgule**, on utilise une représentation similaire à la « notation scientifique» des calculatrices, sauf qu'elle est en **base deux** et non en base dix.

Pour rappel en base 10, pour un nombre réel x, on précise :

- Son **signe s** (**+ ou −**)
- Une **partie décimale m** (appelé *mantisse* de x)
- Un **entier relatif n** (appelé *exposant* de x)

$$x = s\ m \times 10^n$$

Exemple : $173{,}5 = \underset{s}{+}\underbrace{1{,}7395}_{m} \times 10^{2}$

Il s'agit de la **représentation en virgule flottante**.

II. Représentation des flottants en base 2

On va **adapter** cette méthode à la **base 2**.
C'est la norme IEEE-754, tout **nombre réel** peut être représenté sous **la forme** :

$$x = (-1)^s\ m \times 2^E = (-1)^s\ m \times 2^{(n-(N-1))}$$

où :

- **s** correspond au **signe de** x
- **m**, la mantisse, est un **réel de l'intervalle [1,2[**
- **n** est un entier relatif tel que **E = n − (N − 1)** ave **N** le niveau de précision de la machine (**32 ou 64 bits**)

On a le tableau suivant :

Encodage	Signe s	n	Mantisse m	Valeur x
32 bits	1 bit	8 bits	23 bits	$(-1)^s\ m \times 2^{(n-127)}$
64 bits	1 bit	11 bits	52 bits	$(-1)^s\ m \times 2^{(n-1023)}$

Soit sur 32 bits : $\underset{s}{[\ ]}\underbrace{[\quad\quad]}_{\text{n sur 8 bits}}\underbrace{[\quad\quad\quad\quad\quad\quad\quad]}_{\text{m sur 23 bits}}$

Remarques

- Le **premier bit** de la **mantisse** d'un nombre normalisé est **forcément 1**, il **n'est pas nécessaire de le représenter**.
- Par convention, on considère qu'un nombre vaut **zéro** si **tous les bits de son exposant et de sa mantisse est nul**.
- Les valeurs **extrêmes**, **0** et $\mathbf{2^N - 1}$ de l'exposant sont réservées pour **zéro** et les **infinis** (en Python : NaN)
- Les nombres à virgules flottante sont définis sur un **nombre fini de bits**, il y donc un **nombre maximal** représentable par cette méthode.

Exemples

Attention : **Ne pas oublier** le bit implicite de la mantisse

a) On souhaite savoir à quoi correspondent les nombres ci-dessous codé en 32 bits

- $\underset{s}{\mathbf{1}}\ \underbrace{\mathbf{0100\ 0110}}_{\text{n sur 8 bits}}\ \underbrace{\mathbf{1001\ 0000\ 0000\ 0000\ 0000\ 000}}_{\text{m sur 23 bits}}$
  - Le **premier 1** représente s = 1, le signe sera $(-1)^1$ donc **–**
  - **0100 0110** est le n de l'exposant, soit $n = 2^6 + 2^2 + 2^1 = 70$
    D'où $E = n - 127 = 70 - 17 = -57$
  - **1001 0000 0000 0000 0000 000** est la mantisse m et vaut :
    $m = \underset{\text{bit implicite}}{\mathbf{1}} + 2^{-1} + 2^{-4} = 1{,}5625$
  - Le nombre codé en base 10 : $x = -1{,}5625 \times 2^{-57} \approx 1{,}0842 \times 10^{-17}$
    On a ici une *représentation approximative*

- 00111110001000000000000000000000
  - Le **premier 0** représente s = 0, le signe sera $(-1)^0$ donc **+**
  - **01111100** : $n = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 = 124$ donc $E = 124 - 127 = -3$
  - **0100 0000 0000 0000 0000 000** : $m = \underset{\text{bit implicite}}{\mathbf{1}} + 2^{-2} = 1{,}25$
  - Le nombre codé en base 10 : $x = 1{,}25 \times 2^{-3} = 0{,}15625$
    On a ici une *représentation exacte*

b) On souhaite coder en binaire sur 32 bits, le nombre décimal 12,625

On alors :

- $12 = (1100)_2$
- Pour 0,625 :
  - $0{,}625 \times 2 = 1{,}25 = \underline{\mathbf{1}} + 0{,}25$
  - $0{,}25 \times 2 = 0{,}5 = \underline{\mathbf{0}} + 0{,}5$
  - $0{,}5 \times 2 = 1{,}0 = \underline{\mathbf{1}} + 0{,}0$

  On s'arrête, la partie décimale est nulle, donc $0{,}625 = (0{,}101)_2$

Donc $12{,}625 = (1100{,}101)_2 = 2^3 + 2^2 + 2^{-1} + 2^{-3} = 2^3 \times (1 + 2^{-1} + 2^{-4} + 2^{-6})$

- On a alors :
  - Le nombre est positif donc **s = 0**
  - $2^3 \leq 12 < 2^4$ donc l'exposant réel est 3 représenté sous la forme
    $3 + 2^{8-1} = 3 + 127 = 130$
    Donc n = 1000 0010 en binaire sur 8 bits

- On a $12{,}625 = 2^3 \times (1 + 2^{-1} + 2^{-4} + 2^{-6})$ qui est bien de la forme $\mathbf{(-1)^s\, m \times 2^E}$
  Avec $m = 1 + 2^{-1} + 2^{-4} + 2^{-6} = 1{,}578125$
  Donc sur 32 bits en binaire m = 1001 0100 0000 0000 0000 000

Conclusion 12,625 se code en 32 bits par : $\underset{s}{\mathbf{0}}\ \underbrace{\mathbf{1000\ 0010}}_{\text{n sur 8 bits}}\ \underbrace{\mathbf{1001\ 0100\ 0000\ 0000\ 0000\ 000}}_{\text{m sur 23 bits}}$

Comparaison de flottants

La principale conséquence de ces arrondis est que la **notion d'égalité** en virgule flottante n'a bien **souvent pas de sens**.

Si on souhaite comparer deux flottants a et b, en Python, on ne pourra pas utiliser la syntaxe : `>>> a == b`

Exemple : `>>> 0.1 + 0.2 - 0.3 == 0`
La console retourne `False`, ce qui n'a pas de sens.
Si on calcule :

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

Donc une valeur approchée

,

Il faut donc effectuer un test différent, du type $|a-b|<\varepsilon$ où ε est une valeur proche de zéro à définir en fonction de la précision voulue.

```
x = 0.1 + 0.2 - 0.3
y = 0.0
precison = 1E-5 #définition de la précision à 0.00001 près
print(abs(x - y)< precision #test adapté au flottants
```

Dans ce cas la console renvoie `True`

III. Calculs avec les flottants en Python

a) Amplitude

En codant sur 64 bits on peut représenter des nombres entre :

- $2^{-1022} \approx 2{,}23 \times 10^{-308}$ pour **le plus petit**

  et

- $2^{1024} - 2^{971} \approx 1{,}80 \times 10^{308}$ pour le **plus grand**

Des améliorations sont faites pour les nombres très proches de 0.

Quand un flottant dépasse le plus grand nombre possible il est considéré comme *infini*

```
>>> 2.0 * 10**308 # dépasse le plus grand
inf
```

b) Quelques surprises avec `inf`

`inf` se comporte "grosso modo" comme l'infini des mathématiques…

Mais l'implémentation révèle quelques surprises :

```
>>> a = float('inf')    # pour définir inf
>>> a
inf
>>> -a
-inf                    # - inifini
>>> a + a
inf
>>> a - a               # opération interdite
nan                     # not a number
>>> a + a == a
True
>>> b = 2.0 * 10 ** 309  # b = inf
>>> c = 2   * 10 ** 1000 # un integer
>>> c > b                # inf est plus grand que tous les nombres
False
```

Attention donc, les comparaisons entre **grands entiers** et **grands flottants** ne **sont pas correctes** mathématiquement parlant.
Il faut **absolument les éviter**.

c) Deux problèmes dans les calculs avec les flottants

Absorption

```
>>> (1. + 2.**53) - 2.**53  # = 1
0.0                         # 1 a été absorbé par l'enorme nombre 2**53
>>> 2.**53 - 2.**53 + 1     # on change l'ordre...
1                           # et ça fonctionne
```

Annulation

Soustraire deux nombres proches fait perdre de la précision

```
>>> a = 2.**53 + 1
>>> b = 2.**53
>>> a - b
0.0
```

d) Il peut y avoir des conséquences

Les calculs avec des flottants engendrent toujours des erreurs qu'il est possible d'éviter en limitant leur quantité et les répétitions.

Exemple :

> Le 25 février 1991, à Dharan en Arabie Saoudite, un missile Patriot américain a raté l'interception d'un missile Scud irakien, ce dernier provoquant la mort de 28 personnes.

L'enquête a mis en évidence le défaut suivant :

- L'horloge interne du missile mesure le temps en 1/10s. Ce nombre *n'est pas dyadique* et est converti avec une erreur d'environ 0,000000095s
- Le missile a été mis en route 100h avant son lancement, ce qui entraine un décalage de :

$$0{,}000000095\times100\times3600\times10\approx0{,}34\text{s}.$$

- C'est assez pour qu'il rate sa cible.