Flottants : Cours

qkzk

Nombre à virgule flottantes

pdf : pour impression

Comment représenter un nombre à virgule en machine ?

Souvenons nous des contraines de base : chaque nombre doit occuper un nombre fixe de bits.

Approche simpliste : la virgule fixe.

On écrit chaque chiffre dans une case, la position indique la valeur !

Le séparateur décimal est toujours au même endroit.

Voici ce que cela donnerait pour 8 chiffres décimaux avec un séparateur décimal au centre :

1000	100	10	1		0.1	0.01	0.001	0.0001
0	2	3	4	.	4	6	2	1

Pas très efficace :

Les nombres sont limités : ici on ne peut dépasser `9999.9999` et le plus petit est `0.0001`.

Avec 8 chiffres on devrait pouvoir décrire `9999 9999` et `0.000 0001`

La virgule flottante

Comment le faire efficacement pour des nombres

- extrèmement proches de 0

  `0,00000000000000000000000000000123`
- ou extrèmement grands

  `876867867868721637163871687168,78613` ?

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

Bien sûr en machine, on utilise la 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.

0.30000000000000004

Les ordinateurs savent manipuler les "nombres à virgules"

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

mais les résultats sont parfois surprenants :

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

Nombres à virgule flottante

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

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

L'égalité de deux flottants n'a aucun sens

```
>>> 0.1 + 0.2 == 0.3
False
```

Notation positionnelle des décimaux

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

```
Position    100    10   1  virgule    1/10    1/100...
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 : $4 + 2 + 1 + 1/2 + 1/8$ et s'écrit en dyadique :

```
Position      4     2     1    virgule    1/2    1/4    1/8
chiffres      1     1     1    .          1      0      1
```

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

Exactement comme pour les décimaux, n'importe quel nombre réel peut être approché aussi précisément que l'on veut par des dyadiques.

$3.14 < \pi < 3.15$

Décimal vers binaire pour les nombres à virgule

On cherche à convertir 2.3 en dyadique (*on dira souvent "en binaire"*).

1. On commence par la partie entière : `2 = 0b10`
2. On multiplie le nombre précédent, sans sa partie entière, par 2. Le premier chiffre sera 0 ou 1 et c'est le bit correspondant à cette position :

   On répète jusqu'à atteindre un entier ou jusqu'à atteindre la précision souhaitée.

Opération	Résultat	bit	position
0.3	0.3	0	0
0.3×2	0.6	0	1
0.6×2	1.2	1	2
0.2×2	0.4	0	3
0.4×2	0.8	0	4
0.8×2	1.6	1	5
0.6×2	1.2	1	6
0.2×2	0.4	0	7

Opération	Résultat	bit	position
0.4×2	0.8	0	8
0.8×2	1.6	1	9
0.6×2	1.2	1	10
0.2×2	0.4	0	11

On peut s'arrêter ici, étant donné que la suite des bits va se répéter.

$0.3 = 0.010011001100_2$ et $2.3 = 10.01001100110_2$

Vérifions :

```
>>> 2 + 1/4 + 1/32 + 1/64 + 1/512 + 1/1024
2.2998046875
```

Binaire vers décimal

Dans l'autre sens c'est beaucoup plus facile, on compte les positions des bits et on ajoute, lorsque le bit est 1, la puissance de $\frac{1}{2}$ correspondante :

$$x = 0,001001001_2 = \frac{1}{2^3} + \frac{1}{2^6} + \frac{1}{2^9} = 0.142578125$$

Revenons sur 0,1 + 0,2

0,1 et 0,2 ont des notations décimales *finies* (ce sont des *décimaux*)

Leur notation *dyadique* n'est pas finie !

$$0,1 = (0,000110011001100110011001100110011001100110011\cdots)_2$$

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.

Problème :

comment manipuler des nombres très grands et des nombres très petits en même temps ?

La taille de l'univers d'un côté, la masse d'un atome de l'autre : il faudrait des milliers de chiffres.

Rappel : calculer avec la notation scientifique

$A = 300000000 \times 0.00000015$

La notation décimale *n'est pas adaptée.*

On préfère la *notation scientifique* :

$A = (3 \times 10^8) \times (1.5 \times 10^{-7})$

Souvenons nous

- on *multiplie 3 et 1,5*
- on *ajoute les exposants 8 et -7*

$A = (3 \times 1.5) \times 10^{8-7}$

$A = 4.5 \times 10^1$

$A = 45$

La machine procède de la même manière en base 2.

Nombre dyadique

Un **nombre dyadique** s'écrit :

$$\pm(1, b1 \cdots bk)_2 \times 2^e$$

où $b1, \ldots, bk$ sont des bits et e est un entier relatif.

La suite de bits $b1 \ldots bk$ est la *mantisse* du nombre,
La puissance de 2 est *l'exposant* du nombre.

Exemple

$6,25 = (110,01)_2 = (1,1001)_2 \times 2^2$

- La mantisse est la suite `1 0 0 1`
- L'exposant est `2`

Nombres à virgule flottante en détail

On rencontre deux représentations courantes des flottants *simple précision* : 32 bits et *double précision* : 64 bits.

La *norme IEEE 754* de 1985 est adoptée par la majorité des langages informatiques modernes.

Dans cette norme (IEEE 754, double précision), les nombres dyadiques sont codés sur 64 bits en réservant :

- 1 bit pour le signe S,
- 11 bits pour l'exposant E,
- 52 bits pour la mantisse M.

La valeur du nombre est alors :

$$(-1)^S \times M \times 2^{E-1033}$$

Ce qu'on peut résumer ainsi :

Norme	Encodage	Signe	Exposant	Mantisse	Valeur	Précision
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{E-1033}$	53 bits

Pour des questions techniques il est nécessaire d'y inclure d'autres objets comme `NaN` (*not a number*) et des infinis positifs et négatifs.

Amplitude

Sans entrer dans les détails, 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
```

Quelques surprises avec `inf` et `nan`

`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.

Bon okay… c’est bizarre mais on retrouve quelques-unes de ces règles en maths, lorsqu’on calcule des limites.

Et pour `nan` ? Et bien c’est mieux encore :

```
>>> a = float('inf') - float('inf')
>>> a
nan
>>> a == a
False
```

En gros, `nan` est le résultat d’une opération qui ne peut être comparé…

Lorsqu’on a deux nombres *immenses* comme

$a = \text{nombre de grains de sable sur terre}^{\text{nombre d'étoiles de la voie lactée}}$

$b = \text{nombre d'étoiles de la voie lactée}^{\text{nombre de grains de sable sur terre}}$

qu’on soustraie, on ne peut deviner l’ordre de grandeur.

Ainsi ce résultat est évalué à `nan`

Deux `nan` ne sont pas forcément égaux… D’où `nan != nan`

Deux problèmes dans les calculs avec les flottants

Absorption

```
>>> (1.0 + 2.0**53) - 2.0**53  # = 1
0.0                            # 1 a été absorbé par l'enorme nombre 2**53
>>> 2.0**53 - 2.0**53 + 1      # on change l'ordre...
1                              # et ça fonctionne
```

Annulation

Soustraire deux nombres proches fait perdre de la précision

```
>>> a = 2.0 ** 53 + 1
>>> b = 2.0 ** 53
>>> a - b
0.0
```

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.

- 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

$$0,000000095 \times 100 \times 3600 \times 10 \approx 0,34s.$$

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

Source

Outils

Obtenir la représentation interne d'un nombre à virgule flottante en Python :

```python
import struct

def float_rep(num: float) -> str:
    """
    Renvoie les 64 bits de la représentation interne en précision double
    s : signe     : 1 bit
    e : exposant  : 11 bits
    m : mantisse  : 52 bits
    num = (-1)**s * 1,m * 2 ** (e - 1023)
    """
    return ''.join("{:08b}".format(elem) for elem in struct.pack('!d', num))


print(float_rep(8 + 4 + 2 + 1/4))
```

Qui affiche :

`0100000000101100100000000000000000000000000000000000000000000000`

def float_rep(num: float) -> str: “”” Renvoie les 64 bits de la représentation interne en précision double s : signe : 1 bit e : exposant : 11 bits m : mantisse : 52 bits num = (-1)**s * 1,m * 2** (e - 1023) “”” return ’‘.join(“{:08b}”.format(elem) for elem in struct.pack(’!d’, num))

print(float_rep(8 + 4 + 2 + 1/4))

Faisons le calcul :

Commençons par découper :

- signe : 1 bit,
- exposant : 11 bits,
- mantisse : 52 bits

```
0100000000101100100000000000000000000000000000000000000000000000

Soit

0   10000000010   1100100000000000000000000000000000000000000000000000
^   \_________/   \__________________________________________________/
|    exposant e                       mantisse m
|
signe s
```

```
s:  0
le nombre est positif
```

C'était facile.

Pour l'exposant, il faut lui soustraire 1023 :

```
exposant:
10000000010

soit (0b10000000010 - 1023) = (1026 - 1023) = 3
```

Le vrai exposant est donc `3`, le nombre est multiplé par `2 ** 3`.

Cette soustraction permet de représenter les exposants négatifs (proches de 0) par des nombres positifs !

Pour la mantisse, on doit se souvenir que le premier bit est toujours égal à 1 et est effacé :

```
mantisse
1100100000000000000000000000000000000000000000000000

Ces bits étant après la virugle, on peut enlever les 0 finaux :

11001

On ajoute 1 avant la virgule :
1,11001 = 1 + 1/2 + 1/4 + 0 + 0 + 1/32 = 57 / 32
```

On peut maintenant appliquer la formule vue plus haut :

```
n = (-1)**0 * 57/32 * 2**3 = 14.25

Vérifions : 8 + 4 + 2 + 1/4 = 14.25
```

---

1. Un outil en ligne : floating point converter,
2. Un article wikipédia détaillé
3. Flottants dénormalisés : afin de gagner en précision pour des valeurs très proches de zéro, la norme décrit une manière de les représenter. On passe d'une précision de l'ordre de 10^{-38} à 10^{-45}. Cela ne fonctionne que pour des nombres proches de 0.
