Chapitre 7 : Les flottants

1. Problématique

On peut mettre en évidence certains « phénomènes étranges » avec Python. Noter les réponses fournies par la machine aux commandes Python suivantes :

✓
```
>>> 1-1/3==2/3
```

✓
```
s=0
while s!=1:
    s=s+1/10
    print(s)
```

✓
```
>>> a=1/11223344556677; a

>>> 1/(a-(a+a**3))

>>> a**3

>>> a-(a+a**3)

>>>(a-a)-a**3
```

L'addition n'est plus associative ?

✓
```
>>> 1e-200

>>> 1e-400

>>> 1e308

>>> 1e500
```

Pour comprendre l'origine de ces phénomènes, il faut s'intéresser à la représentation en machine des nombres réels.

2. Codage d'un nombre réel

Toute information étant codée en impulsions électriques (c'est-à-dire dans un système à 2 états : impulsion ou absence d'impulsion), le codage naturel des données doit se faire en binaire.

Tout nombre décimal positif peut s'écrire sous la forme :

$$x = n + f \text{ avec } \begin{cases} n \in \mathbb{N} \\ f \in [0;1[\end{cases}$$

- n est appelée la partie entière du nombre x
- f est appelée la partie fractionnaire du nombre x

2.1.Base de numération

Les bases les plus couramment utilisées sont : la base 10 (mathématiques courantes), la base 2 ou base binaire (informatique - codage dans la mémoire de l'ordinateur) et la base 16 dite base hexadécimale (informatique -codage logiciel).

Soit b un entier supérieur ou égal à 2.

✓ Soit $n \in \mathbb{N}$. Il existe une unique suite $(a_i)_{i\in\mathbb{N}}$ d'entiers de $[0;b-1]$, tous nuls à partir d'un certain rang, tels que :

$$n = \sum_{i=0}^{+\infty} a_i b^i$$

Si k est le plus grand entier tel que $a_k \neq 0$, on écrira cette représentation sous la forme synthétique suivante :

$$n = \overline{a_k \ldots a_0}^b$$

Exemples : $5 = \overline{101}^2$; $37 = \overline{100101}^2$

La notion de représentation en base b d'un entier se généralise aux réels, à condition d'accepter des sommes infinies de puissances négatives de b. Ces sommes infinies sont des sommes de séries à termes positifs, à considérer comme la limite de sommes finies.

✓ Soit $f \in [0;1[$. Il existe une suite $(a_i)_{i\in\mathbb{N}^*_-}$ d'entiers de $[0;b-1]$ tels que :

$$f = \sum_{i=-\infty}^{-1} a_i b^i$$

Exemple : $0{,}25 = 0\times 2^0 + 0\times 2^{-1} + 1\times 2^{-2} = \overline{0{,}01}^2$

Pour un nombre décimal (en base b), la suite $(a_i)_{i\in\mathbb{N}^*_-}$ contient un nombre fini de termes c'est-dire qu'il existe un entier négatif $-k'$ tel que :

$$f = \sum_{i=-k'}^{-1} a_i b^i$$

✓ Soit $x \in \mathbb{R}_+^*$ (le cas $x = 0$ étant trivial, et le cas $x \in \mathbb{R}_-^*$ s'obtenant en rajoutant au développement de $|x|$ un signe -). Pour trouver un développement en base b de x , on peut décomposer le calcul en deux étapes :

- trouver la représentation en base b de la partie entière n de x, à l'aide des méthodes exposées précédemment. Cette représentation s'écrit sous la forme

$$n = \overline{a_k \dots a_0}^b$$

- trouver le développement décimal de la partie fractionnaire f de x, par la méthode exposée ci-dessous. Ce développement s'écrit sous la forme :

$$f = \overline{0, a_{-1} a_{-2} \dots}^b$$

- Le développement en base b de x est alors :

$$x = \overline{a_k \dots a_0, a_{-1} a_{-2} \dots}^b$$

Pour trouver le développement propre en base b de f, on peut partir de la remarque évidente suivante : la multiplication par b consiste en le décalage à droite d'un cran de la virgule. Ainsi, a_{-1} est la partie entière de $b \times f$, puis reprenant la partie décimale de cette expression et la remultipliant par b, on obtient a_{-2} en prenant à nouveau la partie entière du résultat. De ce fait, on trouve les chiffres successifs du développement en répétant l'opération consistant à multiplier la partie décimale obtenue précédemment par b et en en prenant la partie entière.

Exercice : Trouver l'écriture en base 2 de $0,7$, $0,1$ et $3,84375$.

2.2.Notion de flottant (float)

En notation décimale, les chiffres a gauche de la virgule représentent des unités, des dizaines, des centaines ... les chiffres a droite, des dixièmes, des centièmes, des millièmes, ... Ce mode d'écriture est appelé virgule fixe. Mais avec ce mode d'écriture, il est difficile de représenter des nombres très grands ou très petits. La notation scientifique, appelée virgule flottante, a été introduite :

$$x = \pm m \times 10^e$$

avec $1 \leq m < 10 \in \mathbb{R}$ et $e \in \mathbb{Z}$. m est appelé la mantisse et e l'exposant.

La notation scientifique peut être généralisée au cas d'une écriture en base b.
Soit $b \in \mathbb{N}$, $b \geq 2$. Soit $x \in \mathbb{R}^*$. Il existe un unique entier $e \in \mathbb{Z}$ et un unique réel $m \in [1;b[$ tel que

$$x = \pm m \times b^e$$

Ainsi, il existe un unique $e \in \mathbb{Z}$, et d'uniques entiers $(m_{-i})_{i \in \mathbb{N}}$ appartenant à $[0;b-1]$, non tous égaux à $b-1$ à partir d'un certain rang, tels que :

$$x = \pm \overline{m_0, m_{-1}m_{-2}\ldots}^{b} \times b^e$$

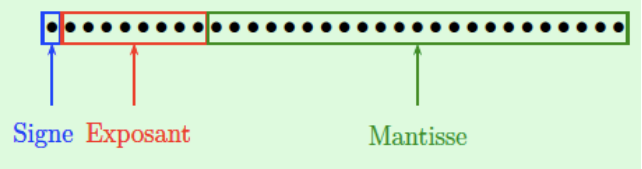
3. Représentation des réels (norme IEEE 754)

Le stockage des nombres réels ne peut pas se faire de façon exacte en général. En effet, même en ne se donnant pas de limite sur le nombre de bits, on sera limité par la capacité de la mémoire (finie, aussi grande soit-elle) de l'ordinateur. Or, entre deux réels distincts, aussi proches soient-ils, il existe une infinité non dénombrable de réels. La mémoire de l'ordinateur est incapable de décrire de façon exacte ce qui se trouve entre deux réels distincts, quels qu'ils soient.
Ainsi, augmenter la capacité de stockage selon les besoins est vain, pour le stockage des réels. Si certains réels peuvent se stocker facilement sur un nombre fini de bits de façon cohérente, la plupart nécessiteraient une place infinie. Ainsi, par convention, on se fixe un nombre fini de bits, en étant conscient que ce qu'on codera sur ces bits ne sera qu'une valeur approchée du réel considéré. Il faudra alors faire attention, dans tous les calculs effectués, à la validité du résultat trouvé, en contrôlant, par des majorations, l'erreur due aux approximations de la représentation informatique des réels. On peut dans certaines situations obtenir une divergence forte entre le calcul effectué par l'ordinateur et le calcul théorique.

- Représentation des réels sur 32 bits
  Le stockage des réels en norme IEEE 754 se fait sous la forme scientifique binaire :

  $$x = \pm \overline{1, m_{-1}m_{-2}\ldots}^{2} \times 2^e$$

  Sur les 32 bits, un bit est réservé au stockage du signe, 8 au stockage de l'exposant et 23 au stockage de la mantisse. Le 1 précédant la virgule n'a pas besoin d'être stocké (valeur imposée).



- Le bit de signe est 0 si $x \geq 0$ et 1 si $x < 0$.
- L'exposant e peut être compris entre -126 et 127. Les 8 bits destinés à cet usage donnent la représentation binaire de $127+e$, compris entre 1 et 254. Les valeurs binaires 00000000 et 11111111 sont interdites (réservées à d'autres usages, correspondant à des représentations non normalisées).
- La mantisse est représentée de façon approchée par son développement en base 2 à 23 chiffres après la virgule (donc les m_{-i}, $i \geq 1$).

Exercices :
- Quel est le nombre représenté par : 11000001010001100000000000000000 ?

- Quelle est la représentation sur 32 bits de 0,1 ?

Remarques :
- La valeur maximale est alors : $\overline{1,11111111....}^{2} \times 2^{127} \simeq 2^{128} \simeq 3,403 \times 10^{38}$
- La valeur minimale est : $2^{-126} \simeq 1,75 \times 10^{-38}$
- La précision de la mantisse est de l'ordre de : $2^{-23} \simeq 1,192 \times 10^{-7}$

Ainsi, cette représentation nous donne des valeurs réelles avec environ 8 chiffres significatifs en notation décimale (en comptant le premier). Cette précision est souvent insuffisante, surtout après répercussion et amplification des erreurs. On utilise de plus en plus fréquemment la norme IEEE 754 avec un codage sur 64 bits, permettant une augmentation des extrêmes et de la précision. C'est le cas de Python que nous utiliserons.

◉ Représentation des réels sur 64 bits
Le stockage des réels en norme IEEE 754 sur 64 bits se fait sur le même principe que sur 32 bits avec les modifications suivantes : sur les 64 bits, un bit est réservé au stockage du signe, 11 au stockage de l'exposant et 52 au stockage de la mantisse.

- Le bit de signe est 0 si $x \geq 0$ et 1 si $x < 0$.
- L'exposant e peut être compris entre -1022 et 1023. Les 11 bits destinés à cet usage donnent la représentation binaire de $1023+e$, compris entre 1 et 2046. Les valeurs

binaires 00000000000 et 11111111111 sont interdites (réservées à d'autres usages, correspondant à des représentations non normalisées).
- La mantisse est représentée de façon approchée par son développement en base 2 à 52 chiffres après la virgule (donc les m_{-i}, $i \geq 1$).

Remarques :
- La valeur maximale est alors : $\overline{1,11111111....}^2 \times 2^{1023} \simeq 2^{1024} \simeq 1,797 \times 10^{308}$
- La valeur minimale est : $2^{-1022} \simeq 2,225 \times 10^{-308}$
- La précision de la mantisse est de l'ordre de : $2^{-52} \simeq 2,22 \times 10^{-16}$

On obtient donc, avec le bit implicite, une précision correcte à 16 ou 17 chiffres significatifs. Cela explique que Python (par exemple) retourne les résultats réels en donnant 16 chiffres après la virgule.

4. Problèmes liées à la représentation non exacte des réels

Il faut faire attention essentiellement à trois problèmes.

4.1.Problèmes d'arrondi

La plupart des nombres n'étant pas représentés de façon exacte, les approximations faites peuvent s'ajouter dans les calculs. Le caractère approché des résultats des calculs peut se constater sur un exemple très simple :

```
>>> 0.3-0.2-0.1
```

Ces petites erreurs s'additionnent, et peuvent devenir grandes si on effectue un grand nombre de calculs :

```
>>> S = 0
>>> for  i in  range(1000):
...       S += 0.1
...
>>> S


>>> S-100
```

4.2.Problème d'absorption

Ce problème se produit lors de la somme de deux réels n'ayant pas du tout le même ordre de grandeur. Si le plus petit des deux a un ordre de grandeur inférieur à la précision du plus grand (donc de l'ordre de grandeur du dernier chiffre significatif), le résultat de la somme admet la même représentation que la valeur la plus grande. On dit que la petite valeur a été absorbée par la grande.

Cette situation se produit dès lors que $\frac{x}{y} > 2^{52}$.

```
>>> 2**100 + 1 - 2**100

>>> 2**100 + 1. - 2**100
```

La première ligne de calcul se fait avec des entiers (aussi grands qu'on veut en Python). Il n'y a dans ce cas pas de problème. La seconde ligne représente le même calcul avec des réels (le point après le 1 force le calcul à se faire en type réel). Le 1 a été absorbé. Nous illustrons le fait que le rapport de grandeurs limite est 2^{52} en remarquant que 1 est absorbé face à 2^{53}, mais pas 2 :

```
>>> 2**53 + 1. - 2**53

>>> 2**53 + 2. - 2**53
```

On peut alors imaginer des algorithmes finissant (rapidement) en théorie, mais ne s'arrêtant pas en pratique à cause de ce problème. On en donne d'abord une version avec des entiers (tapé en ligne de commande) :

```
>>> S=0
>>> i=2**53
>>> while  i < 2**53 + 2:
…         i += 1
…        S += 1
...
>>> S
```

Le comportement est celui qu'on attend. Voici la version réelle du même algorithme :

```
>>> S=0
>>> i=2**53
>>> while  i < 2**53 + 2:
...         i += 1.
...        S += 1
...



>>> S
```

On est obligé d'interrompre le calcul avec Ctrl-C . En demandant ensuite la valeur de s, on se rend compte qu'il est passé un grand nombre de fois dans la boucle. Le fait de travailler en réels a comme conséquence que la valeur de i considérée est toujours la même, car le 1 qu'on lui ajoute est absorbé.

4.3.Problème de cancellation

Il s'agit du problème inverse. Lorsqu'on effectue la différence de deux nombres de même ordre de grandeur, il se peut que de nombreux bits de la représentation se compensent. Par exemple, si seuls les 2 derniers bits diffèrent, le premier bit du résultat sera de l'ordre de grandeur de l'avant-dernier bit des deux arguments, le second bit significatif sera de l'ordre de grandeur du dernier bit des arguments, et les suivants n'ont pas de signification, car provenant de résidus de calculs approchés.
Leur valeur exacte nécessiterait de connaître les chiffres suivants dans le développement des deux arguments. Nous illustrons ceci sur l'exemple suivant :

```
>>> def  f(x):
...         return  (x+1)**2-x**2-2*x-1
...
>>> f(1000000000)


>>> f(1000000000.)


>>> f(10000000000)


>>> f(10000000000.)


>>> f(100000000000)


>>> f(100000000000.)
```

Dans cet exemple, la fonction f est identiquement nulle. Le calcul est correct s'il est effectué avec des entiers, mais plus du tout lorsqu'on passe aux réels.

4.4.Comparaison par égalité

Aux trois problèmes précédents, nous ajoutons le quatrième, qui est en fait dérivé du premier (problème de l'inexactitude de la représentation des réels). Comparer par une égalité deux réels, surtout s'ils sont issus de calculs, n'a pas beaucoup de pertinence, car même s'ils sont égaux en théorie, il est probable que cela ne le soit pas en pratique. Ainsi, en reprenant l'exemple précédent, on obtient le résultat suivant, assez surprenant, pour un test très simple :

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

On y remédie souvent en s'autorisant une petite inexactitude, ce qui oblige à remplacer l'égalité par un encadrement :

```
>>> abs(0.3-0.2-0.1)<1e-15
```