Représentation approximative d'un nombre réel

1 Approximation d'un nombre flottant

Un nombre *flottant* ou *float* comme 0.3 ou 0.1 peuvent poser des problèmes lors de calculs

```
>>> x = 0.3

>>> type(x)
<class 'float'>
```

Les vecteurs suivants $\overrightarrow{u} = \begin{pmatrix} 0,1 \\ 0,3 \end{pmatrix}$ et $\overrightarrow{v} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$ sont colinéaires en effet $\overrightarrow{u} = 0,1 \times \overrightarrow{v}$

Le déterminant de deux vecteurs $\overrightarrow{u} = \begin{pmatrix} x \\ y \end{pmatrix}$ et $\overrightarrow{v} = \begin{pmatrix} x' \\ y' \end{pmatrix}$ est défini par $xy' - yx'$

Deux vecteurs sont colinéaires si et seulement si le déterminant de ces deux vecteurs est nul

La fonction suivante devrait retourner la valeur Vraie or ce n'est pas le cas!!!
`print(colineaires(0.1,0.3,1,3))` retourne False

```
def determinant(xU,yU,xV,yV):
    return xU*yV - yU*xV

def colineaires(xU,yU,xV,yV):
    return determinant(xU,yU,xV,yV) == 0

 print(colineaires(0.1,0.3,1,3))
```

Que s'est il passé ?

1. Les nombres ont une représentation **finie** en machine!

Avec les puissances de 10 on a les nombres **décimaux** par exemple $1,75 = 1 + 7 \times 10^{-1} + 5 \times 10^{-2}$

Par contre $\frac{1}{3}$ n'est pas un nombre décimal car le développement de $\frac{1}{3}$ est infini!

Avec les puissances de 2 on a les nombres **dyadiques** par exemple $1,75 = 1 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1 + \frac{1}{2} + \frac{1}{4}$

On va donc noter $1,75 = (1,11)_2$

Les " premières" puissances de $\frac{1}{2}$ sont dans le tableau suivant :

$(\frac{1}{2})^n = 5^n \times 10^{-n}$

n	1	2	3	4	5	6	7	8
$(\frac{1}{2})^n$	0,5	0,25	0,125	0,0625	0,03125	0,015625	0,0078125	0,00390625

Théorème 1. *0,1 n'est pas dyadique*

Preuve Calculons la représentation dyadique de 0,1 :

On cherche n entier naturel tel que $(\frac{1}{2})^n \leqslant 0,1 \leqslant (\frac{1}{2})^{n-1}$

On trouve $n = 4$

Ensuite on calcule 0,1-0,0625 = 0,0375 et on recommence avec 0,0375

Maintenant 0,0375 = 0,03125 + 0,00625 donc pour l'instant $0,1 = (\frac{1}{2})^4 + (\frac{1}{2})^5 + 0,00625$

On continue avec 0,00625, or 0,00625 = 0,00390625 + 0,00234375

On continue et on obtient que 0,1 a **un développement dyadique illimité** et puisqu'une machine est finie la représentation en machine de 0,1 est **tronquée**

$0,1 = (0,00011001100110011 00.....)_2$

Retenir

Les calculs sur les nombres flottants ne sont parfois pas exacts

Il ne faut JAMAIS tester une égalité entre deux nombre flottants mais utiliser une marge d'erreur relative.

On corrige donc la fonction `sontColineaires(xU,yU,xV,yV)`

```
>>> def determinant(xU,yU,xV,yV):
        return xU*yV - yU*xV

>>> def sontColineaires(xU,yU,xV,yV):
        return determinant(xU,yU,xV,yV) == 0

>>> print(sontColineaires(0.1,0.3,1,3))
False
>>> print(determinant(0.1,0.3,1,3))
5.551115123125783e-17
```

```
>>> def sontColineaires(xU,yU,xV,yV):
        return abs(determinant(xU,yU,xV,yV)) < 6*10**(-17)

>>> print(sontColineaires(0.1,0.3,1,3))
True
>>> print(sontColineaires(0.1,0.3,1.000000000001,3))
False
```

2 Norme IEEE 754

La notation dyadique ne permet pas de représenter des nombres "très grands" ou "très petits".

On va utiliser une représentation basée sur l'équivalent en base 2 de la notation scientifique :

La notation scientifique d'un nombre x à virgule est définie par :

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

Où s est le signe + ou - , m la mantisse un nombre flottant appartenant à l'intervalle $[1, 10[$, et e un entier relatif

Exemples :

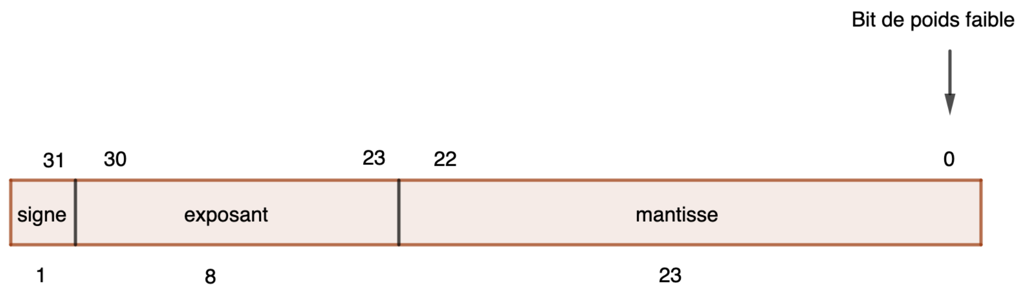
1. $2023 = +2,023 \times 10^3$
2. $0,0123 = +1,23 \times 10^{-2}$
3. $-1234,5678 = -1,2345678 \times 10^3$

La représentation des nombres flottants en machine suit une norme internationale, la norme IEEE 754 (`https://fr.wikipedia.org/wiki/IEEE_754` pour Institute of Electrical and Electronics Engineers)

Si on veut encoder un nombre flottant $x = s \times m \times 2^n$

Il existe deux formats d'encodage

1. **Un format simple précision sur 32 bits**



(a) Le signe s est représenté par 0 si c'est $+$ et le signe moins par 1

(b) Sur 8 bits on ne va pas utiliser la méthode pour coder les nombres relatifs, mais on va décaler de 127 l'intervalle [0,255].
L'exposant n appartient à $[-127, 128]$
On représente n comme l'entier naturel $n + 127 = n + (2^{8-1} - 1)$ appartenant à l'intervalle $[0, 255]$
Par contre les valeurs 0 et 255 sont réservées à des nombres particuliers.
Par exemple 1 encode l'exposant -126 et 254 encode l'exposant 127.
Ainsi si l'exposant n est encodé par c pour retrouver l'exposant n on fait :
$n = c - 127$

(c) La mantisse m est un nombre dans l'intervalle [1,2[mais **les 23 bits servant à encoder la mantisse** représentent les chiffres **après la virgule** qu'on appelle la **fraction**

Par exemple
$2.25 = 1,0125 \times 2 = +(1.001)_2 \times 2^1$
Donc le signe est encodé par 0
Donc l'exposant $n = 1$ est encodé par $c = 1 + 127 = 128 = (10000000)_2$
Enfin la fraction $2^{-3} = 0010\ 0000\ 0000\ 0000\ 0000\ 000$
En tout :
$0\,\underbrace{10000000}_{\text{exposant}}\,0010\ 0000\ 0000\ 0000\ 0000\ 0000\ 000$
Quel est le nombre encodé par $1\,\underbrace{11110000}_{\text{exposant}}\,\underbrace{1010\ 1000\ 0000\ 0000\ 0000\ 000}_{\text{fraction}}$
C'est un nombre négatif, l'exposant est encodé sur 8 bits par 11110000 qui vaut 240 donc l'exposant vaut $240 - 127 = 113$

La mantisse vaut :
$m = 1,1010\,1000\,0000\,0000\,0000\,000 = 1+\dfrac{1}{2}+\dfrac{1}{2^3}+\dfrac{1}{2^5} = \dfrac{2^5+2^4+2^2+1}{2^5} = \dfrac{53}{32}$
Le nombre représenté est $-\dfrac{53}{32} \times 2^{113} \simeq 1,72 \times 10^{34}$

2. **Un format double précision sur 64 bits**

   (a) Le signe s est représenté par 0 si c'est $+$ et le signe moins par 1

   (b) On utilise 11 bits pour l'exposant donc l' intervalle [0,2047].
   On décale par le milieu 1023 pour encoder l'exposant sur l'intervalle $[-1023, 1024]$
   On représente n comme l'entier naturel $n+1023 = n+(2^{11-1}-1)$ appartenant à l'intervalle $[0, 2047]$
   Ainsi si l'exposant n est encodé par c pour retrouver l'exposant n on fait :
   $n = c - 1023$
   Par contre les valeurs 0 et 2047 sont réservées à des nombres particuliers

   (c) La mantisse m est un nombre dans l'intervalle [1,2[mais les 52 bits servant à encoder la mantisse représentent les chiffres **après la virgule** qu'on appelle la **fraction**

Par exemple
$2.25 = +(1.001)_2 \times 2^1$
Donc le signe est encodé par 0
Donc l'exposant $n = 1$ est encodé par $c = 1 + 1023 = 1024 = (10000000000)_2$
Enfin la fraction 2^{-3} = 0010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

3 Les flottants en Python

Les flottants sont représentés **selon la norme IEEE 754 double précision**. Le type est `float`.

Pour avoir de l'aide sur la classe `float` on peut soit consulter la documentation en ligne, soit entrer la commande `help(float` dans une console Python.

Pour avoir la représentation en machine de la mantisse d'un nombre flottant sous la forme d'une chaîne de caractères **en hexadécimal** (l'exposant est en décimal et n'est pas encodé), on a la commande `float.hex()`

Par exemple :

```
>>> float.hex(2.25)
'0x1.2000000000000p+1'
```

On observe bien la mantisse 2000000000000
qui correspond à 0010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
La commande réciproque est `float.fromhex()`

```
>>> float.fromhex('0x1.2000000000000p+1')
2.25
```

De l'écriture dyadique de 0,1 obtenue ci-dessus

$0,1 = (0,00011001100110011 00.....)_2$

Donc $0,1 = 1,1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010 \times 2^{-4}$

(on arrondit) . Or

```
>>> float.hex(0.1)
'0x1.999999999999ap-4'
```

4 Valeurs exceptionnelles

1. +0 a un signe codé par 0, un exposant par 0 et une mantisse par 0
2. -0 a un signe codé par 1, un exposant par 0 et une mantisse par 0
3. $+\infty$ a un signe codé par 0, un exposant par 255 ou 1023, et une mantisse par 0
4. $-\infty$ a un signe codé par 1, un exposant par 255 ou 1023, et une mantisse par 0
5. Une valeur spéciale notée `NaN` pour `Not a Number` permet de représenter des résultats d'opérations invalides comme $0 \times \infty$ ou $\frac{0}{0}$

   Cette valeur spéciale a un signe de 0, un exposant de 255 ou 1023 et une mantisse différente de 0

5 Propriétés

1. En simple précision le plus grand nombre en dehors de la valeur spéciale $+\infty$, a pour exposant 254 et pour mantisse $m = 1,1111\ 1111\ 1111\ 1111\ 1111\ 111 = 1 + \frac{1}{2} + ... + \frac{1}{2^{23}} = 2(1 - \frac{1}{2^{24}}) = \frac{33\ 554\ 430}{16\ 777\ 216}$

   Ce nombre vaut donc $\frac{33\ 554\ 430}{16\ 777\ 216} \times 2^{127} \simeq 3.40282346 \times 10^{38}$
2. En simple précision le plus petit nombre positif en dehors de la valeur spéciale +0, a pour exposant 1 et pour mantisse $m = 1$

   Ce nombre vaut donc $1 \times 2^{-126} \simeq 1.17549435 \times 10^{-38}$
3. Le nombre suivant a pour mantisse $m = 1,0000\ 0000\ 0000\ 0000\ 0000\ 001 = 1 + 2^{-23}$ et pour exposant 1 donc il y a eu un décalage par rapport au nombre précédent de $2^{-23} \times 2^{-126} = 2^{-149}$

   Cet écart grandit au fur et à mesure que l'exposant augmente ainsi lorsqu'il est à son maximum à 127, l'écart vaut $2^{-23} \times 2^{127} = 2^{104} \simeq 10^{31}$
4. **Deux nombres différents peuvent avoir le même représentant**

```
>>> float.hex(123456789876543210.0)
'0x1.b69b4bd9b38efp+56'

>>> float.hex(123456789876543211.0)
'0x1.b69b4bd9b38efp+56'
```

Ce qui explique le calcul suivant :

```
>>> 123456789876543211.0 - 123456789876543210.0
0.0
```

En conclusion ne pas confondre **le nombre et la représentation du nombre en machine**.

On observe le même problème en cartographie, **aucune carte n'est totalement fidèle au territoire qu'elle est censée représenter**

Exercices

Ex 1
Quel est le nombre décimal représenté par $(1010,1010)_2$
Ex 2
Donner la représentation flottante en simple précision de -10,125
Ex 3
Comment est représenté le nombre entier 7 ? et le flottant 7.0 ?
Ex4

1. Décoder
   1 00011011 10100000000000000000000
2. Décoder
   0 10000011 11100000000000000000000

Ex5
Comment est représenté 1024 sur 32 bits ? Comment est représenté 1024.0 en simple précision ?
Ex6
Expliquer les informations obtenues :

```
>>> float.hex(1024.0)
'0x1.0000000000000p+10'

>>> float.hex(1025.0)
'0x1.0040000000000p+10'

>>> float.hex(1026.0)
'0x1.0080000000000p+10'

>>> float.hex(1027.0)
'0x1.00c0000000000p+10'
```

Ex7
La représentation de 0,1 en double précision est donné par

```
>>> float.hex(0.1)
'0x1.999999999999ap-4'
```

1. Quel nombre décimal cette représentation désigne-t-elle en réalité ?
2. Quelles sont les représentations de 0,2 et 0,3 en double précision et quels sont les nombres décimaux que ces représentations désignent en réalité ?

Ex 8
Lire au moins un exemple d'accident technologique dû à des problèmes d'arrondis `http://ta.twi.tudelft.nl/nw/users/vuik/wi211/disasters.html`
Ex 9
Quel est le plus grand nombre flottant en double précision ?
Ex 10
Quel est le plus petit nombre positif flottant en double précision ?
Ex 11

1. Si on calculait avec des nombres rationnels exacts qu'obtiendrait on à la fin de ce programme

```
x = 1.0
for i in range(20):
    x /= 3.0
for i in range(20):
    x *= 3.0
print(x)
```

2. Exécuter le programme qu'observe-t-on ?
3. Proposer une explication

Ex 12

1. Si on calculait avec des nombres rationnels exacts qu'obtiendrait on à la fin de ce programme

```
x = 1.0
y = x + 1.0
while y - x  == 1.0:
    x *= 2.0
    y = x + 1.0
```

2. Exécuter le programme qu'observe-t-on ?
3. Modifier le programme pour faire afficher le nombre de tours de boucle
4. Proposer une explication