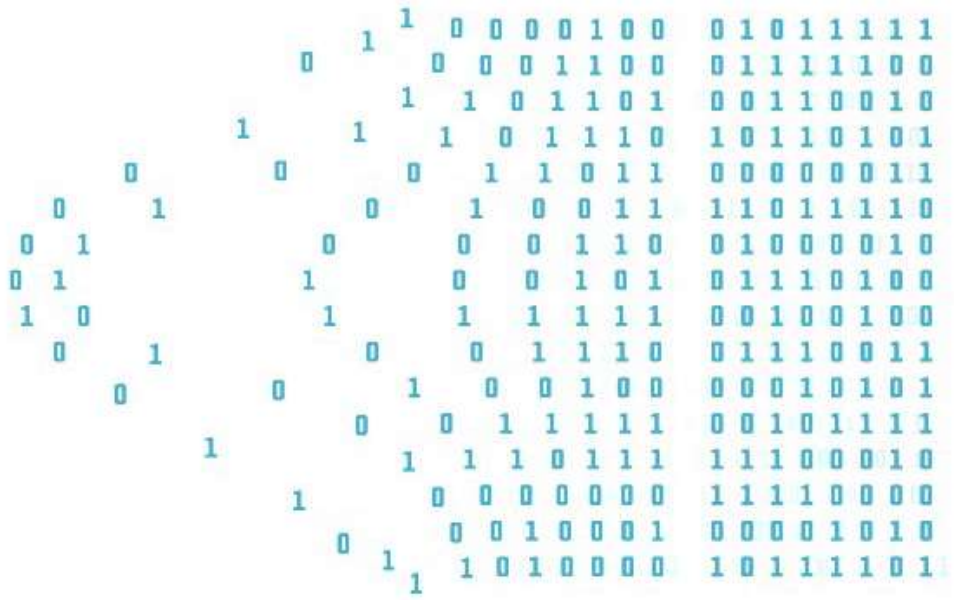
REPRÉSENTATION DES DONNÉES

NSI | 03



REPRÉSENTATION DES NOMBRES RÉELS

[Frédéric PEURIERE - Marion SZPIEG]

Connaître la représentation approximative des nombres réels (à virgule flottante) et en mesurer les conséquences (capacités et arrondis)

1. Petits tests en Python...

Que fait la commande suivante en Python, si a et b sont deux nombres `>>> a - b == 0` ?
Elle teste si les nombres a et b sont égaux : elle renvoie « True » si c'est le cas et « False » sinon.

Taper puis noter ce que renvoie Python :

`>>> 4 - 4 == 0` `>>> 4 - 4.0 == 0` `>>> 0.2 + 0.8 -1 == 0` `>>> 0.2 + 0.6 – 0.8 == 0`

`>>> 0.1 + 0.5 – 0.6 == 0` `>>> 0.1 + 0.2 - 0.3 == 0` `>>> 0.7 + 0.2 - 0.9 == 0`

Êtes vous surpris ? Pourquoi ?

Chercher d'autres cas surprenants et écrivez en quelques uns ici :

Il faut donc ÉVITER de TESTER l'égalité des nombres de type float

Afin de remédier à cela, nous allons effectuer un test du type $|a-b|<\epsilon$ où ϵ est une valeur proche de zéro à définir. Taper la commande suivante :

`>>>abs(0.1+ 0.2 - 0.3)<1E-5` # test adapté aux flottants

Quel résultat est affiché ? Quelle information cela donne t il ?

Créer un programme en Python afin de coder le test ci-dessous. Votre programme contiendra une fonction **égal(***a,b,y,eps***)** qui prendra en argument 4 nombres : *a* et *b* dont on fera la somme qu'on comparera avec *y* le tout avec une précision) *eps* près (10^{-5} dans l'exemple précédent).

Tester ce programme avec *a*=0.1, *b*=0.2 et *y*=0.3 et différentes valeurs de *eps*, et trouver à partir de quelle précision (sous la forme d'une puissance de 10) le test renvoie False.

La représentation des nombres réels est **impossible** car l'ensemble des réels est infini et non dénombrable. (C'est à dire on ne peut pas numéroter les nombres réels). Aucun codage ne peut représenter un intervalle des réels aussi petit qu'il soit. Ainsi la représentation d'un nombre réel est une approximation par un nombre proche. Cela explique les erreurs que vous venez d'observer.

2. Codage des nombres réels en virgule fixe

2.1. Le principe

En base 10, l'expression 652,375 est une manière abrégée d'écrire :

$$6\times10^{2}+5\times10^{1}+2\times10^{0}+3\times10^{-1}+7\times10^{-2}+5\times10^{-3}.$$

On fait exactement pareil pour la base 2. Ainsi, l'expression 110,101 signifie :

$$1\times2^{2}+1\times2^{1}+0\times2^{0}+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}.$$

2.2. Passer de la base 2 à la base 10

Pour convertir un nombre réel de la base 2 vers la base 10, on le décompose à l'aide de puissances de 2 avec des exposants positifs et négatifs. Par exemple :

$110{,}101 = 1\times2^{2}+1\times2^{1}+0\times2^{0}+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}=4+2+0{,}5+0{,}125=6{,}625$

2.3. Passer de la base 10 à la base 2

Le passage de base 10 en base 2 est plus subtil. Par exemple : convertissons 1034,347 en base 2.

- La partie entière se transforme comme vu sur les entiers : $(1034)_{10} = 1024 + 8 + 2 = (10000001010)_2$
- On transforme la partie décimale selon le schéma suivant : on multiplie la partie fractionnaire par 2 et on note la partie entière du résultat, on recommence cette opération avec la partie fractionnaire du résultat. On continue ainsi jusqu'à la précision désirée...On concatène les parties entières obtenues dans l'ordre après la virgule.

$0{,}347 \times 2 = \mathbf{0}{,}694$	0,347 = 0,**0**...
$0{,}694 \times 2 = \mathbf{1}{,}388$	0,347 = 0,0**1**...
$0{,}388 \times 2 = \mathbf{0}{,}776$	0,347 = 0,01**0**...
$0{,}776 \times 2 = \mathbf{1}{,}552$	0,347 = 0,010**1**...
$0{,}552 \times 2 = \mathbf{1}{,}104$	0,347 = 0,0101**1**...
$0{,}104 \times 2 = \mathbf{0}{,}208$	0,347 = 0,01011**0**...
$0{,}208 \times 2 = \mathbf{0}{,}416$	0,347 = 0,0101100**0**...
$0{,}416 \times 2 = \mathbf{0}{,}832$	0,347 = 0,01011000**0**...
$0{,}832 \times 2 = \mathbf{1}{,}664$	0,347 = 0,010110001**1**...
$0{,}664 \times 2 = \mathbf{1}{,}328$	0,347 = 0,0101100011**1**...
$0{,}328 \times 2 = \mathbf{0}{,}656$	0,347 = 0,01011000110**0**...

Attention ! Un nombre à développement décimal fini en base 10 ne l'est pas forcément en base 2 (on dit alors que le nombre n'est pas dyadique)

On dit que cette représentation est **à virgule fixe** car il suffit de définir la position de la virgule et on sait instantanément la valeur du nombre. Par exemple, pour un nombre stocké sur un octet, soit huit bits, si on définit arbitrairement la position de la virgule juste après le quatrième bit, alors on sait que 0110 1001 correspond en fait à 0110,1001.

C'est extrêmement simple **MAIS**, l'inconvénient de cette méthode est que, pour un nombre avec peu de chiffres après la virgule, on perd un espace de stockage significatif. Si le nombre en question est 0110 1000, on perd trois bits "inutilement". De même, le nombre $(1111\ 1111)_2$, qui correspond à 255 en base 10 ne peut pas être représenté sur un octet avec le système de la virgule fixe, alors qu'il n'a besoin que d'un octet pour être écrit. On a donc vite eu envie de trouver une méthode ou il est possible de «changer la virgule de place » en fonction des besoins.

Exercice 1

3. Codage en virgule flottante (écriture scientifique)

3.1. Rappel du fonctionnement avec les puissances de 10

Vue en fin de collège, l'écriture scientifique permet d'écrire un nombre x sous la forme $x = \pm m \times 10^n$ avec :

- le signe du nombre (1 caractère + ou -)
- la mantisse du nombre . $m \in [1,10[$
- l'exposant n : entier relatif.

Exemples : écrire les nombres suivants en écriture scientifique :

$2156 = 2{,}156 \times 10^3$ $\quad -398457{,}5 = -3{,}984575 \times 10^{-5}$ $\quad 0{,}000001452 = 1{,}452 \times 10^{-6}$ $\quad 1{,}34 = 1{,}34 \times 10^0$

Remarque : le nombre 0 n'a pas d'écriture scientifique.

3.2. Adaptation aux puissances de 2

3.2.1. Le principe

Nous représentons des «nombres à virgule» en binaire, sous forme «scientifique» de façon complètement analogue à ceux écrits en base 10.

Ainsi, un nombre réel *x* pourra s'écrire sous la forme : $x=(-1)^s \times m \times 2^e$ où :

- ***s*** est le bit le signe : 0 si positif et 1 si négatif
- ***m*** est un mot binaire appelé mantisse, donc la partie fractionnaire ***f*** (ou fraction) vérifie l'égalité suivante : ***m=1,f*** (en binaire, la mantisse commence forcément par 1,...)
- ***e*** un mot binaire représentant un **entier relatif**, appelé exposant.

Le zéro sera encodé à part.

Par exemple, écrivons l'«écriture scientifique binaire» du nombre *x* = 21, 5 :

$$21{,}5 = (16 + 4 + 1) + \frac{1}{2} = 10101_2 + 0{,}1_2 = 10101{,}1_2 = 1{,}01011_2 \times 2^4 = 1{,}01011_2 \times 2^{100_2}$$

- Le bit de signe est ***s=0***, codé sur 1 bit
- La mantisse est $\boldsymbol{m}=(1{,}01011)_2$ codée sur 5 bits, sa partie fractionnaire est $\boldsymbol{f}=(01011)_2$ codée sur 4 bits
- L'exposant est $\boldsymbol{e}=(100)_2$ codé sur 3 bits.

A l'inverse, avec les informations suivantes, il est possible de retrouver le nombre de départ en base 10 :

- Le bit de signe est ***s=1***, codé sur 1 bit
- La partie fractionnaire est $\boldsymbol{f}=(101)_2$ codée sur 3 bits
- L'exposant est $\boldsymbol{e}=(10000)_2$ codé sur 5 bits.

Le nombre est donc : $-1{,}101_2 \times 2^{10000_2} = -1{,}101_2 \times 2^{16_{10}} = (11010000000000000)_2$

Remarque : sur cet exemple précis, les informations minimales pour retrouver le nombre tiennent sur 1+3+5 = 9 bits, alors que le nombre écrit en entier en base 2 nécessite 17 bits ! Cela signifie qu'avec une place en mémoire de 9 bits, on est capable de représenter un nombre de plus de 9 bits, ce qui est bien sûr très intéressant dans le problème de gain de place en mémoire.

3.2.2. La norme IEEE 754

La norme IEEE 754 est la norme la plus employée pour la représentation des nombres à virgule flottante dans le domaine informatique. La première version de cette norme date de 1985.

Il existe deux formats associés à cette norme : le format dit "simple précision"ou *binary32* (utilise 32 bits pour écrire un nombre décimal) et le format dit "double précision" ou *binary64* (utilise 64 bits pour écrire un nombre décimal).

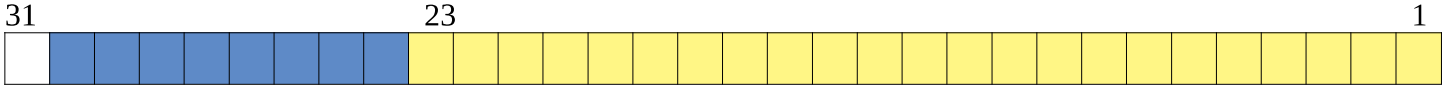
En Python, comme dans la plupart des langages de programmation ou de calcul scientifique, un nombre flottant *x* est représenté sur 32 bits ou 64 bits selon la norme IEEE 754 de la façon suivante :

$$x=(-1)^s \times (m)_2 \times 2^{n-d} = (-1)^s \times 1,(f)_2 \times 2^{n-d}$$

Codage en simple précision (32 bits)

En simple précision,

- le bit de signe ***s*** est le bit de poids fort : si ***s=0***, le signe est positif, sinon il est négatif.
- l'exposant ***n*** est codé sur 8 bits. Il y a un décalage : l'exposant réel est ***n-d*** ou *d=127* (voir ci-dessous)
- la partie fractionnaire ***f*** de la mantisse ***m*** est codée sur les 23 bits restants. Si ***f*** est constituée de moins de 23 bits, on complétera par des zéros à droite.



s exposant ***n*** (8 bits) partie fractionnaire ***f*** de la mantisse (23 bits)

Remarques sur l'exposant :

- Les valeurs extrêmes 0 et 255 pour ***n*** sont des valeurs réservées (voir plus loin)
- Dans la norme IEEE 754, il n'a pas été choisi la méthode du complément à 2 pour représenter l'exposant. En mémoire, le nombre ***n*** est un nombre entier positif. Or, nous avons besoin aussi d'avoir des exposants négatifs pour les nombres inférieurs à 1. Le nombre ***n*** appartenant à l'intervalle [1,254], on choisit un décalage de 127 afin que l'exposant réel ***n-d*** appartienne à l'intervalle [-126,127].

Méthode pour encoder un nombre en virgule flottante en simple précision

Convertir 5,25 en binaire selon la norme IEEE 754, en simple précision

étape 1 : écrire le nombre **non signé** en binaire (virgule fixe) $5{,}25 = 5 + \frac{1}{4} = 5 + \frac{1}{2^2} = 101{,}01_2$

étape 2 : écrire le nombre en « notation scientifique binaire » $5{,}25 = 5 + \frac{1}{4} = 5 + \frac{1}{2^2} = 101{,}01_2 = 1{,}0101_2 \times 2^2$

étape 3 : calculer ***s***, ***f*** et ***n*** :

- le signe est positif donc ***s=0***
- la fraction ***f*** de la mantisse est ce qui se trouve derrière la virgule , donc ***f=0101***
- l'exposant ***n-127***=*2* donc $n = (2+127)_{10} = 129_{10} = 10000001_2$

étape 4 : reprendre les éléments précédents : 5,25 sera représenté en simple précision par :

0	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Exercice 2 question 1)

Méthode pour convertir en base 10 un nombre en virgule flottante en simple précision

Inversement, retrouver le nombre en base 10 qui s'écrit comme ci-dessous en simple précision :

1 0 1 1 1 1 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- Le bit de point fort est un 1, donc le nombre est négatif.
- $n = 01111010_2 = (2^6 + 2^5 + 2^4 + 2^3 + 2^1)_{10} = (64 + 32 + 16 + 8 + 2)_{10} = 122_{10}$

Donc, l'exposant réel sera : ***n***-147 = 122-127 = -5

- la mantisse est : ***m***=1,1011
- le nombre cherché est donc :

$$-1{,}1011 \times 2^{-5} = -0{,}000011011_2 = -(2^{-5} + 2^{-6} + 2^{-8} + 2^{-9}) = -0{,}052734375_{10}$$

Exercice 2 question 2)

Remarque : voici un tableau avec l'encodage des valeurs spéciales :

Bit de signe ***s***	Exposant ***n***	Fraction ***f***	Valeur spéciale
0	0	0	+0
1	0	0	-0
0	11111111	0	+∞
1	11111111	0	-∞
0	11111111	pas 0	NaN

- les valeurs +∞ et -∞ servent à indiquer un dépassement de capacité en mémoire
- NaN, qui signifie « Not a number », sert le résultat d'une opération non valide (division par 0, $\sqrt{-1}$ …)

Petit récap'

Endodage	Signe ***s***	Exposant ***n***	Fraction ***f***	Écriture selon la norme IEEE 754
32 bits (simple précision)	1 bit	8 bits	23 bits	$(-1)^s.(1+f).2^{(n-127)}$
64 bits (double précision)	1 bit	11 bits	52 bits	$(-1)^s.(1+f).2^{(n-1023)}$

La 2e ligne est donnée à titre d'information.

Pour la double précision : exercice 2 question 3) en BONUS

3.3. Conclusion

- Les nombres décimaux et réels n'ont pas tous un codage exact en machine.
- Les calculs et représentations sont nécessairement arrondis et la propagation des erreurs d'arrondi est une problématique délicate.
- Travailler quand on peut avec des entiers plutôt qu'avec des décimaux.
- Il faudra être prudent sur les tests (par exemple, ne pas tester si un nombre flottant A==0, mais plutôt tester si $A<10^{-10}$)
- Les résultats dépendent de propagation d'arrondis faits par la machine.
- On évitera d'additionner deux quantités dont l'écart relatif est très important
- On évitera de soustraire deux quantités très proches

3. A vos machines

Connectez vous alors à https://baseconvert.com/ieee-754-floating-point

1. Que pouvez vous dire de la représentation de 0 ?.....

2. a) Copier la représentation décimale de 0.1 en 32 bits (on peut faire de même en 64 bits) et coller la dans une calculatrice (par exemple https://web2.0calc.fr/)

   b) Additionner de même la représentation de 0.2. Enfin comparez cette valeur avec celle de 0.3 (en faisant pas exemple une soustraction)

   c) Expliquez ce que vous avez trouvé dans la partie 1 (avec le programme)

3. Testez et observez aussi les exemples proposés en bas de page

Exercice 3