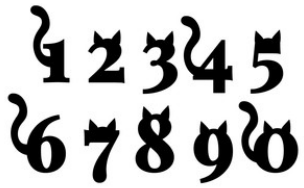
7

Représentation des nombres réels

Extrait du programme



Thème : Types et valeurs de base

Contenus : Représentation approximative des nombres réels : notion de nombre flottant

Capacités attendus : Calculer sur quelques exemples la représentation de nombres réels : 0,1, 0,25 ou 1/3.

Commentaires : 0,2 + 0,1 n'est pas égal à 0,3.
Il faut éviter de tester l'égalité de deux flottants.
Aucune connaissance précise de la norme IEEE-754 n'est exigible.

« Il n'y a que 10 sortes de personnes : celles qui savent compter en binaire et les autres ! »

I. Codage des nombres réels

Une machine n'a pas une mémoire infinie et ne peut donc gérer les nombres réels avec une partie décimale infinie, comme par exemple $\pi, \sqrt{2}, \sqrt{3}, ..., e, \cos(1), ...$ La machine travaille donc avec ce qu'on appelle des **nombres à virgule flottante**, ou plus rapidement des **flottants** ce qui correspond à ce que nous appelons, en mathématique, la notation scientifique.
Par exemple,

- le nombre $0,000\ 145\ 879$ s'écrit en notation scientifique sous la forme $1,458\ 79 \times 10^{-4}$;
- le nombre $-398\ 879,62$ s'écrit $-3,988\ 796\ 2 \times 10^{5}$.

Un nombre à virgule flottante part du même principe, mais en binaire.
Pour commencer voyons comment on peut convertir un nombre à virgule en binaire.

a) Les nombres à virgule en base 2

Nous pouvons nous inspirer ce qui a déjà été vue au **chapitre 3 - Représentation des nombres entiers**.
En base 10, un nombre à virgule peut être décomposé à partir de puissance de 10 :
$348,765 = 3 \times 10^2 + 4 \times 10^1 + 8 \times 10^0 + 7 \times 10^{-1} + 6 \times 10^{-2} + 5 \times 10^{-3}$.
De même un nombre à virgule écrit en base 2 peut se décomposer à partir de puissance de 2 :
$101,011_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 4 + 1 + \frac{1}{4} + \frac{1}{8} = 5 + 0,25 + 0,125 = 5,375.$

Exercice 1 (De la base 2 vers la base 10)
Convertir en base décimale les nombres à virgule suivants écrits en binaire :

- $1101,101_2$ .....
- $10,1101_2$ .....

A nouveau, pour convertir un nombre d'une écriture décimale dans une écriture binaire, c'est une autre paire de manches... Il faut procéder par soustractions successives des puissances de 2 d'exposant négatif... ce qui n'est vraiment pas aisé. Il existe une méthode équivalente que nous allons expliquer ci-dessous avec l'exemple suivant $6,375$.
Nous nous occupons d'abord de la conversion de la partie entière 6 comme nous procédions précédemment : $6 = 110_2$.
Nous nous occupons ensuite de la partie décimale en effectuant des multiplications successives par 2 en mettant de côté la partie entière jusqu'à enfin obtenir une partie décimale nulle. En récupérant les parties entières et en les écrivant de gauche à droite dans leur ordre d'arrivée, nous aurons l'écriture binaire de la partie décimale.

	$0,375 \times 2 = 0,75$	soit $0,75$ plus la partie entière 0	$0..._2$
on reprend la partie décimale précédente $0,75$	$0,75 \times 2 = 1,5$	soit $0,5$ plus la partie entière 1	$01..._2$
on reprend la partie décimale précédente $0,5$	$0,5 \times 2 = 1,0$	soit $\boxed{0,0}$ plus la partie entière 1	011_2

Conclusion : $0,375 = 0,011_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = \frac{1}{4} + \frac{1}{8}$
Donc $6,375 = 110,011_2$.

b) Rigolons un peu

Le programme officiel nous demande de calculer la représentation des nombres réels 0,1 0,25 ou 1/3.

α - Commençons par 0,25 en déroulant la méthode

	$0,25 \times 2 = 0,5$	soit $0,5$ plus la partie entière 0	$0\ldots_2$
on reprend la partie décimale précédente $0,5$	$0,5 \times 2 = 1,0$	soit $\boxed{0,0}$ plus la partie entière 1	01_2

Conclusion : $0,25 = 0,01_2 = 0 \times 2^{-1} + 1 \times 2^{-2} = \frac{1}{4}$.

- Mais Monsieur, votre truc est même pas drôle !
- Patience, patience, petit scarabée...

β - Procédons maintenant avec 0,1

	$0,1 \times 2 = 0,2$	soit $0,2$ plus la partie entière 0	$0\ldots_2$
on reprend la partie décimale précédente $0,2$	$0,2 \times 2 = 0,4$	soit $0,4$ plus la partie entière 0	$00\ldots_2$
on reprend la partie décimale précédente $0,4$	$0,4 \times 2 = 0,8$	soit $0,8$ plus la partie entière 0	$000\ldots_2$
on reprend la partie décimale précédente $0,8$	$0,8 \times 2 = 1,6$	soit $0,6$ plus la partie entière 1	$0001\ldots_2$
on reprend la partie décimale précédente $0,6$	$0,6 \times 2 = 1,2$	soit $\boxed{0,2}$ plus la partie entière 1	$00011\ldots_2$
	...		

- Bon, j'en ai marre, je m'arrête.
- Mais Monsieur, vous ne pouvez pas vous arrêtez comme ça ! ? ? ?
- Si, bien sûr.
- ?
- D'ailleurs pourquoi n'est-ce plus nécessaire de poursuivre ?
- ...
- Petit scarabée ?

Et oui, on retombe sur la partie décimale 0,2 que nous avions au départ. On tombe donc sur un cycle. Autrement dit :
$0,1 = 0,0\underbrace{0011}_{\text{cycle}}\underbrace{0011}_{\text{cycle}}\underbrace{0011}_{\text{cycle}}\underbrace{0011}_{\text{cycle}}\underbrace{0011}_{\text{cycle}}\underbrace{0011}_{\text{cycle}}\ldots_2$

- Mais Monsieur, ..., vous trouvez ça drôle ?
- Tout dépend des jours.

Malgré tout, cet exemple est **important** car il nous montre que ce petit nombre totalement anodin 0,1 n'est pas codable en binaire avec un nombre fini de bits. Donc la machine n'aura jamais la valeur exacte de 0,1 en mémoire mais une valeur approximative ... et quand on est approximatif ... on commet des erreurs.

γ - Allez, un petit dernier pour la route

$$\frac{1}{3} \times 2 = \frac{2}{3} \quad \text{soit } \frac{2}{3} \text{ plus la partie entière } 0 \quad 0\ldots_2$$

on reprend la partie décimale précédente $\frac{2}{3}$ $\quad \frac{2}{3} \times 2 = \frac{4}{3} = \frac{3}{3} + \frac{1}{3} = 1 + \frac{1}{3} \quad$ soit $\boxed{\frac{1}{3}}$ plus la partie entière 1 $\quad 01\ldots_2$

$$\ldots$$

Bon, vous avez compris, à nouveau on retombe sur un cycle :
$\frac{1}{3} = 0,\underbrace{01}_{\text{cycle}}\underbrace{01}_{\text{cycle}}\underbrace{01}_{\text{cycle}}\underbrace{01}_{\text{cycle}}\underbrace{01}_{\text{cycle}}\underbrace{01}_{\text{cycle}}\ldots_2$
Le spécialiste aura remarqué tout de suite qu'avec l'ordinateur russe Setun qui fonctionnait dans une base 3, on aurait eu tout simplement $\frac{1}{3} = 0,1_3$ mais il y aurait quand même eu quelques soucis sur d'autres nombres...

Exercice 2 (De la base 10 vers la base 2)
Convertir en binaire les nombres à virgule suivants :

- $8,25$ .....
- $3,625$ .....

c) Quoi ! ? ? ? $0,1 + 0,2 \neq 0,3$ Mais c'est une escroquerie !

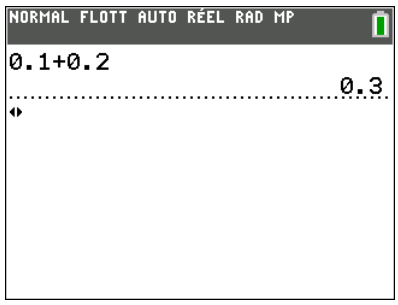
- Mais Monsieur, Python est vraiment nul ! Je lui demande si « `0.1 + 0.2 == 0.3` » est voilà ce qu'il me répond !

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

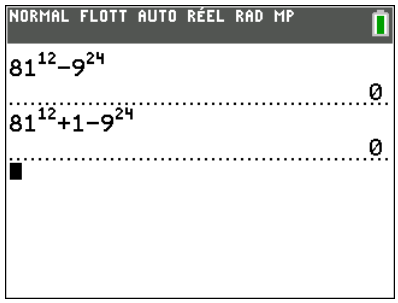
C'est une véritable escroquerie !
- Bien. Je vois que tu n'as pas suivi mon cours. As-tu essayé de faire 0,1 + 0,2 sur Python ?

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

- Oui, c'est bien ce que je dis, c'est une véritable escroquerie ! Parce que ma calculatrice, elle, elle sait faire !



- Effectivement, petit scarabée. Par contre ta calculatrice, elle aussi, elle commet des erreurs. Par exemple, essaye de faire les 2 calculs suivant $81^{12} - 9^{24}$ *puis* $81^{12} + 1 - 9^{24}$*...*



- ...
- *Bon, comme tu le sais* $81^{12} - 9^{24} = 0$*, n'est-ce pas ?*
- ...
- *Donc* $81^{12} + 1 - 9^{24} = 1$*. Serait-ce plutôt ta calculatrice*[1] *qui serait nulle ? L'escroquerie ne serait-elle pas là ? Voyons ce que fait Python ...*

```
>>> 81**12-9**24
0

>>> 81**12+1-9**24
1
```

- ...

d) La norme IEEE 754 (Cette partie est à la limite du programme)

Après quelques tentatives et hésitations, la norme IEEE 754 s'est imposée. C'est une norme pour la représentation des nombres à virgule flottante en binaire. Son nom complet est *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*, autrement dit, norme IEEE pour l'arithmétique binaire en virgule flottante.

Le format général d'un nombre flottant est formé de trois éléments : la mantisse (m bits), l'exposant (e bits) et le signe (1 bit).

Source Image : Wikipédia

L'interprétation d'un nombre est donc :

$$\text{valeur} = \text{signe} \times \text{mantisse} \times 2^{\text{exposant}-\text{décalage}}$$

avec

- *décalage* $= 2^{e-1} - 1$ qui permet de gérer le signe de l'exposant.

A cela, on ajoute trois cas particuliers :

- si l'exposant et la mantisse sont tous deux nuls, le nombre est ± 0, selon le bit de signe.
- si l'exposant est égal à $2^e - 1$ et si la mantisse est nulle, le nombre est $\pm$ infini, selon le bit de signe.
- si l'exposant est égal à $2^e - 1$, mais que la mantisse n'est pas nulle, le nombre est **NaN**, *not a number*.

1. La **numworks**, calculatrice de conception française, ne fait pas cette erreur, elle !

α - Format simple précision (32 bits)

Il s'agit d'un nombre à virgule flottante stocké sur 32 bits. Il s'agit du type **float** pour flottant en **Java**.

Source Image : Wikipédia

- 1 bit pour le signe
- 8 bits pour l'exposant
- 23 bits pour la mantisse
- l'exposant est décalé de $2^{8-1}-1=127$. Ainsi l'exposant d'un nombre normalisé va donc de -126 à +127.
  L'exposant décalé -127 est réservé pour zéro.
  L'exposant décalé +128 est réservé pour les infinis et les **NaN**.

Par exemple le nombre 0 01111100 01000000000000000000000 :
- le signe 0 est nul, il s'agit d'un nombre positif
- l'exposant 01111100_2 équivaut à 124 en écriture décimal. Avec le décalage, on obtient : 124 - 127 = -3
- la mantisse 01000000000000000000000 signifie que la partie significative en binaire est de $1{,}01000000000000000000000 = 1{,}01_2$. Ce qui équivaut en écriture décimal à $1\times 2^0+0\times 2^{-1}+1\times 2^{-2}=1,25$.

Le nombre représenté par 0 01111100 01000000000000000000000 est $1,25\times 2^{-3}=0,15625$.

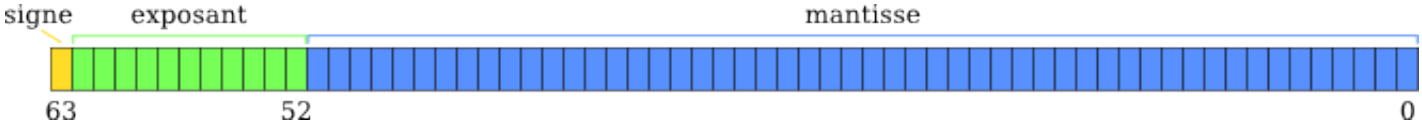
Le format simple précision permet d'obtenir des nombres entre $\approx \pm 10^{-40}$ et $\pm 10^{40}$ avec 6 ou 7 décimales.

Exercice 3
Donner la valeur décimale des nombres flottants suivants codés en simple précision :
- 1 01111110 11110000000000000000000 .....
- 0 10000011 11100000000000000000000 .....

Exercice 4
Donner la représentation flottante en simple précision de :
- 128 .....
- $-32,75$ .....

β - Format double précision (64 bits)

Il s'agit d'un nombre à virgule flottante stocké sur 64 bits.
En **Java** c'est le type **double**, en **Python 3**, il s'agit du type **float** .



Source Image : Wikipédia

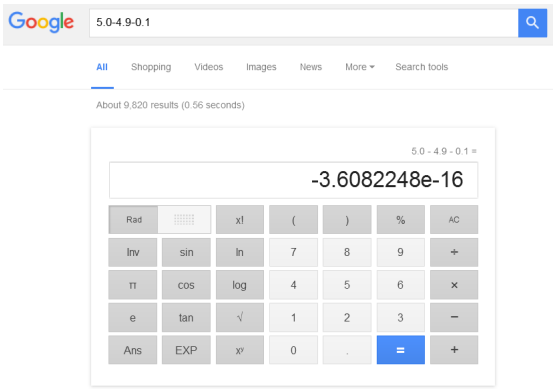
- 1 bit pour le signe
- 11 bits pour l'exposant
- 52 bits pour la mantisse
- l'exposant est décalé de $2^{11-1}-1=1023$. Ainsi l'exposant d'un nombre normalisé va de -1022 à +1023.

Le format double précision permet d'obtenir des nombres entre $\approx \pm 10^{-300}$ et $\pm 10^{300}$ avec 14 ou 15 décimales.

γ - Enfin nous comprenons ce qu'il se passe

Par défaut, les flottants sont codés en double précision, autrement dit les résultats sont approchés 56 bits après la virgule.
$2^{56} \approx 7,2 \times 10^{16}$
Nous retrouvons cette erreur d'approximation sur beaucoup de logiciel.

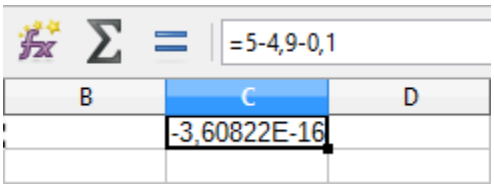
Voici par exemple ce que fait notre ex-ami Google :



Même notre nouvel ami Qwant fait de même, mais avec une « meilleure précision ». Il obtient le résultat :

$$-3,6082248300318 \times 10^{-16}$$

Continuons avec nos logiciels préférés. Calc de LibreOffice 5.1.5.2 ne s'en sort pas mieux :



Et comme nous l'avions déjà vu avec Python, nous avons :

```
>>> 5.0-4.9-0.1
-3.6082248300317596e-16
```

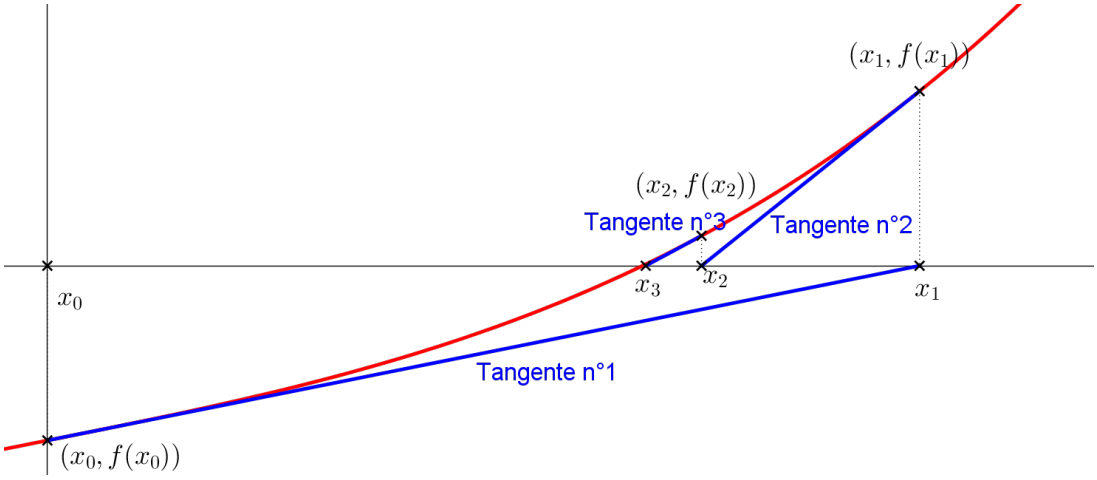
En résumé, il ne faut jamais effectuer de comparaison entre les nombres flottants!!!

- Bien chef! Oui chef!
- C'est bien petit scarabée, tu commences à comprendre.

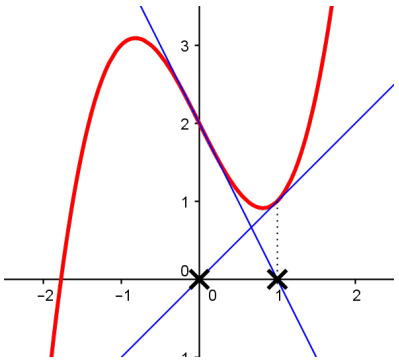
II. Quelques précédents fâcheux

a) Un truc rigolo en math

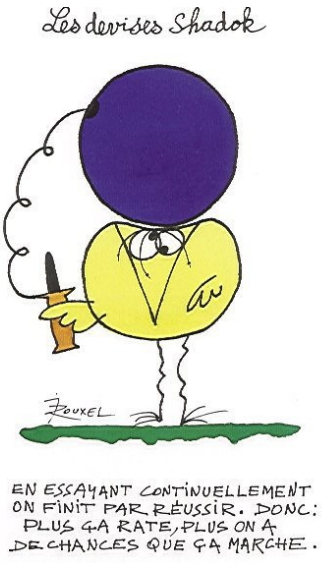
En mathématiques on peut utiliser ce qu'on appelle la méthode de Newton, une méthode qui permet d'obtenir une approximation de l'abscisse du point d'intersection d'une courbe et de l'axe des abscisses. Cette méthode utilise des tangentes, et sous certaines conditions, à chaque itération, le point d'intersection de la tangente avec l'axe des abscisses se rapproche du point recherché.



Pour la fonction $f(x) = x^3 - 2x + 2$, si on cherche le point d'intersection de sa courbe représentative avec l'ax des abscisses, la méthode de Newton peut convenir, mais pas dans n'importe quelle condition. Ainsi, si on cherche en partant de la tangente au point d'abscisse 1, on obtient ensuite la tangente au point d'abscisse 0 puis ensuite la tangente au point d'abscisse 1, puis après ... Bref, on tourne en rond avec uniquement ces 2 tangentes. La méthode, en théorie, ne marche pas.



Bizarrement avec une machine, comme il y a une petite erreur de calcul, on s'éloigne assez rapidement de la boucle entre les deux tangentes car dès la première on ne trouve pour abscisse 0, mais *presque* 0. Cette erreur est reprise dans les calculs suivants et s'amplifie... Au bout de 65 itérations on obtient alors une très bonne approximation du nombres recherchés !
$\alpha \approx -1,76929235...$
Pour une fois que les Shadoks avaient raison !



b) L'explosion d'Ariane 5 en 1996

Source Image : Brèves de maths
http://www.breves-de-maths.fr/meme-les-ordinateurs-font-des-erreurs/

Le premier vol du nouveau lanceur Ariane 5 a eu lieu le 4 juin 1996. Après 30 secondes de vol, le lanceur, alors à une altitude de 3700 m, a soudainement basculé, quitté sa trajectoire, s'est brisé et a explosé. L'échec était dû à la perte totale des informations de guidage et d'altitude, 37 secondes après la mise à feu du moteur Vulcain.
Le Système de Référence Inertiel a calculé une accélération horizontale beaucoup plus grande pour Ariane 5 que pour Ariane 4. Cette valeur flottante sur 64 bits n'a pu être convertie en un entier sur 16 bits.
Comble d'ironie, ce calcul était inutile pour Ariane 5.
Tous les logiciels ont été soigneusement vérifiés avant le lancement d'Ariane 502.
Cette erreur a coûté plus de 500 millions de dollars.
Source Texte : Supinfo
https://www.supinfo.com/articles/single/235-erreur-informatique-plus-couteuse-histoire

c) Moins drôle, l'échec des anti-missiles Patriot

Les anti-missiles Patriot ont été utilisés en 1991 durant la guerre du Golf. Leur rôle était d'intercepter les missiles ennemis avant qu'ils n'atteignent leur but.
A chaque tic de l'horloge (10 fois par seconde), la valeur 1/10 était ajoutée à un compteur. Mais 1/10 n'est pas exact pour un ordinateur (d'environ 0,000001% pour un nombre flottant sur 32 bits) et c'est donc la valeur 0,1000000014901... qui était chaque fois ajoutée. Cette toute petite erreur a été répétée pendant plus de 100 heures et ces erreurs, toujours dans le même sens, se sont accumulées jusqu'à ce que la différence entre les diverses horloges fasse une erreur proche d'une seconde, ce qui, à la vitesse très importante de l'anti-missile, équivalait à un décalage de près de 600 mètres.
Le 25 février 1991, le système radar détecta un Scud irakien mais tous les anti-missiles Patriot lancés le ratèrent inévitablement. Le Scud frappa la caserne de Dhahran, en Arabie Saoudite, tuant 28 soldats du centre de commandement du 14e détachement de l'armée des États-Unis.
Au début, on crut à un défaut de cette batterie et on la retira du service en moins d'une journée. La réalité était toute autre : les Israéliens avaient déjà identifié le problème et informé l'armée des États-Unis ainsi que le fabricant du logiciel de tir le 11 février 1991 mais aucune mise à niveau n'existait alors. On avait demandé à défaut d'autre chose aux commandants d'unités d'effectuer des réinitialisations régulières du système mais cette mesure avait dû se révéler insuffisante pour Dharan car les militaires n'en avaient pas compris l'utilité. Le fabricant parvint à fournir une mise à jour le 26, un jour trop tard pour les 28 militaires de Dharan...
Source Wikipédia et Brèves de Maths
`http://www.breves-de-maths.fr/meme-les-ordinateurs-font-des-erreurs/`