Informatique PCSI
Les nombres flottants : représentation

Nous savons représenter en machine un nombre entier si ce nombre est compris entre deux bornes qui dépendent du nombre d'octets utilisés. Il n'y en a qu'un nombre fini. Pour les nombres réels, c'est plus compliqué. Ces nombres sont de différents types : entier comme 3, décimal comme $-4{,}25$ ou 0,1, rationnel comme $1/3$, irrationnel comme $\sqrt{2}$ et π. Il y a en a une infinité même entre deux bornes quelconques. De plus avec des mots de taille fixe, nous ne pouvons stocker qu'un nombre fini de décimales. Donc, même entre deux bornes comme 1 et 2, nous ne pouvons représenter qu'un nombre fini de réels de manière exacte, et pour tous les autres nous utilisons une valeur approchée.

1 Ensembles de nombres

1.1 Les nombres décimaux

Les nombres décimaux ont un statut particulier qui est lié à l'écriture en base dix. Un nombre décimal peut s'écrire $n/10^p$, où n et p sont des entiers relatifs. En notation décimale positionnelle, il peut s'écrire avec un nombre fini de chiffres après la virgule. En binaire, on utilise une écriture semblable $n/2^p$. En base trois on utiliserait une écriture comme $n/3^p$.

Par exemple, le nombre décimal qui s'écrit en base dix $15/10$, soit 1,5, s'écrit en base deux $1111/1010$ ou $11/10$, soit 1,1. Dans les deux bases, le nombre de chiffres après la virgule est fini. Ce ne serait pas le cas en base trois.

Considérons le nombre $7/3$ qui n'est pas un décimal. Dans une écriture en base dix ou en base deux, le nombre de chiffres après la virgule est infini. Ce n'est pas le cas en base trois puisque ce nombre $(6/3+1/3)$ s'écrit 2,1.

Considérons le décimal $16/10$, soit $1,6$ en base dix. En base deux, le nombre de chiffres après la virgule est infini.

Un nombre qui s'écrit $n/2^p$ peut s'écrire $5^p n/10^p$. Donc si un nombre a une écriture finie en base deux, c'est aussi le cas en base dix.

La réciproque est fausse. Si $n/10^p = m/2^e$, alors $n \times 2^e = m \times 10^p$ et l'arithmétique nous dit que n doit être divisible par 5. Donc parmi les nombres $n/10$ où n est un entier compris entre 10 et 20, seul les nombres $10/10$, $15/10$ et $20/10$ ont une écriture finie en base deux.

En Python, on dispose de deux types de nombres. Les nombres de type int représentent des entiers de manière exacte. Ils constituent un ensemble fini inclus dans $\mathbb{Z}$. Avec des mots de n bits, on peut représenter en complément à deux tous les entiers signés de -2^{n-1} à $2^{n-1}-1$. Les nombres de type float servent à représenter de manière exacte ou approchée des nombres réels.

1.2 Le type float

L'ensemble des nombres de type float en Python est inclus dans $\mathbb{D}$. On peut dire aussi que c'est une partie finie de l'ensemble des nombres réels. Ce sont des nombres écrits en virgule flottante avec des mots de taille fixe, et on abrège en flottants. Pour la suite, on appelle *flottant* un nombre de type float. Il y a un plus petit nombre de type float, un plus grand nombre, et entre les deux, tous les nombres de type float sont répartis de manière non uniforme. Par exemple une partie importante de ces nombres est dans l'intervalle $[-1;+1]$.

Le nombre deux est un entier, un décimal, un rationnel, un réel. Il peut s'écrire sous forme entière 2, sous forme décimale $2,0$ ou $200/100$, mais aussi comme une fraction $6/3$, ou avec un radical $\sqrt{4}$. Ce nombre deux en Python est de type int si on l'écrit par exemple `2`, ou `6//3`, et de type float si on l'écrit `2.0`, `200/100`, `200*1e-2`, `6/3`, `sqrt(4)`.

Le nombre deux est représenté de manière exacte en machine, ce qui entraîne que l'expression `2 == 2.0` vaut `True`. Le nombre décimal $0,1$ n'a pas de représentation exacte avec le type float. Lorsqu'on écrit `0.1` en Python, c'est le nombre de type float le plus proche de $0,1$ qui est enregistré en machine et

utilisé pour les calculs. C'est pourquoi l'expression `0.1+0.1+0.1 == 0.3` vaut `False` par exemple. Mais aussi parce-que 0.3 n'a pas non plus de représentation exacte en machine avec le type float.

En fait `0.3` est représenté en machine par $0.29999999999999989\ldots$, et `0.1` est représenté par $0.10000000000000055\ldots$. Donc si on calcule `0.1+0.1+0.1` on obtient avec les arrondis la représentation $0.30000000000000004\ldots$ qui n'est pas celle de `0.3`.

Le seul nombre représenté de manière exacte parmi 0.1, 0.2, 0.3, ..., 0.9 est 0.5. Cela s'explique par le fait que $0,5$ peut s'écrire $0,1$ en binaire ($=1/2$), comme $0,25$ s'écrit $0,01$ en binaire ($=1/2^2$).

Répartition

L'image des graduations sur une règle, un double décimètre par exemple, donne une idée de la répartition des flottants. Il y a peu de graduations par rapport à l'infinité de nombres réels compris entre 0 et 20. Donc n'importe quel réel compris entre deux graduations sera approché par l'une des deux valeurs.

La principale différence avec la répartition des flottants est que ces derniers ne sont pas répartis de manière uniforme entre deux bornes comme sur la règle, mais ils deviennent de plus en plus rares à mesure qu'on s'éloigne de 0.

Il y en a environ 2^{52} flottants entre 1 et 2 mais il n'y en a aucun entre 9007199254740992 et 9007199254740994 (2^{53} et $2^{53}+2$).

```
>>> 2.0 == 2
True
>>> 2.0**53 == 2**53
True
>>> 2.0**53+1 == 2**53+1
False
>>> 2.0**53+1 == 2.0**53
True
```

La différence entre deux flottants consécutifs (sur une architecture 64 bits), dans $[1;2]$ est 2^{-52}, celle entre deux flottants consécutifs dans $[2;4]$ est 2^{-51}, etc. Celle entre deux flottants consécutifs dans $[2^p;2^{p+1}]$ est 2^{p-52}. Et donc les flottants supérieurs à 2^{52} sont tous des entiers (au sens mathématique).

La moitié des flottants est composée de flottants négatifs, l'autre moitié de flottants positifs. Parmi les positifs, la moitié est composée de flottants inférieurs à 1, l'autre moitié de flottants supérieurs à 1. Les flottants supérieurs à 1 sont (ou représentent) en grande majorité des entiers, avec des trous de plus en plus grands entre deux flottants. Par exemple il n'y a aucun flottant entre les deux nombres 2^{1023} et $2^{1023}+2^{970}$:

```
2.0**1023 == 2.0**1023 + 2.0**970
True
```

2 Les nombres de type float

Nous sommes habitués à la notation positionnelle en base dix. Et comme avec les entiers, la notation positionnelle en base b avec $b \geq 2$ suit le même principe.

$(a_n \ldots a_1 a_0 a_{-1} \ldots a_{-p})_b = a_n b^n + \ldots + a_1 b + a_0 + a_{-1} b^{-1} + \ldots + a_{-p} b^{-p}$.

Par exemple : $(213,03)_4 = 2\times 4^2 + 1\times 4^1 + 3\times 1 + 0\times 4^{-1} + 3\times 4^{-2} = 371/16$.

En base b, nous utilisons b chiffres, de 0 à $b-1$. Ici encore, il apparaît clairement que le rôle du zéro dans l'écriture positionnelle est capital, par exemple pour distinguer 213,3 de 213,03.

2.1 Écriture d'un nombre

Exemple d'une écriture d'un nombre à virgule en base deux :

$101,101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$, soit $5,625$ en base dix.

Réciproquement, si on donne l'écriture en base dix, on commence par séparer la partie entière de la partie décimale :

$13,875 = 13 + 0,875$, et $13_{10} = 1101_2$

La méthode consiste ensuite à effectuer des multiplications par deux successives :

$0,875 = 1,75 \times 2^{-1} = 1/2 + 0,75 \times 2^{-1} = 1/2 + 1,5 \times 2^{-2} = 1/2 + 1/4 + 0,5 \times 2^{-2}$

$= 1/2 + 1/4 + 1 \times 2^{-3} = 1/2 + 1/4 + 1/8$, soit $0,111_2$ en base deux.

Finalement le nombre s'écrit $1101,111_2$.

Considérons le nombre $1000/1010 = 10^3/(10^3 + 10)$ en base dix.

Ce nombre s'écrit $0,990099009900\ldots$

De manière générale, le nombre $1000/1010$ écrit en une base b quelconque, soit $b^3/(b^3 + b)$, s'écrit toujours $0,(b-1)(b-1)00(b-1)(b-1)00(b-1)(b-1)00\ldots$

En base deux $1000/1010$ s'écrit $0,110011001100\ldots$ puisque $2 - 1 = 1$. C'est exactement l'écriture binaire du nombre dont l'écriture décimale est $8/10$. Cette écriture comporte un nombre de chiffres infinis après la virgule et la séquence 1100 est répétée indéfiniment. En divisant par 8, donc en déplaçant la virgule de trois places vers la gauche, on obtient l'écriture de $1/10$ qui est aussi infinie, soit $0,000110011001100\ldots$

On peut en déduire que des nombres comme 2/10, 4/10 ont une écriture infinie, mais aussi 16/10, 32/10, etc. Et plus généralement, tous les nombres de la forme $2^n/10$ en base dix ont une écriture infinie en base deux.

On a une propriété encore plus générale.

Propriété

Si un nombre a une écriture infinie en base deux, alors le nombre obtenu après l'ajout d'un entier, le retrait d'un entier, une multiplication par deux ou une division par deux, a aussi une écriture infinie en base deux.

Par exemple, puisque $8/10$ a une écriture infinie en base deux, on en déduit que c'est le cas pour les nombres suivants : $4/10$, $2/10$, $1/10$, $16/10$ donc $6/10$, $3/10$, $1 + 4/10$ donc $14/10$, donc $7/10$, $1 + 8/10$ soit $18/10$ donc $9/10$.

2.2 Représentation d'un nombre

La représentation en virgule flottante dans un ordinateur ressemble à la notation scientifique en numération binaire. On représente un nombre x par : un signe ($+$ ou $-$), une mantisse a et un exposant p (entier relatif), soit en base dix $x = \pm a \times 10^p$ où $1 \leq a < 10$ et en base deux $x = \pm a \times 2^p$ où $1 \leq a < 2$. En machine le signe est remplacé par 0 ou 1.

Il suffit d'un bit en machine pour coder le signe. Le choix fait est 0 pour les nombres positifs et 1 pour les nombres négatifs.

Observons l'écriture en base deux de quelques nombres décimaux.

2^i	$\ldots$	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	$\ldots$
3	0	0	0	1	1	0	0	0	0
5	0	0	1	0	1	0	0	0	0
4,5	0	0	1	0	0	1	0	0	0
0,25	0	0	0	0	0	0	1	0	0

Le nombre 3 s'écrit 11 soit 1,1 $\times$ 10 en binaire, ($1,5 \times 2$ en décimal).

Le nombre 5 s'écrit 101 soit 1,01 $\times$ 100 en binaire, ($1,25 \times 2^2$ en décimal).

Le nombre $4,5$ s'écrit 100,1 soit 1,001 $\times$ 100 en binaire, ($1,125 \times 2^2$ en décimal).

Le nombre $0,25$ s'écrit 0,01 soit 1,0 $\times$ 0,01 en binaire, ($1,0 \times 2^{-2}$ en décimal).

Ces quatre nombres peuvent donc s'écrire sous la forme $1,\ldots \times 2^p$ en décimal. La partie $1,\ldots$ est la mantisse, p est l'exposant. Cette mantisse appartient à $[1;2[$ et son écriture binaire commence donc toujours par un 1. Il n'est donc pas nécessaire de coder ce 1 en machine.

Prenons le cas du nombre $0,2$ et cherchons son écriture binaire. Il s'agit de l'écrire sous la forme $1,\ldots \times 2^p$ en décimal. $0,2 = 0,4 \times 2^{-1} = 0,8 \times 2^{-2} = 1,6 \times 2^{-3}$. Il faut donc écrire 0,6 en binaire. Or $0,6 = 1,2 \times 2^{-1}$ et il faut écrire $0,2$ en binaire. Nous commençons à "tourner en rond". Ceci signifie que l'écriture binaire nécessite une infinité de décimales exactement comme pour l'écriture en base dix de $1/3 = 1,333333\ldots$. Nous constatons à nouveau que ce nombre ne peut pas avoir une représentation exacte en machine.

Or, $0,2 = 3,2 \times 2^{-4} = 3 \times 2^{-4} + 0,2 \times 2^{-4} = 3 \times 2^{-4} + (3 \times 2^{-4} + 0,2 \times 2^{-4}) \times 2^{-4} = \ldots$
Et nous pouvons montrer que $0,2 = 3 \times 2^{-4} + 3 \times 2^{-8} + 3 \times 2^{-12} + 3 \times 2^{-16} + \ldots$
Son écriture en binaire est donc 0,0011 0011 0011 ....

Remarquons que $0,1 = 0,2 \times 2^{-1}$. Donc le problème est identique pour 0,1, il n'a pas de représentation exacte en machine. Son écriture binaire s'obtient à partir de celle de 0,2 en déplaçant la virgule d'un cran vers la gauche (division par deux), soit 0,0001 1001 1001 ....

Pour représenter les nombres réels sur des mots de taille fixe, nous utilisons donc des valeurs, exactes pour certains nombres, mais approchées pour la plupart.

Une valeur est un nombre décimal qui a une écriture sous forme scientifique en base deux, soit s $m \times 2^p$, avec $1 \leq m < 2$ et $pmin \leq p \leq pmax$.

Les valeurs de la forme s $m \times 2^p$, avec $0 \leq m < 1$ sont aussi autorisées afin de pouvoir coder le zéro et avoir plus de valeurs proches de zéro.

Normes

Différentes normes existent comme la norme IEEE-754 qui est la plus courante. Elle définit précisément le codage des nombres en virgule flottante et les standards pour les représenter en machine et calculer en binaire. Il ne s'agit pas de retenir les détails de cette norme ou d'une autre, mais de comprendre certaines conséquences des choix qui sont faits pour une norme.

Avec des mots de 64 bits, la norme IEEE-754 précise la règle suivante :

- un bit pour le signe, 0 pour le signe $+$ et 1 pour le signe $-$;
- 11 bits pour l'exposant décalé e qui vaut $p + 1023$ avec la condition $-1022 \leq p \leq 1023$, donc $1 \leq e \leq 2046$ (les valeurs 0 et 2047 sont réservées pour coder par exemple $-\infty$ ou $+\infty$) ;
- 52 bits pour la mantisse tronquée m' qui vaut $m - 1$ avec la condition $1 \leq m < 2$.

Ces trois parties sont codées en binaire et concaténées pour former un mot de 64 bits.

Le plus grand nombre flottant a une mantisse égale en binaire à 1,111 ... 1 avec 52 chiffres égaux à 1 après la virgule et un exposant égal à 1023. Sa valeur est donc $(1 + 2^{-1} + 2^{-2} + \ldots + 2^{-52}) \times 2^{1023}$ soit $2^{1023} + 2^{1022} + \ldots + 2^{971}$.

En Python :

```
>>> s = 0
>>> for i in range(971, 1024):
        s = s + 2.0 ** i

>>> s
1.7976931348623157e+308
```

Un nombre réel compris entre ce plus grand nombre et son opposé est alors représenté en machine par le flottant qui en est la meilleure approximation possible.

Exemples

Le réel 20 s'écrit $+1,25 \times 2^4$.

Le signe est $+$ donc le premier bit vaut 0.

L'exposant est 4 donc l'exposant décalé est $4 + 1023 = 1027$, soit 100 0000 0011 en binaire.

La mantisse est $1,25$ qui s'écrit en binaire 1,01. On garde la partie décimale 01 et on complète avec des zéros. Le codage de 20 est donc : 0 100 0000 0011 0100 0000 ... 0000.

Le réel -0.375 s'écrit $-1,5 \times 2^{-2}$.
Le signe est $-$ donc le premier bit vaut 1.
L'exposant est -2 donc l'exposant décalé est $-2 + 1023 = 1021$, soit 011 1111 1101 en binaire.
La mantisse est $1,5$ qui s'écrit en binaire 1,1. On garde la partie décimale 1 et on complète avec des zéros. Le codage de $-0,375$ est donc : 1 011 1111 1101 1000 0000 ... 0000.

Le réel $0,1$ s'écrit $1,6 \times 2^{-4}$. Le signe est $+$, donc le premier bit vaut 0. L'exposant décalé vaut $-4+1023 = 1019$, soit 011 1111 1011. La mantisse de $1,6$ se déduit de celle de $0,8$ rencontrée auparavant. La mantisse tronquée s'écrit en binaire : 1001 1001 ... 1001 1001 ... Il s'agit de l'arrondir. Comme le 53ème chiffre est un 1, on arrondit par valeur supérieure, (1001,1 est arrondi à 1010), et donc on code : 1001 1001 ... 1001 1010.
La valeur approchée représentant $0,1$ est donc : 0 011 1111 1011 1001 1001 ... 1001 1010.

Pour calculer $0,1 + 0,1$, nous calculons $1,6 \times 2^{-4} + 1,6 \times 2^{-4} = 1,6 \times 2^{-3}$. Nous effectuons la somme des deux mantisses :

	1,	1	0	0	1	...	1	0	0	1	1	0	0	1	1	0	1	0	
+	1,	1	0	0	1	...	1	0	0	1	1	0	0	1	1	0	1	0	
1	1			1	1	...	1		1	1			1	1		1			retenues
1	1,	0	0	1	1	...	0	0	1	1	0	0	1	1	0	1	0	0	

Ce nombre représente $1 + 0,6 + 1 + 0,6 = 2(1 + 0,6)$.
On divise alors le résultat par 2, donc on décale la virgule d'un cran vers la gauche et on supprime le 0 à droite en trop. On obtient alors 1,1001 1001 ... 1001 1010 qui est bien la mantisse dans la représentation de 1,6 ou de 0,2.

Calculons $0,2+0,1 = 1,6\times2^{-3}+1,6\times2^{-4} = 1,6\times2^{-3}+0,8\times2^{-3}$. Nous allons ainsi comprendre pourquoi en machine $0,2 + 0,1$ donne 0,30000000000000004. Pour cela nous ajoutons à la mantisse de la représentation de 1,6 la moitié de cette mantisse (pour 0,8).

Nous posons l'addition de ces deux nombres en binaire.

	1,	1	0	0	1	...	1	0	0	1	1	0	0	1	1	0	1	0	
+	0,	1	1	0	0	...	1	1	0	0	1	1	0	0	1	1	0	1	
1	1			1	1	...			1	1			1	1					retenues
1	0,	0	1	1	0	...	0	1	1	0	0	1	1	0	0	1	1	1	

On divise le résultat par 2 donc on supprime le 1 à droite en trop et comme on arrondit, on obtient 1,0011 0011 ... 0011 0100. Or, $0,3 = 1,2 \times 2^{-2}$ et 0,2 s'écrit 0,0011 0011 ... 0011 (voir plus haut). Donc 1,2 est représenté par la mantisse 1,0011 0011 ... 0011 0011 et la différence entre les deux mantisses 1,0011 0011 ... 0011 0100 et 1,0011 0011 ... 0011 0011 est 2^{-52}. En multipliant par 2^{-2} on obtient la différence entre $0,2 + 0,1$ et $0,3$ qui est 2^{-54}, soit environ $5,55 \times 10^{-17}$.

Dans l'interpréteur Python :

```
>>> (0.2 + 0.1) - 0.3
5.551115123125783e-17
>>> 2 ** (-54)
5.551115123125783e-17
```

Nous avons des erreurs d'arrondi mais aussi des troncatures.
Reprenons la mantisse trouvée pour $0,2 + 0,1$ et ajoutons la mantisse de $0,1$ divisée par 4 pour avoir le même exposant -2 (car $0,1 = 1,6 \times 2^{-4} = 0,4 \times 2^{-2}$).

$$\begin{array}{cccccccccccccccccc}
 & 1, & 0 & 0 & 1 & 1 & \dots & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \\
+ & 0, & 0 & 1 & 1 & 0 & \dots & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \\
 & & 1 & 1 & & & \dots & 1 & 1 & & & 1 & 1 & & & 1 & & & & \text{retenues} \\
\hline
 & 1, & 1 & 0 & 0 & 1 & \dots & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 &
\end{array}$$

Nous obtenons la mantisse représentant $1,6$ qui code le nombre $0,4 = 1,6 \times 2^{-2}$. Donc en Python, `(0.2+0.1)+0.1` donne exactement la représentation de `0.4`.

Les résultats qui suivent sont utiles pour les calculs.

- L'écart entre deux nombres flottants consécutifs compris entre 1 et 2 est $\epsilon = 2^{-52}$.
- L'écart entre 2^{52} et 2^{53} est $\epsilon \times 2^{52} = 1$, donc il n'y a aucun flottant entre ces deux nombres.
- Avec un codage sur 64 bits, le plus petit nombre positif non nul est $1,0 \times 2^{-1022}$.

Remarque

Le système qui vient d'être décrit concerne les nombres dit « normaux ». La mantisse de ces nombres appartient à l'intervalle $[1;2[$ et cela ne permet pas de coder le nombre 0.

C'est pourquoi on définit des nombres dits « subnormaux » pour lesquels la mantisse appartient à l'intervalle $[0;1[$. Le plus petit nombre positif non nul est alors $2^{-52} \times 2^{-1022} = 2^{-1074}$.

3 Calculs

3.1 Calculs approchés

La règle pour calculer avec des flottants est la prudence. **Les calculs effectués avec des flottants sont des calculs approchés**.

L'addition n'est pas associative compte tenu des nombres flottants existants. Par exemple, il n'existe pas de flottants entre 2^{53} et $2^{53}+2$.

Donc pour la machine, `2.0 ** 53 + 1.0 + 1.0` vaut `2.0 ** 53 + 1.0` soit `2.0 ** 53`.

Alors que `2.0 ** 53 + (1.0 + 1.0)` vaut `2.0 ** 53 + 2.0`.

De manière générale $(a+b)+c \neq a+(b+c)$.

La multiplication n'est pas associative non plus. Par exemple les nombres $(3\times10^{-1})\times10^{16}$ et 3×10^{15} sont égaux. Mais `3.0 * 10**(-1) * 10**(16)` et `3.0 * 10**(15)` ne sont pas égaux. Leur différence vaut exactement `0.5` en machine.

Il est difficile de prévoir un résultat. Par exemple `(1e15 + 1) - 1e15` a pour valeur 1, alors que `(1e16 + 1) - 1e16` a pour valeur 0.

Un autre exemple : l'expression `2 * 0.1 == 0.2` a la valeur `True` alors que `0.1` et `0.2` ne sont pas représentés de manière exacte. Et l'expression `3 * 0.2 == 0.6` a la valeur `False`.

Des bugs sont possibles à cause de la propagation d'erreurs (arrondis) ou de dépassement de capacité (overflow) qui bloque le programme.

On peut par exemple constater une accumulation d'erreurs en programmant une simple suite.

Voici un code qui calcule les termes d'une suite définie par $u_0 = 1/3$ et pour tout $n > 0$, $u_n = 3(4u_{n-1}/3 - 1/3)$:

```
u = 1/3
for i in range(40):
    u = 3 * ((4*u/3) - 1/3)

print(u)
```

On exécute ce code et on obtient pour u_{40} la valeur 0.3333333333333333. On peut supposer que c'est une approximation de $1/3$ et c'est le résultat attendu.

Cette suite est clairement constante, un calcul montre que tous les termes valent $1/3$. Et on peut vérifier que l'expression `4/9 - 1/3 == 1/9` a la valeur `True`.

Nous simplifions l'expression pour obtenir $u_n = 4u_{n-1} - 1$ et programmons la suite définie par $u_0 = 1/3$ et $u_{n+1} = 4 \times u_n - 1$.

```
u = 1/3
for i in range(40):
    u = 4 * u - 1

print(u)
```

Nous obtenons pour u_{40}, la valeur : `-22369621.0`.

En fait u_{27} prend la valeur 0.0 et ensuite les termes sont de la forme $u_{n+1} = 4u_n - 1$, donc les valeurs sont -1, -5, -21, etc.

Une explication est donnée par la valeur de l'expression `4/3 - 3/3 == 1/3` qui est `False`.

Le langage Python, gère des entiers très grands et permet de faire des calculs exacts avec des nombres qui dépassent la valeur maximale utilisée par le processeur. Ces calculs sont décomposés, peuvent nécessiter un temps non négligeable, mais ils sont exacts. Pour les flottants, ce n'est pas le cas, il y a un plus grand nombre.

```
>>> 1.0e308
1e+308
>>> 1.0e309
inf
>>> 2.0**1023
8.98846567431158e+307
>>> 2.0**1024
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    2.0**1024
OverflowError: (34, 'Result too large')
```

Lorsque dans un calcul la valeur maximale est dépassée, le programme s'arrête et une erreur est signalée : `OverflowError`. Cela peut avoir des conséquences graves comme l'autodestruction de la fusée Ariane 5, lancée le 4 juin 1996, 39 secondes après le décollage. Certains tests n'avaient pas été menés pour des raisons d'économie.

3.2 Comparaisons

La précision étant limitée, pour comparer deux nombres flottants, on ne teste pas une égalité entre les deux nombres. On vérifie s'ils sont suffisamment proches, ce « suffisamment » dépendant du contexte. En effet, s'il n'y a pas de flottant entre deux flottants a et c, tous les réels compris entre a et c sont représentés par a ou c.

Par exemple, 10.0^{16} et $10.0^{16} + 1$ sont égaux pour la machine. Et dans la résolution de certains problèmes, si on ne prend en compte dans les résultats que six chiffres significatifs, $a = 10.0^{16}$ et $b = 10.0^{16} + 1$ sont considérés comme égaux car $(b - a) \leq b \times 10^{-6}$

Pour comparer des flottants, on utilise une précision absolue et une précision relative. Un test de comparaison a donc la forme : `abs(a-b) <= max(absolue, relative * max(abs(a), abs(b)))`. La précision absolue est utile pour comparer les nombres proches de 0, sinon on peut s'en passer.

C'est ainsi qu'est codée la fonction `isclose` dans le module `math`.

```
def estproche(a, b, tol_rel=1e-09, tol_abs=0.0):
    return abs(a-b) <= max(tol_rel * max(abs(a), abs(b)), tol_abs)
```

Pour une tolérance de 5% par exemple, il suffit de prendre `tol_rel = 0.05`.

3.3 Inégalités

On considère les flottants `x = 2.0 - 2.0 ** (-52)` et `y = 2.0`, soit deux flottants consécutifs tels que `x + y` est le même flottant que `2 * y`.

On obtient alors pour `(x + y) ** 2` la valeur 16.0 et pour `2 * (x**2 + y**2)` la valeur $15.9999\ldots98$. On a donc deux nombres x et y tels que $(x+y)^2 > 2(x^2+y^2)$.

```
>>> x = 2.0 - 2.0 ** (-52)
>>> y = 2.0
>>> x < y
True
>>> x + y == 2 * y
True
>>> (x + y) ** 2
16.0
>>> 2 * (x**2 + y**2)
15.999999999999998
>>> (x + y) ** 2 > 2 * (x**2 + y**2)
True
```

On pourrait en déduire l'inégalité $(x+y)^2 \geq 2(x^2+y^2)$. Or, on démontre exactement le contraire en mathématiques : pour tout réels x et y : $2(x^2+y^2) \geq (x+y)^2$.

En effet, $(x-y)^2 \geq 0$ soit, $x^2+y^2-2xy \geq 0$, donc, $x^2+y^2 \geq 2xy$, d'où $2x^2+2y^2 \geq x^2+y^2+2xy$. Et enfin $2(x^2+y^2) \geq (x+y)^2$.

Conclusion

- Deux réels distincts peuvent être égaux pour la machine.
- Deux expressions qui produisent le même résultat en mathématiques peuvent produire des résultats distincts en machine.
- Deux expressions qui produisent des résultats en mathématiques ordonnés dans un sens peuvent produire des résultats ordonnés dans le sens contraire en machine.

La plus grande prudence est donc recommandée.