Codage des nombres

20 janvier 2020

1 Écriture d'un entier naturel dans une base b

1. Généralités

   En base 10, $523 = 5 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$.

   En base 7, $(523)_7 = 5 \times 7^2 + 2 \times 7^1 + 3 \times 7^0$.

   Réciproquement, pour décomposer 523 en base 7, on effectue des divisions euclidiennes successives par 7 :

   algorithme : `decomposition_base`
   Entrées : $n, b \in \mathbb{N}$ avec $b \geqslant 2$
   Résultat : l'écriture en base b de l'entier n
   Variables : a, d deux entiers et une chaîne de caractère mot

   ```
   a ← n
   mot ←' ' # mot vide
   Tant que a ≠ 0, Faire
      d ← le reste de la division de a par b
      mot ← d + mot # ajoute le caractère d à mot
      a ← a/b
   Fin du Tant que
   Renvoyer mot
   ```

   Cet algorithme est de complexité logarithmique. Le nombre d'itérations correspond au nombre de chiffres en base b de l'entier n, qui vaut

   $$\lfloor \log_b(n) \rfloor + 1.$$

2. Le cas de la base 2

   En base deux, il n'y a que deux chiffres 0 ou 1. Par exemple, $(1011)_2 = 8 + 2 + 1 = 11$. Les 4 chiffres qui constituent l'écriture en base 2 de 11 sont appelés des bits. On dit aussi que $\boxed{1011}$ est un mot binaire de 4 bits. Un octet correspond à 8 bits.

   Le mot binaire $\boxed{\underbrace{100\ldots00}_{n \text{ zéros}}}$ code l'entier naturel 2^n (l'équivalent en base 10 est par exemple $1000 = 10^3$).

   Avec n bits, on peut coder 2^n entiers donc tous les entiers de 0 à $2^n - 1$.

Le plus grand entier naturel codé sur n bits est $\underbrace{(111\ldots11)_2}_{n \text{ bits}}$, il vaut $2^n - 1$ (l'équivalent en base 10 est par exemple $999 = 10^3 - 1$).

Remarques :

- multiplier par 2^k revient à rajouter k zéros sur la droite. Par exemple, $(101101)_2 \times 2^3 = (101101\ 000)_2$.
- rappelons une approximation utile en informatique : $2^{10} = 1024 \sim 1000 = 10^3$.

3. Le cas de la base hexadécimale (base 16)

   Le nombre 140 s'écrit en base deux $(1000\ 1100)_2$ et $(8C)_{16}$ en base 16. On remarque que $(1000)_2 = 8 = (8)_{16}$ et $(1100)_2 = 8 + 4 = 12 = (C)_{16}$.

   Un mot de 4 bits est un entier compris entre 0 et 15, donc un chiffre en base 16. En regroupant par paquets de 4 bits à partir de la droite, on obtient une correspondance facile entre la base 2 et la base 16. En effet, $(8C)_{16} = 8 \times 16^1 + 12 \times 1 = 8 \times 2^4 + 12 \times 1 = (1000)_2 \times 2^4 + (1100)_2 = (1000\ 0000)_2 + (1100)_2 = (10001100)_2$.

   L'avantage de la base 16 étant qu'elle nécessite moins de chiffres et est donc plus lisible pour un humain.

   Remarque : les fonctions `bin` (resp. `hex`) de Python renvoie les décompositions binaires (resp. hexadécimales) d'un entier.

```
>>> bin(140)
'0b10001100'

>>> hex(140)
'0x8c'
```

4. Comme à l'école primaire. Additions et multiplications posées...

2 Représentation binaire d'un entier relatif

Commet coder un entier relatif sur n bits ? Une solution pourrait être par exemple d'utiliser le premier bit pour coder le signe de l'entier relatif. Dans ce cas, le nombre zéro aurait deux représentations différentes $\boxed{00\ldots0}$ et $\boxed{10\ldots0}$. Le plus gros inconvénient serrait que l'algorithme de l'addition ne fonctionnerait pas si on ajoute des nombres de signe différent. En effet, par exemple avec $n = 4$ bits, on aurait -2 représenté par $\boxed{1010}$, 1 représenté par $\boxed{0001}$. L'addition des mots $\boxed{1010}$ et $\boxed{0001}$ donne le mot $\boxed{1011}$ qui représente le nombre -3. Mais $-2 + 1 = -1$ et la représentation de -1 est $\boxed{1001}$.

1. Représentation en complément à deux ou modulo 2^n

   On va procéder autrement, et utiliser une représentation dite «en complément à deux» ou «représentation modulo 2^n». Voici le principe : avec n bits nous allons coder 2^n entiers dont la moitié sont positifs ou nuls. Plus précisément nous allons pouvoir représenter les entiers naturels de 0 à $2^{n-1} - 1$ et les entiers négatifs de -1 à -2^{n-1} :

- si $x \in [\![0, 2^{n-1} - 1]\!]$, on représente x par son mot binaire associé
- si $x \in [\![-2^{n-1}, -1]\!]$, on représente x par le mot binaire qui code l'entier naturel $x+2^n$.

Remarque : si $x \in [\![-2^{n-1}, -1]\!]$, le nombre $x + 2^n$ est compris entre $2^n - 2^{n-1}$ et $2^n - 1$, c'est-à-dire entre 2^{n-1} et 2^n-1. Il est donc le reste de la division euclidienne de x par 2^n. La représentation ci-dessus peut donc s'interpréter avec un point de vue plus mathématique :

si $x \in [\![-2^{n-1}, 2^{n-1}-1]\!]$, x est représenté par le mot binaire codant son unique représentant modulo 2^n dans $[\![0, 2^n - 1]\!]$.

Exemple avec $n = 3$: on peut coder les huit entiers $0, 1, 2, 3$ et $-1, -2, -3, -4$.

comme $3 = (011)_2$, 3 est codé par le mot $\boxed{011}$. Le nombre -3 est négatif, on considère alors $-2 + 2^3 = 6 = (110)_2$, son représentant modulo 2^3. On code ainsi -3 par le mot $\boxed{110}$.

Attention, il faut donc bien distinguer un nombre de sa représentation.

2. Avantages et inconvénients

   - il est facile de tester si un nombre est négatif. Sa représentation binaire commence par un 1.
   - additionner deux entiers relatifs revient à additionner leur représentation. C'est une conséquence du fait qu'on puisse additionner deux congruences modulo 2^n. En effet, si x et y sont deux entiers de $[\![0, 2^n - 1]\!]$, en notant $\pi(x)$ et $\pi(y)$ leur représentation, on a $x \equiv \pi(x) \mod 2^n$ et $y \equiv \pi(y) \mod 2^n$, donc $x + y \equiv \pi(x) + \pi(y) \mod 2^n$. Ainsi $\pi(x + y) = \pi(x) + \pi(y)$.
   - L'addition de deux nombres positifs «assez grands» peut donner un résultat négatif. En effet, par exemple avec $n = 4$ bits, on peut représenter les entiers de -8 à 7. Quel résultat est renvoyé si l'on demande $3+6$ sachant que 9 ne peut être représenté avec 4 bits ? Et bien 3 est représenté par $\boxed{0011}$, 6 par $\boxed{0110}$, on additionne leur reprsésentation, cela donne $\boxed{1001}$, qui représente le nombre -7 ($9 \mod 16 = -7$).
   - Dans la plupart des langages de programmation comme C, Java, Fortran,..., les entiers relatifs sont codés sur 64 bits, ce qui correspond à la taille des registres du processeur, et qui permet d'effectuer des calculs très rapides. En Python, un autre choix a été fait, les entiers n'ont pas de taille maximale. Cela a des avantages, mais aussi des inconvénients, les calculs ne seront pas très rapides. Par exemple sur 64 bits, le plus grand entier positif représentable est $x = 2^{63} - 1$ et on voit que Python manipule bien $x + 1$.

     ```
     >>> x = 2**63-1
     >>> x
     9223372036854775807
     >>> x+1
     9223372036854775808
     ```

     Le module `numpy` spécialisé dans le calcul numérique repose sur une librairie C et manipule des entiers 64 bits.

     ```
     >>> import numpy as np
     >>> y = np.int64(x)  # on convertit l'entier x en un entier relatif <<numpy>> c
     >>> y+1
     -9223372036854775808
     ```

Cette fois-ci, $y+1$ donne le nombre négatif -2^{63}, ce qui est conforme au codage modulo 2^{64}.

De même, si on demande de calculer $2y = 2(2^{63}-1) = 2^{64}-2 \equiv -2 \mod 2^{64}$, cela renverra -2.

3 Un peu de mathématiques : écriture décimale et binaire d'un rationnel

1. En base 10

   On a $5,387 = 5 + 3\times 10^{-1} + 8\times 10^{-2} + 7\times 10^{-3}$.

   On appelle nombre décimal, tout nombre réel qui s'écrit avec un nombre fini de chiffres après la virgule, c'est-à-dire de la forme $\frac{a}{10^n}$ avec $a \in \mathbb{Z}$ et $n \in \mathbb{N}$. On note $\mathbb{D}$ l'ensemble des nombres décimaux.

   Tout nombre décimal est rationnel, mais la réciproque est fausse puisque $\frac{1}{3} = 0,333...$ n'est pas décimal. Un autre argument plus arithmétique est de dire que si $\frac{1}{3} = \frac{a}{10^n}$, alors $3a = 10^n$ et donc 3 divise 10, ce qui n'est pas. Pour obtenir la décomposition décimale d'un rationnel, on fait des divisions euclidiennes successives. Exemples : $\frac{43}{5} = 8,6$, mais $\frac{25}{7} = 3,571428\ 571428\ ....$ On parle de développement décimal illimité mais périodique.

   Remarques :

   - si $x = 0,333...$, on peut retrouver la valeur exacte de x car $10x = 3,333... = 3 + x$ donc $9x = 3$ et $x = \frac{1}{9}$. De même, si $x = 0,232323...$, $100x = 23,232323... = 23 + x$ donc $99x = 23$ et $x = \frac{23}{99}$.
   - Derrière ces équations se cachent des limites de sommes géométriques,

   $$0,333... = \lim_{n\to+\infty} \sum_{k=1}^{n} 3\times 10^{-k} = \lim_{n\to+\infty} 3\sum_{k=1}^{n} \frac{1}{10^k} = \lim_{n\to+\infty} 3\frac{1/10 - (1/10)^{n+1}}{1-1/10} = 3\times\frac{1}{9} = \frac{1}{3}.$$

2. En base 2

   On a $(1,011)_2 = 1 + \frac{1}{2^2} + \frac{1}{2^3}$.

   On appelle nombre dyadique, tout nombre réel qui s'écrit avec un nombre fini de chiffres en base deux après la virgule, c'est-à-dire de la forme $\frac{a}{2^n}$ avec $a \in \mathbb{Z}$ et $n \in \mathbb{N}$. On note $\mathbb{D}_2$ l'ensemble des nombres dyadiques.

   Par exemple, $\frac{5}{32} = \frac{1}{32} + \frac{4}{32} = \frac{1}{2^5} + \frac{1}{2^3} = (0,00101)_2$. De même, $6.125 = 6 + \frac{1}{8} = (110,001)_2$.

   Cet exemple est important, il montre que l'on ne peut représenter de manière exacte en base deux, un nombre aussi simple que $\frac{1}{5} = 0,2$. En effet, en base 16, $\frac{1}{5} = (0,333\ldots)_{16}$ d'où en base deux, $\frac{1}{5} = (0,0011\ 0011\ \ldots)_2$.

   Remarques :

   - si $x = (0,111...)_2$, on peut retrouver la valeur exacte de x car $2x = (1,111...)_2 = 1+x$ (en base deux, multiplier par 2 décale la virgule de un cran vers la droite) donc $x = 1$. On peut aussi voir cela avec une somme géométrique de raison $1/2$:

   $$x = (0,111...)_2 = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots = \sum_{n=1}^{+\infty} \frac{1}{2^n} = \frac{1/2}{1/1-2} = 1.$$

- si $x = (0,01\ 01\ 01...)_2$, alors $2^2x = (1,01\ 01\ 01...)_2 = (1)_2 + (0,01\ 01\ 01...)_2 = 1 + x$, donc $4x = 1 + x$ et $x = \frac{1}{3}$. De même, on peut le prouver avec des sommes géométriques :

$$x = (0,01\ 01\ 01...)_2 = \frac{1}{2^2}+\frac{1}{2^4}+\frac{1}{2^6}+\cdots = \sum_{n=1,n\text{ pair}}^{+\infty}\frac{1}{2^n} = \sum_{p=1}^{+\infty}\frac{1}{2^{2p}} = \sum_{p=1}^{+\infty}\frac{1}{4^p} = \frac{1/4}{1-1/4} = \frac{1}{3}.$$

4 Représentation des nombres flottants

Dans de nombreux domaines, on nécessite de faire des calculs avec des «nombres à virgule» qui ne sont pas forcéments entiers (des prix, des masses, des volumes, certaines constantes physiques comme le nombre D'Avogadro, la constante de Boltzmann). On représente ces nombres par un type informatique, appelé «nombre flottant».

```
>>> type(2.5)
<class 'float'>
```

4.1 Principe de codage des flottants à l'aide de l'écriture scientifique binaire

Nous allons pouvoir représenter des «nombres à virgule» écrits en base deux, sous forme «scientifique». Nous les appellerons des nombres flottants :

$$x = \pm 1.m \times 2^e$$

où m est un mot binaire appelé **mantisse** et e un mot binaire représentant un entier relatif, appelé **exposant**. Le zéro sera codé à part.

Par exemple, l'«écriture scientifique binaire» du nombre $x = 21,5$ est $(1.01011)_2 \times 2^4$, car :

$$21.5 = (16+4+1)+\frac{1}{2} = (10101)_2+(0.1)_2 = (10101.1)_2 = (1.01011)_2\times 2^4 = (1.01011)_2\times 2^{(100)_2}.$$

Ici la mantisse est $\boxed{01011}$ codée sur 5 bits, et l'exposant est $\boxed{100}$ codé sur 3 bits.

Supposons maintenant que la mantisse soit codée sur 2 bits, que l'exposant minimal vaut -1 et l'exposant maximal 2 (ainsi il y a 4 valeurs d'exposants possibles, donc l'exposant est codé aussi avec 2 bits). On réserve aussi 1 bit pour le signe. On a ainsi $|\underbrace{s}_{1\ bit}|\underbrace{e}_{2\ bits}|\underbrace{m}_{2\ bits}|$. Avec 5 bits, nous pouvons coder 2^5 nombres flottants. Voici la liste des 16 positifs :

$(1.00)_2 \times 2^{-1} = 0.5$	$(1.10)_2 \times 2^{-1} = 1/2 + 1/4$
$(1.00)_2 \times 2^0 = 1$	$(1.10)_2 \times 2^0 = 1 + 1/2$
$(1.00)_2 \times 2^1 = 2$	$(1.10)_2 \times 2^1 = 3$
$(1.00)_2 \times 2^2 = 4$	$(1.10)_2 \times 2^2 = 6$
$(1.01)_2 \times 2^{-1} = 1/2 + 1/8$	$(1.11)_2 \times 2^{-1} = 1/2 + 1/4 + 1/8$
$(1.01)_2 \times 2^0 = 1 + 1/4$	$(1.11)_2 \times 2^0 = 1 + 3/4$
$(1.01)_2 \times 2^1 = 2 + 1/2$	$(1.11)_2 \times 2^1 = 3 + 1/2$
$(1.01)_2 \times 2^2 = 5$	$(1.11)_2 \times 2^2 = 7$

Le plus grand flottant 7 est obtenu avec une mantisse maximale et un exposant maximal. Le plus petit flottant positif $0,5$ est obtenue avec la mantisse nulle et l'exposant le plus petit ici (-1).

La mantisse étant constituée de 2 bits, on dit que l'on a une précision avec 2 chiffres binaires significatifs.

Si on place ces flottants sur une droite graduée, on remarque qu'ils ne sont pas uniformément répartis sur $[0.5; 7]$, ils sont plus concentrés autour de $0,5$.

Voici un graphique où l'on a représenté les nombres flottants que l'on peut coder avec 8 bits.

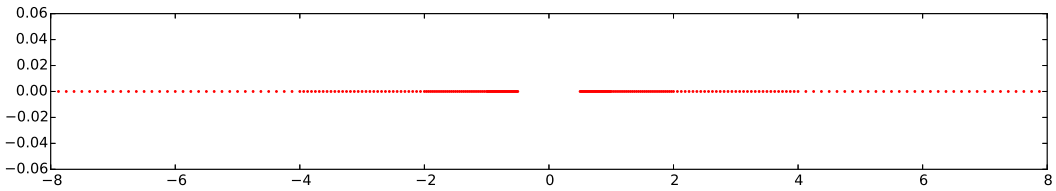

FIGURE 1 – Nombres flottants sur 8 bits

4.2 Norme IEE 754

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

$$x = (-1)^s \times 1.(m_1 m_2 \dots m_{52})_2 \times 2^e.$$

- l'entier s appelé **signe** vaut 0 si x est positif et 1 sinon.
- les bits (valent 0 ou 1) $m_1, \dots, m_{52}$ constituent la **mantisse** m.
- l'entier e est l'**exposant**, on le code par la représentation binaire sur 11 bits de l'entier *naturel* E défini par $e = E - 1023$. L'entier E étant codé sur 11 bits, il prend les valeurs de $[\![0, 2^{11} - 1]\!] = [\![0, 2047]\!]$. Mais les mots «extrêmes» `00000000000` et `11111111111` étant réservés, l'entier E ne prend que les valeurs de $[\![1, 2046]\!]$.

Par exemple, le nombre décimal 5.25 s'écrit en base deux 1.0101×2^2. On a donc $s = 0$, $m = 0101$ et $e = 2$ donc $E = 1025$ donc sa représentation est :

$$\underbrace{\boxed{0}}_{s}\underbrace{\boxed{10000000001}}_{E}\underbrace{\boxed{0101000000000000000000000000000000000000000000000000}}_{m}$$

Remarque :

- la valeur 0 est une exception. Elle sera représentée par une mantisse et un exposant nuls.
- en Python, la fonction si `f` est un nombre flottant, `f.hex()` renvoie la mantisse en hexadécimal (donc 13 chiffres, puisque 13×4 bits $= 52$ bits) et l'exposant e en décimal après le p.

```
>>> f = 5.25
>>> f.hex()
'0x1.5000000000000p+2'
```

Cela dit que 5.25 est représentée par le nombre $1.(5000000000000)_{16} \times 2^{+2}$.

Exercice 1 (questions en vrac)

1. Quel est le nombre dont la représentation IEE 754 sur 64 bits est :

$$\underbrace{\boxed{1}}_{s}\ \underbrace{\boxed{10000000010}}_{E}\ \underbrace{\boxed{0100000000000000000000000000000000000000000000000000}}_{m}$$

   On a $E = 2^{10} + 2 = 1026$ donc $e = E - 1023 = 3$. Ainsi le développement binaire du nombre est $-1.01 \times 2^3 = -1010$, ce qui donne -10 en base 10.

   .

2. Donner la représentation flottante du nombre $\frac{11}{32}$.
   On a $\frac{11}{32} = \frac{8}{32} + \frac{2}{32} + \frac{1}{32} = \frac{1}{4}(1 + \frac{1}{4} + \frac{1}{8}) = 1.011 \times 2^{-2}$. Ainsi $e = -2$ et $E = 1021$. La représentation est donc :

$$\underbrace{\boxed{0}}_{s}\ \underbrace{\boxed{01111111101}}_{E}\ \underbrace{\boxed{0110000000000000000000000000000000000000000000000000}}_{m}$$

3. Quel est le plus petit nombre flottant strictement positif ? Donner sa représentation, puis une valeur approchée.

   Il est obtenu avec une mantisse nulle et un exposant minimal, c'est-à-dire pour $E = 1$ donc $e = 1 - 1023 = -1022$, c'est donc $1.0 \times 2^{-1022} \approx 2.225 \times 10^{-308}$.

4. Quel est le plus grand nombre flottant ? Donner sa représentation, puis une valeur approchée.

   Il est obtenu avec une mantisse maximale donc $111\ldots111$ et un exposant maximal donc $E = 2046$ donc $e = 2046 - 1023 = 1023$. Son développement binaire est donc $1.\underbrace{111\ldots111}_{52\text{ bits}} \times 2^{1023} = \underbrace{111\ldots111}_{53\text{ bits}} \times 2^{-52} \times 2^{1023}$. Ce nombre vaut

$$(2^{53} - 1)2^{971} \approx 1.797 \times 10^{308}.$$

   Remarque : en particulier, le nombre 1.2×10^{309} ne peut être représenté par un flottant, on a un «overflow».

```
>>> 1.2*10**309
Traceback (most recent call last):
  File "<console>", line 1, in <module>
OverflowError: long int too large to convert to float
```

   Mais $1.2 \times 10^{309} = 12 \times 10 * *308$, donc c'est un entier, et on peut donc le représenter en Python par le type `int`.

```
>>> 12*10**308
120000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

5. Quel est le plus petit nombre flottant strictement supérieur à 1 ? Il s'agit de $1+2^{-52}$. Son développement binaire est $1.\underbrace{00\ldots001}_{52 \text{ bits}}$. Sa représentation flottante est donc obtenue avec un exposant nul $e = 0$ donc $E = 1023$.

$$\underbrace{\boxed{0}}_{s}\underbrace{\boxed{01111111111}}_{E}\underbrace{\boxed{0000000000000000000000000000000000000000000000000001}}_{m}$$

4.3 Observations et explications

L'«arithmétique flottante» et la propagation des erreurs d'arrondi est une problématique vraiment délicate. Nous pouvons néanmoins retenir ces quelques points :

1. Phénomène d'absorbtion

```
>>> 1.0 + (2**53 - 2**53)
1.0  # ok
>>> (1.0 + 2**53) - 2**53
0.0  #oups,
>>> 1.0+2**53 == 2**53
True  # phénomène d'absorbtion, la petite quantité 1.0 a été absorbée par la grande
>>> (1+ 2**53) - 2**53
1   # et ici ça marche car on ne travaille pas avec des flottants, mais avec des en
```

   Explication : la mantisse représente le nombre de chiffres binaires significatifs, donc 52. Par exemple, le nombre $1 + 2^{-53} = (1.\underbrace{0000...01}_{53\ bits})_2$ nécessiterait une mantisse de 53 bits.

   Or on ne dispose que de 52 bits, on va donc perdre le dernier bit égal à 1 et représenter $1 + 2^{-53}$ par $(1.\underbrace{0000...0}_{52\ bits})_2$, donc par 1.

   Le nombre 2^{-53} lui est bien représentable car $2^{-53} = 1.0 \times 2^{-53}$, donc s'écrit avec une mantisse nulle et un exposant $e = -53$ donc $E = 1023 - 53$.

   Comme $2^{-52} \approx 2.22 \times 10^{-16}$, cela signifie que lorsqu'on calcule avec des nombres flottants, on ne pourra jamais avoir une précision supérieure à 15 ou 16 chiffres significatifs en base 10 ! Ceci est confirmé par Python :

```
>>> 1+10**-15
1.000000000000001

>>> 1+10**-16
1.0
```

Recommandation : ne pas additionner de nombres dont l'écart relatif est très grand. Dans une somme de plusieurs termes, on commencera par additionner les petites quantités.

Remarque mathématique : l'addition des nombres flottants n'est donc pas associative.

2. Des erreurs d'arrondi

   On a déjà vu que le nombre 0.1 n'était pas dyadique, il a un nombre infini de chiffres binaires après la virgule, on va tronquer son développement binaire et le représenter par un flottant qui ne sera pas tout à fait égal à 0.1. Ceci explique que $(0.2 + 0.1) - 0.3$ ne soit pas tout à fait égal à 0.

   ```
   >>> (0.1+0.2) - 0.3 == 0
   False
   ```

   Par exemple, on a vu que $\frac{1}{5} = 1.(1001\ 1001\ 1001\ldots)_2 \times 2^{-3}$ donc $\frac{1}{5}$ sera arrondi à la valeur $\frac{1}{5} = 1.(1001\ 1001\ 1001\ldots)_2 \times 2^{-3}$.

   Recommandation : ne jamais tester l'égalité entre deux nombres flottants, mais tester si leur distance est inférieure à un nombre très petit.

3. Phénomène de «cancellation»

   Recommandation : ne pas soustraire des quantités voisines sous peine de perdre beaucoup de précision.

   Voici un exemple pour comprendre, on prend $x = (1.\underbrace{000...\mathbf{11011}}_{52\ bits})_2$ et $y = (1.\underbrace{000...\mathbf{00110}}_{52\ bits})_2$.

   Les nombres x et y sont très proches, ils ne diffèrent que par leurs 5 dernières «décimales». On a $x - y = (0.\underbrace{000...0\mathbf{10101}}_{52\ bits})_2 = (1.0101)_2 \times 2^{-48}$. Au lieu des 54 chiffres significatifs après la virgule, on n'en a plus que 4. Ce problème peut se rencontrer si l'on calcule par exemple $\frac{1}{x} - \frac{1}{x+1}$. Dans ce cas, il sera préférable de plutôt calculer $\frac{1}{x(x+1)}$.

Terminons par un exercice :

Exercice 2 (Attention aux flottants) Les deux questions sont indépendantes.

1. Que dire de la représentation flottante de l'entier $2^{54} + 1$? En déduire ce qu'affiche le script Python suivant et expliquer :

   ```
   import math
   a = math.sqrt(1 + 2**54)
   print(a)
   ```

   On a $2^{54} + 1 = 2^{54}(1 + 2^{-54})$, donc son développement binaire est $1.\underbrace{000\ldots001}_{54\ \text{bits}} \times 2^{54}$. La mantisse ne contenant que 52 bits, le dernier 0 et le dernier 1 ne pourront être représentés et seront perdus. L'entier $2^{54} + 1$ sera donc représenté par 2^{54}.

   Comme $\sqrt{2^{54}} = 2^{27} = 134217728$, Python renvoie le flottant `134217728.0`, ce qui constitue une erreur !

   Remarque : en particulier, Python dit que $2^{54}+1$ est un carré, ce qui est faux. C'est pourquoi lorsque l'on travaille avec des entiers, il vaut mieux éviter l'utilisation de flottants.

2. On considère la fonction suivante qui teste la colinéarité de deux vecteurs du plan :

```
def vecteurs_colineaires(a,b,c,d):
    """Données: a,b,c,d quatre flottants
       Résulat: un booléen qui est vrai si les vecteurs de coordonnées
                respectives (a,b) et (c,d) sont colinéaires
       """
    booleen = False
    if a*d - b*c == 0:
        booleen = True
    return booleen
```

L'instruction `f(3,1,0.3,0.1)` renvoie `False`, pourtant les vecteurs de coordonnées respectives $(3, 1)$ et $(0.3, 0.1)$ sont bien colinéaires. Expliquer le problème et envisager une solution ou une amélioration.

Le nombre décimal 0.1 n'est pas dyadique, don admet un développement binaire infini. En particulier, il ne peut être représenté de manière exacte par un nombre flottant. Ainsi le test d'égalité `3*0.1 - 0.3*1 == 0` renvoie Faux. **Il ne faut pas faire de tests d'égalité avec les flottants**. On remplacera le test par une condition de la forme `abs(a*d - b*c) < epsilon` où `epsilon` est une valeur proche de zéro, choisie selon le contexte.