Info 1 | Représentation des nombres et informatique

On introduit ici la numération de position en base $b \geqslant 2$ et on montre comment elle est mise en œuvre pour représenter des entiers relatifs dans un ordinateur. On poursuit par l'étude du codage sur machine des nombres réels et complexes.

1 Représentation des nombres et informatique 1
1 Numération de position en base quelconque 2
1.1 Décomposition d'un entier en base b 2
1.2 Le cas particulier de l'écriture sous forme binaire 3
2 Représentation des entiers relatifs sur machine 4
2.1 Codage des entiers naturels sur n bits 4
2.2 Codage des entiers relatifs sur n bits 4
2.3 Les entiers longs de Python 6
3 Représentation des nombres réels sur machine 6
3.1 Développement décimal et dyadique 6
3.2 Nombres flottants 6
3.3 Conséquences sur les calculs en virgule flottante 8
4 Ce qu'il faut retenir 11
5 Nombres complexes sous Python 11
6 Énoncés des tests 13
7 Solutions 15

1. Numération de position en base quelconque

Le lecteur est renvoyé au cours d'Arithmétique pour les démonstrations et les détails.

1.1. Décomposition d'un entier en base b

La proposition suivante généralise ce résultat à une base b quelconque et établit l'unicité des *chiffres* $a_0, a_1, \ldots, a_m$ de n en base b.

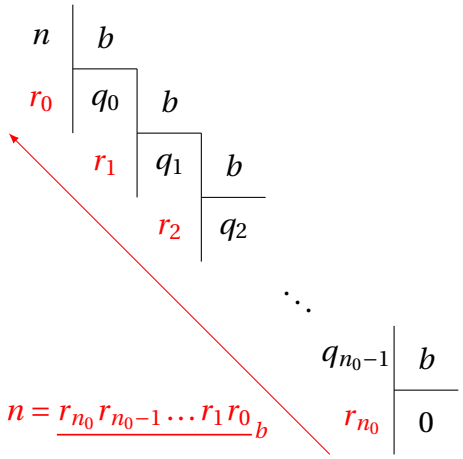
Proposition 1.0. Numération en base b

Soit $n \in \mathbb{N}^*$ et $b \in \mathbb{N}$ tel que $b \geqslant 2$. Il existe un unique entier naturel m non nul et des entiers uniques a_0, a_1, …, a_{m-1} dans $[\![0, b-1]\!]$ tels que

$$n = a_0 + a_1 b + a_2 b^2 + \ldots + a_{m-1} b^{m-1} \quad \text{et} \quad a_{m-1} \neq 0$$

ce que l'on écrit de manière condensée $n = \underline{a_{m-1} \ldots a_1 a_0}_b$. On dit qu'il s'agit du développement (ou encore de l'écriture) en base b de n.

Algorithme de décomposition en base b



On applique la méthode par divisions successives décrite à la proposition 1.1

On calcule le couple quotient-reste (q_0, r_0) dans la division de n par b, puis le couple quotient-reste (q_1, r_1) dans la division de q_0 par b, puis le couple quotient-reste dans la division de q_1 par b, etc.

En notant q_{n_0} le premier quotient nul obtenu au cours de l'algorithme, on a

$$n = \underline{r_{n_0} r_{n_0-1} \ldots r_1 r_0}_b$$

Attention au cas où $b > 10$ Lorsque $b > 10$, on utilise des lettres pour décrire les chiffres en base b. En effet, l'écriture $\underline{10}_{11}$ est ambigüe : s'agit-t-il de $1 \times 11^1 + 0 \times 11^0$ ou de 10×11^0 ? Ainsi, dans le cas de $b = 16$ (base dite *hexadécimale*), la liste des chiffres s'écrit :

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$$

En reprenant les notations de la proposition 1.1, m est le nombre de chiffre(s) en base b de n.

Proposition 1.1. Nombre de chiffres en base b

Soit $b \in \mathbb{N} \setminus \{0,1\}$, $n \in \mathbb{N}^*$ et $p \in \mathbb{N}^*$.

a. Le nombre de chiffre(s) en base b de n vaut p si et seulement si $b^{p-1} \leqslant n < b^p$.

b. Le nombre de chiffres en base b de n est égal à $1 + \left\lfloor \dfrac{\ln n}{\ln b} \right\rfloor$. C'est donc un $O(\ln n)$

1.2. Le cas particulier de l'écriture sous forme binaire

Bits d'un entier naturel

Les chiffres en base deux sont également appelés *bits*. On appelle *poids* d'un bit la puissance de deux en facteur de ce bit. Cf. ci-contre le cas de $n = \underline{1001}_{10} = \underline{1111101001}_2$.

1	1	1	1	1	0	0	1	0	1

Poids décroissant ⟶

✕ Les entiers codables sur n bits sont exactement les éléments de $[\![0, 2^n - 1]\!]$.

Opérations arithmétiques en base deux

Nous avons vu que l'*unité arithmétique et logique* d'un processeur est programmée pour effectuer des opérations sur des nombres écrits en base 2. Les algorithmes d'addition et de multiplication sont très simples :

✕ ADDITION.

On procède *bit par bit* en commençant par les bits de poids faible. Comme $1 \times 2^k + 1 \times 2^k = 1 \times 2^{k+1} + 0 \times 2^k$ pour tout entier naturel k, la somme de deux bits de poids k égaux à 1 a pour résultat un bit de poids k égal à zéro et une retenue au bit suivant. On peut donc présenter comme ci-contre une addition en base 2 (les retenues sont indiquées en rouge).

```
  1 1 1 1 1
  1 0 1 1 1 0 1 0 1
+       1 0 1 1 1 0
= 1 1 0 1 0 0 0 1 1
```

✕ MUTIPLICATION.

Tout repose sur une remarque fondamentale et simple : multiplier un nombre entier n par 2^k revient à ajouter k zéros à la fin de son écriture sous forme binaire (exactement comme pour la multiplication par 10^k en base 10). Dès lors, on peut se ramener à n'effectuer que des additions en utilisant la distributivité du produit sur l'addition. On trouvera ci-contre une version binaire du calcul de 5×3 (on trouve bien 15 !).

```
      1 0 1
  ×     1 1
  -------
      1 0 1
  + 1 0 1 0
  -------
  = 1 1 1 1
```

✕ La division euclidienne par deux ne nécessite aucun calcul : On note $n = \underline{a_{m-1} \cdots a_0}_2$ (où $n \in \mathbb{N}^*$) :

$$n = 2 \times \underbrace{\underline{a_{m-1} \cdots a_1}_2}_{\text{quotient de } n \text{ par } 2} + \underbrace{a_0}_{\text{reste}}$$

2. Représentation des entiers relatifs sur machine

Le codage des entiers relève du langage de programmation choisi.

2.1. Codage des entiers naturels sur n bits

Dans certains langages, il existe un type pour les entiers naturels et un type pour les entiers relatifs. Ainsi, si une machine dispose de n bits pour coder des entiers (décomposés en base 2), elle pourra stocker 2^n entiers naturels distincts, tous les entiers pouvant s'écrire avec n bits : les entiers représentables sont ceux compris entre 0 et $2^n - 1$.

2.2. Codage des entiers relatifs sur n bits

Dans les langages n'utilisant qu'un seul type pour les entiers (comme Python et le type `int`), un bit (généralement celui de poids fort) est utilisé pour décrire le signe : 0 l'entier est positif, 1 il est négatif. On peut alors imaginer plusieurs conventions de codage.

Une première tentative naïve

Une idée simple est de considérer que le bit de poids fort donne le signe et les bits suivants représentent la valeur absolue du nombre. Par exemple sur un octet (ie $n = 8$ bits), on code -45 par `10101101` :

$$\begin{cases} \underbrace{1}_{\text{Signe}} \quad \underbrace{0101101}_{\text{Valeur absolue}} \\ \texttt{0101101} \text{ représente l'entier } 0\times 2^6 + 1\times 2^5 + 0\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 45 \end{cases}$$

Sur n bits, on peut représenter tous les entiers de l'intervalle $[\![1-2^{n-1}, 2^{n-1}-1]\!]$. Cette convention présente l'inconvénient d'avoir 2 représentations pour 0. De plus, l'addition de nombres de signes opposés n'est pas simple.

La solution retenue : représentation par complément à deux

Une autre solution consiste, pour tout $m \in [\![-2^{n-1}, 2^{n-1}-1]\!]$, à représenter m par :

$$\begin{cases} \text{les bits de l'écriture binaire de } m & \text{si } m \geqslant 0 \\ \text{les bits de l'écriture binaire de } 2^n - |m| & \text{si } m < 0 \end{cases}$$

L'ensemble des entiers représentables sur 16 bits est donc $[\![-32768, 32767]\!]$ alors que l'ensemble des entiers représentables sur 32 bits est $[\![-2147483648, 2147483647]\!]$.

Cette convention est retenue dans de nombreux langages (Python par exemple). On vérifie facilement que le bit de poids fort vaut 1 *si et seulement si* le nombre codé est négatif.

Décodage en complément à 2

Pour trouver m à partir de de son code binaire par complément à deux sur n bits :

- On détermine le bit de poids fort dans le code de m. Il vaut 1 *si et seulement si* m est strictement négatif.
- Si $m < 0$, alors le code correspond à $2^n + m$ en base 2 : on calcule ce nombre puis on lui retranche 2^n our trouver m.
- Si $m \geqslant 0$, alors le code correspond à m en base 2.

✗ Décoder 10101 et 1101 (complément à deux sur 5 bits).

On peut également vérifier[1] que l'addition d'un entier positif et d'un entier négatif s'effectue selon le même algorithme que la somme de deux entiers positifs. Ainsi, sur 4 bits, le calcul de $3-7$ s'écrit :

$$\begin{array}{r} \,1\;1 \\ \,0\;0\;1\;1 \\ +\,1\;0\;0\;1 \\ \hline =\,1\;1\;0\;0 \end{array}, \quad \text{le résultat vaut } 1\times 2^3+1\times 2^2-16=-4$$

Le nombre de bits limité alloué à ce type de codage a pour conséquence des *dépassements arithmétiques* : l'ensemble des entiers représentables n'étant stable ni par l'addition, ni par la multiplication, le résultat de certaines opérations sera faux. On parle de *dépassement* dans le cas où le résultat d'une opération entre deux entiers représentables ne l'est pas. On prouvera dans les tests que, s'il n'y a pas dépassement, le résultat d'une addition d'obtient par l'algorithme d'addition binaire.

Le cas du langage Python est plus subtil et sera évoqué au paragraphe suivant.

2.3. Les entiers longs de Python

Depuis la version 3.0, le type `int` de Python ne distingue pas les entiers courts (codés sur 8 bits) des entiers longs (entiers dont la taille n'est limitée que par la mémoire allouée à l'interpréteur). Pour manipuler ces entiers longs, l'interpréteur doit les transformer en entiers courts manipulables par le processeur en les représentant comme des listes d'entiers courts. Ces calculs sont reliés à l'arithmétique multi-précision et dépassent le cadre de ce cours.

3. Représentation des nombres réels sur machine

3.1. Développement décimal et dyadique

En mathématiques, les approximations données sont le plus souvent décimales. L'ensemble $\mathbb{D}$ des nombres décimaux, ie de la forme $k10^{-n}$ avec $(k,n)\in\mathbb{Z}\times\mathbb{N}$, permet d'approcher n'importe quel réel avec une précision arbitraire. Tout réel $x\in\mathbb{R}\setminus\mathbb{D}$ admet un unique[2] développement décimal :

$$x=\pm\sum_{k=-m}^{+\infty}\frac{c_k}{10^k},\ \text{ avec } \forall k\in[\![-m,+\infty[\![,\ \ c_k\in[\![0,9]\!]$$

On résume ce développement en écrivant $x=\pm\underline{c_{-m}c_{-m+1}\cdots c_1c_0\,,\,c_1c_2c_3\cdots}_{10}$.

Ce résultat se généralise à une base $b\geqslant 2$ quelconque.

Représentation dyadique des réels

En informatique, comme le lecteur s'en doute, on manipule plutôt *le développment dyadique* des réels : les nombres dyadiques sont les décimaux de la forme $k2^{-n}$ avec $(k,n)\in\mathbb{Z}\times\mathbb{N}$, on note $\mathbb{D}_2$ l'ensemble formé de ces nombres ; tout réel $x\in\mathbb{R}\setminus\mathbb{D}_2$ admet un unique développement dyadique :

1. Cf. le test **1.10.**
2. Un nombre décimal x admet deux développements décimaux ; par exemple, l'entier naturel 1 s'écrit $\underline{0\,,\,9999\cdots}_{10}$ (développement dit *impropre*) et $\underline{1\,,\,000\cdots}_{10}$ (développment dit *propre*).

$$x = \pm \sum_{k=-m}^{+\infty} \frac{c_k}{2^k}, \text{ avec } \forall k \in [\![-m, +\infty[\![, \quad c_k \in [\![0,1]\!]$$

ce que l'on résume par $x = \pm \underline{c_{-m} c_{-m+1} \cdots c_1 c_0 \,,\, c_1 c_2 c_3 \cdots}_2$.

Par exemple, $3,25 = 13/4$ est un nombre dyadique et, comme $x = 3 + 1/4$, on a $x = \underline{11\,,\,01}_2$.

De même, $x = 0,1$ n'est pas dyadique et s'écrit :

$$0,1 = \underline{0\,,\,0001100110011001100110011001100110011001100110011\cdots}_2$$

3.2. Nombres flottants

Flottants et norme IEEE-754

L'ensemble des nombres réels représentables par un ordinateur est un sous ensemble fini de l'ensemble $\mathbb{D}_2$ des nombres dyadiques appelé ensemble des *nombres flottants*. Il dépend de la précision de la machine (mesurée en nombre de bits alloués pour le calcul numérique) et il est fixé par la norme IEEE-754. Celle-ci est actuellement le standard pour la représentation des nombres à virgule flottante en binaire.

Nous allons en donner une description partielle pour une précision de 64 bits. Un nombre dyadique non nul possède une représentation de la forme

$$\pm \underline{1, c_1 \cdots c_k}_2 \times 2^e, \qquad \text{où } e \text{ est un entier relatif}$$

On dit que c'est *sa forme normalisée*. Par exemple, $3,25 = \underline{11,01}_2$ a pour représentation normalisée $\underline{1,101}_2 \times 2^1$.

On parle également de *forme dénormalisée* dans de le cas d'une représentation du type

$$\pm \underline{0, c_1 \cdots c_k}_2 \times 2^e, \qquad \text{où } e \text{ est un entier relatif}$$

La suite de bits après la virgule est appelée *la mantisse* du nombre, et la puissance de 2, *l'exposant* du nombre.

Dans cette norme, les nombres dyadiques sont codés sur 64 bits en réservant :

- 1 bit pour le signe : 0 pour + et 1 pour − ;
- 11 bits pour l'exposant ;
- 52 bits pour la mantisse.

L'exposant est un entier relatif, mais afin de faciliter la comparaison des flottants, il n'est pas codé par complément à deux mais par un simple *décalage* : l'exposant e est représenté en machine par l'entier positif $e' = e + 2^{10} - 1$.

Les conventions imposées par la norme IEEE-754 sont les suivantes :

- **Nombres sous forme normalisée.**

  Si $\underline{00000000001}_2 \leqslant e' \leqslant \underline{11111111110}_2$, autrement dit si $1 \leqslant e' \leqslant 2^{11} - 2$, le nombre est représenté sous sa forme normalisée. Nous avons alors $-1022 = 2 - 2^{10} \leqslant e \leqslant 2^{10} - 1 = 1023$; ainsi le plus petit nombre représentable sous forme normalisée est

$$\underline{1, \overbrace{0000\cdots0000}^{52 \text{ fois}}}_2 \times 2^{-1022} = 2^{-1022} \simeq 2,23 \times 10^{-308}$$

et le plus grand nombre représentable sous forme normalisée est

$$\underline{1,\overbrace{1111\cdots1111}^{52\text{ fois}}}_2 \times 2^{1023} = 2^{1024} - 2^{971} \simeq 1{,}8 \times 10^{308}$$

⇛ **Nombres sous forme dénormalisée.**

Si $e' = \underline{00000000000}_2$, l'exposant vaut $e = -1022$ par convention, le nombre est représenté sous la forme dénormalisée $\underline{0,c_1\cdots c_k}_2 \times 2^e$. Ceci permet de manipuler des quantités encore plus petites ; le plus petit nombre dénormalisé strictement positif est

$$\underline{0,\overbrace{0000\cdots0000}^{51\text{ fois}}1}_2 \times 2^{-1022} = 2^{-1074} \simeq 4{,}94 \times 10^{-324}$$

et le plus grand nombre dénormalisé strictement positif est

$$\underline{0,\overbrace{1111\cdots1111}^{52\text{ fois}}}_2 \times 2^{-1022} = 2^{-1022} - 2^{-1074} \simeq 2{,}23 \times 10^{-308}$$

Le nombre zéro a deux rerésentations : $e = -1022$, mantisse nulle et bit de signe quelconque.

⇛ **Les exceptions.**

Enfin, le cas où $e' = \underline{11111111111}_2$ permet de coder des valeurs particulières : l'infini qu'on obtient lorsqu'un calcul dépasse le plus grand nombre représentable, et le *NaN* (*Not a Number*) qu'on obtient comme résultat d'une opération invalide.

Nous n'en dirons pas plus car ses aspects sont hors programme.

✕ Représentons par exemple le nombre dyadique $x = -\left(2^2 + \frac{1}{2^2} + \frac{1}{2^4}\right)$ avec cette norme. On a

$$x = -\underline{100,0101}_2 = -\underline{1,000101}_2 \times 2^2$$

Ainsi, x s'écrit sous forme normalisée par un bit de signe égal à 1, un exposant $e = 2$ représenté par $e' = 2 + 2^{10} - 1 = 2^{10} + 1$ et une mantisse $m = \mathbf{000101}0\cdots0$, ce donne le mot binaire suivant :

$$\underbrace{1}_{\text{1 bit de signe}}\ \underbrace{1\overbrace{00\cdots00}^{9\text{ fois}}1}_{\text{11 bits d'exposant}}\ \underbrace{\mathbf{000101}\overbrace{00\cdots00}^{46\text{ fois}}}_{\text{52 bits de mantisse}}$$

Vérifications sous Python

```
c=0
p=1.
while p>0:
    p=p/2
    c+=1
print(c-1)
```

On peut vérifier ces ordres de grandeurs sous Python.

Le programme ci-contre renvoie 1074.

Ceci correspond effectivement au plus petit nombre représentable (sous forme dénormalisée) :

$$2^{-52-1022} = 2^{-1074}$$

En-deça, Python ne distingue plus les nombres du zéro réel :

```
>>> 2**(-1074)
5e-324
```

```
>>> 2**(-1075)
0.0
```

Le programme ci-contre renvoie 1023.

Il s'agit effectivement de la plus grande puissance de 2 représentable sous forme d'un flottant :

$$2^{1023}$$

```
c=0
p=1.
while 2*p>p:
    p=p*2
    c+=1
print(c-1)
```

```
>>> 2.**(1023)
8.98846567431158e+307
>>> 2.**(1024)
Traceback (most recent call last):
File "<console>", line 1, in <module>
OverflowError: (34, "Le résultat numérique est en dehors de l'intervalle")
```

On peut également vérifier la gestion des valeurs particulières :

```
>>> 2.**5600
Traceback (most recent call last):
File "<console>", line 1, in <module>
OverflowError: (34, "Le résultat numérique est en dehors de l'intervalle")
>>> numpy.sqrt(-1)
nan
>>> 2e308
inf
```

3.3. Conséquences sur les calculs en virgule flottante

Les erreurs dans les calculs entre nombres décimaux sont dûes aux arrondis successifs lors des calculs et des calculs d'approximations flottantes. Illustrons notre propos par le cas typique d'une opération $x \star y$ entre deux nombres décimaux ($\star$ désignant $+$ ou $\times$).

Il faut d'abord chercher les représentatiions flottantes $[x]_f$ et $[y]_f$ des nombres décimaux x et y. Puis, on calcule le décimal $[x]_f \star [y]_f$ dont il faut ensuite calculer la représentation flottante $\left[[x]_f \star [y]_f\right]_f$.

Sans entrer dans le détail des conventions d'approximation et de calcul, nous illustrerons trois mécanismes afin de mettre en évidence les effets d'arrondis.

Erreur par conversion

En informatique, on ne manipule bien-sûr qu'un ensemble fini de nombres. Ainsi, on ne pourra manipuler *exactement les nombres décimaux* et la conversion d'un nombre décimal en un nombre dyadique va provoquer une approximation qui sera parfois surprenante :

```
>>> 0.2+0.2
0.4
```

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

Il faut donc être prudent si l'on manipule des flottants ; le programme suivant affiche `Flute !` :

```
if (0.2+0.1==0.3):
    print('Bingo !')
else:
    print('Flute !')
```

Les nombres 0,1 et 0,2 n'étant pas dyadiques, leurs représentations machine ne sont qu'approximatives. Évidemment, un calcul isolé ne produira pas d'erreur importante, mais il peut en être autrement lorsqu'une longue suite de calculs provoque un cumul des erreurs d'arrondi.
Revenons au calcul de $0,1+0,2$ ci-dessus. Lors de la conversion en dyadique, la règle est la suivante, *round to nearest, ties to even* : en cas d'égalité, le nombre pair le plus proche est choisi. Par exemple, la représentation dyadique de 0,1 est infinie : $0,1 = \overline{0,0001100110011001\cdots}_2$ (répétition à l'infini de la séquence 1001) donc la représentation sous forme de flottant de 0,1 est

$$1,\underbrace{100110011001\cdots1001}_{48\text{ bits}}1010\times 2^{-4}$$

et celle de 0,2 est

$$1,\underbrace{100110011001\cdots1001}_{48\text{ bits}}1010\times 2^{-3}$$

Pour additionner ces deux nombres, la plus petite des deux voit ses bits décalés d'un cran : 0,1 est provisoirement représenté par

$$0,\underbrace{110011001100\cdots11001100}_{48\text{ bits}}1101\times 2^{-3}$$

On réalise l'addition de ces deux nombres dyadiques pour obtenir :

$$10,\underbrace{01100110\cdots01100110}_{48\text{ bits}}0111\times 2^{-3}$$

Ce résultat est enfin normalisé (donc arrondi au plus proche), ce qui donne

$$1,\underbrace{00110011\cdots00110011}_{48\text{ bits}}0100\times 2^{-2}$$

Vérifions sous Python :

```
r=2**(-3)+2**(-4)
x=1
for i in range(12):
    x=x+r*16**(-i)
x=2**(-2)*(x+2**(-50))
```

On trouve par ce programme :

$$0.30000000000000004$$

valeur effectivement renvoyée pour l'opération `0.1+0.2`.

Erreur par absorption

En mathématique, l'addition est associative : $(x+y)+z = x+(y+z)$, ce n'est pas le cas pour l'addition entre nombres flottants :

```
>>> (1.+2.**53)-2**53
0.
```

Le résultat de ce calcul est facile à comprendre. L'écriture normalisée de $1+2^{53}$ est égale à :

$$\underline{1,\overbrace{0000\cdots0000}^{\text{52 fois}}1}_2 2^{53}$$

mais comme la mantisse n'occupe que 52 bits, le résultat de cette addition n'est pas représentable en machine et sera approché par :

$$\underline{1,\overbrace{0000\cdots0000}^{\text{52 fois}}}_2 2^{53}$$

Autrement dit, en arithmétique flottante, $1+2^{53} = 2^{53}$. Ce mécanisme porte le nom d'absorption : l'addition de deux quantités dont l'écart relatif est très important entraîne l'absorption de la plus petite de ces deux quantités par la plus grande. En conséquence, il est parfois nécessaire de réorganiser les calculs pour obtenir un résultat plus conforme à nos attentes :

```
>>> 1.+(2.**53-2**53)
1.
```

Erreur par cancellation

Un autre problème se présente lors de la soustraction de deux quantités très proches. Pour fixer les idées, supposons que l'on connaisse deux quantités voisines x et y avec une précision de 20 chiffres significatifs après la virgule. Par exemple :

$$\begin{cases} x = \underline{1,01001001111010111010111}_2 \\ y = \underline{1,01001001111010111010000}_2 \end{cases}$$

Le résultat de l'opération $x-y$ sera normalisé pour être représenté en machine par

$$\underline{1,11}_2 \times 2^{-18}$$

Autrement dit, la précision ne sera plus que de 4 chiffres après la virgule. Cette perte drastique de précision porte le nom de cancellation, car la différence de deux quantités très voisines fait littéralement s'évanouir les chiffres significatifs.
Prendre conscience de ce phénomène permet de conditionner un calcul pour le rendre moins sensible à l'évanescence des chiffres significatifs. Par exemple, pour un calcul numérique il est préférable de calculer

$$\frac{1}{x(x+1)} \quad \text{plutôt que} \quad \frac{1}{x} - \frac{1}{x+1}$$

car, bien que ces deux quantités soient mathématiquement égales, la seconde est beaucoup plus sensible au phénomène de cancellation.

Une triste histoire en guise de conclusion

Le 4 juin 1996, c'est une erreur de programmation qui a causé le crash de la fusée européenne Ariane 5. Plus précisément, un réel codé sur 64 bits donnant la vitesse horizontale de la fusée était converti en un entier signé sur 16 bits. Or l'entier obtenu était plus grand que $32767 = 2^{15} - 1$, le plus grand entier représentable sur 16 bits. La convertion échouait donc. Tous les tests logiciels avaient pourtant réussi, mais ils avaient été effectués avec les données d'Ariane 4 pour laquelle la vitesse horizontale restait inférieure au maximum de 32767.

4. Ce qu'il faut retenir

Trois règles pour le calcul numérique

En règle générale, lors d'un calcul numérique il faut suivre les recommandations suivantes :

- ⇛ on évite d'additionner deux quantités dont l'écart relatif est très grand ;
- ⇛ on évite de soustraire deux quantités très voisines ;
- ⇛ on évite les tests d'égalité.

Vous n'avez pas à connaître la manière dont la norme IEEE-754 calcule les approximations flottantes. En revanche, il faut savoir coder des dyadiques et décoder des mots binaires dans cette norme.

5. Nombres complexes sous Python

Python permet le calcul sur les nombres complexes, représenté par des parties réelles et imaginaires sous forme de flottants. Le nombre imaginaire pur i est codé par `1j`. Les attributs `real` et `imag` permettent d'obtenir la partie réelle et la partie imaginaire. La fonction `abs` calcule le module.

Opération	Code
Addition, soustraction	`a+b`, `a-b`
Produit, puissances, quotient	`a*b`, `a**n`, `a/b`
Parties réelle et imaginaire	`a.real, a.imag`
Module	`abs(a)`

```
>>> a,b=1j,2+3*1j
>>> a/b,a*b,a**3
((0.23076923076923078+0.15384615384615385j), (-3+2j), (-0-1j))
>>> a.real,b.imag,abs(b)
(0.0, 3.0, 3.605551275463989)
```

6. Énoncés des tests

1.1.

Écrire $n = \underline{543234}_{10}$ en base 3 et $m = \underline{110011}_2$ en base 6.

1.2.

Soit $m \in \mathbb{N}^*$. Quel est le plus petit entier naturel non nul ayant m chiffre(s) en base 2 ? Même question avec *le plus grand*.

1.3.

Combien l'entier 10^4 a-t-il de chiffres en base 2 ?

1.4.

En inversant les chiffres en base 2 d'un entier n (0 à la place de 1 et inversement), on obtient un autre entier n'. Exprimer n' en fonction de l'entier n et de son nombre m de chiffres en base 2.

1.5.

Écrire $3n$ en base 3 où $n = \underline{21001}_3$ sans aucun calcul. Généraliser.

1.6.

Écrire $n+m$ en base 4 où $n = \underline{31231}_4$ et $m = \underline{20123}_4$.

1.7.

Soit $n \in \mathbb{N}^*$. On code les entiers de $[\![-2^{n-1}, 2^{n-1}-1]\!]$ par complément à deux sur n bits. Soit $m \in [\![1, 2^{n-1}-1]\!]$. Expliquer comment trouver la représentation de $-m$ à partir de celle m. Par exemple, pour $n = 4$, trouver la représentation de -7.

1.8.

On additionne deux nombres négatifs représentables par complément à 2 sur n bits. Montrer que, s'il n'y a pas dépassement, l'algorithme usuel d'addition binaire donne le bon résultat. On rappelle que les bits dépassant le poids 2^{n-1} sont perdus.

1.9.

On code les entiers de $[\![-8, 7]\!]$ par complément à deux sur 4 bits. Donner un exemple d'addition $a+b$ avec a et b positifs mais donnant pour résultat un entier négatif.

1.10.

Vérifier que l'opération $a+b$ (pour deux entiers $a \geqslant 0$ et $b < 0$ représentables par complément à deux sur n bits) s'effectue selon le même algorithme que pour les entiers positifs.

1.11.

Sur une machine 64 bits, par quel mot binaire est représenté l'entier 11 ? le décimal 11,0 ?

1.12.

Quel est le nombre dyadique représenté par le mot binaire suivant (norme IEEE-754 sur une machine 64 bits) :

$$01111111111010110111\overbrace{0\cdots0}^{44\text{ fois}}$$

1.13.

Que représente le mot binaire suivant (norme IEEE-754 sur une machine 64 bits) :

$$01111111111110110111\overbrace{0\cdots0}^{44\text{ fois}}$$

1.14.

Quel est le nombre dyadique représenté par le mot binaire suivant (norme IEEE-754 sur une machine 64 bits) :

$$00000000000010110111\overbrace{0\cdots0}^{44\text{ fois}}$$

1.15.

On a vu dans le cours que `0.2+0.2` produit sous Python `0.4`. Essayer de comprendre pourquoi le résultat est ici exact.

7. Solutions

1.1.

⇛ On applique l'algorithme du cours :

543234	3	181078	3
0	181078	1	60359
60359	3	20119	3
2	20119	1	6706
6706	3	2235	3
1	2235	0	745
745	3	248	3
1	248	2	82
82	3	27	3
1	27	0	9
9	3	3	3
0	3	0	1
1	3		
1	0		

On trouve donc $\underline{1000121011210}_3$.

⇛ On a $m = 2^5 + 2^4 + 2 + 2^0 = 51_{10}$. On applique l'algorithme du cours,

51	6	8	6
3	8	2	1
1	6		
1	0		

On trouve donc $\underline{123}_6$.

1.2.

Le plus petit :

$$\underbrace{\underline{10\cdots0}_2}_{m \text{ chiffres}} = 2^{m-1}$$

Le plus grand :

$$\underbrace{\underline{11\cdots1}_2}_{m \text{ chiffres}} = 2^{m-1} + \cdots + 2 + 1 = 2^m - 1$$

1.3.

Le nombre de chiffres en base 2 de $n = 10^4$ vaut

$$m = 1 + \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor = 1 + \left\lfloor \frac{4\ln(10)}{\ln(2)} \right\rfloor = 14$$

1.4.

On a

$$n + n' = 2^{m-1} + 2^{n-2} + \cdots + 2^1 + 2^0 = 2^m - 1$$

ainsi $n' = 2^m - n - 1$.

1.5.

D'une manière générale, multiplier n par b^k revient à ajouter k zéro(s) à la fin de l'écriture en base b de n. Ainsi, $3n = \underline{210010}_3$.

1.6.

L'algorithme d'addition en base 2 s'adapte sans peine à d'autre base (on effectue une retenue à chaque « *paquet* » de 4). Ainsi,

$$\begin{array}{r} 1\quad 1\,1\,1 \\ 3\,1\,2\,3\,1 \\ +\quad 2\,0\,1\,2\,3 \\ \hline = 1\,1\,2\,0\,2\,0 \end{array}$$

ie $n + m = \underline{112020}_4$.

1.7.

Il s'agit de trouver les bits de l'entier naturel $2^n - m$ en fonction de ceux de m. Comme

$$2^n - m = 2^n - 1 + 1 - m = \left(2^{n-1} + \cdots + 2^0\right) - m + 1$$

On calcule inverse donc les bits de m (1 à la place de 0 et inversement) pour obtenir $2^{n-1} + \cdots + 2^0 - m$ puis on ajoute 1. Par exemple, 7 est codé `0111`. On inverse, `1000`, puis on ajoute 1 : -7 est représenté par `1001`.

1.8.

Soient a et b des entiers strictement négatifs représentables par compl ément à 2 sur n bits. Supposons $a+b$ représentable, ie $a+b \geqslant -2^{n-1}$. On va vérifier que la représentation de $a+b$ s'obtient par l'algorithme usuel d'adition en base 2. Les nombres a et b sont représentés par les bits des entiers naturels 2^n+a et 2^n+b. Ainsi, par l'algorithme usuel, on obtient

$$2^{n+1}+a+b \geqslant 2^{n+1}-2^{n-1} = 2^n+2^{n-1} \geqslant 2^{n-1}$$

donc, après perte du bit en facteur de 2^n, on trouve donc un entier de bit de poids fort égal à 1 qui représente donc un entier m négatif entre -1 et $1-2^{n-1}$. Puisque

$$m = \left(2^{n+1}+a+b-2^n\right) - 2^n = a+b$$

le résultat obtenu est correct.

1.9.

Considérons $5+5$:

$$\begin{array}{r} \,1\quad 1 \\ 0\,1\,0\,1 \\ +\,0\,1\,0\,1 \\ \hline =\,1\,0\,1\,0 \end{array}$$

On trouve $2^3+2^2-16=-4$.

1.10.

L'entier b est représenté par 2^n+b. Par l'algorithme usuel, on calcule donc 2^n+a+b.

- Si $a \geqslant -b$, alors $a+b \in [\![0, 2^{n-1}-1]\!]$. Le bit de poids fort est perdu, ce qui signifie que le nombre codé est bel et bien l'entier naturel $a+b$.
- Si $a < -b$, alors $a+b \in [\![1-2^{n-1}, -1]\!]$. Comme

$$2^n+a+b = 2^n - \underbrace{(-a-b)}_{\in [\![0,\,2^{n-1}-1]\!]}$$

le résulat est bel et bien la représentation par complément à 2 de l'entier $a+b$.

1.11.

- L'entier 11 est représenté par complément à deux sur 64 bits par :

$$\underbrace{0000\cdots000}^{60\text{ fois}}1011_2$$

- Comme

$$11,0 = \underline{1001}_2 = \underline{1,001}_2 \times 2^3$$

le décimal 11,0 est représenté dans la norme IEEE-754 sur une machine 64 bits par

$$0\,10000000010\,001\,\underbrace{0\cdots0}^{49\text{ fois}}$$

car l'exposant $e=3$ est représenté par $e' = e+2^{10}-1 = 2^{10}+2$.

1.12.

Il s'agit d'un nombre normalisé car $e' = 2^{11}-2$. Le signe est positif, l'exposant vaut $e = e'-2^{10}+1 = 2^{10}-1 = 1023$ et la mantisse vaut $101101110\cdots0$. Ainsi, le nombre vaut

$$x = \underline{1,10110111}_2 \times 2^{1023}$$

1.13.

Il s'agit d'une exception : `inf` ou `NaN`.

1.14.

Il s'agit d'un nombre dénormalisé et, par convention $e=-1022$. La mantisse vaut `10110111` donc il s'agit du nombre

$$x = \underline{0,10110111}_2 \times 2^{-1022}$$

1.15.

On a vu que le représentation de 0,2 est

$$1,\underbrace{100110011001\cdots1001}_{48\text{ bits}}1010 \times 2^{-3}$$

Ainsi, pour les besoins de l'addition, on décale la virgule :

$$0,\underbrace{110011001100\cdots1100}_{48\text{ bits}}1101 \times 2^{-2}$$

L'addition produit alors :

$$1,\underbrace{100110011001\cdots100}_{48\text{ bits}}1010 \times 2^{-2}$$

Or, la représentation binaire de 0,4 vaut

$$1,\underbrace{100110011001\cdots1001}_{48\text{ bits}}1010 \times 2^{-2}$$

(on multiplie par 2 la représentation de 0,2, ce qui ne fait que décaler l'exposant).