Codage des nombres

Les nombres, comme tout autre objet, sont représentés dans la machine par des 0 et des 1 (plus exactement tension haute/basse).

Nous allons voir dans ce chapitre comment cela se passe exactement, et quelles sont les limites de ces représentations.

I Représentation des nombres entiers

1 Écriture en base b

a Définition

Définition

Si $b \in \mathbb{N} \setminus \{0,1\}$, tout entier naturel non nul n s'écrit de manière unique $n = \sum_{k=0}^{N} a_k b = a_0 + a_1 b + \cdots + a_N b^N$ avec $N \in \mathbb{N}, a_0, \ldots, a_N \in [\![0, b-1]\!]$, $a_N \neq 0$.

On note $n = \overline{a_N \cdots a_1 a_0}^b$ ou $n = (a_N \cdots a_1 a_0)_b$.

0 s'écrit 0 dans toutes les bases.

Remarques

R 1 – La base naturelle est la base 10 (décimale).

R 2 – En informatique, les bases usuelles seront la base 2 (binaire, dont les chiffres sont 0 et 1 appelés **bits**) et la base 16 (hexadécimal, dont les chiffres sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) : une unité hexadécimale représente 4 bits soit 1/2 octets.

R 3 – Nombre de chiffres en base b : n possède k chiffres en base b si et seulement si $b^{k-1} \leqslant n < b^k$ soit $k-1 \leqslant \log_b n < k$ donc $\boxed{k = \lfloor \log_b n \rfloor + 1}$ (donc $N = \lfloor \log_b n \rfloor$).

R 4 – Savoir retrouver rapidement les premières puissances de 2 : 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, etc.

b Conversion en décimal

Il est très facile d'écrire un nombre en base 10 connaissant son écriture en base b (la liste L contient les chiffres dans l'ordre inverse, en partant des unités) :

```
def base2decimal(L,b):
    """conversion depuis la base
b en décimal :
L[0]+L[1]*b+L[2]*b**2+..."""
    bpuiss = 1
    resultat = 0
    n = len(L)
    for i in range(n):
        resultat += L[i] * bpuiss
        bpuiss *= b
    return resultat
```

Invariant de sortie : `bpuiss` contient b^i et `resultat` contient $a_0 + a_1 b + \cdots + a_{i-1} b^{i-1}$ où $L = [a_0, a_1, \ldots, a_{n-1}]$

Si L contient les chiffres dans l'ordre de "lecture" : $L = [a_N, \ldots, a_1, a_0]$ on peut utiliser le schéma de Hörner : $n = (\cdots(a_N \times b + a_{N-1}) \times b + \cdots + a_1) \times b + a_0$.

```
def base2decimal_bis(L,b):
    """conversion depuis la base
b en décimal :
L[-1]+L[-2]*b+L[-3]*b**2 +..."""
    resultat = 0
    N = len(L)
    for i in range(N - 1):
        resultat += L[i]
        resultat *= b
    if N != 0:
        resultat += L[-1]
    return resultat
```

Invariant de sortie : `resultat` contient $(\cdots(a_N \times b + a_{N-1}) \times b + \cdots + a_{N-i}) \times b$ où $L = [a_N \ldots, a_1, a_0]$

Exercices

- **Ex 1 –** Donner l'écriture en base 10 de 240_5.
- **Ex 2 –** Donner l'écriture en base 10 de $110\ 0101\ 1111_2$.
- **Ex 3 –** Donner l'écriture en base 10 de $2A3F_{16}$.
- **Ex 4 –** Donner l'écriture en base 10 de 11_2, 111_2, 1111_2, $1\ 1111_2$. Expliquer.
- **Ex 5 –** Combien d'entiers peut-on représenter en binaire sur n bits ?
- **Ex 6 –** Calculer, en les posant, $1011\ 1101_2 + 1001\ 0111_2$ puis $1011\ 1101_2 \times 1101_2$. Vérifier en convertissant en décimal.
- **Ex 7 –** Quel est l'effet sur l'écriture binaire d'une multiplication par 2 ? Par une puissance de 2 ?
- **Ex 8 –** Calculer $0_2 - 0_2$, $1_2 - 0_2$, $1_2 - 1_2$, $10_2 - 1_2$, $100_2 - 1_2$, $1000_2 - 1_2$.
  Calculer ensuite $1011\ 1101_2 - 1001\ 0111_2$.

C Conversion en base b

La conversion en base b s'effectue par divisions euclidiennes successives par b : les restes successifs sont les chiffres $a_0, a_1, \ldots$

```
def decimal2base(n,b):
    """Liste [a_0,...,a_N] des
chiffres de l'écriture en base b
de n"""
    L = []
    m = n
    while m != 0:
        L.append(m % b)
        m //= b
    return L
```

À la fin de l'étape i, `m` contient $\left\lfloor \frac{n}{b^i} \right\rfloor$ et `L` contient $[a_0, a_1, \ldots, a_{i-1}]$.

Exercices

Ex 1 – Convertir 2017 en binaire, hexadécimal, base 7.

Ex 2 – Les adresses IPv4 sont codées sur 4 octets.
Par exemple : 192.168.1.28 (réseau local).
Donner l'écriture en binaire et en hexadécimal de cette adresse.
Même question avec 173.194.78.99 (Google).

2 Les entiers relatifs

Comment représenter des entiers négatifs ?

Une première idée serait de réserver un bit pour le signe (le premier, dit de **poids fort**), et les autres pour la valeur absolue.

Par exemple, pour des mots de 16 bits, le premier bit est pour le signe, les 15 autres pour la valeur absolue, on peut alors coder tous les entiers entre $-111\ 1111\ 1111\ 1111_2 = -\left(2^{15}-1\right) = -32\ 767$ et $111\ 1111\ 1111\ 1111_2 = 2^{15}-1 = 32\ 767$.

Cependant, il y a plusieurs inconvénients :

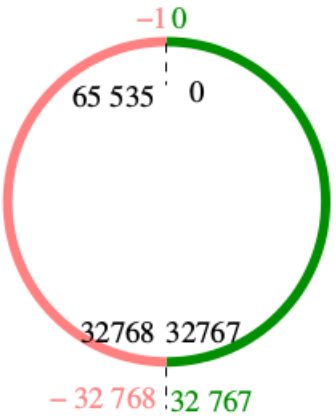
- 0 admet deux représentations : 0000 0000 0000 0000 et 1000 0000 0000 0000 (+0 et −0).
- La soustraction ne se pose pas comme addition de l'opposé ($x - y = x + (-y)$). Il faut donc prévoir électriquement des circuits d'addition et de soustraction différents, et distinguer à chaque fois les différents cas possibles suivant les signes.

a Complément à 2

L'idée est la suivante : par exemple, sur 8 bits, on va coder tous les nombres entre $-128 = -2^7$ et $127 = 2^7 - 1$: l'astuce consiste à regarder les entiers entre 0 et $2^7 + 2^7 - 1 = 2^8 - 1 = 255$ (codés sur 16 bits en binaire), et de réduire modulo 2^8.

- les nombres entre 0 et $127 = 2^7 - 1$ sont codés normalement en binaire sur 8 bits comme précédemment, le bit de poids fort valant forcément 0 (car $< 2^7$).
- pour les nombres $x = -k$ entre $-128 = -2^7$ et -1, on les code avec l'écriture binaire de $2^8 - k = 1111\ 1111_2 - k + 1$, l'opération $1111\ 1111_2 - k$ étant juste un échange de chaque bit de l'écriture de k (devient 0 si c'était 1, 1 si c'était 0). Dans ce cas, le bit de poids fort vaut forcément 1.

1111 1111 1111 1111 | 0000 0000 0000 0000



1000 0000 0000 0000 | 0111 1111 1111 1111

Complément à 2 sur 16 bits.

Exemple

52 est codé par 0011 0100 et -52 est codé par 1100 1100.

Remarque

$100-1=99$, $1000-1=999$, $100...00-1=999...99$. Le principe est le même ici avec $0..0-1=11..1$, puis on retranche 1 à chaque étape.

Définition : Représentation en complément à 2

On peut représenter en complément à 2 sur n bits des entiers relatifs x entre -2^{n-1} et $2^{n-1}-1$:

- si $x \geqslant 0$, il s'agit de l'écriture binaire de x (avec un bit de poids fort à 0)
- si $x < 0$, il s'agit de l'écriture binaire de $x + 2^n$ (avec un bit de poids fort à 1)

Remarques

R1 – Comme dans le cas particulier précédent, pour obtenir l'opposé d'un nombre il suffit de changer tous les bits et de rajouter 1.

R2 – Pour l'opération inverse, si on a la représentation en complément à 2 à n bit de x, soit le bit de poids fort vaut 0 ($x \geqslant 0$) et il suffit de convertir en décimal, soit il vaut 1 ($x < 0$) et il faut retrancher 2^n à la conversion en décimal.

R3 – Cela revient en quelques sorte à travailler modulo 2^n. Les opérations se font normalement, les bits dépassant 2^n n'étant pas pris en compte.

R4 – On peut voir la représentation en complément à deux de la manière suivante : $a_{n-1}\cdots a_1 a_0$ code le nombre $x = a_0 + a_1 \cdot 2 + \cdots + a_{n-2}2^{n-2} - a_{n-1}2^{n-1}$. En effet, si $a_{n-1} = 1$,

$$x + 2^n = a_0 + a_1 \cdot 2 + \cdots + a_{n-2}2^{n-2} + 2^{n-1} = a_0 + a_1 \cdot 2 + \cdots + a_{n-2}2^{n-2} + a_{n-1}2^{n-1}.$$

Exercices

Ex 1 – Donner la représentation en complément à 2 sur un nombre minimal de bits de 113 et −117. Donner deux méthodes pour ce dernier.

Ex 2 – Donner la valeur des nombres représentés en complément à 2 par 0101 0011 et 1100 1100.

Ex 3 – Écrire 99 et 57 sur 8 bits, et calculer la somme des représentations. Étonnant ? Expliquer.

Comme la représentation en complément à deux est une représentation binaire modulo 2^n, soustraire deux nombres revient à faire la somme de deux nombres positifs et à réduire modulo 2^n. Ainsi, avec cette notation, une soustraction revient à un calcul d'opposé (inversion de bits et retenue) puis une addition.

b Dépassement de capacité

Que se passe-t-il lorsque, lors d'une opération, le résultat dépasse $2^{n-1}-1$? Comme on travaille modulo 2^n avec des nombres entre -2^{n-1} et $2^{n-1}-1$, le résultat pourra être négatif !

Exemple

Sur 8 bits, si on fait 0111 1111 + 1, c'est-à-dire 127 + 1, on obtient 1000 0000, c'est-à-dire −128(= 128 − 256).

Par contre, si on fait 1111 1111 + 1, on devrait obtenir 1 0000 0000, mais comme on ne travaille que sur 8 bits, le bit de poids fort est perdu et on obtient en fait 0000 0000 ce qui donne bien −1 + 1 = 0.

Ainsi, non seulement notre représentation est limitée (on ne peut pas représenter tous les entiers avec un nombre de bits fixés, 32 ou 64 sur les machines actuelles), mais il y a en plus un phénomène de dépassement de capacité (*overflow*) à gérer sous peine d'obtenir des résultats inattendus.

Dans certains langages comme le C ou le java, il existe plusieurs types de nombres à choisir suivant les besoins (en complément à 2) :

- sur 8 bits soit 1 octet (`short` / `byte`),
- sur 16 bits soit 2 octets (`int` / `short`),
- sur 32 bits soit 4 octets (`long int` / `int`),
- sur 64 bits soit 8 octets (`long long int` / `long`).

Comment détecter un dépassement de capacité ? On vérifie facilement en utilisant les bornes des entiers, qu'il y a un dépassement de capacité si et seulement si on somme deux entiers de même signe et le résultat a le signe contraire. Il suffit donc d'analyser le bit de poids fort pour le savoir.

Remarque

En décembre 2014, Youtube est obligé de chiffrer le nombre de vues d'une vidéo sur 64 bits au lieu de 32 après le dépassement de 2 147 483 647 = $2^{31}-1$ vues pour la clip Gangnam Style du Coréen PSY ! Prochain dépassement ? 2^{63} = 9 223 372 036 854 775 808 !

Et en Python ?

Python a cette particularité de pouvoir gérer des entiers de taille arbitraire (la seule limite est celle de la mémoire de la machine). Comment cela fonctionne-t-il ?

Sur les machines 32 bits, la représentation binaire des entiers est découpée en paquets de 15 bits, stockés dans des tableaux dont les éléments sont des entiers de 16 bits. Ce sont les "chiffres" de l'entier initial écrit en base 2^{15}. On lui associe aussi un (petit) entier **taille** contenant le nombre de "chiffres" en base 2^{15} et dont le signe est celui de l'entier.

Exemple

Le nombre 1 234 567 891 011 s'écrit en binaire

1 0001 1111 01|11 0001 1111 1011 0|000 1000 0100 0011

ce qui donne en base 2^{15} :

- premier "chiffre" : $000\ 1000\ 0100\ 0011_2 = 2\ 115$
- deuxième "chiffre" : $110\ 0011\ 1111\ 0110_2 = 25\ 590$
- troisième "chiffre" : $100\ 0111\ 1101_2 = 1\ 149$

Notre entier sera donc représenté par le tableau `[2115, 25590, 1149]` et l'entier `3`.

Son opposé $-1\ 234\ 567\ 891\ 011$ est représenté par le tableau `[2115, 25590, 1149]` et l'entier `-3`.

Sur les machines 64 bits, c'est le même principe mais en base 2^{30}.

Pour connaître la base (en puissance de 2) et la taille d'occupation (en octets) de chaque "chiffre" dans cette base, il suffit de taper :

```
>>> import sys
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

Aucun phénomène de dépassement de capacité n'est visible en Python.

II REPRÉSENTATION DES NOMBRES À VIRGULE FLOTTANTE

1 L'arithmétique flottante

On peut représenter en binaire des "nombres à virgule" comme on le fait avec les nombres décimaux : les demis, quarts, huitièmes, etc. remplacent les dixièmes, centièmes, millièmes, etc.

Exemple

$\overline{10{,}01101} = 10{,}01101_2$ est le nombre $2^1 + 2^{-2} + 2^{-3} + 2^{-5} = 2 + \frac{1}{4} + \frac{1}{8} + \frac{1}{32} = \frac{77}{32} = 2{,}40625$

L'avantage de cette représentation est qu'elle est exacte, l'inconvénient est qu'elle ne permet pas de représenter des nombres très grands ou très petits facilement.

On adopte plutôt une représentation binaire proche de la représentation scientifique habituelle :

Définition : Norme IEEE 754

Un nombre à virgule flottante x est représenté sous la forme d'une triplet (s, e, m) où s est **le signe**, n est **l'exposant** et m est **la mantisse**, tels que :

$$x = sm2^e$$

avec $s = \pm 1$, m un nombre à virgule dans l'intervalle $[1, 2[$, et e un entier.

Le signe s est toujours codé sur 1 bit (celui de poids fort).
Puis :
- en 32 bits (simple précision), e est codé sur 8 bits et m sur 23 bits.
- en 64 bits (double précision), e est codé sur 11 bits et m sur 52 bits.

Le codage est le suivant :
- Le premier bit est celui du signe : 0 pour + et 1 pour −.
- Les 8/11 bits suivants sont une représentation de l'exposant en « excédent 127/1023 » : c'est l'écriture binaire de $e + 127$ en simple précision et $e + 1023$ en double précision. Le cas où les bits seraient tous à 1 ou tous à 0 est exclus.
- Les 23/52 derniers bits sont les 23/52 chiffres binaires après la virgule de m (le chiffre des unités étant toujours 1, inutile de le représenter.)

On a alors les limitations suivantes :

- En **simple précision**, l'exposant est codé par $e + 127$ sur 8 bits, non tous nuls/égaux à 1, donc $1 \leqslant e + 127 \leqslant 2^8 - 2 = 254$ et donc $-126 \leqslant e \leqslant 127$.
  Le plus petit nombre (en valeur absolue) représentable est :
  $$1 \cdot 2^{-126} \approx 1{,}2 \cdot 10^{-38}$$
  Le plus grand nombre (en valeur absolue) représentable est :
  $$\overline{1{,}11111\ldots1} \cdot 2^{127} \approx 2^{128} \approx 3{,}4 \cdot 10^{38}$$
  La précision machine (écart relatif maximal entre deux nombres qui se suivent) est
  $$\overline{0{,}00000\ldots1} = 2^{-23} \approx 1{,}2 \cdot 10^{-7}$$
- En **double précision**, l'exposant est codé par $e + 1023$ sur 11 bits, non tous nuls/égaux à 1, donc $1 \leqslant e + 1023 \leqslant 2^{11} - 2 = 2046$ et donc $-1022 \leqslant e \leqslant 1023$.
  Le plus petit nombre (en valeur absolue) représentable est :
  $$1 \cdot 2^{-1022} \approx 2{,}2 \cdot 10^{-308}$$
  Le plus grand nombre (en valeur absolue) représentable est :
  $$\overline{1{,}11111\ldots1} \cdot 2^{1023} \approx 2^{1024} \approx 1{,}8 \cdot 10^{308}$$
  La précision machine (écart relatif maximal entre deux nombres qui se suivent) est
  $$\overline{0{,}00000\ldots1} = 2^{-52} \approx 2{,}2 \cdot 10^{-16}$$

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16,
radix=2, rounds=1)
```

Remarque

Avantage de cette représentation : c'est très facile de faire des produits/divisions. Inconvénient : ça l'est moins pour des sommes...

Pour convertir un nombre binaire codé sur 64 bits en réel, voilà comment procéder : le mot se découpe en

$$s|e_1e_2\cdots e_{11}|m_1m_2\cdots m_{52}$$

Il représente le réel

$$(-1)^s\cdot\overline{1,m_1\dots m_{52}}\cdot 2^{\overline{e_1e_2\cdots e_{11}}-1023}$$

où $\overline{1,m_1\dots m_{52}} = 1+\frac{m_1}{2}+\cdots+\frac{m_{52}}{2^{52}}$.

Cela fonctionne de manière analogue en simple précision.

Exercices

Ex 1 – Écrire la représentation en simple précision de $-245{,}375$.

Ex 2 – Quel est le nombre représenté en double précision par

$$C4693C3800000000\ ?$$

2 Des nombres hors norme (hors-programme)

La norme prévoit aussi quatre types de représentations non normalisées :

- le nombre zéro est représenté par les bits de l'exposant et de la mantisse tous à 0 (qui ne représente donc pas $\pm 2^{-127}$ ou $\pm 2^{-1023}$). Il y a donc deux représentations de 0, l'une positive, l'autre négative suivant le bit de poids fort.

$$*|000..0|000...0$$

- lorsque tous les bits de l'exposant sont à 0 mais que la mantisse n'est pas nulle, on obtient les nombres dits dénormalisés pour lesquels l'exposant vaut -126 ou -1022 et la matisse s'écrit 0,... au lieu de 1,.... Ils permettent d'améliorer la précision des calculs très proches de 0.

$$*|000..0|****...*$$

⋆ **En simple précision** :
- le plus petit nombre (en valeur absolue) ainsi représenté est

$$\overline{0,0000...1}\cdot 2^{-126} = 2^{-23}\cdot 2^{-126} = 2^{-149} \approx 1,4\cdot 10^{-45}.$$

- le plus grand nombre (en valeur absolue) ainsi représenté est

$$\overline{0,1111...1}\cdot 2^{-126} \approx 2^{-126} \approx 1,2\cdot 10^{-38}.$$

⋆ **En double précision** :
- le plus petit nombre (en valeur absolue) ainsi représenté est donc

$$\overline{0,0000...1}\cdot 2^{-1022} = 2^{-52}\cdot 2^{-1022} = 2^{-1074} \approx 5\cdot 10^{-324}.$$

- le plus grand nombre (en valeur absolue) ainsi représenté est

$$\overline{0,1111...1}\cdot 2^{-1022} \approx 2^{-1022} \approx 2,2\cdot 10^{-308}.$$

- $\pm\infty$ est représenté avec les bits de l'exposant tous à 1, et ceux de la mantisse à 0 (ce qui ne représente donc pas $\pm 2^{128}$ ou $\pm 2^{1024}$).

$$0|111..1|000...0 \quad \text{et} \quad 1|111..1|000...0$$

  Notons que les opérations arithmétiques sur $\pm\infty$ sont possibles.
- Lorsque l'on effectue une opération non autorisée, on obtient *NaN (not a number)* représenté par les bits de l'exposant tous à 1 et ceux de la mantisse quelconques mais non tous nuls.

$$*|111..1|****...*$$

3 Limites de la représentation

a Dépassement de capacité

Un nombre flottant étant représenté sur un nombre fini bits, on a bien sur des limites sur les nombres représentables comme on l'a déjà vu.

Ainsi, quand un résultat dépasse la limite supérieure (resp. inférieure), il devient $+\infty$ (resp. $-\infty$) : c'est un *overflow*.

De même, quand un résultat est trop proche de 0 et dépasse la limite des nombres dénormalisés, on obtient un *underflow* qui peut se traduire par $+0$ ou -0, ou par une erreur.

b Erreurs d'arrondi

Seuls des nombres de la forme [1] $\dfrac{k}{2^n}$ peuvent être représentés exactement avec ce codage, donc la plupart du temps il s'agit d'une valeur approchée de la valeur exacte.

1. et encore, pas tous !

Par exemple, le nombre $0,4$ s'écrit en base 2

$$0,0110011001100...$$

avec une période de 2 dans les chiffres après la virgule. En double précision sa représentation donne une nombre environ égal à

$$0,40000000000000002$$

Seuls les 16 premiers chiffres significatifs sont corrects. Cela est dû au fait qu'on a une précision machine (écart relatif maximal) de 2^{-52}, de l'ordre de 10^{-16}.

De même, en 32 bits, on n'a que 7 chiffres significatifs sûrs car la précision machine est 2^{-23}, de l'ordre de 10^{-7}.

Des erreurs d'arrondis apparaissent aussi lors des calculs. Pour chaque résultat obtenu, on a une erreur relative majorée par la précision machine. Ainsi, le calcul de $a+b=c$ renvoie en réalité $c(1+\varepsilon)$ où $\varepsilon \leqslant 2^{-52}$ en double précision et $\varepsilon \leqslant 2^{-23}$ en simple précision (ε dépend de a et b).

Ainsi, la somme ne sera plus associative : $(a+b)+c$ ne renvoie pas le même résultat que $a+(b+c)$! et commutative $a+b+c$ ne renvoie pas le même résultat que $b+a+c$

Enfin, après n opérations sur des nombres à virgule flottante introduisant une erreur relative majorée par ε, on est susceptible d'avoir le résultat multiplié par $(1+\varepsilon)^n \approx 1+n\varepsilon$ donc une erreur relative de l'ordre de $n\varepsilon \xrightarrow[n\to+\infty]{} +\infty$!!!

Premier exemple : en double précision,

```
>>> 2 ** -53 + (1 - 1)
1.1102230246251565e-16
>>> (2 ** -53 + 1) - 1
0.0

>>> 0.1 + 0.2 - 0.3
5.551115123125783e-17
>>> 0.1 + (0.2 - 0.3)
2.7755575615628914e-17

>>> 0.25 + 0.75 - 1
0.0
```

Deuxième exemple : obtention de la précision machine et du plus petit nombre,

```
>>> epsilon, n = 1, 0
>>> while 1 + epsilon / 2 != 1: epsilon, n = epsilon / 2, n-1
>>> epsilon, n
(2.220446049250313e-16, -52)


>>> mini, n = 1, 0
>>> while mini / 2 != 0: mini, n = mini / 2, n - 1
>>> mini, n
(5e-324, -1074)
```

Troisième exemple

```
>>> Pi_1 = 4 * sum((-1) ** k / (2 * k + 1) for k in range(0, 1000000))
>>> Pi_1
3.1415916535897743
>>> Pi_2 = 4 * sum((-1) ** k / (2 * k + 1) for k in range(999999, -1, -1))
>>> Pi_2
3.1415916535897934
>>> Pi_2 - Pi_1
1.9095836023552692e-14
```

Quatrième exemple Si on considère $f(x) = \frac{1}{1-\sqrt{1-x^2}}$ et qu'on veut évaluer f pour $|x|$ petit :

```
>>> def f(x) : return 1 / (1 - (1 - x ** 2) ** .5)
>>> f(2 ** -27)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    f(2 ** -27)
  File "<pyshell#0>", line 1, in f
    def f(x) : return 1 / (1 - (1 - x ** 2) ** .5)
ZeroDivisionError: float division by zero
```

car $\sqrt{1-x^2}$ a été arrondi à 1.

Alors que si l'on écrit $f(x) = \frac{1+\sqrt{1-x^2}}{x^2}$, plus de problème.

```
>>> def f(x) : return (1 + (1 - x ** 2) ** .5) / (x ** 2)
>>> f(2 ** -27)
3.602879701896397e+16
```

Lorsque l'on soustrait des nombres très proches, il y a un phénomène de compensation qui peut donner une importante perte de précision : par exemple, si on a 10 chiffres significatifs, $x \approx 100{,}0000000$ et $y \approx 99{,}9999998$, on obtient $x - y \approx 0{,}0000002$ soit une perte de 9 chiffres significatifs !

On a que $\mathrm{e}^{-100} = \sum_{k=0}^{+\infty} u_k = \sum_{k=0}^{+\infty} (-1)^k \frac{100^k}{k!} \approx 3{,}7 \cdot 10^{-44}$. En calculant directement avec les 1000 premiers termes et en calculant plutôt $1/e^{100}$:

```
>>> terme, som = 1., 0
>>> for k in range(0, 1000):
        som += terme
        terme = -terme * 100 / (k + 1\
 )
>>> som
-2.9137556468915326e+25
```

```
>>> terme, som=1., 0
>>> for k in range(0, 1000):
        som += terme
        terme = terme * 100 / (k + 1)
>>> 1 / som
3.7200759760208386e-44
```

L'erreur absolue pour les plus grands termes, $|u_{99}| = |u_{100}|$ (qui vaut environ 2^{139}) est de l'ordre de $2^{-52}2^{139} = 2^{87} \gg \mathrm{e}^{-100}$.

Morale

- Tout calcul fait par un ordinateur est potentiellement faux !
- Ne jamais tester l'égalité de deux flottants.
- Toujours garder un oeil critique sur des calculs avec des flottants.

4 Quelques catastrophes dues à l'arithmétique flottante

(Source : www.math.univ-paris13.fr/~japhet/Doc/Handouts/RoundOffErrors.pdf et https://www5.in.tum.de/~huckle/bugse.html)

a Missile Patriot

En février 1991, pendant la Guerre du Golfe, une batterie américaine de missiles Patriot, à Dharan (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours, dû à un problème d'arrondi. Les nombres étaient représentés en virgule fixe sur 24 bits, donc 24 chiffres binaires. Le temps était compté par l'horloge interne du système en 1/10 de seconde. Malheureusement, 1/10 n'a pas d'écriture finie dans le système binaire :

$$1/10 = 0{,}1 = \overline{0{,}0001100110011001100110011}...\text{(dans le système binaire).}$$

L'ordinateur de bord arrondissait 1/10 à 24 chiffres, d'où une petite erreur dans le décompte du temps pour chaque 1/10 de seconde. Au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures, ce qui avait entraîné une accumulation des erreurs d'arrondi de 0,34 s. Pendant ce temps, un missile Scud parcourt environ 500 m, ce qui explique que le Patriot soit passé à côté de sa cible. Ce qu'il aurait fallu faire c'était redémarrer régulièrement le système de guidage du missile.

b Explosion d'Ariane 5

Le 4 juin 1996, une fusée Ariane 5, à son premier lancement, a explosé 40 secondes après l'allumage. La fusée et son chargement avaient coûté 500 millions de dollars. La commission d'enquête a rendu son rapport au bout de deux semaines. Il s'agissait d'une erreur de programmation dans le système inertiel de référence. À un moment donné, un nombre codé en virgule flottante sur 64 bits (qui représentait la vitesse horizontale de la fusée par rapport à la plate-forme de tir) était converti en un entier sur 16 bits. Malheureusement, le nombre en question était plus grand que 32 768 (overflow), le plus grand entier que l'on peut coder sur 16 bits, et la conversion a été incorrecte.

c Bourse de Vancouver

Un autre exemple où les erreurs de calcul ont conduit à une erreur notable est le cas de l'indice de la Bourse de Vancouver. En 1982, elle a créé un nouvel indice avec une valeur nominale de 1000. Après chaque transaction boursière, cet indice était recalculé et tronqué après le troisième chiffre décimal et, au bout de 22 mois, la valeur obtenue était 524,881, alors que la valeur correcte était 1098,811. Cette différence s'explique par le fait que toutes les erreurs d'arrondi étaient dans le même sens : l'opération de troncature diminuait à chaque fois la valeur de l'indice