Chapitre 1

Représentation des nombres

Dans ce chapitre, nous allons nous intéresser à la façon dont un nombre (entier naturel, entier relatif ou réel) peut être représenté dans la mémoire d'un ordinateur. Rappelons que cette mémoire est composée de *bits* qui peuvent prendre deux valeurs, 0 ou 1. Un bit n'est en général pas accessible directement par le processeur : on appelle *byte* la plus petite quantité de mémoire adressable directement par ce dernier. Actuellement, pour la très grande majorité des architectures un byte est égal à un *octet* c'est-à-dire 8 bits. Pour cette raison, les nombres sont représentés par un ou plusieurs bytes, dans la pratique un ou plusieurs octets, et pour des raisons évidentes on utilise la décomposition en base 2 des nombres pour les représenter en machine.

1. Décomposition en base deux

1.1 Décomposition des entiers naturels

Nous sommes habitués depuis l'enfance à utiliser l'écriture en base 10 des entiers, qui utilise 10 symboles (les chiffres de 0 à 9) : par exemple, 2985 représente le nombre $2\times10^3+9\times10^2+8\times10+5\times10^0$.
La décomposition en base 2 d'un entier n'utilise que deux symboles, les chiffre 0 et 1 : on convient que l'écriture $(b_pb_{p-1}\cdots b_0)_2$ représente le nombre : $b_p\times2^p+b_{p-1}\times2^{p-1}+\cdots+b_1\times2+b_0\times2^0$, avec pour tout $k\in[\![0,p]\!]$, $b_k\in\{0,1\}$.
Par exemple, $2985=(101110101001)_2$ car $2985=1+8+32+128+256+512+2048$.

Remarque. Deux opérations s'avèrent particulièrement utiles pour manipuler la décomposition d'un entier naturel en base 2 : le quotient et le reste de la division euclidienne par 2. En effet, si $n=(b_p\cdots b_1b_0)_2$ et $m=(b_p\cdots b_2b_1)_2$ alors $n=2m+b_0$ donc $m=\left\lfloor\frac{n}{2}\right\rfloor$ et $b_0=n \bmod 2$.

En Python, le quotient de la division euclidienne de n par 2 se calcule par l'opération `n // 2` et le reste par l'opération `n % 2`.
Ainsi, si $n=(b_pb_{p-1}\cdots b_0)_2$ alors $b_0=$ `n % 2` et $(b_pb_{p-1}\cdots b_1)_2=$ `n // 2`.

Remarque. En Python, la fonction `bin` permet d'obtenir une chaîne de caractères représentant la décomposition binaire d'un entier :

```
In [1]: bin(1492)
Out[1]: '0b10111010100'
```

Remarque. Le préfixe `0b` est utilisé pour indiquer que ce qui suit est à lire en représentation en base 2 et non pas en base 10.

```
In [2]: 0b10111010100
Out[2]: 1492
```

À l'inverse, l'entier n défini par sa représentation binaire : $n=(b_p\cdots b_1b_0)_2$ peut être calculé à l'aide de la formule

$$n=\sum_{k=0}^{p}b_k2^k.$$

L'exemple précédent montre comment entrer au clavier un entier au format binaire ; sous forme de chaîne de caractères on peut aussi utiliser la fonction `int` :

```
In [3]: int('10111010100', 2)
Out[3]: 1492
```

1.2 Utilisation de la base seize

La raison du rôle particulier que joue la base 16 en informatique provient du fait que l'écriture binaire d'un nombre présente l'inconvénient d'être très longue à écrire, ce qui a incité les informaticiens à écrire ces nombres dans une base plus élevée. Le choix de la base 10 pourrait paraître naturel, mais malheureusement convertir un nombre de la base 10 à la base 2 ou inversement n'est pas chose facile. En revanche, nous allons voir que passer de la base 2 à la base 16 est très simple à réaliser.

Pour écrire un nombre en base 16, nous avons besoin d'un caractère pour chacun des entiers de 0 à 15 ; on complète donc les chiffres de 0 à 9 par les lettres a, b, c, d, e et f. Ainsi, $(\mathrm{a})_{16} = 10$, $(\mathrm{b})_{16} = 11$, $(\mathrm{c})_{16} = 12$, $(\mathrm{d})_{16} = 13$, $(\mathrm{e})_{16} = 14$, $(\mathrm{f})_{16} = 15$.
Sachant que $2^4 = 16$, tout nombre écrit en base 2 à l'aide de 4 caractères s'écrit en base 16 à l'aide d'un seul caractère :

$(0000)_2 = (0)_{16}$	$(0100)_2 = (4)_{16}$	$(1000)_2 = (8)_{16}$	$(1100)_2 = (\mathrm{c})_{16}$
$(0001)_2 = (1)_{16}$	$(0101)_2 = (5)_{16}$	$(1001)_2 = (9)_{16}$	$(1101)_2 = (\mathrm{d})_{16}$
$(0010)_2 = (2)_{16}$	$(0110)_2 = (6)_{16}$	$(1010)_2 = (\mathrm{a})_{16}$	$(1110)_2 = (\mathrm{e})_{16}$
$(0011)_2 = (3)_{16}$	$(0111)_2 = (7)_{16}$	$(1011)_2 = (\mathrm{b})_{16}$	$(1111)_2 = (\mathrm{f})_{16}$

Aussi, pour convertir un nombre quelconque de la base 2 à la base 16, il suffit de regrouper les chiffres qui le composent par paquet de 4 et convertir chacun de ces paquets en un chiffre en base 16. Par exemple, $(1011\ 0110\ 1110\ 1001)_2 = (\mathrm{b6e9})_{16}$.

Remarque. . La fonction hex permet d'obtenir une représentation sous forme de chaîne de caractères de la décomposition hexadécimale d'un entier :

```
In [4]: hex(1492)
Out[4]: '0x5d4'

In [5]: 0x5d4
Out[5]: 1492

In [6]: int('5d4', 16)
Out[6]: 1492
```

L'écriture hexadécimale (c'est-à-dire en base 16) est fréquemment utilisée en informatique car un octet (qui rappelons le vaut 8 bits) sera toujours représenté par deux caractères hexadécimaux.

Exemple. Une adresse IP (*Internet Protocol*) est un numéro d'identification attribué à chaque périphérique relié à un réseau informatique ; c'est ce qui permet à deux ordinateurs reliés par l'internet de communiquer.
Le protocole IPv4 utilise 4 octets (donc 32 bits), couramment représentés en notation décimale (donc une valeur comprise entre 0 et 255) séparés par des points ; sur un réseau local une adresse IP va ressembler à : `192.168.1.15`.
Ce protocole permet théoriquement à $2^{32} = 256^4 = 4\,294\,967\,296$ périphériques différents d'être reliés simultanément sur un même réseau internet ; cela peut paraître beaucoup, mais le développement rapide d'internet à cependant vite conduit à une pénurie d'adresses IPv4 disponibles. Ceci a conduit à la création du protocole IPv6 composé cette fois de 16 octets (donc 128 bits), traditionnellement représenté par 8 groupes de 2 octets séparés par un signe deux-points, comme par exemple : `2001:0db8:0000:85a3:0000:0000:ac1f:8001`.
On dispose maintenant de $2^{128} \approx 3.4 \times 10^{38}$ adresses différentes, soit un nombre potentiellement illimité d'adresses.

Exemple. Une couleur d'une page web est définie par trois octets représentant ses composantes RVB [1]. Ainsi, un navigateur web interprète le code couleur $(\mathrm{ffa500})_{16}$ comme du orange : les deux premiers chiffres $(\mathrm{ff})_{16} = 255$ représentent la luminance de la composante rouge (ici à 100%), les deux suivants $(\mathrm{a5})_{16} = 10 \times 16 + 5 = 165$ la luminance de la composante verte (ici à 65%) et les deux derniers $(00)_{16} = 0$ la luminance de la composante bleue (à 0%).

Potentiellement, $256^3 = 16\,777\,216$ couleurs différentes sont donc accessibles.

Remarque. Gardons cependant à l'esprit que l'écriture hexadécimale n'a pas de signification particulière pour la machine, seule l'écriture binaire en a. Ce n'est qu'une représentation commode pour les humains (plus commode que la base 2 car plus concise, et plus commode que la base 10 pour réaliser des conversions en base 2).

1. les composantes Rouge-Vert-Bleu en synthèse additive.

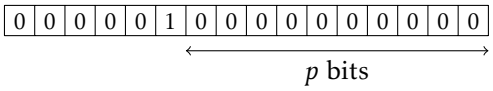
2. Représentation machine des nombres entiers

Pour pouvoir être stocké et manipulé par un ordinateur, un nombre doit être représenté par une succession de bits. Le principal problème est la limitation de la taille du codage : un nombre mathématique peut prendre des valeurs arbitrairement grandes, tandis que le codage dans l'ordinateur doit s'effectuer sur un nombre de bits fixé.

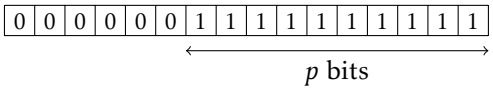
2.1 Les entiers naturels

On les code naturellement par leur décomposition en base 2. Un codage sur n bits permet de représenter tous les entiers naturels compris entre 0 et $2^n - 1$. Ainsi, un octet permet de coder les entiers allant de $0 = (00)_{16} = (0000\,0000)_2$ à $255 = (\texttt{ff})_{16} = (1111\,1111)_2$, et 64 bits (soit 8 octets) tous les nombres allant de 0 à $2^{64} - 1$.
Deux valeurs particulières demandent à être bien connues, la représentation des entiers de la forme 2^p et ceux de la forme $2^p - 1$:

– l'entier naturel 2^p est représenté par



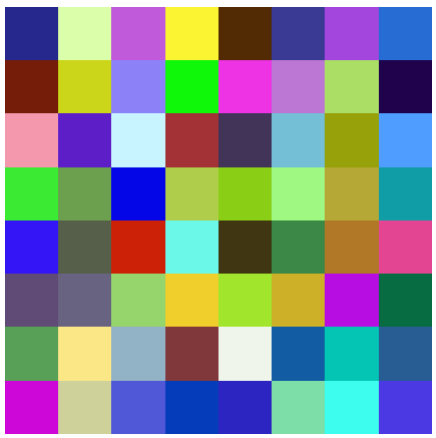
– l'entier naturel $2^p - 1$ est représenté par



En Python, la bibliothèque Numpy offre la possibilité de manipuler des entiers naturels codés sur 8, 16, 32 ou 64 bits : ce sont les types `uint8`, `uint16`, `uint32`, `uint64` (pour *unsigned integer*).
C'est par exemple le type `uint8` qui est utilisé par Python pour représenter chacune des trois composantes RVB d'un pixel d'une image en couleur. À titre d'illustration, le script suivant crée une image de 8×8 pixels, chaque pixel étant défini par un triplet RVB :

```
import numpy as np
import numpy.random as rd
import matplotlib.pyplot as plt

img = rd.randint(256, size=(8, 8, 3), dtype=np.uint8)
plt.imshow(img)
plt.show()
```



• Opérations sur les entiers naturels
Les entiers naturels étant codés sur un nombre fini de bits, les opérations mathématiques peuvent produire des résultats non représentables en machine. Observons le fonctionnement de Python dans ce cas :

```
In [1]: x = np.uint8(250)
In [2]: y = np.uint8(10)
In [3]: x + y
RuntimeWarning: overflow encountered in ubyte_scalars
Out[3]: 4
```

Python détecte un débordement, mais fournit néanmoins un résultat, qui n'est autre que le reste modulo 256 du résultat du calcul. Autrement dit, *les entiers naturels codés sur n bits restent dans l'intervalle* $[\![0, 2^n - 1]\!]$ *et les calculs sont effectués modulo* 2^n.

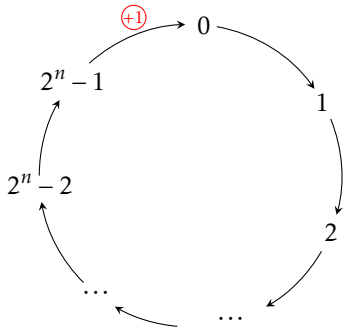


FIGURE 1 – Représentation machine des entiers naturels codés sur n bits.

Ce principe est facile à comprendre si on observe l'addition en base 2 de $250 = (11111010)_2$ et de $10 = (1010)_2$:

	1	1	1	1	1	0	1	0
+	0	0	0	0	1	0	1	0
1	0	0	0	0	0	1	0	0

Le résultat devant être stocké sur 8 bits, le bit de poids le plus fort est ignoré (mais c'est lui qui provoque le RuntimeWarning) et le résultat renvoyé est $(100)_2 = 4$.

2.2 Les entiers relatifs

Pour représenter un entier relatif x sur n bits on code le signe sur le premier bit avec la convention 0 pour les nombres positifs et 1 pour les nombres négatifs. Il reste donc $n-1$ bits pour représenter la valeur absolue de l'entier. Le plus grand entier relatif positif représentable sur n bits est donc égal à $2^{n-1} - 1$. En base 2, il s'écrit : $(0\underbrace{1111111\cdots1111}_{n-1 \text{ chiffres } 1})_2$.

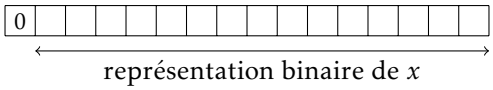


FIGURE 2 – Représentation machine d'un entier relatif positif.

• Le codage des nombres négatifs

On pourrait s'attendre à ce que la représentation d'un entier relatif négatif soit un 1 (le bit de signe) suivi de la représentation binaire de la valeur absolue de l'entier. *Il n'en est rien*, car cette méthode possède plusieurs inconvénients :

- Le nombre 0 posséderait deux représentations (et il est toujours préférable d'avoir unicité de la représentation d'un objet) ;
- L'algorithme d'addition ne s'applique qu'à des entiers de même signe et cette représentation le rendrait inapplicable pour additionner des entiers relatifs.

C'est pourquoi on utilise un codage particulier pour représenter les entiers négatifs, dit en *complément à deux* :

le nombre négatif x est représenté en mémoire par la représentation binaire de l'entier (positif) $2^n + x$.

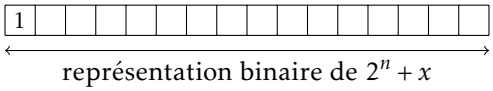


FIGURE 3 – Représentation machine d'un entier relatif négatif.

Pour que cette représentation sur n bits commence par un 1, il est donc nécessaire que $2^n + x$ soit compris entre $2^{n-1} = (1\underbrace{000\,0000\cdots0000}_{n-1\text{ chiffres}})_2$ et $2^n - 1 = (\underbrace{1\,1111\,1111\cdots1111}_{n-1\text{ chiffres}})_2$ soit :

$$2^{n-1} \leqslant 2^n + x \leqslant 2^n - 1 \iff -2^{n-1} \leqslant x \leqslant -1$$

Le plus petit entier négatif représentable sur une architecture à n bits est donc égal à -2^{n-1} et est représenté par : $(1\underbrace{000\cdots0000}_{n-1\text{ bits}})_2$.

Ainsi, les entiers relatifs représentables sur une architecture à n bits sont $\boxed{\text{compris entre } -2^{n-1} \text{ et } 2^{n-1} - 1.}$

Attention donc à bien distinguer un nombre de sa représentation, qui ne coïncide pas avec sa représentation binaire dans le cas des nombres négatifs.

Exemple. Pour simplifier les calculs, considérons une représentation sur 8 bits ($n = 8$).
L'entier relatif 105 est représenté par l'octet `01101001`, ce qui correspond à la représentation en base 2 de l'entier $105 = (1101001)_2$.
L'entier relatif -105 est représenté par l'octet `10010111`, ce qui correspond à la représentation en base 2 de l'entier $256 - 105 = 151 = (10010111)_2$.

Exercice 1. Dans une représentation en complément à deux sur 8 bits, quels sont les entiers relatifs représentés par `01101101` et par `10010010` ?

• Opérations sur les entiers relatifs
La bibliothèque Numpy permet de manipuler des entiers relatifs codés sur 8, 16, 32 ou 64 bits : ce sont les types `int8`, `int16`, `int32`, `int64`. Observons le calcul suivant :

```
In [1]: x = np.int8(127)
In [2]: y = np.int8(1)
In [3]: x + y
RuntimeWarning: overflow encountered in byte_scalars
Out[3]: -128
```

Là encore, Python détecte un débordement, mais fournit néanmoins un résultat : *les entiers relatifs codés sur n bits restent dans l'intervalle* $[\![2^{n-1}, 2^{n-1} - 1]\!]$ *et les calculs sont effectués modulo* 2^n.

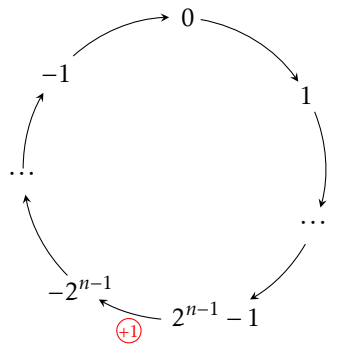


FIGURE 4 – Représentation machine des entiers relatifs codés sur n bits.

L'intérêt majeur de la représentation en complément à 2 est que *l'addition des représentations coïncide avec l'addition des entiers relatifs*. Par exemple, observons l'addition sur 8 bits des entiers 105 (représenté par l'octet `01101001`) et -25 (représenté par l'octet `11100111`) :

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

Puisque le résultat est stocké sur 8 bits, le bit de poids fort est ignoré est le résultat est l'entier codé en complément à 2 par `01010000` ; le premier bit est un zéro donc il s'agit d'un entier positif, précisément $(1010000)_2 = 80$, et on a bien $105 + (-25) = 80$.

Exercice 2. Sur le même modèle, réaliser l'addition en complément à 2 sur 8 bits des entiers :

a) $a = 48$ et $b = -100$;

b) $a = 95$ et $b = 42$.

Remarque. La notion de dépassement de capacité des entiers signés est la cause de l'explosion en vol du premier essai du lanceur spatial Ariane 5. En effet, ce dernier utilisait un capteur de mesure de l'inclinaison qui stockait la valeur de cette dernière sur 22 bits. Mais ce capteur initialement conçu pour Ariane 4 était insuffisant pour Ariane 5, un lanceur cinq fois plus puissant. Ainsi, lorsqu'après quelques secondes le fusée s'incline un peu vers la gauche, *la valeur déborde* et l'ordinateur de bord interprète cela comme une inclinaison *à droite*. Dès lors la correction de trajectoire amplifie le problème, et la dislocation de la fusée devient inéluctable…

Entiers de taille arbitraire

Notons pour conclure que le type `int`, natif dans Python, est un type un peu plus complexe : il utilise la représentation par complément à 2 sur 64 bits lorsque c'est possible (pour des raisons d'efficacité), mais lors d'un débordement de capacité la représentation des entiers change pour adopter une représentation non limitée en taille. Cela présente bien évidemment des avantages : on peut manipuler des entiers arbitrairement longs, mais aussi des inconvénients : les opérations sur ces grands entiers longs prennent plus de temps que sur les entiers classiques. Nous n'en dirons pas plus sur ce type de représentation.

3. Représentation machine des nombres décimaux

3.1 Les nombres flottants

Rappelons qu'un nombre décimal est un rationnel qui peut s'écrire sous la forme $\frac{x}{10^n}$ où x est un entier relatif. Si on considère la représentation en base 10 de l'entier $x = \pm(a_p a_{p-1} \cdots a_1 a_0)_{10}$, on obtient l'écriture décimale de $\frac{x}{10^n}$ en plaçant une virgule séparant les n chiffres les plus à droite des autres : $\frac{x}{10^n} = \pm(a_p \cdots a_n \,,\, a_{n-1} \cdots a_0)_{10}$.
Par exemple, $\frac{25}{4}$ est un nombre décimal car il peut aussi s'écrire $\frac{625}{100}$, et son écriture décimale est $6,25$.

De la même façon, un nombre *dyadique* est un rationnel qui peut s'écrire sous la forme $\frac{x}{2^n}$ où x est un entier relatif, et son développement dyadique s'obtient en plaçant une virgule séparant les n chiffres les plus à droite des autres.
Par exemple, $\frac{25}{4}$ est un nombre dyadique et son développement dyadique est $(110,01)_2$. En effet,

$$25 = (11001)_2 \quad \text{et} \quad \frac{25}{4} = (110,01)_2 = 1\times 2^2 + 1\times 2^1 + 0\times 2^0 + 0\times\frac{1}{2^1} + 1\times\frac{1}{2^2} = 6,25.$$

On s'en doute, les ordinateurs ne manipulent pas des nombres décimaux mais des nombres dyadiques. Ce n'est pas un problème quand il s'agit d'entiers puisque dans ce cas la conversion binaire/décimal est exacte, mais cela pose un problème pour les nombres décimaux car *le développement dyadique d'un nombre décimal peut être infini* [2].
Par exemple, le nombre $0,1$ a une représentation décimale finie et une représentation dyadique infinie :

$$0,1 = (0,0001100110011001100110011001100110011001100110011\cdots)_2$$

Ainsi, sa représentation machine sera nécessairement tronquée et ne correspondra pas exactement au nombre $0,1$ (mais en sera néanmoins très proche).

2. En revanche, la conversion réciproque ne pose pas de problème, puisque tout nombre dyadique est aussi décimal.

La conversion d'un nombre décimal en nombre dyadique va donc souvent provoquer une *approximation* qui dans certains cas conduit à des résultats qui peuvent paraître étranges à un utilisateur non averti. Par exemple :

```
In [1]: 0.1 + 0.2
Out[1]: 0.30000000000000004

In [2]: (0.1 + 0.2) - 0.3
Out[2]: 5.551115123125783e-17
```

De ceci il faudra retenir qu'un calcul sur des nombres décimaux sera toujours entaché d'une certaine marge d'erreur dont il faudra tenir compte, avec une conséquence importante :

L'égalité entre nombres flottants n'a pour ainsi dire aucun sens.

```
In [3]: 0.1 + 0.2 == 0.3
Out[3]: False
```

Quand on utilise le type `float`, il est rarissime (hormis dans un but pédagogique) d'utiliser l'égalité de valeur `==` ; on fera toujours intervenir une marge d'erreur (absolue ou relative).

```
In [4]: abs(0.1 + 0.2 - 0.3) <= 1e-10
Out[4]: True
```

(On a choisi ici une marge d'erreur absolue en considérant que toute quantité inférieure ou égale à 10^{-10} est nulle.)
Sur la figure 5 sont représentées les trois étapes d'une opération arithmétique $f(x,y)$ entre deux nombres décimaux x et y :

- la conversion de x et de y en nombres dyadiques $\tilde{x}, \tilde{y}$;
- le calcul approché de $\widetilde{f(\tilde{x},\tilde{y})}$;
- la conversion du résultat en nombre décimal.

Les deux premières étapes s'accompagnent d'une marge d'erreur dont il faut avoir conscience.

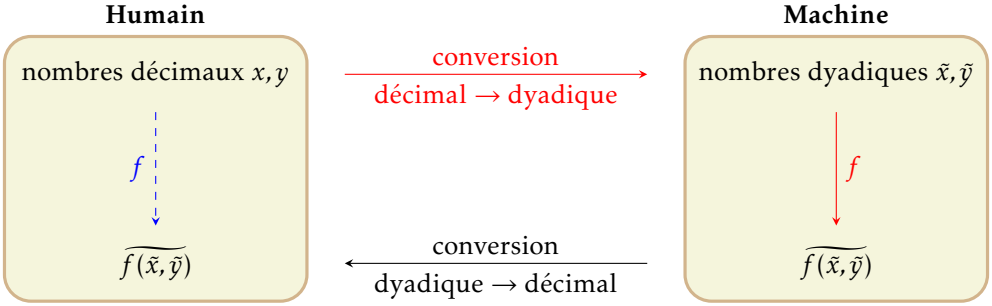


FIGURE 5 – Réalisation d'une opération arithmétique $f(x,y)$ entre flottants.

La norme IEEE-754

Cette norme est actuellement le standard pour la représentation des nombres à virgule flottante en binaire. Nous allons en donner une description incomplète.

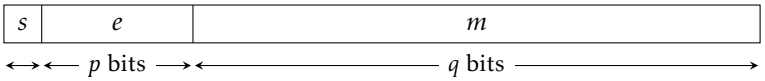
Un nombre dyadique non nul possède une représentation normalisée de la forme $\pm(1, b_1 \cdots b_k)_2 \times 2^e$, où e est un entier relatif. Par exemple, $6,25 = (110,01)_2$ a pour représentation normalisée $(1,1001)_2 \times 2^2$ et $-0,375 = -(0,011)_2$ la représentation normalisée $-(1,1)_2 \times 2^{-2}$.
La suite de bits $b_1 \cdots b_k$ est appelée la *mantisse* du nombre, et e, la puissance de 2, l'*exposant*.

Les nombres dyadiques sont représentés sur n bits en réservant :

- 1 bit pour représenter le signe ;
- p bits pour représenter l'exposant ;

– q bits pour représenter la mantisse

avec $1+p+q=n$.



Remarque. L'exposant est un entier relatif, mais pour permettre une comparaison plus aisée des nombres flottants, il n'est pas codé suivant la technique de complément à deux mais suivant la technique du décalage : l'exposant e est représenté en machine par l'entier naturel $e' = e + 2^{p-1} - 1$. Sachant que $0 \leqslant e' \leqslant 2^p - 1$, on a $-(2^{p-1}-1) \leqslant e \leqslant 2^{p-1}$.
Dans la réalité c'est même un peu plus compliqué. Les valeurs extrêmes $e' = 0$ et $e' = 2^p - 1$ sont dévolues à la représentation machine de valeurs particulières telles 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.

```
In [1]: 2e308          # dépasse le plus grand nombre représentable
Out[1]: inf

In [2]: sqrt(-1)       # calcul invalide
Out[2]: nan
```

La bibliothèque Numpy possède trois types de nombres flottants : `float16`, `float32` et `float64` qui représentent des nombres flottants sur respectivement 16, 32 et 64 bits, avec la répartition suivante :

– pour le type `float16` : 1 bit pour le signe, 5 bits pour l'exposant, 10 bits pour la mantisse ;
– pour le type `float32` : 1 bit pour le signe, 8 bits pour l'exposant, 23 bits pour la mantisse ;
– pour le type `float64` : 1 bit pour le signe, 11 bits pour l'exposant, 52 bits pour la mantisse.

Par exemple, dans le type `float16` le plus grand nombre exactement représentable (abstraction faite des valeurs particulières évoquées précédemment) correspond à la suite de bits (0|11110|1111111111) et vaut

$$+(1,1111111111)_2 \times 2^{15} = (1111111111100000)_2 = 65\,504.$$

Le nombre qui le précède correspond à la suite de bits (0|11110|1111111110) et vaut

$$+(1,1111111110)_2 \times 2^{15} = (1111111111000000)_2 = 65\,472.$$

L'approximation se faisant au flottant représentable le plus proche, tous les réels de l'intervalle $[65\,472, 65\,488]$ ont la même représentation en `float16`, ainsi que les réels de l'intervalle $]65\,488, 65\,504]$. Illustration :

```
In [1]: np.float16(65472) == np.float16(65488)
Out[1]: True

In [2]: np.float16(65488.0000001) == np.float16(65504)
Out[2]: True

In [3]: np.float16(65488) == np.float16(65488.0000001)
Out[3]: False
```

Exercice 3. Dans le type `float16`, déterminer le plus petit nombre x strictement supérieur à 1 qui soit représentable de manière exacte.
La quantité $\epsilon = x - 1$ est appelé l'*epsilon numérique*. Cette quantité représente la meilleure erreur relative qu'on peut espérer obtenir lors d'un calcul numérique (dans la pratique l'erreur relative est souvent plus élevée). Quelle est la valeur de l'epsilon numérique pour les types `float32` et `float64` ?

3.2 Calculs sur les nombres flottants

La représentation interne des nombres flottants a une incidence sur les opérations que l'on est susceptible de demander à l'ordinateur, et provoque quelques désagréments dont il faut avoir conscience. Sans prétendre à l'exhaustivité, nous allons passer en revue quelques-unes des conséquences de cette représentation.

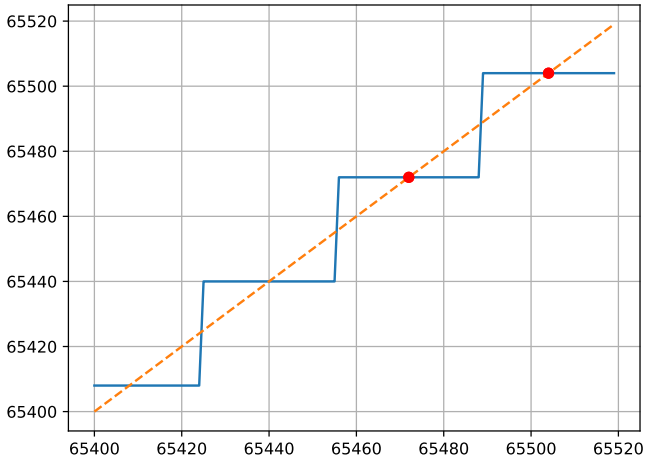


FIGURE 6 – Représentation des réels dans le type `float16`. Les deux points indiqués correspondent aux deux plus grandes valeurs représentables de manière exacte.

• Erreurs d'arrondis dues aux changement de bases

Nous l'avons déjà constaté, la conversion d'un nombre décimal en nombre dyadique et réciproquement provoque des erreurs d'arrondis qui ne sont pas sans conséquences :

```
In [5]: (0.1 + 0.2) - 0.3
Out[5]: 5.551115123125783e-17
```

Aucun des trois nombres 0,1, 0,2 et 0,3 n'est un nombre dyadique, donc leurs représentations machine n'est qu'approximative.
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 (le fameux « effet papillon » dans l'étude de phénomènes chaotiques).

• Cancellation

L'erreur d'élimination (*cancellation* en anglais) se produit lors de la soustraction de deux quantités très proches. Considérons par exemple la soustraction de 1 à $(1+3^{-33})$. Définissons la valeur x calculée par Python lors du calcul de cette différence, la valeur exacte $y = 3^{-33}$ de cette différence, et évaluons l'erreur relative commise lorsqu'on approche y par x :

```
In [1]: x = (1 + 3**-33) - 1
In [2]: y = 3**-33
In [3]: (x - y) / y
Out[3]: 0.23435940725514182
```

L'erreur relative est de l'ordre de 23%, ce qui est énorme. *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 $\dfrac{1}{x(x+1)}$ plutôt que $\dfrac{1}{x} - \dfrac{1}{x+1}$ car, bien que ces deux quantités soient mathématiquement égales, la seconde est beaucoup plus sensible au phénomène de cancellation.

• Absorption

L'erreur d'absorption se produit lorsqu'on additionne ou soustrait deux nombres x et y d'ordres de grandeurs très différents : lorsque $x \ll y$ alors $x+y \approx y$ et on « perd » partiellement ou totalement x.
En effet, $x+y = y\left(1+\dfrac{x}{y}\right)$ donc lorsque $\dfrac{x}{y}$ est inférieur à l'epsilon numérique, x est totalement absorbé par y :

```
In [1]: x = 2**-53
In [2]: y = 1.
In [3]: x + y == y
Out[3]: True
```

Les conséquences d'une erreur d'absorption peuvent être importantes lorsque :

- elle est suivie d'une erreur d'élimination : si $x \ll y$ alors $(x+y)-y$ peut être très différent de x ;
- on cherche à évaluer une somme de nombreux termes. Dans ce cas, il faut réorganiser le calcul de manière à sommer les termes du plus petit jusqu'au plus grand, pour éviter que les premiers soient absorbés par les seconds.

Et pour finir...

Il ne faudrait surtout pas croire que ces petites erreurs de calcul et d'arrondi soient négligeables, et l'anecdote suivante devrait vous convaincre de l'importance qu'il y a à en prendre conscience.

Le 25 février 1991, à Dharan en Arabie Saoudite, un missile Patriot américain a raté l'interception d'un missile Scud irakien, ce dernier provoquant la mort de 28 personnes. La commission d'enquête chargée de comprendre la raison de cet échec a mis en évidence le défaut suivant.
L'horloge interne du missile Patriot mesure le temps en 1/10s. Pour obtenir le temps en seconde, le système multiplie ce nombre par 10 en utilisant un registre de 24 bits en virgule fixe. Or 1/10 n'est pas un nombre dyadique donc a été arrondi : le registre de 24 bits contient $(0,00011001100110011001100)_2$ et induit une erreur binaire de $(0,00000000000000000000011001100\ldots)_2$, soit approximativement $0,000000095$ en notation décimale.
En multipliant cette quantité par le nombre de 1/10s pendant 100h (le temps écoulé entre la mise en marche du système et le lancement du missile Patriot), on obtient le décalage entre l'horloge interne de missile et le temps réel, soit :

$$0,000000095 \times 100 \times 3600 \times 10 \approx 0,34\text{s}.$$

Or un missile Scud vole à la vitesse approximative de $1,676$m/s donc parcourt plus de 500m en $0,34$s, ce qui le fait largement sortir de la zone d'acquisition de sa cible par le missile d'interception [3].

4. Exercices

Exercice 4 On appelle *espace binaire* d'un entier naturel n toute séquence consécutive de `0` délimités par deux `1` dans la décomposition en base 2 de n. Par exemple, le nombre 529 possède deux espaces binaires de longueurs respectives 3 et 4 car $529 = (1000010001)_2$. En revanche, 32 ne possède pas d'espace binaire puisque $32 = (100000)_2$.
Rédiger une fonction `espacemax` qui prend pour argument un entier naturel et renvoie la longueur du plus grand espace binaire présent dans n s'il en existe, et la valeur 0 sinon.

Exercice 5 Que représente la suite de 16 bits (2 octets) suivante : `1010101000000000` lorsqu'elle est interprétée comme :

a) un entier naturel de type `uint16` ?

b) un entier relatif de type `int16` ?

c) un nombre flottant de type `float16` ?

Exercice 6 Rédiger une fonction `addition` qui prend pour arguments deux listes *de mêmes longueurs* contenant les représentations binaires de deux entiers positifs x et y et qui renvoie :

- la liste binaire représentant l'entier $x+y$ si elle a même longueur que les deux premières listes ;
- la valeur `None` sinon.

Pour ce faire, on appliquera l'algorithme d'addition appris à l'école primaire, mais en l'exécutant en base 2.

3. Référence : http://ta.twi.tudelft.nl/nw/users/vuik/wi211/disasters.html.