5 Représentation des réels

Toute information étant codée en impulsions électriques (c'est-à-dire dans un système à 2 états : impulsion ou absence d'impulsion), le codage naturel des données doit se faire en binaire. Nous faisons dans ce chapitre quelques rappels sur les bases de numération, puis nous indiquons plus précisément les normes de codage des entiers et des réels.

I Bases de numération

Soit b un entier supérieur ou égal à 2.

Théorème 5.1.1 (existence et unicité de la représentation en base b)

Soit $N \in \mathbb{N}$. Il existe une unique suite $(x_n)_{n\in\mathbb{N}^*}$ d'entiers de $[\![0, b-1]\!]$, tous nuls à partir d'un certain rang, tels que :

$$N = \sum_{n=0}^{+\infty} x_n b^n.$$

◁ **Éléments de preuve.**

L'application $(x_0, \dots, x_{n-1}) \mapsto \sum_{n=0}^{+\infty} x_n b^n$ est injective de $[\![0, b-1]\!]^n$ dans $[\![0, b^n - 1]\!]$ (sinon, isoler la plus petite différence entre x_n et y_n et obtenir une contradiction par un argument de divisibilité). Par cardinalité, c'est une bijection. ▷

Définition 5.1.2 (représentation d'un entier dans une base)

On dit que l'écriture de N sous la forme de la somme

$$N = \sum_{n=0}^{+\infty} x_n b^n,$$

où les x_n sont nuls à partir d'un certain rang, et vérifient $x_n \in [\![0, b-1]\!]$, est la représentation en base b de N. Si k est le plus grand entier tel que $x_k \neq 0$, on écrira cette représentation sous la forme synthétique suivante :

$$N = \overline{x_k x_{k-1} \dots x_1 x_0}^b.$$

Il convient de bien distinguer l'entier N de sa représentation sous forme de caractères. Il en est de même des x_i, qui dans la somme représente des valeurs (des entiers), alors que dans la notation introduite, ils

représentent des caractères (les chiffres de la base de numération). De fait, lorsque $b \leqslant 10$, on utilise pour ces caractères les chiffres de 0 à $b-1$, tandis que si $b > 10$, on introduit au-delà du chiffre 9 des caractères spécifiques (souvent les premières lettres de l'alphabet). Ainsi, usuellement, les chiffres utilisés pour la représentation des entiers en base 16 (hexadécimal), très utilisée par les informaticiens, sont les 10 chiffres numériques de 0 à 9, et les 6 premières lettres de l'alphabet, de A à F.
Il faut bien comprendre la notation sous forme d'une juxtaposition de caractères (il ne s'agit pas du produit des x_i). Par exemple, $\overline{1443}^5$ représente l'entier

$$N = 3 \times 5^0 + 4 \times 5^1 + 4 \times 5^2 + 1 \times 5^3 = 248.$$

Une même succession de caractères représente en général des entiers différents si la base b est différente. Ainsi $\overline{1443}^6$ représente l'entier

$$N = 3 + 4 \times 6 + 4 \times 6^2 + 1 \times 6^3 = 387.$$

En revanche, la notation $\overline{1443}^4$ n'a pas de sens, puisque seuls les caractères 0, 1, 2 et 3 sont utilisés en base 4.

Notation 5.1.3

Une représentation d'un entier N sous forme de juxtaposition de chiffres sans mention de la base (et sans surlignage) désignera suivant le contexte la numération en base 10 (numération usuelle en mathématiques) ou en base 2 (dans un contexte informatique). On s'arrangera pour que le contexte lève toute ambiguïté.

On remarquera que le seul cas d'ambiguïté dans ce contexte est le cas où seuls les chiffres 0 et 1 sont utilisés.

Méthode 5.1.4 (Convertir d'une base b à la base 10)

Soit N représenté sous la forme $\overline{x_k x_{k-1} \dots x_0}^b$. On trouve l'entier N en effectuant le calcul :

$$N = \sum_{i=0}^{k} x_i b^i.$$

Pour ce calcul, consistant en l'évaluation d'un polynôme, on utilisera de préférence l'algorithme de Hörner, basé sur la factorisation suivante :

$$N = x_0 + b(x_1 + b(x_2 + \cdots + b(x_{k-1} + bx_k))).$$

Ainsi, initialisant à x_k, on trouve N en répétant l'opération consistant à multiplier par b et ajouter le chiffre précédent.
Ce calcul peut être effectué en base 10, en convertissant les chiffres x_i en leur analogue en base 10, ainsi que b, puis en utilisant les algorithmes usuels de calcul des sommes et produits en base 10 (poser les opérations).

Remarque 5.1.5

L'algorithme de Hörner est plus généralement un algorithme permettant d'évaluer tout polynôme en une valeur donnée a. Ici, la correspondance entre la valeur de N et sa représentation de N s'exprime sous forme d'un polynôme en b. L'algorithme de Hörner, consistant à mettre en facteur les termes b autant que possible, permet d'éviter certaines multiplications par b : on ne reprend pas le calcul des puissances de b depuis le début à chaque fois. Cela permet d'obtenir calcul nécessitant un nombre d'opérations linéaire en le nombre de chiffres (c'est-à-dire à peu près égal à une quantité αk, lorsque k

devient grand), alors qu'un algorithme naïf reprenant le calcul des puissances au début à chaque fois nécessite un nombre quadratique d'opérations (c'est-à-dire de l'ordre de αk^2. Lorsque k est de l'ordre de 1000, on gagne un facteur 1000 entre les deux temps de calcul, ce qui n'est pas négligeable !

Remarque 5.1.6

La méthode décrite ci-dessus permet en théorie de convertir tout entier d'une base b vers une base c. Les règles arithmétiques du calcul d'une somme et d'un produit se généralisent en effet en base quelconque. Dans la pratique, pour mener un tel calcul à la main, il vaut mieux connaître ses tables d'addition et de multiplication en base c (on peut dresser ces tables avant de commencer le calcul pour les avoir sous les yeux).

Exemple 5.1.7

Convertir $\overline{342}^5$ en base 6 et en base 2, sans passer par la base 10.

Cependant, le manque d'habitude qu'on a des calculs en base différente de 10 fait qu'en pratique, on préfère souvent faire une étape par la base 10, de sorte à n'avoir à utiliser des algorithmes de calculs que sur des numérations en base 10. Ainsi, on commence par convertir de la base b à la base 10, puis de la base 10 à la base c, en utilisant cette fois la méthode suivante, permettant de faire cette étape en n'utilisant que la numération en base 10.

Méthode 5.1.8 (Convertir de la base 10 à une base b)

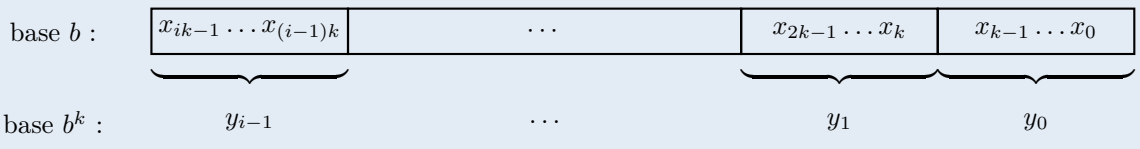
Soit N (dont on connait la représentation en base 10). Pour trouver la représentation de N en base b, on effectue les divisions euclidiennes de N, puis des quotients successifs, par b, jusqu'à obtenir un quotient nul. Les restes successifs sont les chiffres de N dans la numération en base b, dans l'ordre croissant des poids (le premier reste est le chiffre des unités).

Exemple 5.1.9

Convertir 2016 en base 7, en base 2.

Méthode 5.1.10 (cas particulier : convertir d'une base b à la base b^k)

Il suffit pour cela de regrouper k par k les chiffres de la représentation de N en base b, en commençant par les unités. Les nombres représentés par ces groupements de k chiffres sont dans $[\![0, b^k - 1]\!]$, et donnent les chiffres de la représentation de N en base b^k :



(dans cette représentation, on complète éventuellement la tranche la plus à gauche par des 0).
On peut également faire la démarche inverse sur le même principe.

Exemples 5.1.11

1. Donner la représentation hexadécimale de 2016, en utilisant la représentation binaire calculée ci-dessus.

2. Donner la représentation en base 2 de $\overline{af19b}^{16}$.

Ce dernier principe est très largement utilisé par les informaticiens, pour passer du binaire à l'hexadécimal et réciproquement.

II Codage des entiers

Dans un système où le nombre de bits destinés à la représentation d'un entier est constant (langages traditionnels), seule une tranche de l'ensemble des entiers peut être codée. Plus précisément, avec n bits, on peut coder une tranche de longueur 2^n, par exemple $[\![0, 2^n - 1]\!]$, ou $[\![-2^{n-1}, 2^{n-1} - 1]\!]$, ou toute autre tranche de la même longueur suivant la convention adoptée. Nous nous bornerons à l'étude de cette situation où le nombre n de bits est une constante, même si en Python, le langage que nous utiliserons par la suite, le nombre n de bits alloués à la représentation d'un entier varie suivant les besoins (les entiers sont aussi grand qu'on veut en Python).

Pour pouvoir considérer des entiers de signe quelconque, de façon la plus équilibrée possible, on cherche à donner un codage des entiers de l'intervalle $[\![-2^{n-1}, 2^{n-1} - 1]\!]$. On pourrait imaginer différents codages, ayant chacun leurs avantages, mais aussi leurs inconvénients ; par exemple :

- coder la valeur absolue sur 7 bits, par la numération en binaire, et attribuer la valeur 0 ou 1 au 8$^{\text{e}}$ bit suivant le signe ; il faudrait ensuite corriger bien artificiellement le doublon du codage de 0 et l'absence du codage de -2^{n-1}.
- Compter les nombres à partir de -2^{n-1}, à qui on attibue la valeur initiale 0 (autrement dit, considérer le code binaire de $x + 2^{n-1}$)

Ces deux codages sont simples à décrire et à calculer, mais s'avèrent ensuite assez peu commodes dans l'implémentation des calculs élémentaires.

Le codage adopté généralement est le codage appelé codage en complément à 2, correspondant grossièrement à une mise en cyclicité des entiers de -2^{n-1} à $2^{n-1} - 1$, donc à un codage modulo 2^n. On commence à énumérer les entiers depuis 0 jusqu'à $2^{n-1} - 1$, puis, les codages suivants en binaire vont correspondre aux entiers de -2^{n-1} à -1. Plus précisément :

Définition 5.2.1 (Codage en complément à 2)

Dans ce codage sur n bits des entiers de $[\![-2^{n-1}, 2^{n-1} - 1]\!]$:

- le code binaire d'un entier $k \in [\![0, 2^{n-1} - 1]\!]$ est sa représentation binaire
- le code binaire d'un entier $k \in [\![-2^{n-1}, -1]\!]$ est la représentation binaire de $k + 2^n = 2^n - |k|$.

Remarque 5.2.2

On distingue facilement par ce codage les entiers de signe positif (leur premier bit est 0) des entiers de signe négatif (leur premier bit est 1). On dit que le premier bit est le bit de signe.

L'intérêt de ce codage est la facilité d'implémentation des algorithmes usuels de calcul (somme, produit...). En effet, ce codage évite d'avoir à distinguer dans ces algorithmes différents cas suivant le signe des opérandes : l'algorithme usuel pour la somme (et les autres également), correspondant à poser l'addition, est valable aussi bien pour les entiers négatifs, en posant la somme de leur représentation binaire en complément à 2, à condition :

- d'oublier une éventuelle retenue qui porterait sur un $n + 1$-ième bit
- que le résultat théorique de l'opération soit bien compris dans l'intervalle $[\![-2^{n-1}, 2^{n-1} - 1]\!]$.

En effet, les nombres théoriques et leur représentation binaire sont congrus modulo 2^n. De plus, l'oubli des retenues portant sur les bits non représentés (au delà du n^{e}) n'affecte par le résultat modulo 2^n

du calcul. Ainsi, le résultat obtenu (sur le codage binaire, puis sur l'entier codé ainsi) est égal, modulo 2^n, au résultat théorique. Cette congruence est une égalité si on sait que le résultat théorique est dans $[\![-2^{n-1}, 2^{n-1}-1]\!]$, intervalle constitué de représentants de classes de congruences deux à deux distinctes.

Exemple 5.2.3

Pour un codage à 8 bits, illustrer ce qui précède pour le calcul de $-45+17$, de $-45+65$ et de $100+100$.

Certains langages gèrent les **dépassements de capacité** (le fait que le résultat sorte de la tranche initialement allouée) en attribuant des octets supplémentaires pour le codage du résultat (c'est le cas de Python), d'autres ne le font pas (comme Pascal par exemple). Dans ce cas, le résultat obtenu n'est correct que modulo 2^n. Ainsi, pour des entiers standard de type `integer` en Pascal, l'opération $32767+1$ renvoie le résultat -32768. Le programmeur doit être conscient de ce problème pour prendre les mesures qui s'imposent.

Pour terminer ce paragraphe, observons qu'il est possible d'obtenir très simplement le codage en complément à 2 d'un nombre $m < 0$, connaissant le codage de sa valeur absolue $|m|$. En effet, le codage de m est le développement binaire de $2^n - |m| = (2^n - 1) - |m| + 1$. Or, $2^n - 1$ est le nombre qui, se représente en binaire par n chiffres tous égaux à 1. Par conséquent, effectuer la soustraction $(2^n-1)-|m|$ est particulièrement simple : elle n'engendre aucune retenue, et consiste simplement à changer tous les 0 en 1 et réciproquement dans la représentation binaire de $|m|$. Par exemple :

$$\begin{array}{cccccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ - & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Il suffit ensuite de rajouter 1 au résultat obtenu, ce qui se fait, par reports successifs des retenues, en transformant le dernier bloc $011\ldots1$ en $100\ldots0$. Il convient ici de noter qu'un tel bloc existe toujours (éventuellement uniquement constitué du terme 0), car sinon, le nombre obtenu est $111\ldots1$, ce qui correspond à $|m| = 0$, ce qui contredit l'hypothèse $m < 0$.
On résume cela dans la méthode suivante :

Méthode 5.2.4 (Codage en complément à 2 de $-|m|$)

Pour trouver le codage en complément à 2 de $-|m|$, on échange le rôle des chiffres 0 et 1 dans la représentation binaire de $|m|$, puis on ajoute 1, ce qui revient à changer le dernier bloc $01\ldots1$ (éventuellement réduit à 0) en $10\ldots0$.
Réciproquement si un codage correspond à un nombre négatif (*i.e.* le bit de signe est 1), on trouve le codage de la valeur absolue en effectuant la démarche inverse, c'est-à-dire en remplaçant le dernier bloc $10\ldots0$ par $01\ldots1$, puis en échangeant les 0 et les 1.

On peut remarquer que cela revient aussi à changer les 0 en 1 et réciproquement, sauf dans le dernier bloc $10\ldots0$ (algorithme valable dans les deux sens)

III Développement en base b d'un réel

La notion de représentation en base b d'un entier se généralise aux réels, à condition d'accepter des sommes infinies de puissances négatives de b. Ces sommes infinies sont des sommes de séries à termes positifs, à considérer comme la limite de sommes finies. Nous obtenons :

Théorème 5.3.1 (Développement en base b d'un réel positif)

Soit $x \in \mathbb{R}_+^*$. Il existe $n \in \mathbb{Z}$ et des entiers $x_i \in [\![0, b-1]\!]$, pour tout $i \in]\!]-\infty, n]\!]$, tels que $x_n \neq 0$ et

$$x = \sum_{i=-\infty}^{n} x_i b^i = \lim_{N \to -\infty} \sum_{i=N}^{n} x_i b^i.$$

De plus :

- *si pour tout $m \in \mathbb{N}$, $b^m x \notin \mathbb{N}$, alors ce développement est unique (c'est-à-dire que n et les x_i sont uniques)*
- *s'il existe $m \in \mathbb{N}$ tel que $b^m x \in \mathbb{N}$, alors il existe exactement deux développements, l'un « terminant » (vers $-\infty$) par une succession infinie de 0, l'autre terminant par une infinité de $b-1$.*

◁ **Éléments de preuve.**

- Unicité : si ce n'est pas le cas, on a égalité de deux nombres se développant différemment. Si aucun des deux ne termine par des 9, en multipliant par b^n assez grand pour ramener une différence avant la virgule, on contredit l'unicité de la décomposition en base b de l'entier $\lfloor b^n x \rfloor$. Si l'un des développements termine par des 9, remplacer par un développement temrinant par des 0 et utiliser le même argument.
- Existence : Considérer les développements successifs de $\lfloor b^n x \rfloor$, montrer que les premiers chiffres sont toujours les mêmes, et que la série infinie obtenue par ce procédé converge bien vers x (on pourra encadrer la somme partielle).

▷

Notation 5.3.2 (Représentation du développement en base b)

Soit $x = \sum_{i=-\infty}^{n} x_i b^i$ un développement en base b de x. Si $n \geqslant 0$, on note :

$$x = \overline{x_n x_{n-1} \dots x_0 \ , \ x_{-1} x_{-2} \dots}^{b}.$$

Si $n < 0$, on pose $x_0 = x_{-1} = \dots = x_{n+1} = 0$, et on écrit

$$x = \overline{x_0, x_{-1} x_{-2} \dots}^{b} = \overline{0, 0 \dots 0 x_n x_{n-1} \dots}^{b}.$$

De plus, s'il existe $m \leqslant 0$ tel que pour tout $k < m$, $x_k = 0$, on utilisera une représentation bornée, en omettant l'ultime série de 0 :

$$x = \overline{x_n x_{n-1} \dots x_0, x_{-1} x_{-2} \dots x_m}^{b}.$$

Enfin, si $b = 10$ (ou $b = 2$ dans un contexte informatique sans ambiguïté), on omettra la barre et la référence à b dans la notation.

Terminologie 5.3.3 (développement décimal, développement dyadique)

On parlera de développement décimal plutôt que de développement en base 10. De même, on parlera de développement dyadique plutôt que de développement en base 2. On rencontre aussi parfois la notion de développement triadique.

Exemples 5.3.4

1. En base 10, on a l'égalité entre 1 et $0,999999\dots$
2. En base 3, on a l'égalité entre les réels $\overline{1,0210000000\dots}^{3}$ et $\overline{1,0202222222\dots}^{3}$

Terminologie 5.3.5 (développement propre)

On dira que le développement en base b est propre s'il ne termine pas par une série de chiffres $b-1$.

Ainsi, d'après ce qui précède, tout nombre réel admet une unique décomposition propre en base b.

Méthode 5.3.6 (Développement en base b d'un réel)

Soit $x \in \mathbb{R}_+^*$ (le cas $x = 0$ étant trivial, et le cas $x \in \mathbb{R}_-^*$ s'obtenant en rajoutant au développement de $|x|$ un signe $-$). Pour trouver un développement en base b de x, on peut décomposer le calcul en 2 étapes :

- trouver la représentation en base b de la partie entière $\lfloor x \rfloor$ de x, à l'aide des méthodes exposées précédemment. Cette représentation s'écrit sous la forme
$$\lfloor x \rfloor = \overline{x_n \dots x_0}^b;$$
- trouver le développement décimal de la partie décimale $\{x\}$ de x, par la méthode exposée ci-dessous. Ce développement s'écrit sous la forme
$$\{x\} = \overline{0, x_{-1}x_{-2}\dots}^b.$$

Le développement en base b de x est alors :
$$x = \overline{x_n \dots x_0, x_{-1}x_{-2}\dots}^b.$$

Pour trouver le développement propre en base b de $\{x\}$, on peut partir de la remarque évidente suivante : la multiplication par b consiste en le décalage à droite d'un cran de la virgule. Ainsi, x_{-1} est la partie entière de $b\{x\}$, puis reprenant la partie décimale de cette expression et la remultipliant par b, on obtient x_{-2} en prenant à nouveau la partie entière du résultat. On énonce :

Méthode 5.3.7 (Trouver le développement en base b d'un réel $x \in [0,1[$)

Soit $x \in [0,1[$. On a alors
$$x = \overline{0, x_{-1}x_{-2}\dots}^b,$$
où :
$$x_{-1} = \lfloor b\{x\} \rfloor, \qquad x_{-2} = \lfloor b\{b\{x\}\} \rfloor, \qquad x_{-3} = \lfloor b\{b\{b\{x\}\}\} \rfloor \qquad \text{etc.}$$
Ainsi, on trouve les chiffres successifs du développement en répétant l'opération consistant à multiplier la partie décimale obtenue précédemment par b et en en prenant la partie entière.

Exemples 5.3.8

1. Trouver le développement dyadique de $0,7$
2. De même pour $0,1$.
3. De même pour $3,84375$.

On peut toujours écrire un nombre réel x (en base 10) sous la forme $y \times 10^k$, où $y \in [1,10[$ et $k \in \mathbb{Z}$. Il s'agit de la notation scientifique usuelle. Ici, l'exposant k correspond au rang du premier chiffre non nul de x. De la même façon :

Proposition/Définition 5.3.9 (Notation scientifique en base b)

Soit $b \in \mathbb{N}$, $b \geqslant 2$. Soit $x \in \mathbb{R}_+^*$. Il existe un unique entier $k \in \mathbb{Z}$ et un unique réel $y \in [1,b[$ tel que $x = y \times b^k$.

Ainsi, il existe un unique $k \in \mathbb{Z}$, et d'uniques entiers y_i de $[\![0, b-1]\!]$, $i \in [\![-\infty, 0]\!]$, non tous égaux à $b-1$ à partir d'un certain rang, tels que

$$x = \overline{y_0, y_{-1}y_{-2}\dots}^{b} \times b^k \qquad \text{et} \qquad y_0 \neq 0.$$

Il s'agit de la notation scientifique en base b.

◁ **Éléments de preuve.**

k est la position du premier chiffre non nul dans la décomposition *propre* en base b (compté négativement s'il est après la virgule ; le chiffre le position 0 est celui juste avant la virgule). Si k est plus petit, $y \geqslant b$, et si k est plus grand, $y < 1$. ▷

Remarque 5.3.10

Si $b = 2$, le chiffre y_0 est nécessairement égal à 1. Le stockage d'un nombre réel sous ce format pourra se dispenser du bit correspondant à y_0, tout déterminé.

IV Codage des réels (norme IEEE 754)

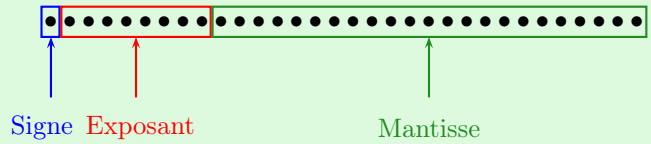
Le stockage des nombres réels ne peut pas se faire de façon exacte en général. En effet, même en ne se donnant pas de limite sur le nombre de bits, on sera limité par la capacité de la mémoire (finie, aussi grande soit-elle) de l'ordinateur. Or, entre deux réels distincts, aussi proches soient-ils, il existe une infinité non dénombrable de réels. La mémoire de l'ordinateur est incapable de décrire de façon exacte ce qui se trouve entre 2 réels distincts, quels qu'ils soient.
Ainsi, augmenter la capacité de stockage selon les besoins est vain, pour le stockage des réels. Si certains réels peuvent se stocker facilement sur un nombre fini de bits de façon cohérente, la plupart nécessiteraient une place infinie. Ainsi, par convention, on se fixe un nombre fini de bits, en étant conscient que ce qu'on codera sur ces bits ne sera qu'une valeur approchée du réel considéré. Il faudra alors faire attention, dans tous les calculs effectués, à la validité du résultat trouvé, en contrôlant, par des majorations, l'erreur dûe aux approximations de la représentation informatique des réels. On peut dans certains situations obtenir une divergence forte entre le calcul effectué par l'ordinateur et le calcul théorique. Nous en verrons un exemple frappant en TP lors de l'étude de l'algorithme d'Archimède pour le calcul de π.

Définition 5.4.1 (Représentation des réels sur 32 bits)

Le stockage des réels en norme IEEE 754 se fait sous la forme scientifique binaire :

$$x = \pm \overline{1, \underbrace{x_{-1}x_{-2}\dots}_{\text{mantisse}}}^{2} \times 2^k.$$

Sur les 32 bits, 1 bit est réservé au stockage du signe, 8 au stockage de l'exposant et 23 au stockage de la mantisse. Le 1 précédant la virgule n'a pas besoin d'être stocké (valeur imposée) :



- Le bit de signe est 0 si $x \geqslant 0$ et 1 si $x < 0$.
- L'exposant k peut être compris entre -126 et 127. Les 8 bits destinés à cet usage donnent la représentation binaire de $127 + k$, compris entre 1 et 254. Les valeurs binaires 00000000 et 11111111 sont interdites (reservées à d'autres usages, correspondant à des représentations non normalisées)

- La mantisse est représentée de façon approchée par son développement dyadique à 23 chiffres après la virgule (donc les x_i, $i < -1$).

IEEE = Institute of Electrical and Electronics Engineers

Exemples 5.4.2

- La représentation sur 32 bits de $0,1$ est :
$$0\,0\,1\,1\,1\,1\,0\,1\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0$$
- Donner la représentation sur 32 bits de $-5890,3$
- Quel est le nombre représenté par :
$$1\,1\,0\,0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,?$$

Remarque 5.4.3 (Valeurs extrémales, précision pour un codage sur 32 bits)

- La valeur maximale est alors $\overline{1.11111111\ldots}^2 \times 2^{127} \simeq 2^{128} \simeq 3,403 \times 10^{38}$.
- La valeur minimale est $2^{-126} \simeq 1,75 \times 10^{-38}$.
- La précision de la mantisse est de l'ordre de $2^{-23} \simeq 1,192 \times 10^{-7}$.

Ainsi, cette représentation nous donne des valeurs réelles avec environ 8 chiffres significatifs en notation décimale (en comptant le premier). Cette précision est souvent insuffisante, surtout après répercussion et amplification des erreurs. On utilise de plus en plus fréquemment la norme IEEE 754 avec un codage sur 64 bits, permettant une augmentation des extrêmes et de la précision. C'est le cas en Python, que nous utiliserons.

Définition 5.4.4 (Représentation des réels sur 64 bits)

Le stockage des réels en norme IEEE 754 sur 64 bits se fait sur le même principe que sur 32 bits avec les modifications suivantes : sur les 64 bits, 1 bit est réservé au stockage du signe, 11 au stockage de l'exposant et 52 au stockage de la mantisse.

- Le bit de signe est 0 si $x \geqslant 0$ et 1 si $x < 0$.
- L'exposant k peut être compris entre -1022 et 1023. Les 11 bits destinés à cet usage donnent la représentation binaire de $1023 + k$, compris entre 1 et 2046. Les valeurs binaires 00000000000 et 11111111111 sont interdites (reservées à d'autres usages, correspondant à des représentations non normalisées)
- La mantisse est représentée de façon approchée par son développement dyadique à 52 chiffres après la virgule.

Remarque 5.4.5 (Valeurs extrémales, précision pour un codage sur 64 bits)

- La valeur maximale est alors $\overline{1.11111111\ldots}^2 \times 2^{1023} \simeq 2^{1024} \simeq 1,797 \times 10^{308}$.
- La valeur minimale est $2^{-1022} \simeq 2,225 \times 10^{-308}$.
- La précision de la mantisse est de l'ordre de $2^{-52} \simeq 2,22 \times 10^{-16}$.

On obtient donc, avec le bit implicite, une précision correcte à 16 ou 17 chiffres significatifs. Cela explique que Python (par exemple) retourne les résultats réels en donnant 16 chiffres après la virgule.

Définition 5.4.6 (Nombres dénormalisés)

- La valeur 11111111111 de l'exposant permet le codage des nombres infinis (NaN, not a nomber). Nous n'en parlerons pas.
- La valeur 00000000000 de l'exposant permet de coder des nombres un peu plus petits que la borne trouvée ci-dessous, mais en diminuant la précision (nombres normalisés). Dans ce cas, le bit implicite n'est plus considéré égal à 1, mais à 0. Un codage :

  1 00000000000 0001000000000000000000000000000000000000000000000000

  correspond par exemple au nombre $-2^{-1022} \times \overline{0,0001}^2 = -2^{1026}$.

Le plus petit nombre qu'on puisse alors obtenir est $2^{-1022-52} = 2^{-1074}$. Python renvoie `5.e-324` pour cette valeur (on a perdu toute la précision derrière la virgule). Le calcul de 2^{-1075} renvoie 0.

V Problèmes liés à la représentation non exacte des réels

Il faut faire attention essentiellement à trois problèmes :

1. **Problèmes d'arrondi**

   La plupart des nombres n'étant pas représentés de façon exacte, les approximations faites peuvent s'ajouter dans les calculs. Le caractère approché des résultats des calculs peut se constater sur un exemple très simple :

   ```
   >>> 0.3-0.2-0.1
   -2.7755575615628914e-17
   ```

   Ces petites erreurs s'additionnent, et peuvent devenir grandes si on effectue un grand nombre de calculs :

   ```
   >>> S = 0
   >>> for i in range(1000):
   ...     S += 0.1
   ...
   >>> S
   99.9999999999986
   >>> S-100
   -1.4068746168049984e-12
   ```

2. **Problème d'absorption**

   Ce problème se produit lors de la somme de deux réels n'ayant pas du tout le même ordre de grandeur. Si le plus petit des deux a un ordre de grandeur inférieur à la précision du plus grand (donc de l'ordre de grandeur du dernier chiffre significatif), le résultat de la somme admet la même représentation que la valeur la plus grande. On dit que la petite valeur a été absorbée par la grande. Cette situation se produit dès lors que $\frac{x}{y} > 2^{52}$. Par exemple, comparons les 2 calculs suivants :

   ```
   >>> 2**100 + 1 - 2**100
   1
   >>> 2**100 + 1. - 2**100
   0.0
   ```

   La première ligne de calcul se fait avec des entiers (aussi grands qu'on veut en Python). Il n'y a dans ce cas pas de problème. La seconde ligne représente le même calcul avec des réels (le point après le 1 force le calcul à se faire en type réel). Le 1 a été absorbé.

   Nous illustrons le fait que le rapport de grandeurs limite est 2^{52} en remarquant que 1 est absorbé face à 2^{53}, mais pas 2 :

```
>>> 2**53 + 1. - 2**53
0.0
>>> 2**53 + 2. - 2**53
2.0
```

On peut alors imaginer des algorithmes finissant (rapidement) en théorie, mais ne s'arrêtant pas en pratique à cause de ce problème. On en donne d'abord une version avec des entiers (tapé en ligne de commande) :

```
>>> S=0
>>> i=2**53
>>> while i < 2**53 + 2:
...     i += 1
...     S += 1
...
>>> S
2
```

Le comportement est celui qu'on attend. Voici la version réelle du même algorithme :

```
>>> S=0
>>> i=2**53
>>> while i < 2**53 + 2:
...     i += 1.
...     S += 1
...
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
>>> S
30830683
```

On est obligé d'interrompre le calcul avec `Ctrl-C`. En demandant ensuite la valeur de S, on se rend compte qu'on est passé un grand nombre de fois dans la boucle. Le fait de travailler en réels a comme conséquence que la valeur de i considérée est toujours la même, car le 1 qu'on lui ajoute est absorbé.

3. **Problème de cancellation**

   Il s'agit du problème inverse. Lorsqu'on effectue la différence de deux nombres de même ordre de grandeur, il se peut que de nombreux bits de la représentation se compensent. Par exemple, si seuls les 2 derniers bits diffèrent, le premier bit du résultat sera de l'ordre de grandeur de l'avant-dernier bit des deux arguments, le second bit significatif sera de l'ordre de grandeur du dernier bit des arguments, et les suivants n'ont pas de signification, car provenant de résidus de calculs approchés. Leur valeur exacte nécessiterait de connaître les chiffres suivants dans le développement des deux arguments. Nous illustrons ceci sur l'exemple suivant :

```
>>> def f(x):
...     return (x+1)**2-x**2-2*x-1
...
>>> f(1000000000)
0
>>> f(1000000000.)
-1.0
>>> f(10000000000)
```

```
0
>>> f(10000000000.)
-2049.0
>>> f(100000000000)
0
>>> f(100000000000.)
-905217.0
>>>
```

Dans cet exemple, la fonction f est identiquement nulle. Le calcul est correct s'il est effectué avec des entiers, mais plus du tout lorsqu'on passe aux réels.

4. **Comparaisons par égalité**

   Aux trois problèmes précédents, nous ajoutons le quatrième, qui est en fait dérivé du premier (problème de l'inexactitude de la représentation des réels). Comparer par une égalité 2 réels, surtout s'ils sont issus de calculs, n'a pas beaucoup de pertinence, car même s'ils sont égaux en théorie, il est probable que cela ne le soit pas en pratique. Ainsi, en reprenant un exemple vu plus haut, on obtient le résultat suivant, assez surprenant, pour un test très simple :

   ```
   >>> 0.3==0.2+0.1
   False
   ```

   On y remédie souvent en s'autorisant une petite inexactitude, ce qui oblige à remplacer l'égalité par un encadrement :

   ```
   >>> abs(0.3-0.2-0.1)<1e-15
   True
   ```