Chap. 6 : Représentations des entiers et réels en informatique

Table des matières

1 Des maths : l'écriture des entiers naturels en base dix, deux, seize,..

1.1 Les entiers naturels qu'on connaît

Nous avons tous une idée *intuitive* de la notion de nombre entier naturel. En mathématiques, on montre que *toutes* les propriétés des entiers viennent des trois axiomes suivants :

> Il existe un ensemble *essentiellement unique* noté $\mathbb{N}$, muni d'une application $s : \mathbb{N} \to \mathbb{N}$ avec les propriétés :
> (i) s est injective,
> (ii) il existe dans $\mathbb{N}$ un élément noté 0 tel que $0 \notin s(\mathbb{N})$,
> (iii) pour tout sous-ensemble A de $\mathbb{N}$, si A contient 0 et est stable par s, alors $A = \mathbb{N}$.

N.B. Ces axiomes disent *tout* sur $\mathbb{N}$. Si on les cite ici, c'est pour signifier que la *nature* des nombres entiers est purement liée à cette notion de *succession* et pas à la *façon de les écrire*. Ils disent aussi que *le raisonnement par récurrence contient la clef de* toutes *les propriétés des entiers....*

Nous avons *l'habitude* de manipuler les entiers *en base dix*, dire $n = 340587$ signifie pour nous $n = 3.10^5 + 4.10^4 + 0.10^3 + 5.10^2 + 8.10^1 + 7.10^0$.

D'une manière générale, dire que *l'écriture décimale* d'un nombre entier naturel est de la forme :

$$n = a_r a_{r-1} \dots a_1 a_0, \text{ avec }, \forall\, i \in [\![0, r]\!], a_i \in [\![0, 9]\!],$$

signifie en fait que :

$$n = a_r 10^r + a_{r-1}.10^{r-1} + \cdots + a_1.10 + a_0.$$

Le « *l'* » devant le mot *écriture* est justifié par le fait que cette écriture est *unique* pour chaque nombre entier, à condition de demander que si $n \neq 0$, $a_r \neq 0$. Pour $n = 0$, on choisit, bien sûr, en maths, comme unique écriture, l'écriture $a_0 = 0$.

Cette habitude n'est qu'une *convention*, qui a correspondu au *choix* de dix chiffres pour coder tous les nombres.

1.2 L'écriture en base deux

Aujourd'hui, avec les ordinateurs, une autre écriture est devenue incontournable : l'écriture en base deux.

Une telle écriture va coder une décomposition du nombre entier n sous la forme :

$$n = b_r.2^r + b_{r-1}2^{r-1} + \cdots + b_1.2 + b_0, \text{ avec } \forall\, i \in [\![0, r]\!], b_i \in \{0, 1\} \quad (\dagger)$$

Pour un exemple concret : voici $n_0 = 2^4 + 2^2 + 2 + 2^0$

Pour signifier qu'on a l'écriture $(\dagger)$ ci-dessous, on peut noter :

$$n = b_r b_{r-1} \dots b_{0[2]},$$

et dans l'exemple concret $n_0 = 10111_{[2]}$.

Moins rigoureusement peut-être, on dira aussi simplement $n = b_r b_{r-1} \dots b_0$ *en base deux*!

Bien sûr, dans notre exemple $n_0 = 16 + 4 + 2 + 1 = 23$ en base dix. Donc on écrira $23 = 10111_{[2]}$ et encore, de façon moins rigoureuse : $23 = 10111$ en base deux.

Théorème 1. Toute nombre entier naturel admet une unique écriture de la forme :

$$n = b_r.2^r + b_{r-1}2^{r-1} + \cdots + b_1.2 + b_0, \text{ avec } \forall\, i \in [\![0, r]\!], b_i \in \{0, 1\}$$

à condition de spécifier que pour $n = 0$ l'écriture est donnée par $r = 0$, $b_0 = 0$ et pour $n \neq 0$, $b_r \neq 0$.

Avec ces conventions, tout entier naturel non nul admet donc une unique écriture en base deux :

$$n = b_r b_{r-1} \cdots b_1 b_{0[2]},$$

avec $b_r \neq 0$.

L'unicité dans le théorème précédent permet la :

Définition. Avec les notations du théorème précédent, le nombre $l = r + 1$ sera appelé ici *longueur de l'écriture en base deux de n*. Par exemple, cette longueur vaut 1 si, et seulement si, $n = 0$ ou $n = 1$.

Exercice 1.

a) Donner l'écriture en base deux de tous les entiers de 1 à 12. A partir de maintenant vous savez compter en base deux. Cette écriture des entiers vérifie tout aussi bien les axiomes de Peano.

b) Quel est le plus grand entier que l'on peut écrire avec 8 chiffres en base deux ? Même question en remplaçant 8 par un entier l quelconque ?

c) Démontrer si on note $l = r + 1$ la longueur de l'écriture d'un entier n strictement positif en base deux vérifie alors $r = \lfloor \log_2(n) \rfloor$, où $\lfloor\ \rfloor$ désigne la *partie entière*.

1.3 Comment obtenir l'écriture en base deux d'un nombre écrit en base dix ?

(M1) Des poids forts vers les poids faibles : Mettons qu'on veuille savoir comment s'écrit 54 en base deux. On commence par se demander entre quelles puissances de deux il est : ici 32 et 64. On sait alors quel est le premier chiffre (en partant de la gauche, celui qu'on appelle le poids fort) de son écriture en base deux : comme $32 = 2^5$, $54 = 2^5 + (54 - 32) = 2^5 + 22$. On fait le même travail pour 22.

Exercice 2. Finir ce travail, pour obtenir l'écriture en base deux de 54, puis faire de même pour 361.

(M2) Des poids faibles vers les poids forts : L'idée de départ est que si $n = b_r 2^r + \cdots + b_1 2 + b_0$ alors en particulier $n = 2B + b_0$ où B est un entier et $b_0 \in \{0, 1\}$. Donc b_0 est le *reste* de la division euclidienne de n par 2. Donc on sait calculer b_0, le dernier chiffre de l'écriture de n en base 2 (le poids le plus faible). On a bien sûr $b_0 =$ si n est pair et $b_0 =$ si n est impair. Pour 54 ce chiffre fait donc .

On peut ensuite appliquer la même méthode au quotient B de cette division euclidienne. A partir de l'écriture de départ, $B = b_r 2^{r-1} + \cdots b_2.2 + b_1$ et obtenir b_1.

Exercice 3. Appliquer cette méthode pour obtenir l'écriture en base deux de 54, puis de 361.

On comparera et implémentera ces deux méthodes en T.P.

1.4 Culture : la base 16 ou hexadécimal

L'écriture en base 16 utilise traditionnellement les dix chiffres 0,1,. . .,9 suivis par les six premières lettres de l'alphabet A,B,C,D,E,F. Ainsi la lettre A correspond au 10 de la base 10, et F correspond à 15 en base dix.

Pour comprendre un nombre écrit en hexadécimal, on utilise une écriture analogue à celle vue au 1.1, avec les puissances de 16, en remplaçant les lettres par leur valeur décimale : par exemple si n=3E5D en hexadécimal, on a :

$$n = 3 \times 16^3 + 14 \times 16^2 + 5 \times 16 + 13.$$

Exercice 4.
a) A quelle âge, en hexadécimal, peut-on passer le permis de conduire en France, si on n'a pas fait la conduite accompagnée ?

b) En informatique, le système RGB (Red Green Blue) définit une couleur comme la donnée de trois nombres chacun entre 0 et 255, définissant pour le premier le niveau de Rouge, le second le niveau de vert, le troisième le niveau de bleu. (i) Quelle couleur donne le code html #FF0000 ? (ii) Plus dur : même question pour #FFFF00 ?

Notation en PYTHON(mais aussi C, Java...) La notation `0x27` signifie qu'on considère le nombre qui admet l'écriture 27 en hexadécimal. La commande `hex` convertit de décimal en hexa.

Exercice 5. Expliquer comment obtenir l'écriture binaire d'un nombre à partir de son écriture hexadécimale. Le faire sur l'exemple de $n = 3E5D$.

2 Les entiers à l'intérieur d'un ordinateur... et suivant les programmes

En informatique, les variables ont un *type* qui désigne leur nature : ici, nous allons d'abord parler du type entier `integer` [1].

1. Ce qui se prononce 'intedjer et pas integeur comme on entend souvent en France.

2.1 Le bit, l'octet

Le bit est l'unité d'information : il vaut 0 ou 1. Au début de l'informatique, l'information était transmise par paquets de 8 bits : on appelle *octet* un mot de 8 bits. Avec des octets, on peut donc écrire $2^8 = 256$ suites de 0 et de 1, différentes, comme par exemple `01001011`.

2.2 Le codage des entiers aujourd'hui : jusqu'à 64 bits

Les ordinateurs personnels les plus récents ont des microprocesseurs 64 bits, ce qui signifie que l'information arrive par paquets de 64 bits.

En PYTHON 2, mais aussi dans beaucoup de langages, les entiers standard (type `int`, en PYTHON) sont codés sur 64 bits, c'est-à-dire encore 8 octets. (Si vous avez une machine 32 bits, ce sera sur 32 bits. De même si vous avez une machine 64 bits, mais un système d'exploitation gérant 32 bits).

Cependant quand, on parle d'entiers en informatique, il s'agit d'entiers avec signe, qui peuvent être négatifs, on va garder un bit (par convention le premier) pour coder le *signe*.

Définition (Bit de signe)**.** Pour tous les entiers, le premier bit code le signe : `0` code les positifs, `1` code les négatifs. Ainsi, si les entiers sont codés sur 64 bits, les entiers positifs seront tous de la forme $0b_1 \dots b_{63}$

On dispose donc de 63 bits pour coder les entiers positifs d'où l' :

Exercice 6. a) Quel est donc le plus grand entier de type `int` en PYTHON 2 ?

b) Essayer en PYTHON 2.7.

```
>> a=2**62
>> type(a)
```

puis

```
>> a=2**63
>> type(a)
```

Commentez. Puis faites la même chose en PYTHON 3.

Avec NUMPY, on peut spécifier davantage le type des nombres qu'on manipule. Voici un extrait de la doc. :

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C `long`; normally either `int64` or `int32`)
intc	Identical to C `int` (normally `int32` or `int64`)
intp	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for `float64`.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

Par exemple : on peut déclarer `a=np.int8(77)` pour fabriquer un entier, avec signe, codé sur 8 bits.

Exercice 7 (Les mauvais gag de manip. des entiers sur un nombre de bits donnés)**.**

a) Que donnent les commandes : `a=np.int8(120) b=np.int8(100) a+b`

b) Expliquer la réponse du a).

c) A quoi sert donc ce type entier `int8` (resp. `int16` etc), s'il est si dangereux ?

2.3 Le cas des entiers négatifs : la méthode des compléments

2.3.1 Echauffement sur l'exemple des `int8`

On considère les entiers codés sur 8 bits, (les `int8` de `numpy`) : pour les entiers positifs, le plus grand est $01111111 = 2^7 - 1 = 127$, car on n'oublie pas le bit de signe `0`.

Une première idée, non retenue, de codage pour les négatifs : on aurait pu coder les négatifs $n \in [\![-127, -1]\!]$ en mettant `1` devant l'écriture de $|n|$ en base deux, *mais ce n'est pas ce qui a été choisi*! Une première raison évidente est qu'alors on aurait *perdu de la place* car on aurait eu deux façons d'écrire 0. Mais la raison la plus importante est liée à l'addition.

En effet, avec un tel codage : 2 serait codé par `00000010` et -2 par `10000010`, mais alors la somme usuelle sur les nombres en binaires donnerait comme résultat `10000100` c'est-à-dire -4. Il faudrait donc coder l'addition de manière différente suivant qu'on ajoute des positifs entre eux (addition usuelle) ou des positifs avec des négatifs.

Il s'avère qu'il y a un moyen de contourner cette difficulté avec la :

Définition (La vraie convention de codage pour les négatifs). Si n est un entier négatif, avec $n \in [\![-128, -1]\!]$, on le code, directement sur 8 bits, par $c_2(n) \overset{def}{=} 2^8 - |n| = 256 + n \in [\![128, 255]\!]$, écrit en binaire. En effet, le premier bit est alors forcément un `1` et donc on *sait* qu'il s'agit du code d'un nombre négatif.

Exemple. Le plus petit entier négatif, -128 sera codé par 128 écrit en binaire, donc $c_2(n)$ =`10000000`. C'est un mauvais exemple car $-128 + 256 = 128$. D'où l'

Exercice 8. Quel est le codage $c_2(-103)$ de -103 ?

2.3.2 Obtention commode du codage choisi pour les entiers négatifs

La notion de complément à 1 : Si à partir d'un `int8`, `x`, on en modifie les 8 bits en changeant les `0` en `1` et les `1` en `0`, on obtient un nombre `c(x)` qui s'appelle le *complément à* 1 de x.

Par exemple `c(10011101)=01100010`.

Par construction, il est évident qu'en notant + l'addition des nombres écrits en base 2, `x+c(x)=`

Cela donne une relation simple entre `x` et `c(x)`, qu'on écrira : `c(x)=`

Application à l'interprétation du codage des entiers négatifs :

On a dit au § 2.3.1 que si $n \in [\![-128, -1]\!]$, on le codait par $c_2(n) = 2^8 + n \in [\![128, 255]\!]$.

Cette relation ressemble un peu à ce qu'on vient de dire pour le complément à 1. Précisément, pour obtenir le codage de $n \in [\![-128, -1]\!]$ en `int8`, il suffit de faire les opérations suivantes :

a) On considère l'entier naturel $x = -(n+1) \in [\![0, 127]\!]$, toujours en base deux,

b) puis on considère son complément à 1, $c(x)$. On a alors le codage cherché :

$$c_2(n) = c(-(n+1))$$

Le codage c_2 défini au § 2.3.1, s'appelle (bizarrement) le *complément à deux* (il vaudrait mieux dire *complément à* 2^8 puisqu'au total on a obtenu $2^8 - |n|$).

Exemple Pour $n = -103 = -01100111_{[2]}$, on a $x = -(-103+1) = (103-1) = 01100110_{[2]}$, puis avec le complément à 1, on obtient : $c(x) = 10011001_{[2]}$, qui est bien $c_2(n)$ comme à l'exercice 8.

Exercice 9. Justifier que, d'une manière générale, la méthode donnée dans le cartouche ci-dessus est valide, c'est-à-dire qu'on a toujours l'égalité : $c_2(n) = c(-(n+1))$.

2.3.3 Application au calcul des différences

Remarque 1. On a vu au chapitre 1 comment il est facile pour un ordinateur d'ajouter des entiers *positifs* écrits en base deux (avec une porte logique par bit). Le problème que l'on posait au début du § 2.3.1 était celui de traiter, avec la même opération `+`, l'addition d'un entier positif et d'un entier négatif. On va voir que le codage des négatifs par les compléments à deux permet cela *efficacement*.

Un petit exemple valant mieux qu'un long discours, avec des nombres à trois chiffres en base deux, à chaque fois le codage machine est à droite (N.B. le codage machine est alors en complément à 2^3).

On code les nombres sur trois bits $0, 1, 2, 3$ sont codés resp. par $000, 001, 010, 011$ et les négatifs $n = -4, -3, -2, -1$ codés par les compléments à 2^3 comme 2^3+n écrit en base 2 donc $100, 101, 110, 111$.

	1		001
$+$	-2	$+$	110
	-1		111
	-1		111
$+$	-2	$+$	110
	-3		101

Notez qu'ici dans le résultat, on n'écrit pas le dernier 1 qui devrait apparaître à gauche par retenue. La propriété générale est la suivante :

Proposition 1. Pour ajouter deux entiers a et b dans $[\![-128, 127]\!]$ *dont la somme reste dans cet intervalle*, il suffit de faire la somme de leurs représentations en `int8`. Si cette somme a un 9-ième chiffre à gauche on l'enlève ! Le résultat obtenu est bien le codage machine de $a + b$.

Exercice 10. Prouver la propriété. Si vous avez du mal, lisez l'appendice A.

2.3.4 Cas des entiers en 64 bits

Pour les entiers codés sur 64 bits, on a la même convention de codage, avec le premier bit qui donne le signe, et le même système de complément à deux pour les entiers négatifs : cette fois, si $n < 0$, $c_2(n) = 2^{64} + n$.

L'intervalle des entiers codés sur 64 bits en PYTHON est donc $[\![-2^{63}, 2^{63} - 1]\!]$.

3 Représentation binaire des réels : maths

3.1 A propos de l'écriture décimale

Nous avons l'habitude de l'écriture décimale, qui pour tout réel positif x est de la forme :

$$x = a_r 10^r + a_{r-1}10^{r-1} + a_1 10 + a_0 + a_{-1}10^{-1} + \cdots + a_{-k}10^{-k} + \cdots \quad (\dagger)$$

Dans cette écriture, tous les a_i sont entre 0 et 9, les ... à la fin signifie que cette écriture peut être *infinie*.

Par exemple $1/3 = 0,3333....$

Les nombres qui admette une écriture *finie* de la forme $(\dagger)$ s'appellent les *nombres décimaux*.

Attention : les nombres décimaux ont aussi la particularité d'avoir une *autre* écriture décimale, dite impropre, qui consiste à écrire une infinité de 9 à la fin.

Par exemple $1 = 0,9999....$ oui *égal* et pas *approximativement égal* (cf. appendice maths sur les limites) : une telle écriture d'un décimal avec que des 9 à la fin s'appelle *écriture impropre*.

A part ce problème des écritures impropres pour les nombres décimaux, l'écriture décimale d'un nombre réel est *unique*.

3.2 Ecriture binaire (ou dyadique)

On a de même la :

Proposition 2. Tout réel positif x s'écrit :

$$x = b_q 2^q + \cdots + b_1 2 + b_0 + b_{-1}2^{-1} + \cdots b_{-k}2^{-k} + \cdots$$

où les b_i valent tous 0 ou 1, les $(b_{-i})_{i\in\mathbb{N}}$ ne sont pas tous égaux à 1 à partir d'un certain rang (écriture *propre*).

Définition. Les nombres qui admette une développement binaire *fini* sont appelés *nombres dyadiques*. Ils sont de la forme $a/2^k$ où a est un entier.

3.3 Comment calculer la représentation binaire d'un nombre réel donné par son écriture décimale ?

On considère un nombre réel positif x. Grâce aux méthodes du § 1.2, on sait trouver l'écriture binaire de sa partie entière $\lfloor x \rfloor$.

Pour trouver les coefficients b_{-k} pour $k > 0$, qui apparaissent dans l'écriture de la prop. du § 3.2, on multiplie x par 2^k, ce qui donne :

$$2^k x = \cdots + b_{-k+1}2 + b_{-k} + b_{-k-1}2^{-1} + \cdots$$

En considérant le reste de la division euclidienne par 2 de $\lfloor 2^k x \rfloor$, on obtient b_{-k}.

Exercice 11. Déterminer l'écriture binaire des nombres qui s'écrivent en base dix sous la forme $3,25$ et $0,1$.

4 Représentation informatique des réels : les flottants

4.1 La représentation mantisse-exposant en base 10

Pour tous les nombres décimaux, on va convenir d'une *écriture normalisée* : considérons par exemple le nombre décimal $x = 1234,5678$

On peut aussi l'écrire $123,45678.10^1$ ou $12,345678.10^2$ ou $1,2345678.10^3$ ou $0,12345678.10^4$ ou $12345,678.10^{-1}$ etc.

Parmi toutes ces écritures, on va choisir comme écriture normalisée : $x = 1,2345678.10^3$ et d'une manière générale :

Définition. L'écriture *normalisée*, en base dix, d'un nombre décimal x strictement positif avec une *mantisse* de longueur $r+1$ et un exposant $e \geq 0$ est une écriture de la forme :

$$x = a_r, a_{r-1} \ldots a_0 . 10^e$$

avec $\forall\, i \in [\![0, r]\!]$, $a_i \in [\![0, 9]\!]$ *et* $a_r \neq 0$.

Dans cette représentation, la *mantisse* est le nombre $a_r a_{r-1} \ldots a_0$.

Remarque 2. En informatique, la longueur de la mantisse va être *fixée*, ce qui bien sûr, va nous obliger à considérer des *valeurs approchées* des nombres réels (même décimaux) s'ils s'écrivent avec davantage de chiffres.

Définition (Notation "scientifique" en base dix)**.** En PYTHON la notation `23e17` signifie 23×10^{17}. L'ordinateur renverra l'écriture normalisée `2.3e18`.

4.2 La représentation mantisse-exposant en base 2

C'est celle qui nous sera utile pour passer ensuite au *vrai* codage des nombres en machine à la section suivante.

On a l'analogue exact de la définition précédente :

Définition (mantisse mathématique binaire sans signe)**.** L'écriture normalisée, en base 2, d'un nombre $x > 0$ avec une *mantisse* de longueur $r+1$ et un exposant $e \in \mathbb{Z}$ est une écriture de la forme :

$$x = a_r, a_{r-1} \dots a_{0[2]}.2^e$$

avec $\forall\, i \in [\![0, r]\!]$, $a_i \in [\![0, 1]\!]$ *et* $a_r \neq 0$, donc $a_r = 1$.

Dans cette représentation, la *mantisse mathématique binaire*, sans signe, est le nombre

$$a_r a_{r-1} \dots a_{0[2]} = a_r 2^r + a_{r-1} 2^{r-1} + \dots + a_1 2 + a_0.$$

Remarque 3. Souvent on convient de noter un tel nombre par le couple (m, e) où m est la mantisse et e est l'exposant. Cependant, à ce stade, on n'a parlé que des nombres strictement positifs. Nous aborderons le codage des négatifs et de zéro, ainsi que le codage des exposants, dans la section 4.3 suivante consacrée à la norme IEEE. La *vraie mantisse machine* sera en fait légèrement différente de la mantisse mathématique de la définition précédente, notamment à cause d'un bit rajouté pour le signe, et aussi parce qu'en binaire, pour les nombres strictement positifs, le nombre $a_r \neq 0$ est forcément un 1, ce qui permettra de *l'enlever* du codage du nombre.

Pour l'instant, contentons nous de nous familiariser avec un :

Un bébé exemple : considérons les nombres *strictement positifs* que l'on peut représenter avec une mantisse mathématique binaire m de longueur 3, et un exposant $e \in [\![-2, 1]\!]$.

Exercice 12. a) Combien a-t-on de nombres strictement positifs de cette forme ?

b) Quels sont les *entiers* parmi ces nombres ?

c) Combien y-a-t-il de tels nombres dans $\left[\frac{1}{2}, 1\right[$?

d) Même question dans $\left[\frac{1}{4}, \frac{1}{2}\right[$?

e) Quel est le plus grand et le plus petit parmi ces nombres ?

4.3 Le standard *double précision* sur les flottants : 64 bits, illustré en Python

Les différences de représentation des nombres flottants d'un ordinateur à un autre obligeaient au départ à reprendre les programmes de calcul pour les porter d'une machine à une autre. Pour assurer la compatibilité entre les machines, une norme a été proposée par l'IEEE (Institute of Electrical and Electronics Engineers) et ceci, dès 1985.

4.3.1 La définition du standard

Définition. Le format *simple précision* est codé sur 32 bits, et le *double précision* utilise 64 bits. En Python comme de plus en plus partout, le format utilisé est le format *double précision* (même si votre ordinateur est un 32 bits). Le signe est codé sur un bit, l'exposant sur 11 bits, et la mantisse est codée sur 52 bits. On va voir plus précisément comment chacun est codé.

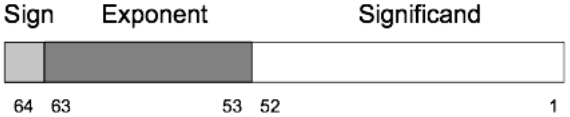

Figure 1: An IEEE-754 64 bits binary floating point number.

a) Le bit de signe S porte 0 pour les positifs stricts et 1 pour les négatifs stricts. On va voir que le nombre zéro a [2] deux codages l'un avec 0 l'autre avec 1 comme bit de signe.

b) Le code E pour l'exposant : Les 11 bits d'exposants peuvent coder $2^{11} = 2048$ nombres. Comme on veut coder aussi bien des exposants positifs que négatifs, plutôt que d'utiliser un bit de signe, on va *décaler* les exposants. En notant e l'exposant mathématique, on code $E = e + 1023$ en base deux (on parle d'« exposants biaisés »). On convient que les valeurs mathématiques autorisées pour e sont dans $[\![-1022, 1023]\!]$ et donc $E \in [\![1, 2046]\!]$. Reste deux valeurs spéciales disponibles pour E.

   i) $E = 0$ codera les nombres *très petits* et notamment 0.

   ii) $E = 2047$ codera le nombres *trop grands* traités comme *infinis*.

c) Le code M de mantisse machine :

   i) Pour le nombre 0, on le code avec la mantisse machine $M = 0 \dots 0$ et l'exposant machine $E = 0$ réservé.

   ii) Pour les nombres non nuls, on sait que la mantisse mathématique $m = a_r a_{r-1} \dots a_0$ commence par $a_r = 1$. La mantisse machine M codera $a_{r-1} \dots a_0 0 \dots 0$ en complétant avec des 0 pour avoir 52 chiffres si nécessaire.

Exemple. Le nombre 1, qui avait une mantisse mathématique $m = 1$ et un exposant $e = 0$ sera codé par $S = 0$, $E = 01111111111$, $M = 0 \dots 0$. Car $E = e + 1023$ écrit en base deux et M s'obtient à partir de m en enlevant le premier 1.

Exercice 13. Quel sera le codage machine des nombres suivants écrits en base deux : $a = 1,0111.2^3$, $b = -1,101.2^{-5}$?

Définition (Nombres machines normalisés)**.** On appelle *nombres machines normalisés* le sous-ensemble des nombres *dyadiques* qui peuvent être écrits *exactement* par le codage IEEE précédent. C'est bien sûr un ensemble *fini*. Pour les autres nombres, les nombres machines serviront d'approximation.

4.3.2 Conséquence : nombre de chiffres significatifs

Exercice 14 (Nombres de chiffres significatifs)**.** a) Suivant le format défini précédemment, on peut coder exactement des nombres qui n'ont besoin « que » d'une mantisse mathématique de 53 chiffres en binaires ($53 = 52+1$ car le premier 1 est omis dans la mantisse machine). On dit qu'il y a 53 chiffres significatifs en binaire. Combien a-t-on alors de chiffres significatifs en écriture décimale ?

b) Tester le programme suivant :

```
i=1
while 1.+10**(-i)>1.:
    i=i+1
```

4.3.3 Le plus grand et le plus petit `float` : l'intervalle des nombres machines (normalisés)

Exercice 15 (Dépassement de capacité : `overflow, underflow`)**.** Exécutez les lignes suivantes en PYTHON et commentez le résultat par rapport à la déf. donnée au 4.3.1

a) `1e-323, 1e-324`

b) `1e308, 1e309`

Remarque 4. Comme expérimenté dans l'exercice précédent a), le code IEEE permet de gérer des nombres plus petits que les nombres décrits au § 4.3.1. Ceci est possible par le fait que pour l'instant avec le code d'exposant $E = 0 \dots 0$, on n'a codé que le nombre 0. Les nombres codés avec $M \neq 0$ et $E = 0 \dots 0$ sont dits *sous-normaux*, nous n'insisterons pas sur ce point.

2. contrairement au cas des entiers

Remarque 5. Toutes ces informations sont disponibles en PYTHON, via le module `sys` qui contient des informations systèmes.

```
from sys import *
#  on importe tout le contenu du module
float_info.max_exp
```

qui répond bien 1024, mais il faut comprendre que 1024 est le premier interdit.

Vous pouvez jouer avec les options de complétion de la commande `float_info.` pour apprendre plus de choses.

Définition (L'intervalle des nombres machines et les nombres normalisés)**.** D'après ce qui précède [3], tous les flottants normalisés représentés en machines en PYTHON, sont donc dans l'intervalle $[x_{\min}, x_{\max}[$ où $x_{\min} = 2^{-1022}$ et $x_{\max} = 2^{1024}$.

Remarque 6 (Différence `overflow/underflow`)**.** Si au cours d'un calcul avec des *floats* l'ordinateur tombe sur un nombre plus grand que $x_{\max}$, on dit qu'il y a un `overflow` (dépassement), et le calcul s'arrête, pour donner un infini. S'il tombe sur un nombre positif strictement plus petit que $x_{\min}$, le calcul *continue*, mais en donnant au nombre en question la valeur zéro (en fait, on peut modifier la gestion de ces cas).

4.3.4 Format d'affichage des flottants en PYTHON

En python : on doit à chaque fois préciser le format qu'on veut pour l'affichage. La raison est qu'il s'agit d'un langage de programmation : le code doit dire exactement le résultat qu'on veut, indépendamment d'un réglage global.

Par exemple, en PYTHON, avec le module math pour avoir `pi` :

```
>>> '{:.22f}'.format(pi)
'3.1415926535897931159980'
```

Ici le `.22f` veut dire qu'on veut 22 chiffres après la virgule. Nous reviendrons sur l'utilisation de la méthode `format` en PYTHON.

Ceci donne un nombre π qui est différent de celui de MATHEMATICA à partir de la 16eme décimale ! Cela peut paraître étrange non ? On pourrait se dire que *de toutes façons* tous les calculs numériques n'auront pas une précision de plus de 16 chiffres, donc cette erreur sur π n'affecterait pas les calculs : c'est faux, à cause des phénomènes de « cancellation » que l'on verra plus tard en calcul numérique. Alors, pourquoi avoir « rentré » un nombre π faux ? Voir l'exercice suivant :

Exercice 16 (L'explication du mystère sur π)**.** Le nombre `pi` du module `math` est un `float`, codé donc sur avec une mantisse binaire de 53 chiffres. Une façon de savoir exactement quel est le nombre codé en machine pour pi est d'utiliser la commande `pi.as_integer_ratio()` qui répond

```
(884279719003555, 281474976710656)
```

Ce couple dit que le `pi` du module math est le nombre rationnel (884279719003555/281474976710656)

a) Obtenir l'écriture en base deux de chacun de ces deux nombres à l'aide de PYTHON.

b) Conclure : quelle est la représentation (m, e) binaire de ce nombre `pi` ?

c) Aurait-il été possible de représenter `pi` plus précisément ? Question difficile !

3. en laissant de côté les nombres sous-normaux

A Excursion niveau collège : les compléments en base dix

En base dix, on parle de *méthode des compléments* pour désigner la méthode suivante :

Définition. Pour chaque *chiffre* i, son complément est par définition $9 - i$.

Pour un *nombre* n à r chiffres par exemple $n = 356291$, on appelle *complément* de n le nombre c obtenu en remplaçant chaque chiffre de n par son complément.

Sur l'exemple $c = 643708$.

L'intérêt d'introduire les compléments vient de la facile :

Proposition 3. Pour calculer la différence $a - b$ de deux nombres entiers a et b, on peut écrire le complément c de a, calculer la *somme* $S = c + b$ puis écrire le complément C de S. Alors $C = a - b$.

Exemple Si on veut calculer $24563 - 15743$. On écrit le complément de 24563 qui est $c = 75436$. On calcule la somme $b + c = 75436 + 15743 = 91179$. Le complément de la somme s'écrit 08820, c'est la différence cherchée.

Exercice 17. Démontrez cette propriété.