Codage des nombres dans la machine

Table des matières

1 Codage d'un entier naturel (entier positif)

L'ensemble de entiers naturels est $\mathbb{N}$. On les appellera aussi entiers positifs pour les différentier des entiers les plus généraux qui peuvent aussi bien être positifs que négatifs.

La représentation naturelle d'un entier $n > 0$ dans un système informatique est sa représentation en base deux.

Commençons par un petit entier naturel, 12 par exemple. On voit rapidement que :

$$12 = 8 + 4 = 1 \times 2^3 + 1 \times 2^2$$

que nous pouvons compléter en :

$$12 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

Par définition, la succession des quatre signes 1100 est **l'écriture** de l'entier 12 **en base deux** . On dit aussi la **représentation** de 12 en base 2 ou encore la **représentation binaire** de 12.

Afin de bien préciser qu'il s'agit de la représentation de 12 en base 2 et pas de l'entier 1100 (mille cent) en base 10, nous l'écrirons avec un **b** en indice tout à droite :

$$12 = 1100_b$$

De cette manière, tout entier naturel peut être écrit d'une *façon unique* sous la forme d'une suite de 0 et 1.

Plus généralement, la représentation binaire d'un entier naturel s'écrira $b_m b_{m-1} ... b_2 b_1 b_0$ avec $b_i \in \{0, 1\}$. Dans cette écriture, chacun des bits b_i possède un **rang** :

rang	m	$m-1$	...	2	1	0
	b_m	b_{m-1}	...	b_2	b_1	b_0

Le bit de rang 0 (le plus à droite) est le bit de **poids faible**. Celui de rang m (le plus à gauche) est le bit de **poids fort**. L'entier n se calcule alors suivant :

$$n = b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + ... + b_{m-1} \times 2^{m-1} + b_m \times 2^m$$

Voici un tableau qui donne les représentations binaires des entiers naturels de 0 à 15.

0	0000_b	8	1000_b
1	0001_b	9	1001_b
2	0010_b	10	1010_b
3	0011_b	11	1011_b
4	0100_b	12	1100_b
5	0101_b	13	1101_b
6	0110_b	14	1110_b
7	0111_b	15	1111_b

Figure 1 Représentation binaire est entiers de 0 à 15.

Étant donné un entier naturel écrit en base 10, prenons 313 par exemple, comment trouver rapidement sa représentation binaire ?

Un moyen simple consiste à commencer par dessiner un tableau des puissances de 2 comme ci-dessous :

Puissance	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Valeur	1	2	4	8	16	32	64	128	256	512	1024

On commence par chercher quelle est la plus grande puissance de 2 que peut contenir 313 : c'est $256 = 2^8$. Nous avons donc :

$$313 = 256 + 57 = 2^8 + 57$$

On recommence alors avec 57 en cherchant la plus grande puissance de 2 que contient ce nombre : c'est $32 = 2^5$. Cela permet d'écrire :

$$313 = 2^8 + 32 + 25 = 2^8 + 2^5 + 25$$

On recommence le même processus avec 25 et ainsi de suite :

$$313 = 2^8 + 2^5 + 25 = 2^8 + 2^5 + 16 + 9 = 2^8 + 2^5 + 2^4 + 8 + 1$$

c'est à dire :

$$313 = 2^8 + 2^5 + 2^4 + 2^3 + 2^0$$

Il suffit ensuite de compléter les termes comme ci-dessous :

$$313 = 1\times2^8+0\times2^7+0\times2^6+1\times2^5+1\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$$

ce qui donne la représentation binaire suivante :

$$313 = 100111001_b$$

2 Octet, seizet, trente-deuzet

Pour une raison pratique liée à l'architecture de la mémoire des premiers ordinateurs et de ceux qui ont suivis et qui ont repris cette architecture en la rendant simplement plus ample et efficace, il est d'usage de regrouper les bits en paquets ayant un nombre bien déterminé de bits et désignés par un nom spécifique.

De cette façon, un paquet de :

- 8 bits forme un **octet** ;
- 16 bits forme un **seizet** qui vaut donc 2 octets ;
- de 32 bits forme un **trente-deuzet** qui vaut 4 octets ou 2 seizets.

selon la terminologie française [1].

Le bit mis à part, l'octet est la plus petite unité d'information pouvant exister au sein d'un système informatique : n'importe quel entier naturel sera codé en utilisant au moins 1 octet, c'est à dire 8 bits.

1. Les anglo-saxons parleront plutôt de **byte** pour désigner un paquet de 8 bits, de **word** pour 16 bits et de **long-word** ou plus simplement **long** pour un paquet de 32 bits.

Par exemple 3 s'écrit 11_b mais dans une machine il sera écrit au minimum sous la forme 00000011_b en le complétant avec 6 zéros à gauche pour former un octet.

Il peut être nécessaire d'utiliser plus d'un octet pour coder un entier et dans ce cas, un seizet ou un trente-deuzet seront les formats utilisés. On peut bien sûr aller encore plus loin en utilisant 8, 16 (ou plus encore) octets. Pour la majorité des applications 1, 2, 4 ou 8 octets suffiront.

Reprenons l'exemple de $313 = 100111001_b$ qui nécessite 9 bits. Le plus petit format qui puisse les contenir est un seizet et l'écriture binaire sera complétée avec 7 zéros à gauche pour former :

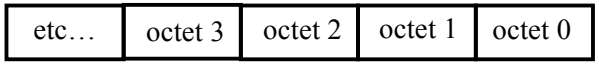
$$313 = 0000000100111001_b$$

Afin de rendre cette écriture plus lisible, mettons un "espace" entre les deux octets :

$$313 = 00000001\ \ 00111001_b$$

Chaque octet est alors numéroté en commençant par celui qui est le plus à droite, qui reçoit le numéro 0, puis en augmentant au fur et à mesure qu'on se déplace vers la gauche. Ce numéro est le **rang** de l'octet [2]. Les deux octets ayant le rang le plus petit (c'est à dire 0) et le rang le plus haut sont appelés respectivement **octet de poids faible** et **octet de poids fort**.

Ainsi, un entier naturel codé sur plusieurs octets s'écrira sous la forme :



2. Tout comme les bits qui sont affecté d'un rang, il en va donc de même pour les octets.

Figure 2 Représentation d'un entier naturel codé sur plusieurs octets. Chaque octet possède un rang. L'octet de rang 0 est appelé octet de poids faible et l'octet de plus haut rang est l'octet de poids fort.

3 Codage d'un entier relatif

Les entiers relatifs sont les éléments de $\mathbb{Z}$. Ils peuvent être aussi bien positifs que négatifs. Si $k \in \mathbb{Z}$ est positif, son codage naturel est la représentation en base deux. La question qui se pose est : "Comment coder les entiers < 0 ? ".

3.1 Premier essai

Pour coder -3 par exemple, on peut penser coder $+3$ en binaire, c'est à dire 11_b, puis ajouter un bit égal à 1 à gauche (bit de poids fort donc) pour indiquer le signe négatif : on aurait donc 111_b.

Comment faire alors la différence entre 7 dont la représentation est aussi 111_b et -3 ? C'est simple si on décide que les entiers doivent être codés sur un nombre fixe d'octets, défini à l'avance.

Prenons par exemple un codage sur 1 octet, c'est à dire 8 bits. Le bit de signe est le bit de poids fort (bit n°7) et il vaut nécessairement 1 pour un entier negatif et 0 pour un entier positif. Cela donne :

1	0	0	0	0	1	1

pour -3

et

0	0	0	0	1	1	1

pour 7.

Malheureusement, cette méthode de codage entraîne qu'il y a deux représentations différentes pour 0 et -0 et, de plus, si on additionne les représentations de $+3$ et -3, on n'obtient pas 0.

D'où une autre solution...

3.2 Codage en complément à deux

Commençons par expliquer comment coder des entiers relatifs selon ce principe, en utilisant 8 bits, c'est à dire un seul octet. 256 entiers différents peuvent être représentés de la façon suivante :

- Les octets de 0000000_b à 0111111_b, c'est à dire ceux dont le **bit de poids fort est nul**, sont réservés au codage naturel des entiers positifs compris entre 0 et 127. L'octet nul $00....0_b$ représente bien 0.

- Il reste alors les 128 derniers octets, de 10000000_b à 11111111_b, dont le **bit de poids fort vaut 1** qui pourraient coder des entiers naturels de 128 à 255, ...oui mais... dans ce cas ils sont destinés à coder les entiers négatifs de -128 à -1 selon le principe suivant :

  À tout entier $k \in [\![-128, -1]\!]$, on associe l'unique entier naturel $n_k = k + 256 \in [\![128, 255]\!]$. C'est alors le *code naturel* de n_k qui va représenter k.

En conséquence :

- Un entier k positif est toujours représenté par un octet dont le bit de poids fort est égal à 0, tandis qu'un entier k négatif est représenté par un octet dont le bit de poids fort vaut 1 : ce bit contient donc l'information sur le signe de l'entier et on l'appelle pour cette raison, le **bit de signe**.

- Étant donné un octet $1xy...z_b$, on trouve l'entier $k < 0$ qu'il représente en :

  1. Déterminant l'entier naturel $n_k > 0$ dont cet octet est le codage naturel ;
  2. puis en retranchant 256 à n_k pour trouver k : $k = n_k - 256$.

Exemple : 10011101_b donne $n_k = 157$ d'où $k = 157 - 256 = -99$.

3.3 Généralisation

Nous pouvons aisément généraliser cette approche pour des entiers codés sur 2 ou 4 (ou peut être encore plus) octets. De façon générale, soit p le nombre d'octets utilisés pour coder un entier, ce qui donne un nombre de bits égal à $8 \times p$. Chaque bit pouvant prendre deux valeurs, cela offre 2^{8p} possibilités, que nous divisons en deux parties égales :

- Aux $2^{8p}/2 = 2^{8p-1}$ possibilités dont le bit de poids fort vaut 0, nous associons les entiers $k \geqslant 0$, variant de 0 à :

$$0111111 \underbrace{11111111 \ ... \ 11111111_b}_{p-1 \text{ octets}}$$

  Le plus grand entier positif que l'on peut représenter ainsi vaut :

$$k_{max} = 1 + 2^1 + 2^2 + ... + 2^{8p-2} = 2^{8p-1} - 1$$

- Les 2^{8p-1} autres possibilités, pour lesquels le bit de poids fort vaut 1, servent à coder les entiers $k < 0$ que l'on code en leur ajoutant 2^{8p}. Ainsi, un entier $k < 0$ sera d'abord transformé en :

$$n_k = k + 2^{8p}$$

  qui sera codé de façon naturelle. Le bit de poids fort de ces entiers n_k est toujours égal à 1 : il permet d'identifier le signe de k et possède donc la fonction de bit de signe.

Exemple :

Un codage sur 2 octets donne $2^{8\times2} = 2^{16} = 65536$ possibilités. Les 65 536 / 2 = 32 768 octets dont le $16^{\text{ème}}$ bit est mis à 0 représentent les entiers naturels qui s'étendent de 0 à $2^{8\times2-1} - 1 = 2^{15} - 1 = 32$ 767.

L'autre moitié représente les entiers k négatifs de -1 à -32768 que l'on code en leur ajoutant $2^{8\times2} = 2^{16} = 65$ 536. Un entier $k < 0$ sera donc d'abord transformé en :

$$n_k = k + 65536$$

avant d'être codé en binaire.

3.4 Une autre façon de voir

Soit k est un entier strictement négatif codé sur p octets. Nous pouvons toujours l'écrire : $k = -n$, avec $n > 0$. Soit alors n_k l'entier naturel associé à k. Nous avons :

$$n_k = k + 2^{8p} = -n + 2^{8p} \iff n_k + n = 2^{8p}$$

Comme :

$$1 + 2^1 + 2^2 + ... + 2^{8p-1} = 2^{8p} - 1$$

nous obtenons :

$$n_k + n - 1 = 1 + 2^1 + 2^2 + ... + 2^{8p-1}$$

c'est à dire, en notation binaire :

$$n_k - 1 + n = \underbrace{11111....11111_b}_{\text{que des 1, } 8p \text{ fois}}$$

Cela signifie que les représentations en base deux des entiers naturels $n_k - 1$ et n sont **complémentaires** l'une de l'autre, c'est à dire que l'on passe de l'une à l'autre en inversant tous les bits : les 1 deviennent des 0 et les 0 des 1. Autrement dit [3] :

$$\boxed{n_k - 1 = \overline{n} \quad \text{et donc} \quad n_k = \overline{n} + 1}$$

Prenons l'exemple de -5 avec un codage sur 1 octet :

- On commence par coder son opposé + 5 : 00000101_b
- On inverse tous les bits : les "0" deviennent des "1" et les "1" des "0" : 11111010_b
- On ajoute 1 : $11111010_b + 1 = 11111011_b$

Le codage en complément à deux de - 5 est donc 11111011_b et le bit de poids fort (bit de signe) est bien égal à 1.

4 Codage des nombres réels dans la machine

4.1 Notation décimale d'un réel

Rappelons qu'un nombre réel r qui s'écrit par exemple, en notation décimale, $r = 34.213$ (on utilise la notation anglo-saxonne qui met un point à la place de la virgule) se calcule de la façon suivante :

$$r = 3 \times 10^1 + 4 \times 10^0 + 2 \times 10^{-1} + 1 \times 10^{-2} + 3 \times 10^{-3}$$

Plus généralement, la notation décimale "code" le réel sous la forme :

$$r = d_n...d_3d_2d_1d_0.d_{-1}d_{-2}d_{-3}...d_{-m}$$

où les d_i sont des **chiffres** pouvant prendre une valeur 0, 1, 2, ..., 8, 9.

d_0 est le chiffre des unités, d_1 celui des dizaines, d_2 celui des centaines, etc... Après le point, d_{-1} est le chiffre des dizièmes, d_{-2} celui des centièmes, etc...De façon générale, r se calcule par la relation :

$$r = \sum_{i=-m}^{n} d_i \times 10^i$$

3. $\bar{n}$ représente l'entier obtenu à partir de n en inversant tous les bits

4.2 Notation d'un réel en base deux

Il s'agit de la même logique de notation. Simplement, les chiffres d_i de la notation décimale sont remplacés par des chiffres b_i (dans ce cas, on parle plutôt de **bits**) ne pouvant prendre que les valeurs 0 ou 1 et, à la place d'avoir des puissances de 10, on utilisera les puissances de 2. Par exemple :

$$r = 101.1011$$

signifie :

$$r = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$$

La représentation du réel s'écrira de façon générale sous la forme :

$$r = b_n ... b_2 b_1 b_0 . b_{-1} b_{-2} ... b_{-m}$$

ce qui signifie que la formule permettant de calculer r est :

$$r = \sum_{i=-m}^{n} b_i \times 2^i$$

4.3 Codage binaire des nombres réels

Tout d'abord, l'ordinateur utilise la représentation en base deux. Il y a cependant plusieurs représentations possibles d'un même nombre réel r dans cette base. Par exemple :

$$0.110 \times 2^5 \quad \text{ou} \quad 110 \times 2^2 \text{ ou} \quad 1.10 \times 2^4$$

sont trois représentations différentes d'un même nombre réel. Il est donc convenu de se ramener à la représentation suivante :

$$r = 1.b_{-1}b_{-2}....b_{-m} \times 2^e$$

On choisit donc **l'exposant** e, qui peut être > 0 ou < 0, de sorte que le **bit des unités soit toujours** $b_0 = 1$ et que les b_1, b_2, etc... b_i avec $i > 0$ soient tous nuls. Il ne reste alors que les bits des chiffres après la virgule et l'exposant.

Le nombre réel pouvant aussi être négatif, on partira sur un format de codage de la forme :

$$r = \underbrace{\pm}_{\text{signe}} 1.\underbrace{b_{-1}b_{-2}....b_{-m}}_{\text{mantisse}} \times 2^e$$

Exemple : -110.1001 sera transformé en -1.101001×2^2 et un nombre comme 0.001101001 sera d'abord transformé en 1.101001×2^{-3}

- La **mantisse** est l'ensemble des m bits $b_{-1}b_{-2}....b_{-m}$ qui se trouvent après le point (c'est à dire après la virgule).
- L'**exposant** est le nombre entier e que l'on va aussi *coder en base deux*.
- D'autre part, comme le 1 du b_0 est toujours présent, on l'omet car il est implicite ! Le codage se dirige alors vers le format suivant :

signe (1 bit)	exposant (k bits)	mantisse (m bits)

Dans cette notation :

- Le signe est codé par un seul bit : 0 pour un réel positif et 1 pour un réel négatif. Il s'agit du **bit de signe**.
- L'exposant e est codé sur k bits.
- La mantisse est codée sur m bits

Comme le nombre de bits de codage de l'exposant est fini (k), on ne pourra pas représenter des nombres réels aussi grands ou aussi petits que l'on veut.

De même, le nombre fini de bits de la mantisse (m) entraîne que les nombres représentés auront forcément un *nombre fini de chiffres après la virgule.*

L'ensemble des nombres que l'on peut représenter ainsi est un sous ensemble fini de $\mathbb{R}$ et même de $\mathbb{Q}$: il s'agit de l'ensemble des **nombre flottants**.

Les plus perspicaces d'entre vous aurons peut être remarqué qu'en ne codant pas le 1 juste devant la virgule (le b_0), on ne peut plus faire la différence entre le réel 1.000000..., c'est à dire 1 et le réel nul 0.000000... c'est à dire 0. Comment faire alors ? La réponse est dans la section suivante.

4.4 Codage de l'exposant

L'exposant e est un entier positif, négatif ou nul. On pourrait donc penser qu'on le code comme n'importe quel entier, c'est à dire en complément à deux s'il est négatif... mais ici, ce n'est pas la solution qui a été retenue ! (on va voir pourquoi plus loin)

Pour la suite, il va falloir bien faire la différence entre l'exposant e et le nombre binaire de k bits $e_{k-1}e_{k-2}...e_1e_0$, avec $e_i \in \{0,1\}$ qui est placé dans la zone dédiée à l'exposant et qui *sert à le coder*. Nous allons donc poser :

$$\boxed{\textbf{codexp} = e_{k-1}e_{k-2}...e_1e_0}$$

On peut lui associer une valeur entière positive en écrivant :

$$\boxed{n_{\textbf{codexp}} = \sum_{i=0}^{k-1} e_i \times 2^i}$$

RÈGLES D'ENCODAGE :

- La valeur **codexp** $= \underbrace{000...000}_{k \text{ fois}}$ est réservée pour le réel nul 0.

  Aucun autre réel ne peut avoir les k bits de son exposant tous nuls.

- La valeur **codexp** $= \underbrace{111...111}_{k \text{ fois}}$ est réservée pour désigner l'infini, c'est à dire un truc plus grand que n'importe quel autre réel qu'on va coder. En d'autres termes, aucun nombre réel fini ne peut avoir les k bits de son exposant tous égaux à 1.

- Il reste donc $2^k - 2 = 2 \times \left(2^{k-1} - 1\right)$ valeurs possibles, de $\underbrace{000...001}_{k \text{ fois}}$ à $\underbrace{111...110}_{k \text{ fois}}$. Cela signifie que $n_{\textbf{codexp}}$ va donc varier de :

  $$\left(n_{\textbf{codexp}}\right)_{min} = 1$$

  à

  $$\left(n_{\textbf{codexp}}\right)_{max} = 2^1 + 2^2 + ... + 2^{k-1} = 2 \times \left(2^{k-1} - 1\right)$$

  De plus :

- On souhaite que ces valeurs servent à coder à peu près autant de valeurs négatives que de valeurs positives de e, plus la valeur $e = 0$.

Pour parvenir à ce but, il est nécessaire d'introduire le **décalage** :

$$\boxed{\text{dec} = 2^{k-1} - 1}$$

de sorte que la valeur de l'exposant e se calcule à partir de sa représentation **codexp** sur k bits au moyen de la formule :

$$\boxed{e = n_{\textbf{codexp}} - \text{ dec} \iff n_{\textbf{codexp}} = e + \text{ dec}}$$

De cette façon : $e_{min} = \left(n_{\textbf{codexp}}\right)_{min} - \left(2^{k-1} - 1\right) = -2^{k-1} < 0$ et $e_{max} = \left(n_{\textbf{codexp}}\right)_{max} - \left(2^{k-1} - 1\right) = 2^{k-1} - 1 > 0$. On a bien à peu près autant de valeurs négatives que de valeurs positives de e.

Prenons un exemple avec un codage sur 8 bits, ce qui donne : dec $= 2^7 - 1 = 127$.

- Un codage de l'exposant sous la forme **codexp** $= 00000111 \Longrightarrow n_{\mathbf{codexp}} = 2^2 + 2^1 + 2^0 = 7$ représentera un exposant e égal à :

$$e = 7 - 127 = -120$$

- Si maintenant le **codexp** $= 11000011 \Longrightarrow n_{\mathbf{codexp}} = 2^7 + 2^6 + 2^1 + 2^0 = 195$, la valeur de l'exposant e sera :

$$e = 195 - 127 = 68$$

Le tableau ci-dessous donne un autre exemple concret de codage d'exposant avec $k = 4$ bits. Le décalage est alors : dec $= 2^3 - 1 = 7$. On aura donc, $e = n_{\mathbf{codexp}} - 7$:

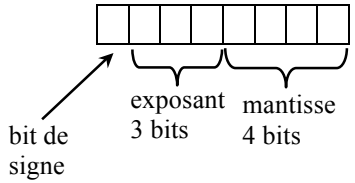
codexp	$n_{\mathbf{codexp}}$	e	**codexp**	$n_{\mathbf{codexp}}$	e
0000	0	réservé à $r = 0$	1000	8	1
0001	1	-6	1001	9	2
0010	2	-5	1010	10	3
0011	3	-4	1011	11	4
0100	4	-3	1100	12	5
0101	5	-2	1101	13	6
0110	6	-1	1110	14	7
0111	7	0	1111	15	réservé à $+\infty$

La représentation de **codexp** est donc très différente d'un entier codé en complément à deux. On voit que des exposants $e < 0$ ont un bit de poids fort égal à 0 dans leur **codexp** et que des exposants e positifs ont un **codexp** avec un bit de poids fort valant 1.

Un point important est que l'ordre dans lequel sont classés les exposants e est le même que celui des entiers $n_{\mathbf{codexp}}$. Cela est très commode lorsqu'on veut comparer deux réels.

4.5 Exemples

Prenons un cas très simple, comme ci-dessous avec une codage du réel sur 8 bits : 1 bit de signe, 3 bits d'exposant et 4 bits pour la mantisse.



Le décalage vaut : déc $= 2^2 - 1 = 3$.

- Codage du réel $r = 0$. Par convention :

0	000	0000

Remarque : le réel nul 0 est toujours codé avec des zéros partout (c'est à dire dans le bit de signe, dans l'exposant et dans la mantisse)

$$\underbrace{0}_{\text{signe}}\ \underbrace{000...000}_{\text{exposant}}\ \underbrace{000...000}_{\text{mantisse}}$$

- Codage du réel $r = 1$ qui s'écrit encore $1.00000... \times 2^0$. L'exposant e valant 0, on a $n_{\mathbf{codexp}} = e + \text{dec} = 3$, d'où **codexp** $=$ 011. La mantisse vaut 0000 et le bit de signe vaut 0, d'où :

0	011	0000

- Codage du réel $r = -3.125$. Dans la base deux r peut s'écrire : $r = -11.001 = -1.1001 \times 2^1$. Le codage de l'exposant est : $n_{\mathbf{codexp}} = 1 + 3 = 4$, c'est à dire **codexp** = 100. La mantisse est égale à 1001 et le bit de signe vaut 1. Il vient :

1	100	1001

En pratique, sur les ordinateurs actuels il y a deux formats de codage des nombres réels qui diffèrent par les nombres de bits k et m alloués à l'exposant et à la mantisse. Il s'agit des codes en **simple précision** et en **double précision**.

4.6 Codage d'un réel en simple précision

Dans ce cas :

- Le nombre réel est codé sur 32 bits
- La mantisse occupe 23 bits
- L'exposant est codé sur 8 bits. Le décalage est donc : dec = $2^7 - 1 = 127$.

Exemple : r = 1 10000010 00110000000000000000000

bit de signe = 1 $\Longrightarrow$ réel négatif. $n_{\mathbf{codexp}} = 2^7 + 2^1 = 130$ donc e = 130 - 127 = 3. Mantisse = $2^{-3} + 2^{-4} = 0.125 + 0.0625 = 0.1875$ à laquelle il faut ajouter 1 pour tenir compte du b_0 omis. Il vient donc :

$$r = -1.1875 \times 2^3 = -9.5$$

4.7 Codage d'un réel en double précision

C'est le type de codage utilisé par Python ! On a ici :

- Le nombre réel est codé sur 64 bits
- La mantisse occupe 52 bits
- L'exposant est codé sur 11 bits. Le décalage est donc : dec = $2^{10} - 1 = 1023$.

Dans ce codage :

- Le plus petit nombre positif non nul est codé :

$$r = 0\ \underbrace{00000000001}_{\text{11 bits}}\ \underbrace{000...00}_{\text{52 zéros}}$$

  c'est à dire : $r = 1.000\ ...00 \times 2^{-1022} = 2.2250738585072014... \times 10^{-308}$.

- Le plus grand nombre positif est :

$$r = 0\ \underbrace{11111111110}_{\text{11 bits}}\ \underbrace{111...11}_{\text{52 uns}}$$

  c'est à dire $r = \left(1 + 2^{-1} + 2^{-2} + ... + 2^{-52}\right) \times 2^{1023} = 1{,}7976931348623157... \times 10^{308}$.