# Numération et Codage

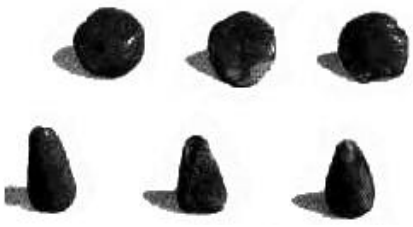
La création de la numération est un des faits les plus marquants de l'histoire de l'humanité. Si la plupart des civilisations ont adopté **le système décimal**, c'est qu'il a toujours été naturel de compter sur ses doigts. L'utilisation des phalanges et des articulations permit même d'améliorer ce simple procédé connu de tous : les chinois ont, par exemple, compté jusqu'à 100 000 sur une main et dix milliards sur les deux mains. Les Mayas, Aztèques, Celtes et Basques, sans doute plus sportifs, ont utilisé aussi les orteils, et la base 20. Les Sumériens ont bizarrement utilisé la base 60, et les Romains la base 12…Les Shadoks, ne connaissant que quatre mots, Ga, Bu, Zo et Meu, ont naturellement utilisé la base 4 (voir la vidéo)



*Bois de renne entaillé*

Les archéologues ont découvert en Europe des ossements d'animaux, essentiellement des radius, vieux de 20 000 à 35 000 ans, dont les entailles servaient très probablement à dénombrer les ours, bisons ou autres gibiers abattus. L'homme alors incapable de compter est tout au plus capable de concevoir l'unité et la multitude. Ce type d'entailles, ou de crans, retrouvé aussi sur les parois d'une caverne préhistorique, au côté de dessins d'animaux, ne laisse aucun doute sur la fonction de dénombrement.

Dès 9 000 avant J.-C., les peuples du Proche-Orient utilisent des cailloux : en latin calculi, à l'origine du mot calcul. Cette méthode est encore utilisée aujourd'hui (bouliers chinois). Au IVème millénaire avant J.-C., ce sont les habitants de Suse, capitale de l'Élam (sud-ouest de l'Iran actuel), qui ont les premiers l'idée de remplacer les cailloux par des objets dont la dimension et la forme correspondent à un ordre d'unité d'un système de numération : un bâtonnet symbolise l'unité simple, une bille plate la dizaine, une sphère la centaine.



*Calculi*

À la même époque, les Sumériens, habitants du pays de Sumer (basse Mésopotamie, actuel Irak), créent un système similaire. Seule différence, le système utilisé est sexagésimal plutôt que décimal, un petit cône vaut 1, une bille 10, un grand cône 60, un grand cône perforé 600, une sphère 3 600, etc... Notre culture a d'ailleurs conservé un souvenir des Sumériens puisque nous utilisons encore leur système pour exprimer la mesure du temps en heures, minutes et secondes, et la mesure d'angle plan en degrés, minutes, secondes.

Les Sumériens enferment ces objets dans des boules creuses en argile, qui servent pour la comptabilité des biens. Elles portent à leur surface le sceau du propriétaire ou du contrôleur, et doivent être cassées pour qu'on puisse en vérifier le contenu.

Vers 3200 avant J.-C., les différents *calculi* enfermés sont symbolisés par diverses empreintes sur la paroi externe de chaque boule. Il n'est donc plus nécessaire de briser la boule d'argile pour procéder à des vérifications ou inventaires, il suffit de « lire » les informations. Ce sont là les plus vieux chiffres de l'histoire (chiffres protoélamites).

Vers 3000 avant J.-C., les *calculi* disparaissent, seules restent les marques sur la boule qui s'aplatit peu à peu pour devenir une tablette. A la même époque, les Égyptiens inventent une écriture et un système de numération reposant sur une base décimale (numération hiéroglyphique). C'est vers 2700 avant J.-C. que l'on trouve les premiers chiffres sumériens cunéiformes (formes de clous). Vers 2200 avant J.C. les Akkadiens (basse Mésopotamie), succédant aux Sumériens, adaptent le système sexagésimal à une base décimale : le 60 et le 600 deviennent 100 et 1000. Mais tous ces systèmes se bornent à répéter le chiffre de chaque classe autant de fois que nécessaire…

Toujours à Babylone, vers 1900/1800 avant J.-C., on trouve les plus anciennes traces de numération positionnelle sexagésimale. Vers les années 300/200 avant J.-C., apparaît la première utilisation du zéro, qui n'est alors pas un chiffre. Il est intéressant de noter que le mot « chiffre » trouve ses origines dans le monde arabe où le zéro était appelé « sifr » (ٱلصِّفْر *aṣ-ṣifr*, à prononcer avec l'accent).

Mais c'est aux Indiens que l'on doit la découverte ultime. Si on a longtemps cru que notre numération provenait du monde arabo-musulman, il semble qu'elle soit plutôt d'origine indienne. Cette notation est apparue pour la première fois au milieu du IIIème siècle avant notre ère. Nos chiffres dits « arabes » sont en fait « des chiffres indiens, un peu déformés par l'usage, le temps et les voyages ».

Au début de notre ère, les Grecs et les Romains utilisent des lettres pour écrire les chiffres, mais leur système de numération est une « régression dans l'histoire du calcul » (G. Ifrah). C'est pourquoi les comptables romains, et les calculateurs européens du Moyen Âge après eux, utiliseront toujours des tables de calcul.

C'est au IVème ou Vème siècle que les neuf premiers chiffres indiens reçoivent une valeur de position selon une base décimale et qu'ils sont complétés par le zéro.

En 629, le mathématicien et astronome indien Brahmagupta publie son Brahmasphutasiddhânta, qui révèle une parfaite maîtrise de la notation décimale de position au moyen de neuf chiffres et du zéro. Il utilise aussi, le premier, des nombres négatifs et énonce les quatre opérations fondamentales. Dans ce système, les nombres de 1 à 9 reçoivent une valeur qui, selon leur position dans l'énonciation des nombres, correspond à plusieurs ordres d'unités. En disant un, trois, huit pour « notre 831 d'aujourd'hui » par exemple, on donne une valeur d'unité simple au mot un, une valeur de dizaine à trois et une valeur de centaine à huit. Il est amusant de remarquer que si les chiffres utilisés aujourd'hui, dits arabes, constituent une sorte de langage universel, des chiffres « hindi » représentant les mêmes nombres coexistent encore dans certains pays du Moyen-Orient.

En 976, lors d'un voyage en Espagne, le moine Auvergnat Gerbert d'Aurillac (il sera pape de 999 à 1003 sous le nom de Sylvestre II) s'initie aux chiffres dits arabes et les introduit en Europe occidentale. Ils ne seront réellement utilisés qu'à la fin du XIIème siècle.

Vers 1440, la mise au point par Johannes Gensfleisch, dit Gutinberg, du procédé de composition en caractères mobiles fondus en alliage d'imprimerie stabilise graphiquement les chiffres « arabes » en Europe occidentale, et donne naissance à la forme définitive qu'ils ont actuellement.

Quant à l'origine graphique des chiffres, elle est relativement simple pour le 1, le 2 et le 3. La superposition de deux ou trois traits horizontaux, réunis d'abord en un seul signe par une ligature, a donné naissance à des graphismes de même facture que le 2 et le 3 indiens. Les chiffres 4 à 9 ont subi beaucoup plus de modifications au cours du temps.

## 1 - SYSTÈMES DE NUMÉRATION

On utilise aujourd'hui des systèmes de numération dits pondérés : c'est une numération positionnelle, comme énoncé précédemment.

La définition d'un système de numération pondéré repose sur trois notions :

- La « base » du système : c'est un nombre entier, noté B.
- Les « digits » du système : ce sont des caractères, tous différents, représentant chacun un élément de la base. Il y en a donc B au total, notés 0, 1, 2, 3, 4, ... Pour écrire un nombre, on associe plusieurs digits dans un ordre déterminé, par exemple : N = 1354 ou N = 4153 ou ...
- Le « poids » de chaque digit selon son rang (sa position). Compté de la droite vers la gauche, ce poids est $B^0$ (c'est à dire 1) pour le premier digit, $B^1$ (c'est à dire B) pour le second digit, $B^2$ pour le troisième digit, etc...

## 1.1 - Système décimal

Dans ce système, la base B est 10. Il y a 10 digits notés : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Les nombres 3997 et 195,28 exprimés en décimal signifient :

$3997 = 3 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 7 \times 10^0$ $\quad 195{,}28 = 1 \times 10^2 + 9 \times 10^1 + 5 \times 10^0 + 2 \times 10^{-1} + 8 \times 10^{-2}$

**Convention :** en base 10, on ne note pas $(3997)_{10}$ ou $(195{,}28)_{10}$, mais 3997 ou 195,28.

On ne peut représenter exactement en décimal que les nombres de la forme $N / 10^k$ avec N et k entiers.

Rappelons que ces chiffres indiens nous sont parvenus par le monde arabo-musulman où on écrit de la droite vers la gauche. En partant de la droite, on peut lire aisément un nombre, chiffre après chiffre :




La lecture de gauche à droite, à la mode occidentale, est impossible sans une vision d'ensemble du nombre. En effet, on lit 3, puis trente neuf, puis trois cent quatre vingt dix neuf, avant de comprendre que c'est en fait trois mille neuf cent quatre vingt dix sept.




Cette lecture de gauche à droite est d'autant plus difficile que le nombre est grand (il faut commencer par faire des « paquets de trois » en partant de la droite, avant de pouvoir lire le nombre en partant de la gauche). Pour se consoler de cet inconvénient, on peut dire qu'en annonçant « trois mille… » on a immédiatement un ordre de grandeur du nombre, ce qui ne serait pas le cas si les unités étaient annoncées en premier.

## 1.2 - Système binaire

Le mathématicien et philosophe allemand Leibniz (1646-1716) a été un des premiers à étudier la numération binaire. Mais le système binaire est réellement utilisé depuis le XIXème siècle et les travaux du mathématicien anglais George Boole (nous étudierons plus tard l'algèbre de Boole). C'est aujourd'hui le système qui permet de traiter et représenter les informations par ordinateur.

Dans ce système, la base B est 2. Il y a 2 digits notés : 0 et 1

Les nombres 1101 et 101,01 exprimés en binaire signifient :

$(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$ (en base 10).

$(101{,}01)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1 \times 4 + 0 \times 2 + 1 \times 1 + 0 \times 0{,}5 + 1 \times 0{,}25 = 5{,}25$ (en base 10).

On ne peut représenter exactement en binaire que les nombres de la forme $N / 2^k$ avec N et k entiers.

Les opérations arithmétiques simples, comme l'addition et la multiplication, sont faciles à effectuer en binaire :

L'addition

| | 2 | 6 |
|---|---|---|
| + | 6 | 0 |
| | 8 | 6 |

| Poids | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| | | | 1 | 1 | 0 | 1 | 0 |
| + | | 1 | 1 | 1 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

La multiplication

| | 1 | 3 |
|---|---|---|
| x | | 5 |
| | 6 | 5 |

| Poids | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| | | | | 1 | 1 | 0 | 1 |
| x | | | | | 1 | 0 | 1 |
| | | | | 1 | 1 | 0 | 1 |
| | | | 0 | 0 | 0 | 0 | |
| | | 1 | 1 | 0 | 1 | | |
| | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

## 1.3 - Vocabulaire

Un « Bit » (contraction américaine de Binary digiT) est un digit du système binaire (valeur 0 ou 1).

Un « Mot » (Word en américain) est un ensemble de bits dont il faut préciser le nombre. Par exemple : un mot de 16 bits, un mot de 12 bits, .... Un mot de 10 bits sera nécessaire pour exprimer un nombre compris entre 0 et 1023.

Un « Quartet » est un mot de 4 bits.

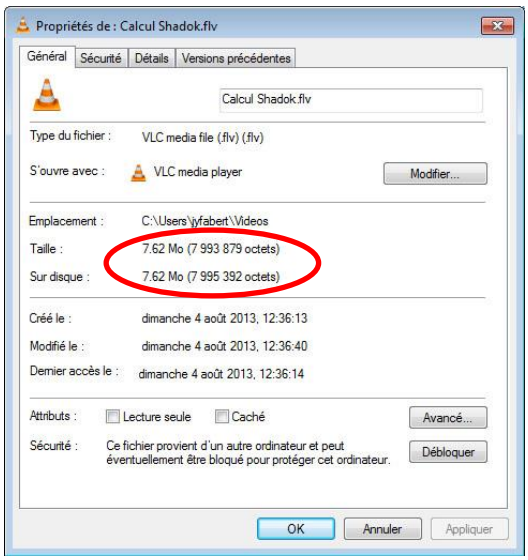
Un « Octet » est un mot de 8 bits. On dit aussi « Byte », mais on peut trouver des Bytes qui n'ont pas 8 bits. Il ne faut pas confondre aussi bit (b) et Byte (B).

Dans un octet, ou un mot, le « Bit de poids faible » (L.S.B. en américain, Less Significant Bit) est le bit situé le plus à droite, donc de plus faible poids. Le « Bit de poids fort » (M.S.B. en américain, Most Significant Bit) est le bit situé le plus à gauche.

Depuis décembre 1998, une norme internationale a imposé que les préfixes kilo, méga, giga, téra, etc., correspondent aux mêmes multiplicateurs que dans tous les autres domaines. Ce qui donne :
1 kilooctet (ko ou kB) = $10^3$ octets = 1 000 octets
1 mégaoctet (Mo ou MB) = $10^6$ octets = 1 000 ko
1 gigaoctet (Go ou GB) = $10^9$ octets = 1 000 Mo
1 téraoctet (To ou TB) = $10^{12}$ octets = 1 000 Go
1 pétaoctet (Po ou PB) = $10^{15}$ octets = 1 000 To

Mais on peut encore compter comme avant 1998. Il faut alors utiliser d'autres appellations :

1 kilo binaire octet, appelé kibioctet (kio) = $2^{10}$ octets = 1 024 octets
1 méga binaire octet, appelé mébioctet (Mio) = $2^{20}$ octets = 1 024 Kio
1 giga binaire octet, appelé gibioctet (Gio) = $2^{30}$ octets = 1 024 Mio
1 téra binaire octet, appelé tébioctet (Tio) = $2^{40}$ octets = 1 024 Gio
1 péta binaire octet, appelé pébioctet (Pio) = $2^{50}$ octets = 1 024 Tio




Comme le montre l'image ci-contre, il est intéressant de noter que Microsoft, même dans sa version 10 de Windows, utilise encore la notation antérieure à 1998, sans respecter la norme.

## 1.4 - Système octal

Dans ce système, la base B est 8. Il y a 8 digits notés : 0, 1, 2, 3, 4, 5, 6, 7

Les nombres 547 et 47,12 exprimés en octal signifient :

$(547)_8 = 5x8^2 + 4x8^1 + 7x8^0 = 5x64 + 4x8 + 7x1 = 359$ (en base 10).

$(47,12)_8 = 4x8^1 + 7x8^0 + 1x8^{-1} + 2x8^{-2} = 4x8 + 7x1 + 1x0,125 + 2x0,015625 = 39,15625$ (en base 10).

On ne peut représenter exactement en octal que les nombres de la forme $N / 8^k$ avec N et k entiers.

## 1.5 - Système hexadécimal

Dans ce système, la base B est 16.

Il y a 16 digits notés : 0, 1, 2, 3, …9, A, B, C, D, E, F.

Les nombres 1A5F et 5F,2 exprimés en hexadécimal signifient :

$(1A5F)_{16} = 1x16^3 + Ax16^2 + 5x16^1 + Fx16^0 = 1x4096 + 10x256 + 5x16 + 15x1 = 6751$ (en base 10).

$(5F,2)_{16} = 5x16^1 + Fx16^0 + 2x16^{-1} = 5x16 + 15x1 + 2x0,0625 = 95,125$ (en base 10).

On ne peut représenter exactement en hexadecimal que les nombres de la forme $N / 16^k$ avec N et k entiers.

## 2 - CHANGEMENT DE SYSTÈME DE NUMÉRATION

Le premier paragraphe présente quatre systèmes de numération parmi les plus utilisés. Il existe des règles pour passer d'un système à un autre, pour les entiers naturels :

### 2.1 - Conversion Octal en Binaire, et Binaire en Octal

On remarque que la base du système octal (8) est égale à la puissance troisième de la base du système binaire (2) : $8 = 2^3$

A chaque digit d'un nombre exprimé en octal, on peut donc faire correspondre un ensemble de 3 bits du même nombre exprimé en binaire.

*Exemple* $(1547)_8 = (001)(101)(100)(111) = (1101100111)_2$

La conversion inverse se fait de la même manière, en partageant le nombre binaire en ensembles de 3 bits à partir de la droite.

*Exemple* $(11010110100)_2 = (011)(010)(110)(100) = (3264)_8$

### 2.2 - Conversion Hexadécimal en Binaire, et Binaire en Hexadécimal

On remarque que la base du système hexadécimal (16) est égale à la puissance quatrième de la base du système binaire (2) : $16 = 2^4$

A chaque digit d'un nombre exprimé en hexadécimal, on peut donc faire correspondre un ensemble de 4 bits du même nombre exprimé en binaire.

*Exemple* $(6B3)_{16} = (0110)(1011)(0011) = (11010110011)_2$

La conversion inverse se fait de la même manière, en partageant le nombre binaire en ensembles de 4 bits à partir de la droite.

*Exemple* $(1101011101101)_2 = (0001)(1010)(1110)(1101) = (1AED)_{16}$

### 2.3 - Conversion Décimal en Binaire, Octal ou Hexadécimal

Notons B = 2 (binaire) ou 8 (octal) ou 16 (hexadécimal), en base 10 on a :

Soit un entier naturel $N = a.B^4 + b.B^3 + c.B^2 + d.B^1 + e.B^0$

$(N - e)/B = a.B^3 + b.B^2 + c.B^1 + d.B^0$ est un entier, donc e est le reste de la division entière de N par B.
Le quotient est $N_1 = a.B^3 + b.B^2 + c.B^1 + d.B^0$

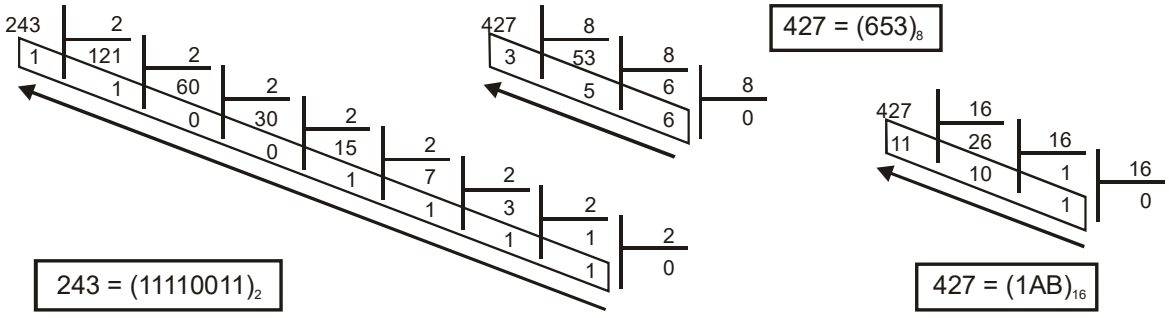
$(N_1 - d)/B = a.B^2 + b.B^1 + c.B^0$ est un entier, donc d est le reste de la division entière de $N_1$ par B.
Le quotient est $N_2 = a.B^2 + b.B^1 + c.B^0$

$(N_2 - c)/B = a.B^1 + b.B^0$ est un entier, donc c est le reste de la division entière de $N_2$ par B.
Le quotient est $N_3 = a.B^1 + b.B^0$

$(N_3 - b)/B = a.B^0$ est un entier, donc b est le reste de la division entière de $N_3$ par B. Le quotient est $N_4 = a.B^0$

$(N_4 - a)/B = 0$ est un entier, donc a est le reste de la division entière de $N_4$ par B. Le quotient est nul.

Il suffit donc d'effectuer des divisions entières successives : du nombre par la base, puis du quotient obtenu par la base, puis du nouveau quotient par la base, ... jusqu'à ce que le quotient devienne nul. L'expression recherchée est constituée par l'ensemble des restes successifs des divisions, lu à l'envers, comme on le voit ci-dessous :




Une autre méthode (suite des dividendes) est disponible ici : http://unprof.com/ellipses/binaire.html

**Cas d'un nombre décimal :**

Dans le cas d'un nombre décimal $X = a.B^4 + b.B^3 + c.B^2 + d.B^1 + e.B^0 + f.B^{-1} + g.B^{-2} + h.B^{-3}$

Ayant déjà converti la partie entière, il reste à convertir $f.B^{-1} + g.B^{-2} + h.B^{-3}$

En multipliant successivement par B les parties décimales, on trouve f, g et h

*Exemple* La conversion en binaire de 0,375 est $(0.\boxed{011})_2$ car :

0,375 x 2 = $\boxed{0}$,75 0,75 x 2 = $\boxed{1}$,5 0,5 x 2 = $\boxed{1}$,0 (on attend que la partie décimale soit nulle)

Mais on ne peut pas convertir exactement 0,3 en binaire, car on se souvient qu'on ne peut représenter exactement en binaire que les nombres de la forme $N / 2^k$ avec N et k entiers.

La conversion en binaire de 0,3 est $(0.\boxed{1001}\boxed{1001}\boxed{1001}\boxed{1001}\ldots\ldots)_2$ car :

0,3 x 2 = $\boxed{0}$,6 0,6 x 2 = $\boxed{1}$,2 0,2 x 2 = $\boxed{0}$,4 0,4 x 2 = $\boxed{0}$,8 0,8 x 2 = $\boxed{1}$,6 0,6 x 2 = $\boxed{1}$,2 0,2 x 2 = $\boxed{0}$,4 …….

## 3 - CODAGE BINAIRE

Tout traitement informatique ou automatique d'une information implique que celle-ci soit codée, c'est à dire représentée à l'aide de bits à 2 états distincts, 0 et 1, ce qui facilite le stockage et la manipulation. On distingue deux catégories de codes : les codes numériques et les codes alphanumériques. Voici les codes les plus courants :

### 3.1 - Codage numérique d'entiers naturels

#### 3.1.1 - Code Binaire Naturel (BN)

Le code binaire naturel est le code dans lequel on exprime un nombre selon le système de numération binaire. C'est le code le plus simple, il est pondéré et il se prête parfaitement bien au traitement des opérations arithmétiques.

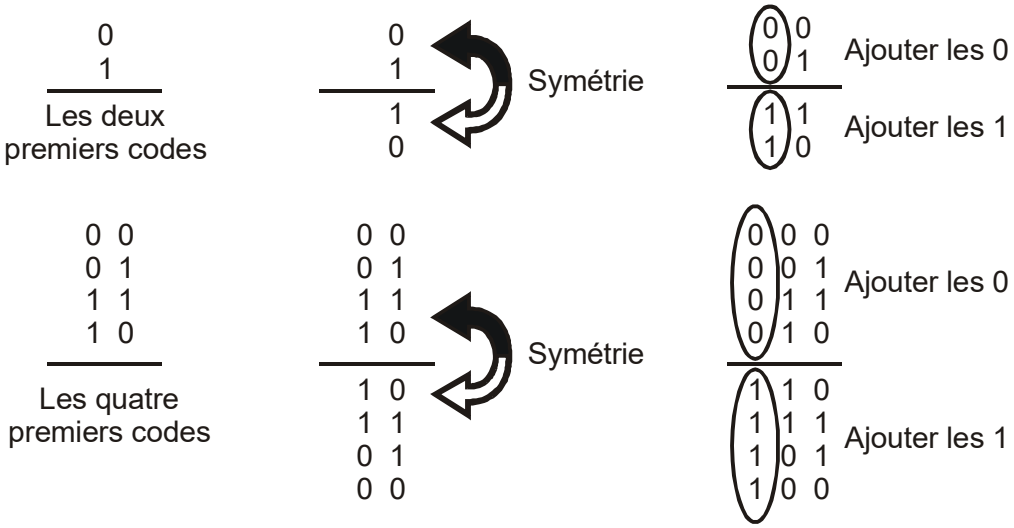
Les poids successifs des bits à partir de la droite (1, 2, 4, 8, 16, 32, ...) sont très faibles par rapport à ceux du système décimal (1, 10, 100, 1000, ...), ce qui est un inconvénient à cause du nombre de bits nécessaires au codage.

Un autre inconvénient du code binaire naturel est qu'il peut introduire des erreurs lors du codage de grandeurs variant de façon ordonnée. Entre deux codes successifs, plusieurs bits doivent changer simultanément. Par exemple : entre le code $(01)_2$ représentant 1 en base 10 et le code $(10)_2$ représentant 2, les deux bits changent en même temps, ce qui est impossible physiquement : il y a deux transitions possibles, $(11)_2$ et $(00)_2$. Pendant un court instant, un code parasite risque donc d'introduire une erreur.

#### 3.1.2 - Code Binaire Réfléchi (BR) ou Code GRAY

Pour pallier l'inconvénient du codage Binaire Naturel, il existe un codage, appelé code Binaire Réfléchi ou code Gray, tel qu'entre deux codes successifs, un seul bit change de valeur. Le tableau page suivante donne quelques correspondances entre le codage Binaire Naturel et le codage Binaire Réfléchi.

Pour construire le tableau des codes Binaires Réfléchis, on procède par réflexions successives. Les deux premiers codes étant 0 et 1, on opère une symétrie et on ajoute à gauche deux « 0 » et deux « 1 » : on obtient ainsi les quatre premiers codes. En opérant de nouveau une symétrie, en ajoutant quatre « 0 » et quatre « 1 », on obtient les huit premiers codes.



Il faudra opérer une nouvelle symétrie, ajouter huit « 0 » et huit « 1 » pour obtenir les seize premiers codes… etc… Voir le tableau ci-dessous.

Le Code Gray est notamment utilisé pour les informations fournies par des codeurs optiques. Ce codage n'est pas pondéré et ne permet pas l'utilisation des opérations arithmétiques.

| Décimal | Binaire Naturel | Binaire Réfléchi |
|---|---|---|
| 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 |
| 1 | 0 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 1 |
| 2 | 0 0 0 0 0 0 1 0 | 0 0 0 0 0 0 1 1 |
| 3 | 0 0 0 0 0 0 1 1 | 0 0 0 0 0 0 1 0 |
| 4 | 0 0 0 0 0 1 0 0 | 0 0 0 0 0 1 1 0 |
| 5 | 0 0 0 0 0 1 0 1 | 0 0 0 0 0 1 1 1 |
| 6 | 0 0 0 0 0 1 1 0 | 0 0 0 0 0 1 0 1 |
| 7 | 0 0 0 0 0 1 1 1 | 0 0 0 0 0 1 0 0 |
| 8 | 0 0 0 0 1 0 0 0 | 0 0 0 0 1 1 0 0 |
| 9 | 0 0 0 0 1 0 0 1 | 0 0 0 0 1 1 0 1 |
| 10 | 0 0 0 0 1 0 1 0 | 0 0 0 0 1 1 1 1 |
| 11 | 0 0 0 0 1 0 1 1 | 0 0 0 0 1 1 1 0 |
| 12 | 0 0 0 0 1 1 0 0 | 0 0 0 0 1 0 1 0 |
| 13 | 0 0 0 0 1 1 0 1 | 0 0 0 0 1 0 1 1 |
| 14 | 0 0 0 0 1 1 1 0 | 0 0 0 0 1 0 0 1 |
| 15 | 0 0 0 0 1 1 1 1 | 0 0 0 0 1 0 0 0 |
| 16 | 0 0 0 1 0 0 0 0 | 0 0 0 1 1 0 0 0 |
| ... | . . . . | . . . . |
| 31 | 0 0 0 1 1 1 1 1 | 0 0 0 1 0 0 0 0 |
| 32 | 0 0 1 0 0 0 0 0 | 0 0 1 1 0 0 0 0 |
| ... | . . . . | . . . . |
| 63 | 0 0 1 1 1 1 1 1 | 0 0 1 0 0 0 0 0 |
| 64 | 0 1 0 0 0 0 0 0 | 0 1 1 0 0 0 0 0 |
| ... | . . . . | . . . . |
| 127 | 0 1 1 1 1 1 1 1 | 0 1 0 0 0 0 0 0 |
| 128 | 1 0 0 0 0 0 0 0 | 1 1 0 0 0 0 0 0 |
| ... | . . . . | . . . . |
| 255 | 1 1 1 1 1 1 1 1 | 1 0 0 0 0 0 0 0 |

- **Conversion du Binaire Naturel en Binaire Réfléchi**

Pour trouver l'expression d'un nombre binaire dans le code réfléchi, on additionne sans retenue le nombre complété par un 0 avec lui-même, et on abandonne le bit de poids faible. Pourquoi ? Voir la solution de l'exercice 15 du chapitre 3.

*Exemple* 7 en Binaire Naturel vaut 111.
Conversion : 1 1 1 0 le nombre complété par un zéro,
+ 1 1 1 le nombre en Binaire Naturel
Somme sans retenue = 1 0 0 ~~1~~, donc 7 vaut 100 en Binaire Réfléchi.

- **Conversion du Binaire Réfléchi en Binaire Naturel**

On part de la gauche vers la droite : le premier chiffre est recopié, les suivants sont recopiés quand le chiffre précédent, en codage Binaire Naturel, est 0 ; si le chiffre précédent, en codage Binaire Naturel, est 1, on remplace par le complément. Pourquoi ? Voir la solution de l'exercice 15 du chapitre 3.

*Exemple* 13 en BR vaut 1011.
Conversion en BN : le premier chiffre 1 est recopié ;
le second chiffre 0 est changé en 1, car le premier chiffre BN est un 1 ;
le troisième chiffre 1 est changé en 0, car le second chiffre BN est un 1 ;
le quatrième chiffre 1 est recopié, car le troisième chiffre BN est un 0.
On obtient donc : 13 en BN vaut 1101.

Une autre méthode (suite des dividendes) est disponible ici : http://unprof.com/ellipses/pdf/chapitre_1_bn_et_br.pdf

N.B. : les transcodages BN→BR et BR→BN peuvent être réalisés avec des « OU Exclusifs ».

### 3.1.3 - Code « Décimal Codé Binaire » (DCB)

Pour coder les 10 chiffres du système décimal, on doit utiliser 4 bits. Dans le code DCB (ou BCD en américain : Binary Coded Decimal), on code chaque chiffre selon son équivalent binaire :
$0 \rightarrow (0000)_2$, $1 \rightarrow (0001)_2$, ... , $9 \rightarrow (1001)_2$, les 6 combinaisons de $(1010)_2$ à $(1111)_2$ ne sont pas utilisées.

La représentation d'un nombre se fait donc avec autant de groupes de 4 bits que ce nombre a de chiffres.

*Par exemple :* $9708 = (1001\ 0111\ 0000\ 1000)_{DCB} = (1001011100001000)_{DCB}$

Ce codage est pondéré, et les poids des bits successifs, en partant de la droite, sont respectivement : 1, 2, 4, 8, 10, 20, 40, 80, 100, 200, 400, 800, ...

### 3.1.4 - Code Excédent 3

Comme dans le code DCB, on code chaque chiffre selon son équivalent binaire, mais augmenté de 3 :
$0 \rightarrow (0011)_2$, $1 \rightarrow (0100)_2$, ... , $9 \rightarrow (1100)_2$, les 6 combinaisons de $(0000)_2$ à $(0010)_2$ et de $(1101)_2$ à $(1111)_2$ ne sont pas utilisées.

*On obtient ainsi, par exemple :* $9708 = (1100\ 1010\ 0011\ 1011)_{exc\ 3} = (1100101000111011)_{exc\ 3}$

Ce codage n'est pas pondéré, mais il présente l'avantage d'être autocomplémentaire, c'est à dire que la complémentation des 4 bits d'un chiffre donne son complément à 9. En effet : $7 = (1010)_{exc\ 3}$, complémenté donne $(0101)_{exc\ 3} = 2$, et 7 + 2 = 9.

## 3.2 - Codage numérique binaire d'entiers relatifs

Sur un octet (8 bits), on peut coder les entiers naturels de 0 à 255. Sur 2 octets (16 bits), on peut coder les entiers naturels de 0 à 65535. De manière générale, sur n bits on peut coder les entiers naturels de 0 à $2^n - 1$.

Si on décide sur un octet que le bit de poids fort, celui de gauche, est consacré au signe (0 pour « + », et 1 pour « - »), on pourra coder les entiers relatifs de -128 à +127 :

$+127 = (01111111)_{bns}$, $+126 = (01111110)_{bns}$, ... , $+1 = (00000001)_{bns}$, $+0 = (00000000)_{bns}$, $-1 = (11111111)_{bns}$, ... , $-126 = (10000010)_{bns}$, $-127 = (10000001)_{bns}$, $-128 = (10000000)_{bns}$.

*Exemple :* $62 = (00111110)_{bns}$, pour trouver le codage de -62, inverser (ce qui donne 11000001) et ajouter 1, soit $-62 = (11000010)_{bns}$. Ou encore, en partant de la droite, conserver tous les chiffres jusqu'au premier 1 inclus, et inverser le reste. C'est la méthode du complément à 2.

De manière générale, sur n bits on peut coder les entiers relatifs de $-2^{n-1}$ à $2^{n-1} - 1$.

## 3.3 - Codage numérique binaire des nombres réels

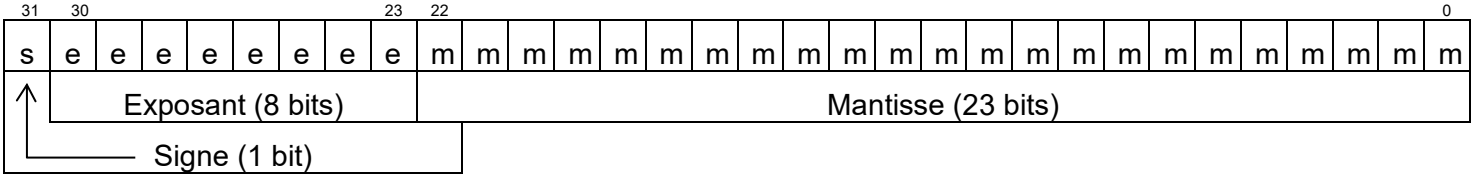
Il ne sera pas possible de coder exactement en binaire tous les réels, comme par exemple $\pi$ ou $\sqrt{3}$.

Le standard IEEE 754 (1985) définit un format de représentation des nombres à virgule flottante (float) en binaire. C'est ce codage qui sera présenté ici. Parmi les formats proposés, nous ne retiendrons que la représentation binaire simple précision (32 bits) et double précision (64 bits).

On se propose de coder **$243,25 = (11110011,01)_2$** en simple précision, soit sur 32 bits, ou sur 4 octets.

Le standard impose d'écrire ce nombre sous la forme normalisée $(1,mmmm\ldots \times 2^n)_2$, soit pour notre exemple $(1,111001101 \times 2^7)_2$. L'exposant est 7 et la mantisse est 111001101 (chiffres après la virgule). Le chiffre 1 avant la virgule étant invariant (pour la forme normalisée) n'est pas représenté dans le code.

Le code simple précision va se mettre sous cette forme :




Pour le signe c'est facile, on prend « 0 » pour positif et « 1 » pour négatif, comme pour les entiers relatifs.

Pour l'exposant, il faut ajouter 127 (0111 1111) à l'exposant 7 (0000 0111), ce qui donne 134 (1000 0110).

La mantisse est tout simplement complétée par des 0, soit 1110 0110 1000 0000 0000 000.

Le code est donc **0100 0011 0111 0011 0100 0000 0000 0000** (43734000 en hexadécimal).

On remarque que l'exposant a été décalé de 127. C'est pour permettre les exposants négatifs, mais surtout éviter la méthode du complément à 2. L'exposant peut ainsi varier de -126 à +127, en étant codé de 1 à 254.

L'exposant -127, codé 0000 0000, est réservé pour zéro et les nombres « non normalisés ».

L'exposant 128, codé 1111 1111, est réservé pour coder les infinis et les « NaN » (Not a Number).

Vérification : pour vérifier un codage, il existe des codeurs en ligne, comme :
http://www.binaryconvert.com/convert_float.html

Mais on peut aussi faire cette vérification « à la main ».
Exemple : $(41\ BD\ 00\ 00)_{16}$ est le code d'un nombre à virgule flottante.
On remarque qu'il est codé en simple précision (4 octets).
Son code binaire est 0100 0001 1011 1101 0000 0000 0000 0000 = 0 10000011 01111010000000000000000
C'est un nombre positif.
La puissance de 2 est $(1000\ 0011)_2 = 131$ auxquels il faut retrancher 127, soit 4.
Le nombre est donc $(1{,}0111101 \times 2^4)_2 = (1\ 0111{,}101)_2 = 23{,}625$

Exercice : montrer que le code de « -0,625 » est « BF 20 00 00 » en hexadécimal.

Cas de l'exposant codé 0000 0000 :

Si la mantisse est nulle, c'est le code du zéro. Il y a deux zéros suivant la valeur du signe, +0 et -0.

Les autres codes sont des codes non normalisés. Ils s'écriront $(0{,}mmmm\ldots \times 2^{-126})_2$, afin de garantir une continuité entre cette représentation et la représentation normalisée. On n'utilise pas $(1{,}mmmm\ldots \times 2^{-127})_2$.

Le + petit nombre positif non normalisé est 0 00000000 00000000000000000000001 = $2^{-149} \approx 1{,}401298464.10^{-45}$
Le + grand est 0 00000000 11111111111111111111111 = $(1 - 2^{-23}).2^{-126} \approx 1{,}175494211.10^{-38}$

Les nombres normalisés :

Le + petit nombre positif normalisé est 0 00000001 00000000000000000000000 = $2^{-126} \approx 1{,}175494351.10^{-38}$
Le + grand est 0 11111110 11111111111111111111111 = $(2 - 2^{-23}).2^{127} \approx 3{,}402823466.10^{38}$

Cas de l'exposant codé 1111 1111, soit 255 :

L'infini est codé 0 11111111 00000000000000000000000

Les autres codes sont des NaN (Not a Number). Il existe deux sortes de NaN, les avertisseurs (premier bit à gauche de la mantisse = 0) et les silencieux (premier bit à gauche de la mantisse = 1). L'utilisation des NaN n'est pas au programme de CPGE.

Double précision :

Pour le codage en double précision, la méthode est identique, sauf qu'on utilise 64 bits (8 octets).
Il y a : 1 bit de signe,
11 bits d'exposant (-1022 à 1023). Il faut ajouter 1023 à l'exposant au lieu de 127.
52 bits de mantisse (le 1 avant la virgule étant implicite)

Le + petit nombre positif non normalisé est $2^{-1074} \approx 4{,}94065645841246544 18.10^{-324}$.
Le + petit nombre positif normalisé est $2^{-1022} \approx 2{,}2250738585072013831.10^{-308}$.
Le + grand nombre positif normalisé est $(2 - 2^{-52}).2^{1023} \approx 1{,}7976931348623157081.10^{308}$.

Révision du standard IEEE 754 de 1985 :

Le standard IEEE 754 a été révisé en 2008, mais conserve les formats « simple précision » et « double précision » qui sont nommés binary32 et binary64. A ces formats binaires de base à virgule flottante est ajouté le format binary128. Deux nouveaux formats décimaux de base à virgule flottante decimal64 et décimal128, utilisant le 10 à la place du 2.

Opérations sur les nombres à virgule flottante :

Pour les calculs, on préfèrera toujours utiliser des nombres entiers car les nombres à virgule flottante n'étant pas toujours codés « exactement » peuvent réserver des surprises.

Par exemple, avec le langage Python, 3,11 + 2,08 = 5,1899999999999995 !

## 3.4 - Codage pour les transmissions

Pour se prémunir contre les éventuelles erreurs pendant une transmission, on ajoute par exemple un bit vérificateur au mot : on l'appellera le bit de parité. Si, dans le mot initial, le nombre de bits égaux à 1 est pair, on met le bit de parité à 0. Si, dans le mot initial, le nombre de bits égaux à 1 est impair, on met le bit de parité à 1. Ainsi le nombre total de bits égaux à 1 (dans le mot initial et le bit de parité) est toujours pair, c'est ce que devra vérifier le détecteur d'erreur. *Exemple de la transmission sur un octet (8 bits) :* on peut réserver 7 bits pour les données et 1 bit de parité.

Il existe des codes qui permettent de détecter et de corriger une erreur survenue dans la transmission de bits. Par exemple, le code de Hamming qui, pour transmettre un bloc de m bits (avec $m = 2^k - k - 1$), ajoute k bits de contrôle. On transmet donc n bits, avec n = m + k. Avec 3 bits de contrôle (k = 3), on peut transmettre 4 bits de données (m = 4) et corriger une erreur de transmission. Avec 5 bits de contrôle (k = 5), on peut transmettre 26 bits de données (m = 26) et corriger une erreur de transmission.

## 3.5 - Codage alphanumérique : exemple du Code ASCII

Utilisé pour les échanges en informatique, le code ASCII (American Standard Code for Information Interchange) permet de coder les 26 lettres de l'alphabet (majuscules et minuscules), les 10 chiffres et les signes de ponctuation : il utilise un octet (8 bits). Cet octet donne une certaine souplesse d'utilisation, puisqu'il permet de coder des commandes de contrôle en plus des caractères alphanumériques (bits 1 à 7), d'utiliser le bit 8 comme bit de parité ou pour définir un deuxième tableau de caractères (caractères étendus).

Table des codes ASCII pour les bits 1 à 7 :

| **Codes ASCII 0 à 127** | | | | | bit b7 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | bit b6 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| | | | | | bit b5 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | | Décimal | | 0 | 16 | 32 | 48 | 64 | 80 | 96 | 112 |
| b4 | b3 | b2 | b1 | | Hexa | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | 0 | NUL | DLE | SP | **0** | **@** | **P** | **`** | **p** |
| 0 | 0 | 0 | 1 | 1 | 1 | SOH | DC1 | **!** | **1** | **A** | **Q** | **a** | **q** |
| 0 | 0 | 1 | 0 | 2 | 2 | STX | DC2 | **"** | **2** | **B** | **R** | **b** | **r** |
| 0 | 0 | 1 | 1 | 3 | 3 | ETX | DC3 | **#** | **3** | **C** | **S** | **c** | **s** |
| 0 | 1 | 0 | 0 | 4 | 4 | EOT | DC4 | **$** | **4** | **D** | **T** | **d** | **t** |
| 0 | 1 | 0 | 1 | 5 | 5 | ENQ | NAK | **%** | **5** | **E** | **U** | **e** | **u** |
| 0 | 1 | 1 | 0 | 6 | 6 | ACK | SYN | **&** | **6** | **F** | **V** | **f** | **v** |
| 0 | 1 | 1 | 1 | 7 | 7 | BEL | ETB | **'** | **7** | **G** | **W** | **g** | **w** |
| 1 | 0 | 0 | 0 | 8 | 8 | BS | CAN | **(** | **8** | **H** | **X** | **h** | **x** |
| 1 | 0 | 0 | 1 | 9 | 9 | HT | EM | **)** | **9** | **I** | **Y** | **i** | **y** |
| 1 | 0 | 1 | 0 | 10 | A | LF | SUB | ***** | **:** | **J** | **Z** | **j** | **z** |
| 1 | 0 | 1 | 1 | 11 | B | VT | ESC | **+** | **;** | **K** | **[** | **k** | **{** |
| 1 | 1 | 0 | 0 | 12 | C | FF | FS | **,** | **<** | **L** | **** | **l** | **\|** |
| 1 | 1 | 0 | 1 | 13 | D | CR | GS | **-** | **=** | **M** | **]** | **m** | **}** |
| 1 | 1 | 1 | 0 | 14 | E | SO | RS | **.** | **>** | **N** | **^** | **n** | **~** |
| 1 | 1 | 1 | 1 | 15 | F | SI | US | **/** | **?** | **O** | **_** | **o** | DEL |

| | | | |
|---|---|---|---|
| SP | Espace | DEL | Effacement |
| NUL | Nul | DLE | Echappement de liaison données |
| SOH | Début d'en-tête | DC1 | Contrôle dispositif 1 |
| STX | Début de texte | DC2 | Contrôle dispositif 2 |
| ETX | Fin de texte | DC3 | Contrôle dispositif 3 |
| EOT | Fin de transmission | DC4 | Contrôle dispositif 4 |
| ENQ | Interrogation | NAK | Acquittement négatif |
| ACK | Acquittement positif | SYN | Synchronisation au repos |
| BEL | Sonnerie | ETB | Bloc fin de transmission |
| BS | Espacement arrière | CAN | Annuler |
| HT | Tabulation horizontale | EM | Fin de support |
| LF | Changement de ligne | SUB | Remplacer |
| VT | Tabulation verticale | ESC | Echappement |
| FF | Alimentation papier | FS | Séparateur de fichiers |
| CR | Retour chariot | GS | Séparateur de groupes |
| SO | Shift désactivé | RS | Séparateur d'enregistrements |
| SI | Shift activé | US | Séparateur d'unités |

*Exemples :*
« K » = 4B en hexa = 75 en décimal (64+11) = 1001011 en binaire
« z » = 7A en hexa = 122 en décimal (112+10) = 1111010 en binaire