1| Architecture des ordinateurs

1 Les composants et leurs connexions

1.1 Ordinateur

Les ordinateurs vont de la plus petite taille à la plus grande : puce en silicium, appareils photos, caméras intelligentes, box internet, calculatrices, smartphones, tablettes, ordinateurs portables, serveurs (permettant les échanges sur internet), stations de travail (ordinateurs de calcul scientifique puissants), supercalculateurs.

Le point commun entre ces machines est leur fonction, à savoir traiter de l'information. Cette information est numérique, c'est-à-dire qu'elle est stockée, échangée ou manipulée sous forme de mots binaire.

Définition :

Un ordinateur est un système de traitement numérique de l'information, capable d'exécuter des programmes, c'est-à-dire une suite d'opérations enregistrées en mémoire.

1.2 Architecture interne

L'architecture de l'ordinateur se décompose en 3 parties :

- Le **processeur**, ou **unité centrale**, comprend en réalité 2 éléments distincts : l'unité arithmétique et logique (UAL) chargée de réaliser les opérations élémentaires, et l'unité de contrôle (UC) chargée du séquençage de ces opérations. L'ensemble de l'unité centrale correspond donc au « cœur » de l'ordinateur, chargé d'interpréter les instructions et d'effectuer les opérations nécessaires au traitement des données. Il est généralement associé à des périphériques entrées/sorties lui permettant de recevoir des instructions de l'extérieur (clavier, souris, etc.) ou de les transmettre (écran, imprimante, haut-parleur, etc.).
  Le processeur possède également une très petite mémoire (**registres**, mémoire cache) destinée à effectuer des calculs élémentaires et au **temps d'accès très court**. Il est caractérisé en partie par sa **fréquence d'horloge** (ou cycle) exprimée en Hertz correspondant aux nombres d'impulsions (et indirectement, d'instructions élémentaires) qu'il peut effectuer par seconde. Un processeur de 1GHz peut donc traiter un milliard d'instructions élémentaires chaque seconde.

- La **mémoire** est utilisée pour le stockage des données. De façon simplifiée on peut distinguer 4 sortes de mémoire : la mémoire morte, la mémoire vive, les mémoires de masse et les mémoires flash.

La **mémoire morte**, ou **ROM** (Read-Only Memory) contient entre autre le BIOS et ne peut être effacée ou reprogrammée simplement. Comme son nom l'indique, elle est destinée à une **lecture seule**, et les données y sont inscrites « en dur ». Elle intervient lors de la mise sous tension de

l'ordinateur pour donner les premières instructions. Son temps d'accès est relativement grand (de l'ordre de 150 ns)

La **mémoire vive**, ou **RAM** (Random Access Memory) est une **mémoire volatile**. Composée essentiellement de condensateurs qui se déchargent dès coupure de l'alimentation électrique. Les données qu'elle contient sont donc vouées à disparaître dès que l'on éteint l'ordinateur. Elle est utilisée pour la mise en mémoire des données temporaires nécessaires qui serviront pour des calculs effectués par le processeur et se caractérise par un temps d'accès relativement court (de l'ordre de 10 ns).

Les **mémoires de masse et mémoires flash** se présentent très souvent sous la forme de périphériques (disques durs, clef USB, CD, DVD, bandes magnétiques, etc.) reliés au processeur. Ces mémoires utilisées pour conserver des **données permanentes** sont nécessairement non volatiles et peuvent avoir de très grosses capacités de stockage. En contrepartie, leur temps d'accès est généralement très grand comparativement à celui des autres mémoires (de l'ordre de $10\ ms$ ou plus).

- Les **bus** (de données, de contrôle et d'adresses) sont chargés de l'**échange d'informations entre la mémoire et l'unité centrale**. Il s'agit en fait de câbles physiques destinés à transmettre les impulsions électriques (ou **bit**) et sont caractérisés par le nombre d'impulsions électriques qu'ils peuvent transmettre simultanément ainsi que la fréquence à laquelle ils peuvent les transmettre. Un bus de 16 bits et de fréquence 20 MHz peut par exemple transmettre environ 38 Mo par seconde. Ils ont donc un rôle essentiel dans la limitation des performances d'un ordinateur.

Figure 1: Processeur, mémoire morte (ROM), mémoire vive (RAM)

1.3 Composants et communication binaire

Les éléments constitutifs de l'ordinateur minimal (processeur + mémoire + bus) sont essentiellement composés de fils de connexion, de transistors, de condensateurs ou encore d'amplificateurs opérationnels. Sans entrer dans le détail, on peut donc aisément comprendre que ce sont des paramètres physiques qui vont impacter directement sur les « performances » d'un ordinateur (diamètre et longueur d'un fil, constante de temps des condensateurs, conductivité des matériaux, etc.) De même, les signaux transmis sont toujours de nature électrique et un choix, arbitraire, de considérer un état binaire a été fait : absence de signal électrique (0) ou présence de signal électrique (1). Par extension on peut ramener les états de tous les composants de l'ordinateur à une forme binaire (interrupteur ouvert/fermé par exemple, diode passante/bloquée, pôle Nord/Sud, etc.)

L'état global de la mémoire de l'ordinateur peut donc être défini comme la **superposition des états de l'ensemble de ses composants à un instant donné**. Ainsi, si on considère un circuit-mémoire 2 bits constitué de 2 interrupteurs l'un ouvert et l'autre fermé, les **mots binaires** 1 0 ou 0 1 permettent de décrire l'état de ce circuit-mémoire.

2 Le codage des données en mémoire

2.1 Du matériel à l'information

Un ordinateur est en premier lieu un système de traitement de l'information. Pour être manipulée, stockée et transformée, cette information doit être codée sous une forme conventionnelle et s'adapter à la structure matérielle de l'ordinateur.
Le microprocesseur étant constitué de portes logiques ne traite que des signaux « tout ou rien » donc une information binaire.
Cette représentation binaire de l'information est dite représentation numérique et s'oppose à la représentation analogique utilisant des évolutions continues de grandeurs physiques. D'une manière générale numériser consiste à coder une information analogique dans un format numérique.

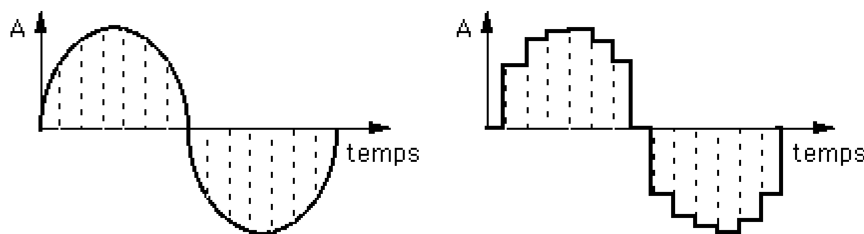


Figure 2: représentation analogique - numérique

Exemple : La TNT (Télévision Numérique Terrestre) remplace désormais la transmission analogique des chaînes de télévision. Dans sa version analogique, un faisceau d'électrons balaye l'écran et le signal analogique reçu de l'antenne module l'intensité du faisceau de façon à reconstituer une image à l'écran. Dans sa version numérique, une suite de 0 et de 1 sont envoyés sous forme d'ondes et décodés pour former une image sur les pixels de l'écran.

Définition : Une information élémentaire binaire est appelée en informatique un bit, diminutif de Binary digIT.

Le support matériel de l'information binaire 0 ou 1 , appelé couche physique, est très divers :

- Aux bornes du microprocesseur : potentiel 0 ou 5V environ
- Sur une clef USB : état actif ou non d'une porte logique
- Sur un disque dur : orientation du moment magnétique de petites cellules
- Sur une fibre optique : présence ou non d'une intensité lumineuse

Définition :
Un octet est un mot binaire de 8 bits.

Pour des raisons de performance, les « cases mémoires » associées à une adresse unique contiennent 8 bits, soit un octet. L'avantage est de limiter la taille du bus d'adresse. L'inconvénient est de ne pas pouvoir allouer moins de 8 bits en mémoire.

Quelques unités

Depuis 1998, la convention électrotechnique internationale recommande d'adopter le système de conversion suivant, afin de respecter les us du système international :

Système international			**Système binaire (pour les octets)**	
1 kilo-octet	10^3 octets		1 kibi-octet (kio)	2^{10} octets
1 méga-octet	10^6 octets		1 mébi-octet (Mio)	2^{20} octets
1 giga-octet	10^9 octets		1 gibi-octet (Gio)	2^{30} octets
1 tera-octet	10^{12} octets		1 tébi-octet	2^{40} octets

Donc avant 1998 : $40\ Mo = 40 \times 2^{20} = 41\ 943\ 040$ octets

Après 1998 : $40\ Mo = 40 \times 10^6$ octets $= 38\ Mio$ (toutes les machines n'ont pas encore adopté cette norme)

2.2 Différents types de données standard

Dans bon nombre de langage de programmation, les variables doivent être déclarées avant de les utiliser, de façon à ce qu'un espace mémoire dans la RAM leur soit alloué. Pour allouer la bonne taille mémoire, cette déclaration inclut le type de la variable. En Python, les variables n'ont pas à être déclarées : le langage utilise le type le mieux approprié et change de type si nécessaire. Cette fonctionnalité apporte du confort au programmeur, au détriment des performances d'exécution du programme.

Nous traiterons ici les types standards suivants :

- Les booléens (bool)
- Les nombres entiers (int)
- Les nombres décimaux (float)
- Les chaînes de caractères (string)

2.2.1 Les booléens

Une variable booléenne est une variable valant vrai ou faux, souvent notés 0 ou 1. Il est donc très simple pour un ordinateur de coder un booléen car il ne travaille lui-même qu'avec des booléens en mémoire.

Sous Python, les états booléens sont décrits par les mots réservés True et False.

Exemple : (le symbole # indique que le contenu qui suit est un commentaire, et non du code).

```
2+2==4     #Le symbole == traduit une égalité (vraie ou fausse). Ce code renvoie True
x = 4<3     #Définit la variable x comme le résultat de 4<3 (donc x vaut False).
type(x)     #renvoie le type de la variable x, donc bool
```

Ce sont nos premières lignes de codes, notez la deuxième ligne, où le symbole = désigne que le membre de gauche (une variable) se voit affecter (« reçoit ») la valeur du membre de droite. Ce symbole = n'a pas en python le sens ni l'usage dont vous avez l'habitude, on y reviendra. En pseudo-code, on note $x \leftarrow 4<3$

2.2.2 Les nombres entiers naturels

Les entiers sont codés en mémoire en utilisant une base (binaire, décimale, hexadécimale…)

Définition :

On fixe un entier naturel $b \geq 1$ (la « base »). Pour tout nombre N entier naturel, il existe une décomposition unique telle que :

$N = \sum_{k=0}^{\infty} \alpha_k b^k$ avec $0 \leq \alpha_k \leq b-1$ tous nuls à partir d'un certain rang.

Ainsi, il est possible de décomposer tout nombre entier naturel de manière unique. Par exemple, en base b=10 (base décimale usuelle) :

$$(392)_{10} = 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

← -
poids croissant

Ces décompositions ne posent généralement aucun problème d'implémentation mais les algorithmes rédigés au début sont parfois assez naïfs et généralement peu efficaces. Souvent, on préfèrera les programmer en utilisant des factorisations successives par b : c'est la **méthode de Hörner**. Cela consiste à écrire :

$$N = \sum_{k=0}^{n} \alpha_k b^k = \left(\left(\left((\alpha_n b + \alpha_{n-1})b + \alpha_{n-2}\right)b + \cdots\right) + \alpha_1\right)b + \alpha_0$$

Ainsi, par exemple : $(1392)_{10} = \left((1 \times 10 + 3) \times 10 + 9\right) \times 10 + 2$

On a fait trois multiplication seulement, donc moins de calculs pour la machine.

La base 10 est constituée de 10 chiffres (de 0 à 9). Au-delà de 10, il faut écrire un nombre en incrémentant la dizaine.

La base 2 fonctionne sur le même principe avec seulement deux chiffres (0 et 1). Un nombre binaire constitué de n chiffres permet de coder 2^n nombres, c'est-à-dire de compter de 0 à 2^n -1.

La base hexadécimale (base 16) est souvent utilisée pour manipuler des mots binaires, car elle permet de représenter 4 bits (donc $2^4 = 16$ possibilités) à l'aide d'un seul chiffre. On complète nos chiffres habituels de 0 à 9 par des lettres naturellement, voir tableau ci-dessous. Par exemple $(F)_{16}=15_{10}$.

Ainsi, un octet (donc 8 bits qui peuvent correspondre à deux nombre en base 16) se représente par deux chiffres hexadécimaux : $(F0)_{16} = (11110000)_2$.

Exercice : Combien cela vaut-il en base 10 ?

Base 10	Base 2	Base 16		Base 10	Base 2	Base 16
0	0000	0		8	1000	8
1	0001	1		9	1001	9
2	0010	2		10	1010	A
3	0011	3		11	1011	B
4	0100	4		12	1100	C
5	0101	5		13	1101	D
6	0110	6		14	1110	E
7	0111	7		15	1111	F

Notations Python : Un langage machine doit bien différencier 11 en base 10 et 11 en binaire (qui vaut 3 en base 10). On fait cela à l'aide d'un préfixe :

- Aucun préfixe pour les nombres en base 10 : 255
- Préfixe 0b pour les nombres binaires : 0b11111111
- Préfixe 0x pour les nombres hexadécimaux : 0xFF
- Préfixe 0o pour les nombres en base octale (c'est-à-dire avec 8 chiffres, de 0 à 7) : 0o377

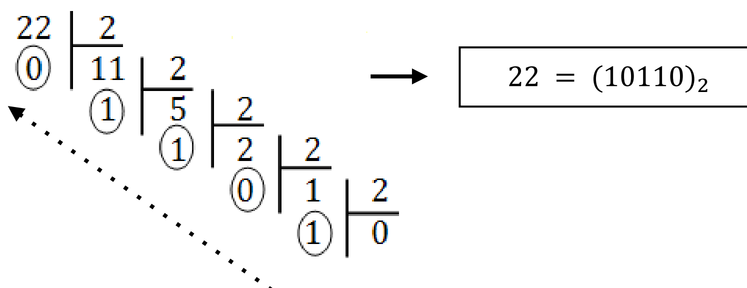
Remarque : L'interpréteur affiche par défaut les nombres en décimal. Pour afficher un nombre au format binaire ou hexadécimal il faut utiliser les fonctions bin() et hex().

Exemple : Conversion de 22 (en base 10) vers la en base binaire :

Une méthode qui fonctionne est d'effectuer une série de divisions euclidiennes par 2, puis de développer :

22=2$\times$ $11 = 2 \times (2 \times 5 + 1) = 2 \times (2 \times (2 \times 2 + 1) + 1) = 2^4 + 2^2 + 2$

Cela peut se schématiser comme suit :



Astuce : La base hexadécimale permet de compter de 0 à 15, un terme représente donc 4 bits (et réciproquement). Pour effectuer une conversion binaire à hexadécimale il faut donc regrouper des

paquets de 4 bits en partant du bit de poids faible (la droite) et le traduire en un décimal puis en hexadécimal. L'opération marche dans le sens inverse en traduisant chaque terme d'un hexadécimale sur 4 bits.

1011110 -> l 0101 l 1110 l -> l 5 l 14 l -> 5E (On remarque l'ajout d'un 0 pour être sur 4 bits)

Exemple : Conversion de $(A1)_{16}$ en base binaire

A1 -> l 10 l 1 l -> l 1010 l 0001l -> 10100001 (On remarque l'ajout de trois 0 devant le 1 afin de rester sur 4 bits et respecter le poids final de chacune de nos valeurs)

Représentation des entiers relatifs négatifs, notation en complément.

Les entiers relatifs sont représentés par des entiers naturels en réservant 1 bit pour le signe. On fixe le nombre de bit, n. Ainsi, sur n $bits$ *on peut coder tous les entiers allant de* -2^{n-1} *à* $2^{n-1}-1$ *(ce qui fait bien* 2^n *valeurs). Mais de quelle manière procéder ? La méthode standard est celle du « complément à 2 » que nous décrivons ci-dessous.*

L'idée est de « décaler vers les négatifs » les codages en binaires des nombres entre à 2^{n-1} *et* à 2^n-1. *Exemple ci-contre avec 3 bits, soit 8 combinaisons cad une représentation de -4 à 3, les codes en binaires des nombres de 4 à 7 ont été utilisés pour coder les nombres de -4 à -1.*

entier positif	bits	entier négatif
	0 0 0	
	0 0 1	
	0 1 0	
	0 1 1	
	1 0 0	-4
	1 0 1	-3
	1 1 0	-2
	1 1 1	-1
0	0 0 0	0
1	0 0 1	1
2	0 1 0	2
3	0 1 1	3
4	1 0 0	
5	1 0 1	
6	1 1 0	
7	1 1 1	

Méthodes générales et exemple :

Situation : On dispose de n bits et on souhaite donner la représentation des *entiers allant de* -2^{n-1} *à* $2^{n-1}-1$. *Avant cela on a besoin du :*

Codage par complément à deux. On inverse tous les bits (complément à un), puis on ajoute 1. En effet, étant donné un nombre entier positif k, effectuer son complément à un revient à calculer 2^n-1-k, et donc son complément à deux revient à calculer2^n-k. Commencer par le vérifier sur le tableau ci-dessus, puis essayer de le montrer.

Codage d'un nombre négatif par complément à deux *: Soit x entre* -2^{n-1} *et 0. On le code en binaire par le complément à deux de* $|x|$.

Pourquoi cela marche-t-il ? Si on note c(x) le complément à deux de $|x|$, *alors on a c(x)+* $|x|$=2^n. Or, si on prend les n premiers bits de 2^n, on trouve 0.

Exemple *: on a 4 bits et on veut représenter -5. On peut représenter tous les nombres allant de* -2^3 *à* 2^3-1 *c'est-à-dire de* -8 *à* 7. On peut donc bien représenter -5 en complément à deux. On suit les étapes ci-dessus :

- On cherche la représentation binaire de $|-5|$ c'est-à-dire 5, ce qui donne 0101 (sur 4 bits)

- On inverse tous les bits obtenus, ce qui donne : 1010 (complément à un)

On ajoute 1 au résultat obtenu, ce qui donne 1011.

Notez qu'on pouvait aussi procéder par décalage, et cherche la représentation standard de $-5 + 2^4$ (décalage) soit $-5 + 16 = 11$

<u>Remarque</u> : Avec cette représentation, le premier bit vous indique bien si vous codez de manière standard un nombre positif entre 0 et $2^{n-1} - 1$ *(le bit vaut 0), et ou si vous codez un nombre négatif (le bit vaut 1).*

2.2.3 Les nombres décimaux

Il existe deux conventions standard pour coder les nombres décimaux :

- la convention à virgule fixe, où une partie des bits sont réservé pour la partie entière et une pour la partie décimale (plus un bit de signe). (Aujourd'hui obsolète sauf dans des cas particuliers)
- La convention à virgule flottante, où une partie des bits est réservés aux chiffres significatifs (mantisse) et une partie pour l'exposant de 2 (plus un bit de signe).

Norme IEEE 754 – Virgule flottante

La convention à virgule flottante est plus complexe en termes d'opérations, mais s'avère beaucoup plus performante pour le calcul numérique : elle permet d'assurer un nombre de chiffres significatifs constant (c'est-à-dire une précision relative constante).

Un nombre à virgule flottante est exprimé sous la forme $\boldsymbol{sm2^E}$ où s est le signe du nombre (0 : positif ou 1 : négatif), E son exposant (entier relatif tenant compte du décalage) et m sa mantisse (nombre à virgule compris entre 1 inclus et 2 exclu). Pas de panique! Lisez, un exemple arrive ! Un flottant peut être codé dans un format simple, double, voire quadruple précision :

Format	Taille (en bits)	Exposant	Précision	E_{min}	E_{max}	Valeur max
Simple	32	8 bits	23 bits + 1	-126	+127	$3.403...10^{38}$
Double	64	11 bits	52 bits + 1	-1022	+1023	$1.798...10^{308}$

En mémoire, ces nombres sont représentés sous la forme :

Bit de signe	Bits d'exposant	Bits de mantisse (dont 1 caché)

Exemple en simple précision 32 bits :

La valeur du nombre est donnée par :

$N = (-1)^s \times (1 + \sum_{i=1}^{23} m_i 2^{-i}) \times 2^{e-E_{max}}$ avec $e = e_1 e_2 \dots e_8$ les bits d'exposant et $E_{max} = 127$ (décalage)

On constate que la somme sur i commence à $i = 1$. En effet, comme le bit tout à gauche de la mantisse est toujours égal à 1 (puisque m est compris entre 1 inclus et 2 exclu), ce bit n'apparaît jamais dans la

représentation de N et le décompte dans la mantisse s'effectue donc à partir du bit $i = 1$ (on l'appelle bit caché).

Prenons par exemple la représentation : N= $1\ \ 10000010\ \ 00110000000000000000000$

Le premier bit permet de calculer le signe, les 8 suivants permettent de calculer l'exposant e, et les 23 derniers la mantisse m.

donc :

- $s = 1$ donc N est de signe négatif
- $e = 2^7 + 2^1$ (somme des bits d'exposant) donc $e - E_{max} = 130 - 127 = 3$
- $m = 1 + 2^{-3} + 2^{-4}$ (bit caché + somme des bits de mantisse) $= 1.1875$

Finalement, on a : $N = (-1)^1 \times 1.1875 \times 2^3 = -9.5$

Remarque : Il n'est pas nécessaire de connaître la norme IEEE 754, ni de savoir déchiffrer un nombre flottant. Néanmoins, on retiendra que les flottants peuvent raisonnablement avoir jusqu'à 7 chiffres significatifs en simple précision et **16 chiffres significatifs en double précision**. On évitera donc toujours d'attendre une précision supérieure de la part des programmes que l'on écrira.

2.2.4 Problèmes liés à la représentation des nombres

S'il est important de savoir comment les nombres sont représentés dans la mémoire de l'ordinateur, il est sans doute encore plus important pour le futur ingénieur de comprendre quelles sont les conséquences directes de ces représentations sur le calcul numérique.

Le premier problème qui se pose à l'informaticien souhaitant pratiquer la simulation numérique vient tout d'abord de la mémoire disponible pour le calcul. On comprend rapidement qu'il va être difficile de représenter le nombre entier 24 avec un circuit-mémoire de 2 bits, on parle alors de **dépassement d'entier** (ou **overflow**).

D'autres problèmes peuvent survenir, notamment avec les flottants, en raison de leur **représentation limitée en mémoire**. En effet, il est impossible de représenter correctement un nombre possédant une infinité de chiffres après la virgule tels que $\sqrt{2}$, π ou tout simplement $\frac{1}{9}$. Nous allons passer en revue quelques uns des problèmes les plus courants qui surviennent en ingénierie numérique lorsqu'on utilise les flottants.

- **Erreurs d'arrondis :**

Il est important de comprendre que les résultats affichés ne sont que des approximations :

```
>>> 0.3 == 0.2 + 0.1       # l'instruction == réalise une comparaison de terme de gauche à celui de droite
False

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

Un autre exemple classique sur les erreurs d'arrondis :

```
>>> 1.2000000000000001 > 1.2
True          #(double précision : 16 chiffres derrière la virgule, les 16 chiffres sont pris en compte)

>>> 1.20000000000000001 > 1.2
False         #(17 chiffres derrière la virgule, le 17ème chiffre est ignoré)
```

Ces problèmes peuvent conduire à des choses étonnantes et même venir contredire certaines lois mathématiques courantes. Illustrons par exemple la perte d'associativité :

```
>>> 0.3-(0.2+0.1)
-5.551115123125783e-17

>>> (0.3-0.2)-0.1
-2.7755575615628914e-17

>>> (2.55*2.06)*1.55 == 2.55*(2.06*1.55)
False
```

- **Erreurs d'absorption :**

Ce problème peut survenir dès que l'on **somme 2 flottants a et b ayant des ordres de grandeur très différents**, typiquement lorsque $\frac{a}{b} > 2^{23}$ en simple précision et $\frac{a}{b} > 2^{52}$ en double précision. Dans une telle situation, le plus petit nombre pourra être absorbé par le plus grand jusqu'à être négligé dans les calculs :

```
>>> 2**30 + 1 – 2**30
1                    #(en double et simple précision)

>>> 2**30 + 1.0 – 2**30
0.0                  #(en simple précision)

>>> 2**30 + 1.0 – 2**30
1.0                  #(en double précision)


>>> 2**53 + 1.0 – 2**53
0.0                  #(en simple et double précision)
```

- **Erreurs de cancellation (ou annulation) :**

Il s'agit du problème inverse de la situation précédente. Lorsqu'on **soustrait 2 flottants a et b d'ordres de grandeur proches**, certains bits servant à la représentation de ces nombres peuvent se compenser et conduire à des résultats totalement erronés.
On comprend bien ce phénomène en prenant un exemple trivial en base décimale :

$a = 13{,}2349$. On imagine que « faute de mémoire suffisante » le résultat est arrondi à $13{,}235$ (voir erreurs d'arrondis)
$b = 13{,}2347$. De la même façon le résultat est arrondi à $13{,}235$
Alors $a - b = 0$ (et $0{,}0002$ a été simplement « ignoré » du calcul). L'erreur peut alors se propager si un calcul est effectué à partir de ce résultat (division, exponentiation, etc…).

On pourrait imaginer que cette erreur est anecdotique mais il n'en est rien car ramenée à un décalage de bits cela peut conduire à des aberrations, dont voici un exemple avec l'identité remarquable $(x + 1)^2$:

```
def f(x):
        return (x+1)**2 - x**2 - 2*x - 1

>>> f(10000000000)        # On remplace x par 10000000000 dans la fonction f définie
0                         # Avec un entier, le problème ne se pose pas.

>>> f(1E10)
-2049.0                   # Constater l'ampleur des dégâts avec un flottant…
```

- **Propagation des erreurs :**

Lorsqu'un calcul isolé est réalisé, on peut généralement admettre (sauf exception) que les résultats obtenus sont à peu près fiables. Cependant, on constate qu'à chaque nouvelle opération sur le résultat, le nombre de chiffres significatif diminue jusqu'à conduire parfois à des résultats totalement aberrants.

```
>>> import math
>>> math.sqrt(100)
10.0                        (résultat à priori fiable)
```

```
>>> math.sqrt(exp(log(100)))          # « log » désigne  sous Python le logarithme népérien
« ln (x) »
10.000000000000002                    (17 chiffres significatifs)

>>> math.sqrt(exp(log(1000)-log(10)))
9.999999999999998                     (16 chiffres significatifs)
```

Finalement, on retiendra que la représentation des nombres en mémoire n'est pas sans poser de problème en ingénierie numérique. Dans le courant de l'année, nous rencontrerons parfois des situations où ces problèmes peuvent devenir très handicapants et conduisent à des erreurs importantes (comme par exemple un algorithme qui ne se termine pas alors que la preuve de la terminaison en est faite). Ces erreurs ne sont pas à prendre à la légère et contourner les situations qui les engendrent est parfois un vrai défi en soi.

On retiendra 2 règles simples :
→ Toujours considérer un résultat provenant d'opérations avec des flottants comme étant une approximation résultant d'une **représentation limitée en mémoire**.

→Ne **<u>jamais utiliser de comparaisons entre des flottants</u>**. C'est un non-sens grave en algorithmique !

2.2.5 Les chaînes de caractères

Le codage des caractères en mémoire relève d'une simple association entre un code binaire et un symbole. Beaucoup de standards existent, nous nous attacherons à décrire le standard ASCII, ancien mais très connu, puis plus sommairement, le principe des codes modernes comme l'UTF8 et l'Unicode.

L'ASCII est un code définissant une liste de caractères codés sur 7 bits, soit 128 caractères. Il s'agit des caractères latins principaux permettant d'écrire en anglais (il n'y a donc pas d'accents). Les 32 premiers caractères ainsi que le dernier sont des caractères de contrôle (retour à la ligne, tabulation, effacement etc…)

Il n'y a donc que 95 caractères affichables donc la majeure partie se trouvent sur les claviers :

- Les lettres (majuscules et minuscules), 52 caractères
- Les chiffres, 10 caractères
- La ponctuation (espace ! « ‘() , . : ; ? [] `), 16 caractères
- Les opérations (* + - / =), 5 caractères
- Quelques caractères spéciaux (# $ % & < > @ ^ _ | ~), 16 caractères

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

Figure 3: Table ASCII - code hexadécimal

Le code n'utilise que 7 bits alors que les octets en mémoire en contiennent 8, un certain nombre d'extension ASCII existent, permettant de coder les caractères accentués de différentes langues entre les caractères 129 et 256.

Exemple : Ecriture du mot « Informatique » en code hexadécimal :

49 6E 66 6F 72 6D 61 74 69 71 75 65

L'ASCII ou son extension ne sont pas suffisants pour coder les caractères spéciaux des différentes langues. Unicode a été créé en 1996 afin de permettre de définir un code unique pour un maximum d'un million de caractères différents (actuellement seul 250 000 caractères sont définis).

L'UTF-8 permet de coder n'importe quel caractère sur 1 à 4 octets. De façon à optimiser l'espace mémoire, les caractères les plus courants (les ASCII) sont codés sur 1 octet (ce qui assure une compatibilité avec l'ASCII) tandis que les caractères les plus rares sont codés sur 2, 3 ou 4 octets.

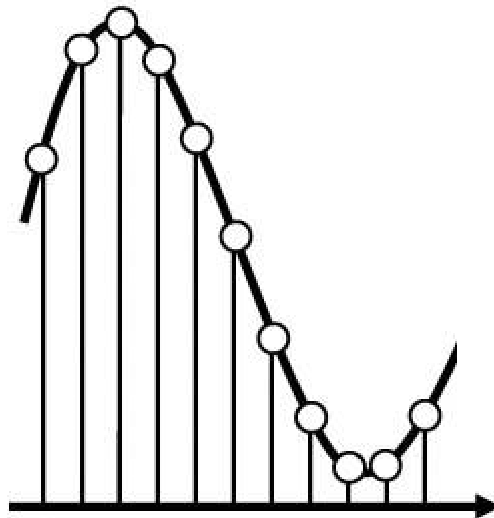
2.3 Codage des données complexes

2.3.1 Codage des signaux et courbes

Les signaux analogiques et les courbes d'une façon générale ne peuvent pas être mémorisés en tant que tel dans un système numérique. Il est nécessaire de les numériser. Cette opération implique généralement deux phases :

- L'échantillonnage : évaluation de la courbe sur un nombre fini de points en abscisse
- La quantification : approche de la valeur en ordonnée dans un ensemble fini de valeur

La représentation d'une courbe se fait donc sous la forme d'un tableau de points de coordonnées x_i et y_i. La représentation de la courbe est alors une polyligne reliant les points du tableau.



Plus le nombre de point augmente, plus la courbe apparaît parfaitement et plus le calcul risque d'être long…

2.3.2 Codage du son

Le son est une onde de pression. Son amplitude caractérise le volume et les fréquences les « notes ». Un micro permet de convertir l'onde de pression en signal électrique lui-même converti en signal numérique (par un convertisseur analogique-numérique). Un morceau musical stocké dans l'ordinateur subit la transformation inverse via un haut-parleur (et un convertisseur numérique-analogique).

Le son est donc codé comme un signal, donc comme une liste de points régulièrement répartis au cours de temps.

2.3.3 Codage des images

Les images bitmap forment une représentation numérisée d'une image ou d'une photographie sous forme de réseaux de pixels. La numérisation conduit à une discrétisation spatiale (suivant x et y) et une quantification de couleurs. La discrétisation spatiale est un découpage en pixels. La couleur de chaque pixel est mémorisée sous la forme de 3 niveaux de couleurs rouge, vert, bleu (RVB) généralement codés sur un octet chacun.

Table des matières

1 Les composants et leurs connexions 1
1.1 Ordinateur 1
1.2 Architecture interne 1
1.3 Composants et communication binaire 2
2 Le codage des données en mémoire 3
2.1 Du matériel à l'information 3
2.2 Différents types de données standard 4
2.2.1 Les booléens 4
2.2.2 Les nombres entiers naturels 4
2.2.3 Les nombres décimaux 8
2.2.4 Problèmes liés à la représentation des nombres 9
2.2.5 Les chaînes de caractères 12
2.3 Codage des données complexes 13
2.3.1 Codage des signaux et courbes 13
2.3.2 Codage du son 13
2.3.3 Codage des images 14