Information, Calcul et Communication
Représentation de l'information

R. Boulic & J.-C. Chappelier

EPFL

Objectif du cours d'aujourd'hui

Répondre aux questions suivantes :

- Existe-t-il une représentation universelle de l'information ?
- Par quels moyens peut on **représenter** des symboles et **des nombres** ?
- Est-il possible de construire une représentation exacte du monde réel ?

Plan du cours d'aujourd'hui

Quelle(s) représentation(s) ?

- Rappel des domaines d'applications
- Une représentation est une convention
- Vers l'unité élémentaire d'information (bit)

Représentation des nombres entiers

- Entiers positifs
- Entiers positifs et négatifs

Représentation des nombres décimaux

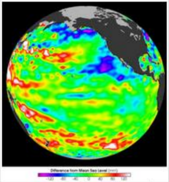
- La virgule flottante : Pourquoi ? Comment ?
- Erreur relative contrôlée
- Au voisinage de 0

Annexe : Représentation des symboles

- De l'alphabet aux idéogrammes

Lien avec les leçons précédentes

Domaines d'application



En 2003, mise en évidence du "Niño" par simulation numérique de la circulation océanique - © *INRIA / Projet IDOPT*

Calcul scientifique /Simulation

-> nombres

Information ?

Gestion d'information

-> texte

Conduite de processus

-> signaux (mesures, contrôle...)

Google datacenter

Une représentation est une convention

- pour faciliter l'activité d'un groupe d'utilisateurs
- correspondance entre un ensemble de signes et leur signification.

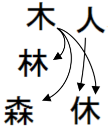
☞ Fragilité de cette convention : langues mortes, codes perdus,etc...

Il n'existe pas de représentation universelle :

- standard de facto = porté par le marché, l'usage (ex : pdf)
- standard de jure = normalisation (IEEE, ACM, ISO...).

Exemples : alphabet romain, chiffres arabes, code de la route, papier monnaie

Vers l’unité élémentaire d’information



214 motifs graphiques (appelés « clés ») ont été utilisés pour construire ≃100’000 idéogrammes chinois

(Lesquels idéogrammes sont utilisés pour exprimer des mots plus complexes : 3–4 idéogrammes par mot)

A B C ... Les **26 lettres** de l’alphabet latin ont été utilisées pour construire ≃1’000’000 de mots (langues occidentales)

123... Les **10 chiffres** indo-arabes permettent de construire une infinité de nombres
(et par là même encoder tous les mots existants)

QUESTION : quel est le plus petit alphabet permettant de représenter *efficacement* tous les nombres (entiers) ?

Le binaire

Limites de l'écriture unaire :
un alphabet *unaire* permet de représenter tous les nombres :
|, | |, | | |, | | | |, | | | | |, ...
mais la taille de l'écriture est proportionnelle au nombre écrit !

$$|\mathrm{rep}_1(n)| \in \Theta(n)$$

Un tel système d'écriture n'est pas **efficace**.

Le système *binaire* est le système de signes le plus simple pour représenter des nombres de façon *efficace* (en $\log(n)$ symboles)

$$|\mathrm{rep}_2(n)| \in \Theta(\log n)$$

Unité élémentaire d'information : le bit

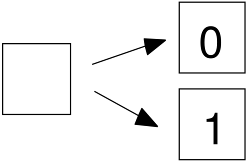
Toute information peut être représentée à l'aide d'une **suite d'éléments binaires** (appelé « *motif binaire* »).

Par convention, nous choisirons `0` et `1` comme éléments binaires.

L'expression anglaise « *binary digit* » a été abrégée « **bit** » pour désigner un tel élément.

Note : Dans cette leçon, nous faisons abstraction de la manière dont les éléments binaires sont réalisés (états magnétiques, tensions, courants, etc.). Cela sera abordé dans le Module 3.

1 bit

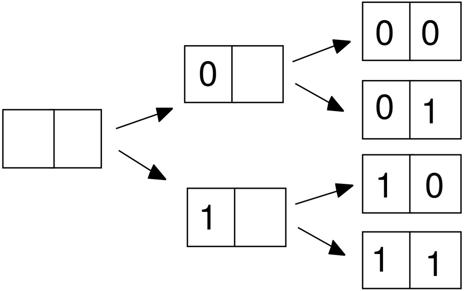


2^1 informations distinctes

Capteurs à seuil

Comment représenter plus d'informations ?

2 bits



2^2 informations distinctes

♣

♠

♥

♦

n bits permettent de représenter **2^n** informations distinctes

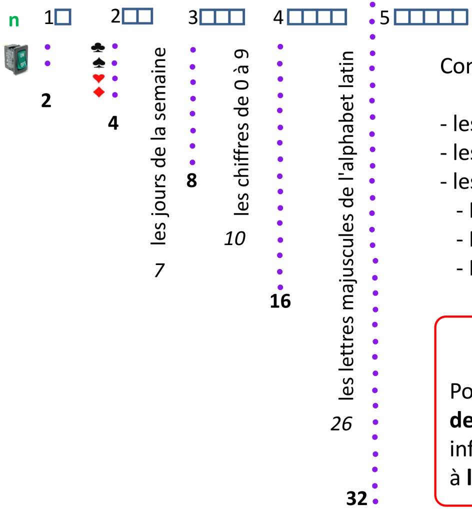
n	2^n
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
10	1024
20	1048576
30	1073741824
32	4294967296

Bonne pratique pour estimation rapide:

2^{10} = kibi (Ki) $\approx 10^3$ = kilo (k)
2^{20} = mébi (Mi) $\approx 10^6$ = méga (M)
2^{30} = gibi (Gi) $\approx 10^9$ = giga (G)
$2^{32} = 2^{30+2} = 2^{30} .2^2 \approx 4$ G

n bits permettent de représenter **2^n** informations distinctes

réciproquement, **2^n** informations distinctes sont représentables par $\log_2(2^n) = n \log_2(2) = n$ bits



Exercice:

Combien de bits suffisent pour représenter :

- les jours de la semaine :
- les chiffres de 0 à 9 :
- les lettres de l'alphabet:
  - Majuscules
  - Minuscules + Majuscules
  - Min + Maj + chiffres + signes ...

règle générale:

Pour **K informations distinctes**, le **nombre de bits n** suffisant pour représenter ces informations est l'entier supérieur ou égal à **$\log_2 K$** : $n = \lceil \log_2 K \rceil$

Organisation de l'information

Il est plus simple de lire

234, 149

que

`11101010, 10010101`

La complexité cognitive de la lecture de messages écrits en binaire étant trop grande pour un humain normalement constitué, des *organisations plus compactes* ont été mises en œuvre, comme par exemple des séquences de 8 bits appelée « **octet** » (« *byte* » en anglais)

Un octet peut donc représenter $2^8 = 256$ informations.

Note : en français on utilise la lettre 'o' pour mesurer les octets : 200 Mo, en anglais c'est la lettre 'B' à ne pas confondre avec 'b' (pour « bit ») :
2.36 Mib = 302 kiB

Plan

Quelle(s) représentation(s) ?

☞ **Représentation des nombres entiers**

- Entiers positifs
- Entiers positifs et négatifs

Représentation des nombres décimaux

- La virgule flottante : Pourquoi ? Comment ?
- Erreur relative contrôlée
- Au voisinage de 0

Annexe : Représentation des symboles

- De l'alphabet aux idéogrammes

Comment représenter un nombre entier ?

Commençons par les entiers naturels (= positifs et nul)

Rappel : tout entier naturel peut être représenté à l'aide d'un *motif binaire* (suite d'éléments binaires)

Mais un motif binaire isolé est insuffisant pour comprendre ce qui est codé !

☞ Il faut une **convention d'interprétation** du motif binaire

Une solution : la notation positionnelle en base 2

Notation positionnelle des nombres

Exemple d'un nombre entier en base 10 :
le nombre **703** est la notation abrégée de l'expression :

$$\mathbf{7} \cdot 10^2 + \mathbf{0} \cdot 10^1 + \mathbf{3} \cdot 10^0$$

- Le chiffre de droite est toujours associé à la puissance 0 de la base 10
- La puissance de la base augmente d'une unité de chiffre en chiffre, en allant de la droite vers la gauche

Cette convention de **notation positionnelle** peut être exploitée dans n'importe quelle base

Notation positionnelle en base 2

La notation positionnelle en base 2 utilise exactement les mêmes convention qu'en décimal (base 10)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

poids forts à gauche

poids faibles à droite

Pratique: Conversions

Du binaire vers le décimal: additionner les puissances de 2 présentes dans le motif binaire

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	1	0	1	1

$$0 + 0 + 0 + 0 + 8 + 0 + 2 + 1 = 11$$

Du décimal vers le binaire :
décomposer un nombre entier X en une somme de puissances de 2 par division entières successives tant que le quotient ≥ 2

$$\begin{aligned}
11 &= 2 \cdot 5 + 1 \\
&= 2 \cdot (2 \cdot 2 + 1) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot 1 + 0) + 1) + 1 \\
&= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1011
\end{aligned}$$

Écriture en binaire

écriture en binaire

entrée : $n \in \mathbb{N}$

sortie : *écriture binaire de n*

$L \longleftarrow ()$ *// liste vide*

Répéter

  Si *n est pair*

    $L \longleftarrow 0 \oplus L$ *// ajouter 0 devant*

  Sinon

    $L \longleftarrow 1 \oplus L$ *// ajouter 1 devant*

  $n \longleftarrow \left\lfloor \frac{n}{2} \right\rfloor$

tant que $n > 0$

Sortir : L

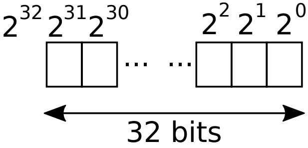
Entiers : domaine couvert

Une représentation destinée à une machine est associée à une capacité fixe exprimée en nombre de bits (d'octets).

Exemple : entier manipulé par une machine « 32 bits ».
Une telle machine dispose d'instructions pour réaliser très rapidement les opérations de base (addition, multiplication, etc.) pour des entiers représentés sur 4 octets.

☞ *limitation* du nombre d'entiers représentables = 2^{32}

Si cette représentation est uniquement destinée aux nombres positifs, son **domaine couvert** est alors (pour 32 bits) :



Min = motif binaire avec des 0 partout
= **zéro**

Max = motif binaire avec des 1 partout
= $2^{32} - 1$

Entiers : domaine couvert (2)

Les calculs sur des entiers sont exacts si le résultat correspondant

- est un **entier**
- et **appartient au domaine couvert**

Plusieurs causes possibles de **dépassement de capacité** :

- division : perte de partie fractionnaire (lorsque le résultat n'est pas un entier)
- multiplication, addition, soustraction : sortie du domaine couvert p.ex. pour entiers positifs sur 32 bits : lorsque le résultat n'est pas compris entre 0 et $2^{32} - 1$

Exemples de dépassement de capacité

Exemple 1 : addition de 2 entiers sur **un seul bit** ☞ 4 cas :

0	0	1	1
+0	+1	+0	+1
--	--	--	--
0	1	1	$\boxed{1}$0
			la retenue est perdue

☞ la retenue est perdue dans le dernier cas : dépassement de capacité

Exemple 2 : addition sur **32 bits** : dépassement sur $(2^{32} - 1) + 1$:

2^{32}	2^{31}																															2^0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
~~1~~	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Plan

Quelle(s) représentation(s) ?

Représentation des nombres entiers

- Entiers positifs
- ☞ **Entiers positifs et négatifs**

Représentation des nombres décimaux

- La virgule flottante : Pourquoi ? Comment ?
- Erreur relative contrôlée
- Au voisinage de 0

Annexe : Représentation des symboles

- De l'alphabet aux idéogrammes

Entiers négatifs : première tentative

Le signe d'un nombre est une information binaire ($+$ ou $-$).
Il suffit donc d'un bit pour le représenter.

Exactement comme nous le faisons en décimal, on pourrait écrire le signe (par convention : `0` pour $+$, `1` pour $-$) *devant* la valeur absolue.
Par exemple (sur 4 bits) :

$-$ 5	(décimal)
`1 101`	(binaire)

Propriétés :

- 1 bit étant réservé pour le signe, le domaine couvert va de $-2^{n-1} + 1$ à $2^{n-1} - 1$ (p.ex. sur 32 bits, de $-2^{31} + 1$ à $2^{31} - 1$)
- Symétrie parfaite du domaine couvert, mais 2 représentations pour 0
- La soustraction **ne** peut **pas** être effectuée en additionnant l'opposé d'un nombre

Problème de l'écriture « signe et valeur absolue »

Le problème de fond de la représentation précédente est qu'additionner l'opposé est différent de la soustraction !

Exemple :

$$
\begin{aligned}
11 &: \texttt{00001011} \\
-11 &: \texttt{10001011} \\
\text{somme des deux} &: \texttt{?0010110}
\end{aligned}
$$

☞ plus ou moins 22, mais en tout cas pas 0 !

Comment représenter les nombres pour que rep(11) + rep(-11) = rep(0) ?

Entiers : dépassement de capacité et représentation des nombres négatifs

Question: comment tirer partie d'une capacité limitée de n bits pour en déduire une représentation des entiers négatifs permettant de remplacer la soustraction par l'addition de l'opposé ?

Rappel: n bits permettent de représenter 2^n nombres entiers positifs de 0 à $(2^n - 1)$

La valeur 2^n elle-même n'est pas représentable sur n bits, on a:

$$(2^n - 1) + 1 = 2^n \quad \text{(en théorie)}$$

Mais $$(2^n - 1) + 1 = 0 \quad \text{(sur n bits)}$$

Conséquence: le motif binaire de $(2^n - 1)$ est une bonne représentation de -1 car on obtient 0 quand il est ajouté à 1

Représentation des entiers signés

<u>Propriétés à vérifier:</u> si a et b sont deux nombres opposés

Alors :

$$a + b = 0$$

de plus $-(-a) = a$

Avec n bits de capacité, on pose que l'opposé d'un nombre **x** est donné par l'expression $\mathbf{2^n - x}$ appelée le Complément à 2 de **x**.

Pour a (c.-à-d. l'opposé de b), posons donc $a = 2^n - b$.

Alors : $a + b = (2^n - b) + b = 2^n = 0$ (sur n bits)

de plus $-(-a) = 2^n - (2^n - a) = a$

Pratique: comment calculer l'opposé (2^n – **x**) d'un entier **x** ?

Une transformation est nécessaire pour :

- faire apparaitre des quantité représentables sur n bits
- les manipuler à l'aide d'opérations simples

$$2^n - x$$
$$= 2^n \underline{-1 + 1} - x$$
$$= (\underline{(2^n -1)} - x) + 1$$

1	1	1	1	1	1	1	1

<u>Cas particulier</u> : ($(2^n -1) -$ **x**) est très facile à obtenir !

Il suffit d'<u>inverser</u> chaque bit de **x** : 0 -> 1 ou 1 -> 0
Cette valeur est appelée le <u>Complément à 1 de **x**</u>.

<u>Complément à 2 de **x**</u> = <u>Complément à 1 de **x**</u> + 1

Exemple1: Complément à 1 du nombre onze en binaire = (2^n-1) - onze

0	0	0	0	1	0	1	1	motif binaire de onze

	1	1	1	1	1	1	1	1	2^n-1
-	0	0	0	0	1	0	1	1	soustraction de onze

1	1	1	1	0	1	0	0	complément à 1 de onze

Exemple2: calcul de l'opposé de onze avec son complément à 2=(complément à 1)+1

	0	0	0	0	1	0	1	1	motif binaire de onze
	↓	↓	↓	↓	↓	↓	↓	↓	
	1	1	1	1	0	1	0	0	complément à 1 de onze
+	0	0	0	0	0	0	0	1	addition de + 1
=	1	1	1	1	0	1	0	1	complément à 2 de onze = opposé de onze
+	0	0	0	0	1	0	1	1	addition de onze
	1	1	1	1	1	1	1		
	0	0	0	0	0	0	0	0	0 sur n bits

Entiers signés : domaine couvert

On a donc à ce stade :

11 : `00001011`
-11 : `11110101`

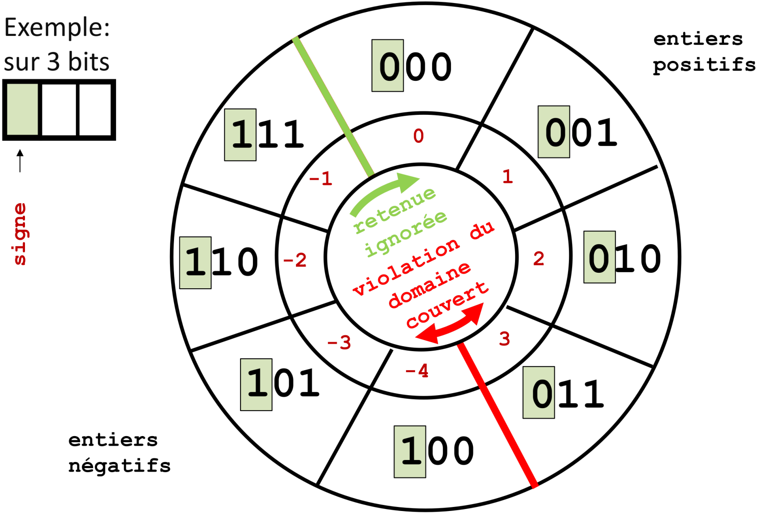
Quelques autres valeurs :

`00000000` : 0	
`00000001` : 1	`11111111` : -1
`01111111` : $2^7 - 1 = 127$	`10000001` : $-2^7 + 1 = -127$
	`10000000` : $-2^7 = -128$

Domaine couvert : de -2^{n-1} à $2^{n-1} - 1$

ATTENTION : il s'agit bien d'une *nouvelle* convention : de fait, le bit de poid fort est utilisé pour le signe :
`10000001` ne s'interprète plus comme 129, mais bien comme -127

Entiers signés : domaine couvert



Résumé à ce stade

Bilan : **DEUX** conventions *différentes* vues jusqu'ici :

- représentation non signée : de 0 à $2^n - 1$
- représentation signée : de -2^{n-1} à $2^{n-1} - 1$

(sur *n* bits)

Cela correspond en C++ aux **deux** types *différents* :
`unsigned int` et `int`, respectivement

L'addition se fait **de la même façon** dans les deux cas ;
c'est *l'interprétation* du motif binaire résultant qui est différente.

Plan

Quelle(s) représentation(s) ?

Représentation des nombres entiers

- Entiers positifs
- Entiers positifs et négatifs

☞ **Représentation des nombres décimaux**

- La virgule flottante : Pourquoi ? Comment ?
- Erreur relative contrôlée
- Au voisinage de 0

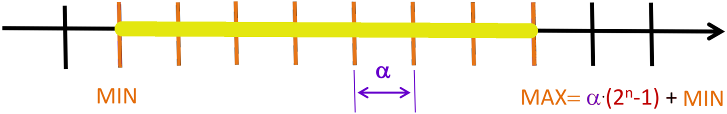
Annexe : Représentation des symboles

- De l'alphabet aux idéogrammes

Représentation à virgule fixe

Avec n bits, on peut représenter 2^n valeurs.

En représentation **à virgule fixe** sur n bits, les 2^n valeurs représentées sont *uniformément réparties* dans un intervalle [MIN, MAX] fixé et séparées par la quantité α :



$$\alpha = \frac{\text{MAX} - \text{MIN}}{2^n - 1}$$

Remarque : il existe un nombre infini de nombres à virgule à l'intérieur de [MIN, MAX] qui sont donc représentés de façon *approchée* par l'une des 2^n valeurs représentées.

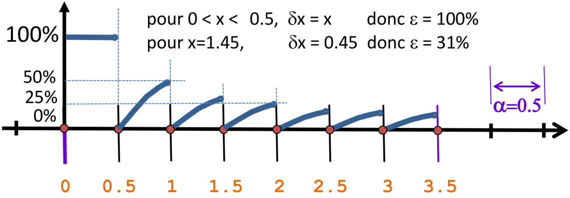
☞ production d'une **erreur absolue de quantification** (ou « de discrétisation ») au maximum égale à α.

Erreur relative sur le domaine couvert

L'erreur relative décrit l'importance relative de l'erreur de discrétisation **δx** par rapport au nombre à représenter **x**.
|δx|/|x| <u>n'est pas uniforme</u> sur le domaine couvert.

<u>Exemple</u>: avec 3 bits et α =0.5, le domaine couvert est [0, 3.5]. Seules 8 valeurs sont représentées exactement: 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5 .
L'erreur relative (en %) est importante lorsque **δx** est grand par rapport à **x**.

Erreur relative ε en %
pour une approximation
par <u>troncation</u>



Cette répartition hétérogène de l'erreur relative n'est pas acceptable pour de nombreux problèmes

Erreur relative uniforme ?

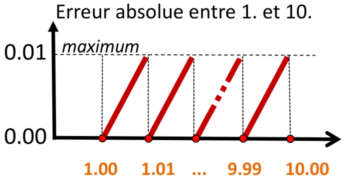
Le compromis retenu: garantir une erreur relative uniforme veut dire <u>accepter une croissance de l'erreur de discrétisation</u> au sein du domaine couvert !

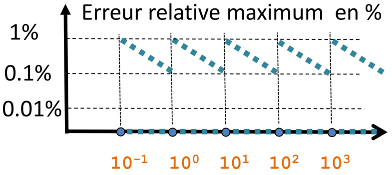
<u>Inspiration</u>: notation scientifique en base 10 avec un nombre fixe de chiffres significatifs.

Exemples avec 3 chiffres significatifs :

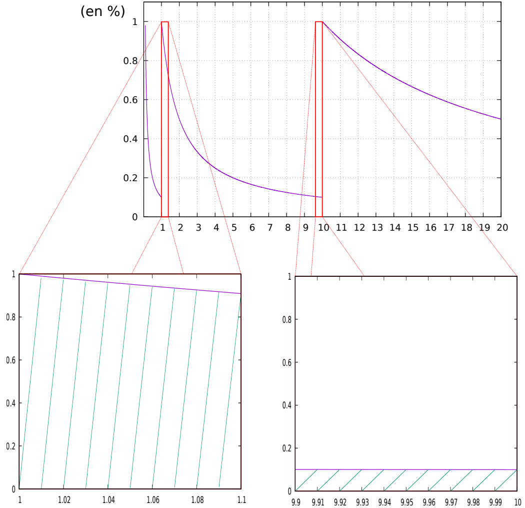
3.1415	s'écrit	$3.14\ 10^{0}$
0.0125	s'écrit	$1.25\ 10^{-2}$
7354	s'écrit	$7.35\ 10^{3}$

l'erreur relative est (presque) uniforme.
le pire des cas est en début d'intervalle $[10^k, 10^{k+1}[$ lorsque $\delta x = 0.01 * 10^k$
pour $x = 1.00999.. * 10^k$
ce qui donne $\varepsilon = \delta x / x \approx 0.01 = 1\%$





Erreur relative en notation scientifique (à 3 chiffres significatifs)



La représentation en virgule flottante

= une notation scientifique en base 2

Représentation flottante en base 2: comporte 3 parties qui se partagent le nombre de bits à disposition: le **signe**, **l'exposant** de la base 2 et le **nombre normalisé** en base 2. La partie fractionnaire du nombre normalisé est appelée la **mantisse**.

Particularité de la base 2: le chiffre le plus significatif du nombre normalisé est constant et toujours égal à 1. Il est donc implicite.

$$\textbf{signe} \cdot 2^{\textbf{exposant}} \cdot 1,\textbf{mantisse}$$

Comme pour la notation scientifique, l'erreur relative maximum est définie par la puissance la plus faible de la mantisse.

n bits de mantisse ☞ 2^{-n}

Exemple

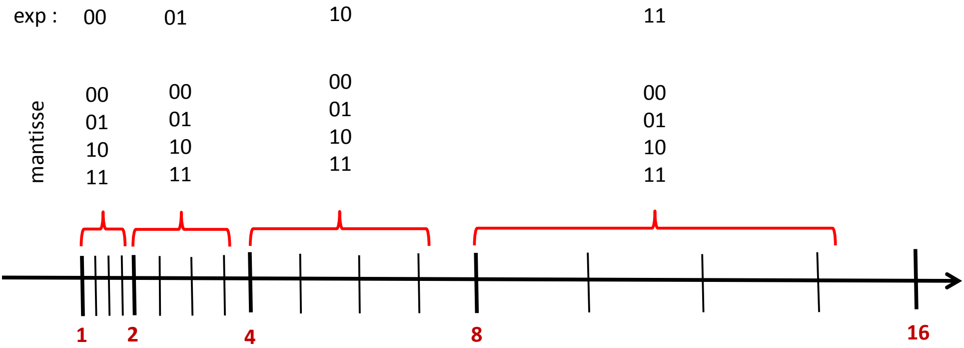
Supposons que l'on ait 2 bits d'exposant et 3 bits pour la mantisse.
(dans l'ordre : signe, exposant, mantisse)

Que représente alors `101011` ?

$\texttt{101011} = \texttt{1 01 011} = -2^1 \times 1,011 = -(2+1/2+1/4) = -2.75$

La représentation en virgule flottante

Exemple avec 2 bits pour l’exposant et 2 bits pour la mantisse. On a la forme: $2^{\square\square} \cdot 1,\square\square$. L’erreur relative est au plus de $2^{-2} = ¼$.



Et vers 0 ?

Le plus petit nombre positif représentable à ce stade est alors :
0 pour l'exposant et
0 pour la mantisse

soit donc $2^0 \times 1,0 = 1$! !

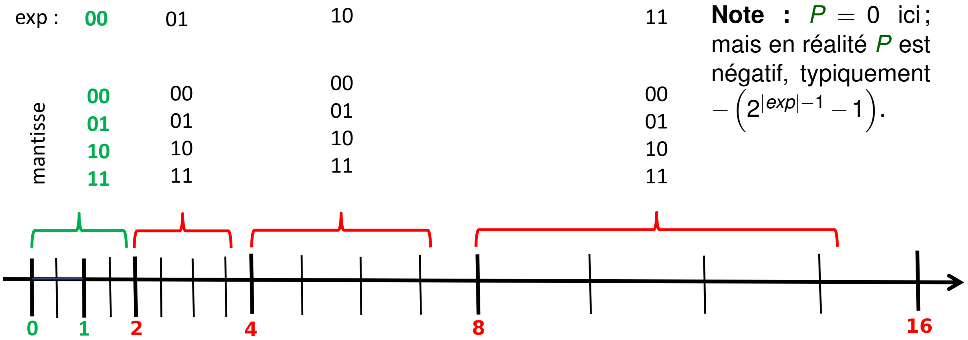
Comment représenter des nombres plus petits ? (voire 0 lui-même) ?

Trois changements par rapport à ce qui précède :

- La mantisse est interprétée différemment lorsque l'exposant est `00..0`
- Cet exposant `00..0` ne représente pas 0, mais un nombre $P \leq 0$ (la valeur de P fait partie de la convention de représentation)
- Tous les exposants sont « décalés de P » : `00..0` représente P, `00..1` représente $1+P$, ..., `11..1` représente $2^{|exp|}-1+P$ ($|exp|$ est le nombre de bits pour l'exposant)

La représentation de 0 en virgule flottante

Même exemple, mais pour inclure 0 on traite la plus petite puissance de la base, notée P, comme un cas particulier qui permet de couvrir l'intervalle $\left[0, 2^{P+1}\right[$ avec la formule : $2^{P+1} \times 0,$**mantisse**



Note : $P = 0$ ici ; mais en réalité P est négatif, typiquement $-\left(2^{|exp|-1} - 1\right)$.

Autre exemple

Supposons que l'on ait 3 bits d'exposant, 4 bits pour la mantisse et $P = -(2^{3-1} - 1) = -3$.

Que représente alors `00000110` (dans l'ordre : signe, exposant, mantisse) ?

`00000110` = `0 000 0110`

Comme l'exposant est ici `000`, on interprète alors la mantisse différement : `0,0110`

soit $(1/4 + 1/8) = 0.375$

et le nombre comme $2^{P+1} \times 0.375 = 2^{-2} \times 0.375 = 0.09375$

0 lui-même est représenté avec simplement que des `0` : `0000000`

L'erreur d'arrondi est la règle

On appelle aussi **erreur d'arrondi** l'écart entre un nombre *x* et sa représentation approchée en virgule flottante.

Exercice : quel est l'écriture binaire de « un dixième » ?

`0.0 0011 0011 0011 0011 0011 0011 0011 0011 0011 ...`

☞ « un dixième » ne peut pas être représenté de manière exacte avec un nombre fini de bits

Remarques :

- Ce problème existe dans toutes les bases, pensez à « un tiers » en base 10.
- Même si l'erreur d'arrondi est inévitable pour la majorité des nombres, on peut **garantir qu'elle se situe en dessous d'un seuil défini** par les besoins du problème traité, en choisissant une représentation adaptée.

Erreurs d'arrondis : un exemple

Exemple : $a \cdot x^2 + b \cdot x + c$
avec les valeurs *décimales* $a = 0.25$, $b = 0.1$, $c = 0.01$

Discriminant : $\Delta = b^2 - 4ac$

en théorie Δ est nul pour cet exemple

Mais sur machine (représentation et calculs sur 64 bits) :

$$\Delta = 9.02 \cdot 10^{-19}$$

- 4 et 0.25 sont représentés exactement ; leur produit donne 1 (qui est aussi correctement représenté)
- Par contre 0.1_{10} et 0.01_{10} sont représentés par des valeurs approchées
- Conséquence : sur les calculs effectués en binaire $0.1_{10} \times 0.1_{10} \neq 0.01_{10}$

Conséquence des erreurs d’arrondi

Tester l’égalité de résultats en virgule flottante est une absurdité.

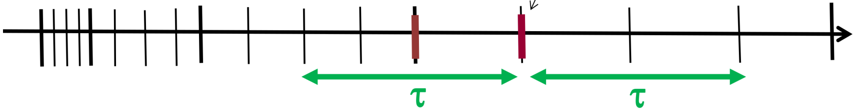
En règle générale, le résultat obtenu est (légèrement) différent de la valeur théorique.

Le test d’égalité de valeur de résultats en virgule flottante doit être fait avec **tolérance autour** de la valeur théorique :

$$| \text{résultat} - \text{valeur\_théorique} | < \tau$$

τ dépend de :

- la valeur théorique
- l'algorithme de calcul du résultat

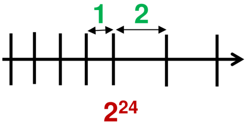


La valeur exacte de la tolérance est difficile à prévoir et dépend des calculs effectués. Mais si quelques ordres de grandeur au dessus de l’erreur relative de représentation sont acceptables, c’est une bonne valeur pour cette tolérance.

Conséquence des erreurs d'arrondi

Le résultat est différent selon l'ordre des opérations.
L'addition n'est plus associative :
il existe des valeurs a, b, c telles que $(a+b)+c \neq a+(b+c)$

Exemple avec un standard sur 32 bits
possédant 23 bits de mantisse:



$(2^{24} + 1) + 1$ -> $2^{24} + 1$ -> 2^{24}
$2^{24} + (1 + 1)$ = $2^{24} + 2$ qui est représenté

☞ **Bonne pratique :** d'abord additionner les petits nombres entre eux, avant de les additionner aux plus grands.

La maîtrise de la précision est possible

Pour un problème donné et son algorithme de résolution, il est important de se poser les questions suivantes :

- De quelle précision ai-je besoin pour mes résultats ?
- Quelle est l'influence de l'algorithme sur la précision des résultats ?
- Quelle est la précision maximum disponible sur la machine cible ?

En cas de précision insuffisante, il faut reconsidérer l'algorithme de résolution et/ou adapter la représentation pour obtenir une précision désirée.

☞ Compromis : Précision / Coûts calcul et mémoire

Résumé

Existe-t-il une représentation universelle de l'information ?

Une représentation est une **convention** humaine d'interprétation d'un ensemble de signes. Sa force est directement liée au nombre de personnes qui la partage, d'où l'importance des **standards** (ex : code ASCII, UTF).

Par quels moyens peut on représenter des symboles et des nombres ?

La **représentation binaire** suffit pour représenter de façon efficace ($\log$) un nombre arbitrairement grand de signes.
Par convention nous utilisons les symboles 0 et 1.

Est il possible de construire une représentation exacte du monde réel ?
Les calculs avec la *représentation positionnelle entière* donnent des résultats exacts pour autant qu'on reste dans le **domaine couvert**.

Pour la *représentation à virgule flottante*, il faut se poser la question de la précision dont on a besoin. La représentation peut être **adaptée pour garantir une précision relative désirée**.

Plan

Quelle(s) représentation(s) ?

Représentation des nombres entiers

- Entiers positifs
- Entiers positifs et négatifs

Représentation des nombres décimaux

- La virgule flottante : Pourquoi ? Comment ?
- Erreur relative contrôlée
- Au voisinage de 0

☞ **Annexe : Représentation des symboles**

- De l'alphabet aux idéogrammes

Comment représenter un alphabet ?

Ensemble fini de signes

Considéré avec des variantes : Majuscule / minuscule

Associé aux symboles des chiffres, de ponctuation

Convention la plus large possible est nécessaire:

Table ASCII de base codifie 2^7 caractères
American Standard Code for Information Interchange
`http://www.asciitable.com/`

Code ASCII

```
--------------                                      --------------   --------------   --------------
 0  0 000 NUL (null)                                32 20 040 SPACE  64 40 100 @       96 60 140 `
 1  1 001 SOH (start of heading)                    33 21 041 !      65 41 101 A       97 61 141 a
 2  2 002 STX (start of text)                       34 22 042 "      66 42 102 B       98 62 142 b
 3  3 003 ETX (end of text)                         35 23 043 #      67 43 103 C       99 63 143 c
 4  4 004 EOT (end of transmission)                 36 24 044 $      68 44 104 D      100 64 144 d
 5  5 005 ENQ (enquiry)                             37 25 045 %      69 45 105 E      101 65 145 e
 6  6 006 ACK (acknowledge)                         38 26 046 &      70 46 106 F      102 66 146 f
 7  7 007 BEL (bell)                                39 27 047 '      71 47 107 G      103 67 147 g
 8  8 010 BS  (backspace)                           40 28 050 (      72 48 110 H      104 68 150 h
 9  9 011 TAB (horizontal tab)                      41 29 051 )      73 49 111 I      105 69 151 i
10  A 012 LF  (NL line feed, new line)              42 2A 052 *      74 4A 112 J      106 6A 152 j
11  B 013 VT  (vertical tab)                        43 2B 053 +      75 4B 113 K      107 6B 153 k
12  C 014 FF  (NP form feed, new page)              44 2C 054 ,      76 4C 114 L      108 6C 154 l
13  D 015 CR  (carriage return)                     45 2D 055 -      77 4D 115 M      109 6D 155 m
14  E 016 SO  (shift out)                           46 2E 056 .      78 4E 116 N      110 6E 156 n
15  F 017 SI  (shift in)                            47 2F 057 /      79 4F 117 O      111 6F 157 o
16 10 020 DLE (data link escape)                    48 30 060 0      80 50 120 P      112 70 160 p
17 11 021 DC1 (device control 1)                    49 31 061 1      81 51 121 Q      113 71 161 q
18 12 022 DC2 (device control 2)                    50 32 062 2      82 52 122 R      114 72 162 r
19 13 023 DC3 (device control 3)                    51 33 063 3      83 53 123 S      115 73 163 s
20 14 024 DC4 (device control 4)                    52 34 064 4      84 54 124 T      116 74 164 t
21 15 025 NAK (negative acknowledge)                53 35 065 5      85 55 125 U      117 75 165 u
22 16 026 SYN (synchronous idle)                    54 36 066 6      86 56 126 V      118 76 166 v
23 17 027 ETB (end of trans. block)                 55 37 067 7      87 57 127 W      119 77 167 w
24 18 030 CAN (cancel)                              56 38 070 8      88 58 130 X      120 78 170 x
25 19 031 EM  (end of medium)                       57 39 071 9      89 59 131 Y      121 79 171 y
26 1A 032 SUB (substitute)                          58 3A 072 :      90 5A 132 Z      122 7A 172 z
27 1B 033 ESC (escape)                              59 3B 073 ;      91 5B 133 [      123 7B 173 {
28 1C 034 FS  (file separator)                      60 3C 074 <      92 5C 134 \      124 7C 174 |
29 1D 035 GS  (group separator)                     61 3D 075 =      93 5D 135 ]      125 7D 175 }
30 1E 036 RS  (record separator)                    62 3E 076 >      94 5E 136 ^      126 7E 176 ~
```

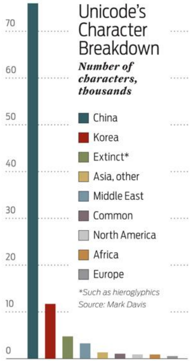
Au-delà du code ASCII de base

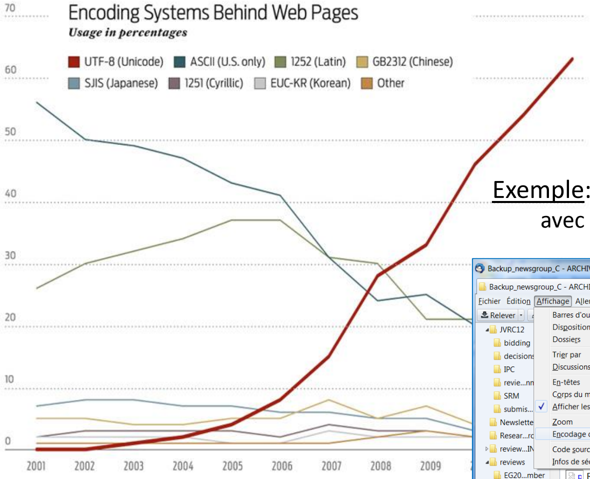
ASCII étendu sur 8 bits: Codes 0x80 à 0xFF
Le code étendu ISO 8859 Latin1 offre les caractères accentués minuscules et majuscules des langues occidentales: é è ê à ä ö ü …

La norme **UNICODE** permet d'intégrer les autres langues, > 109’000 caractères pour 93 écritures dont le chinois.

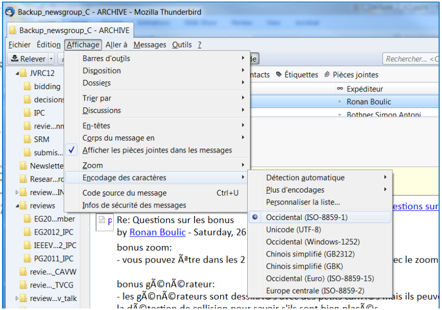
- les 256 codes d' ISO 8859 sont au début de l'UNICODE

- **UTF-8** est un codage des caractères UNICODE comprenant de 1 à 4 octets. Il est recommandé mais son usage n’est pas encore généralisé.





<u>Exemple</u>: texte d'un message interprété avec ISO 8859 au lieu de UTF-8

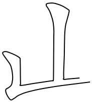


Taux d'utilisation de UTF-8 pour l'écriture de pages web [IEEE Spectrum 7/12]

Du code multi-byte (idéogramme) à l'image

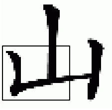
shan = montagne
le symbole peut être codé par 1 à 4 octets en UTF-8

MAIS **la représentation du symbole = son image** demande plus d'information.
Plusieurs approches sont possibles, du plus haut vers le plus bas niveau :



1) Préciser la **police de caractères** = *« style classique »*.

2) Caractériser les **contours** de la forme par un ensemble de **courbes** mathématiques paramétrées (silhouette). C'est la manière dont les polices de caractères sont construites.



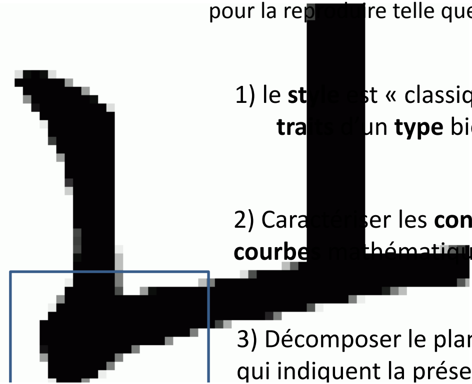
3) Décomposer le plan de l'image en un **ensemble de cellules** qui indiquent la quantité d'encre (pixel).

Du code multi-byte (idéogramme) à l'image

shan = montagne

le symbole peut être codé par 1 à 4 octets en UTF-8

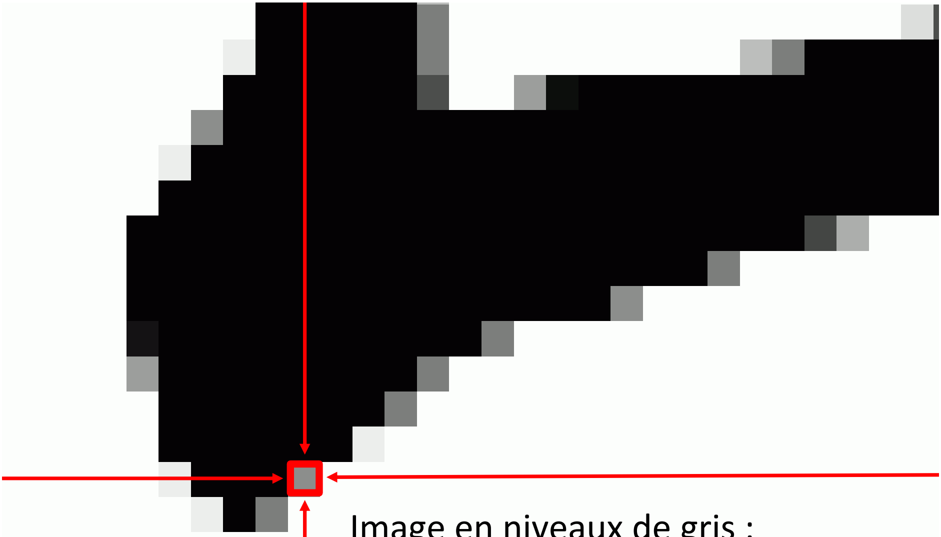
MAIS **l'image du symbole** demande plus d'information pour la reproduire telle quelle. Plusieurs approches:



1) le **style** est « classique ». Il comporte plusieurs **traits** d'un **type** bien précis parmi 37

2) Caractériser les **contours** de la forme par un ensemble de **courbes** mathématiques paramétrées (silhouette).

3) Décomposer le plan de l'image en un **ensemble de cellules** qui indiquent la présence d'encre (pixel).



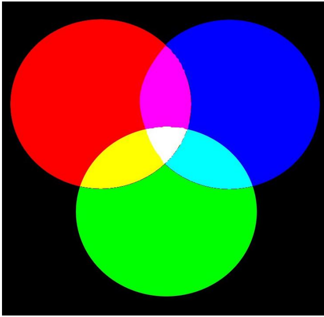
bitmap

Image en niveaux de gris :

chaque pixel mémorise une intensité entre 0 (noir) et le maximum définit par le nombre de bits retenu (blanc)

Image en couleur:

chaque pixel mémorise l'intensité de 3 composantes primaires dont la combinaison permet de restituer un espace de couleurs.



Le codage RGB (Red, Green, Blue)

- synthèse additive des couleurs: Rouge + Vert donne ....

- niveaux de gris lorsque les trois composantes sont égales noir(0,0,0) et le maximum définit par le nombre de bits retenu (blanc)

- parfois complété d'une 4ième composante Alpha (transparence) pour les applications graphiques.

Taille d'une image UXGA 1600x1200 x 3octets/pixel = 5'760'000 octets