Architecture des ordinateurs

Chapitre 1 : Représentation de l'information
& arithmétique des ordinateurs

*
* *

Clément Moreau, Olivier Ploton

{clement.moreau, olivier.ploton}@univ-tours.fr

Université de Tours ~ Département informatique de Blois

Licence 2 - Informatique

Sommaire

1. Représentation de l'information
2. Représentation des entiers
3. Arithmétique binaire
4. Représentation des flottants
5. Arithmétique flottante
6. Représentation des caractères

Représentation de l'information

En informatique, toute donnée est codée fondamentalement sous la forme d'une séquence de bits (0,1). Mais pourquoi ?

- Question relative à la théorie de l'information.
- Modélisation formelle de l'information. (Shannon et Weaver, 1948)
  Un bit $\Leftrightarrow$ unité élémentaire (i.e minimale) d'information.

Dès lors, il nécessaire d'obtenir un *système de numération* qui permette de représenter les entités communes que l'on utilise en informatique : les entiers (`int`), réels (`float`), données numériques brutes, information textuelle, etc. uniquement à partir du 0 et du 1.

Représentation de l'information

Définition (Système de numération)

Un *système de numération* est un triplet (X, I, Φ), où X est l'ensemble des nombres à énumérer, I est un ensemble fini ou dénombrable de symboles (e.g. chiffres) et Φ, une application injective telle que $\Phi : \begin{cases} X & \to I^{\mathbb{N}} \\ x & \mapsto (\varepsilon_n(x))_{n\geq 1} \end{cases}$

Ainsi, une *représentation en base* $b \in \mathbb{N}$ (ou b-adique) avec $b > 1$ est un système de numération tel que :

- $X = [\![0, b^n - 1]\!]$,
- $I = [\![0, b - 1]\!]$
- $\Phi(x) = (\varepsilon_i(x))_{0\leq i\leq n}$ est bijective avec $\varepsilon_i(x) = x_i$ telle que :

$$\forall x \in X, \exists!(x_n, x_{n-1}, ..., x_0) \in I^{n+1} \text{ avec } x_n \neq 0 \,\middle|\, x = \sum_{i=0}^{n} \varepsilon_i(x) b^i$$

C'est-à-dire, que pour tout nombre entier x, il existe une unique manière de représenter x dans la base b, on notera ceci $x = \langle x_n x_{n-1} ... x_0 \rangle_b$.

Représentation de l'information

Exemple

- Décimal $(b = 10)$

  $\langle 42 \rangle_{10} = 4 \times 10 + 2 \times 10^0$

- Binaire $(b = 2)$

  $\langle 101010 \rangle_2 =$
  $1 \times 2^5 + 1 \times 2^3 + 1 \times 2^1 = \langle 42 \rangle_{10}$

- Hexadécimal
  $(b = 16) : [\![0, 9]\!] \cup \{A, B, C, D, E, F\}$

  $\langle 2A \rangle_{16} = 2 \times 16^1 + A \times 16^0 = \langle 42 \rangle_{10}$

42			
21×2	+	0	*Bits de poids faible*
10×2	+	1	
5×2	+	0	
2×2	+	1	
1×2	+	0	
0×2	+	1	*Bits de poids fort*

Représentation de l'information

Soient les opérations $div(a,b)$ et $mod(a,b)$ qui correspondent respectivement à la division entière de a par b et au résultat de a modulo b.

L'obtention des coefficients x_i de l'écriture d'un nombre x dans une base b est assurée par l'algorithme suivant :

Data : $b > 1, x \in \mathbb{N}$
Result : $(x_n, x_{n-1}, ..., x_0)$
$i \leftarrow 0$;
while $x \neq 0$ **do**
    $x_i \leftarrow mod(x,b)$;
    $x \leftarrow div(x,b)$;
    $i \leftarrow i+1$;
end

Algorithme - Calcul d'un entier x dans une base b

Il est possible d'établir des correspondances entre deux bases a et b (avec $b > a$) si :

$$\log_a(b) \in \mathbb{N}$$

Exemple

- $\log_2(16) = 4 \Leftrightarrow 2\log_2(16) = 8$
  Deux chiffres hexadécimaux correspondent à un octet.
- $\langle 186 \rangle_{10} = \langle 10111010 \rangle_2 = \langle (1011)(1010) \rangle_2 = \langle BA \rangle_{16}$

Représentation de l'information

Soit $x \in \mathbb{N}^*$ avec $x = \langle x_n x_{n-1} ... x_0 \rangle_b$. On note $N_b(x) = n + 1$ le nombre de chiffres nécessaires pour exprimer x dans la base b.

Théorème

Soit $x \in \mathbb{N}^*$ et $b > 1$. Alors

$$N_b(x) = \lfloor \log_b(x) \rfloor + 1$$

Démonstration.

On rappelle que $\forall \alpha \in \mathbb{R}$, $\lfloor \alpha \rfloor$ est la partie entière de $\alpha \in \mathbb{R}$, càd l'unique entier p tel que $p \leq \alpha < p + 1$.

Soit $x \in \mathbb{N}^*$ avec $x = \langle x_n x_{n-1} ... x_0 \rangle_b$. On chercher à déterminer n. Alors :
$b^n \leq x < b^{n+1} \Leftrightarrow n \leq \log_b(x) < n + 1 \Leftrightarrow \lfloor \log_b(x) \rfloor$

$\square$

Représentation des entiers

On se place dans le cas où $b = 2$.

Si $x \in \mathbb{Z}$? Comment fait-on ?

Il existe plusieurs modes de représentation :

- En binaire signé $\rightarrow$ `signed` ou `unsigned`
- En complément à 2

Dans tous les cas, on note que l'on travaille sur des nombres d'une taille k fixée. Par défaut, en dehors de toute précision, on travaille sur un octet avec $k = 8$.

Représentation des entiers

La représentation en *binaire signé* réserve le premier bit de la représentation du nombre en binaire pour connaitre son signe.

Représentation en binaire signé

Soit $x \in [\![0, 2^{k-1} - 1]\!]$ tel que $x = \langle 0x_{k-2}...x_0\rangle_2$. L'opposé $-x$ est représenté tel que :

$$-x = \langle 1x_{k-2}...x_0\rangle_2$$

Exemple

- $\langle 3\rangle_{10} = \langle 0000\ 0011\rangle_2$
- $\langle -3\rangle_{10} = \langle 1000\ 0011\rangle_2$

- Présence de deux valeurs pour 0 (0000 0000) et -0 (1000 0000).
- Principes d'addition et soustraction non valables.

La méthode utilisée pour représenter les entiers est celle du *complément à 2*.

Définition (Représentation en complément à 2)

- On travaille sur k bits et sur les nombres $x \in \llbracket 0, 2^{k-1} - 1 \rrbracket$.
- Le bit de signe est placé en tête (i.e. x_{k-1}).

$$\begin{cases} \langle x \rangle_{c2} & = \langle x \rangle_2 \\ \langle -x \rangle_{c2} & = \langle \overline{x} \rangle_2 + 1 \end{cases}$$

Représentation des entiers

Exemple

-63	:		1	1	0	0	0	0	0	1
$+28$	:		0	0	0	1	1	1	0	0
report		0	0							
$0\|$			1	1	0	1	1	1	0	1
$+$										0
$\langle -35 \rangle_{2c8}$	$=$		1	1	0	1	1	1	0	1

63	:		0	0	1	1	1	1	1	1	
-28	:		1	1	1	0	0	1	0	0	
report		1	1	1	1	1	1				
$1\|$			0	0	1	0	0	0	1	1	
$+$										$\cancel{1}$	$\leftarrow$retenue négligée
$\langle 35 \rangle_{2c8}$	$=$		0	0	1	0	0	0	1	1	

Représentation des entiers

On note $\langle x \rangle_{c2} = \langle x_{k-1} x_{k-2} ... x_0 \rangle$ la représentation en complément à 2 de l'entier x. Alors, le décodage d'un nombre en complément à 2 est assuré par la formule :

$$x = -(x_{k-1}) \times 2^{k-1} + \sum_{i=0}^{k-2} x_i 2^i$$

Pour $k = 8$ bits, on a les entiers signés :

$$\begin{array}{ccc} -128 & = & \langle 10000000 \rangle_{2c8} \\ \vdots & & \\ -1 & = & \langle 11111111 \rangle_{2c8} \\ 0 & = & \langle 00000000 \rangle_{2c8} \\ 1 & = & \langle 00000001 \rangle_{2c8} \\ \vdots & & \\ 127 & = & = \langle 01111111 \rangle_{2c8} \end{array}$$

Représentation des entiers

- Codage / décodage facile,
- Représentation unique du 0,
- Opérations arithmétiques simples. On néglige toujours les retenues finales.

- Taille de la mémoire fixée.

Représentation des entiers

Taille des différents types numériques en C (pareil en Java, il me semble...).

Type	nb bits	Borne inf	Borne sup
`char`	8	-128	127
		0	255
`short`	16	$-32\ 768$	32 767
		0	65 535
`int`*	32	$-2\ 147\ 483\ 648$	2 147 483 647
		0	4 294 967 295
`long`	64	-2^{63}	$2^{63} - 1$
		0	$2^{64} - 1$

* Si le processeur $\geq$ 32 bits, sinon = `short`

Représentation des entiers

On observe la relation d'ordre suivante :

$$\mathrm{Dom}(\texttt{char}) \subseteq \mathrm{Dom}(\texttt{short}) \subseteq \mathrm{Dom}(\texttt{int}) \subseteq \mathrm{Dom}(\texttt{long})$$

- Toutes les valeurs possibles pour une variable du type `char` sont aussi utilisables pour les autres types,
- Il est possible qu'une valeur valide pour le type `int` puisse ne pas être représentable dans une variable de type `short`.

Si vous ne savez pas quel type donner à une variable de type entier, le type `int` est par défaut le meilleur choix car `int` $\equiv$ mot machine, c'est-à-dire qu'il est adapté à la taille que la machine peut traiter directement.

Représentation des entiers

Considérons qu'on travaille sur un PC en 32 bits, en complément à 2.

Si on utilise des valeurs hors du domaine, par exemple 2^{32}, et qu'on essaye de la stocker dans une variable de type `int` sur une telle machine, que se passe-t-il ?

Dépassement de capacité

La réponse dépend du type :

- Si on essaye d'enregistrer une valeur hors domaine dans une variable de type *signé*, le programme affiche le résultat de l'addition sur les k bits.
- Si on essaye d'enregistrer une valeur hors domaine dans une variable de type *non signé*, la conversion se fait modulo la valeur maximale représentable par le type $+ 1$.

Représentation des entiers

Exemple

```
#include <stdio.h>

int main(void) {
   unsigned char x = 300;
   /* %hhu sert à dire à printf() qu'on veut afficher un unsigned char */
   printf("La variable x vaut %hhu.\n", x);
   return 0;
}
---------
La variable x vaut 44.
```

En effet, on a $300 - (255 + 1) = 44$

Exemple

```c
#include <stdio.h>
int main(void) {
    signed char x = -103;
    signed char y = -65;
    /* %hhd sert à afficher un signed char */
    printf("x + y = %hhd.\n", x+y);
    return 0;
}
---------
x + y = 88.
```

-103	:		1	0	0	1	1	0	0	1
-65	:		1	0	1	1	1	1	1	1
report		1	0	1	1	1	1	1	1	
$1\vert$			0	1	0	1	1	0	0	0
$+$										$\cancel{1}$
$\langle -168 \rangle_{2c8}$	$\neq$		0	1	0	1	1	0	0	0

Exemple

Soit le calcul des coefficients binomiaux C_n^k (ou $\binom{n}{k}$) tel que :

$$C_n^k = \frac{n!}{k!(n-k)!}$$

1. Que pensez-vous de cette méthode de calcul ?
2. À partir de quel rang p a-t-on $p! > 2^{64} - 1$?

Représentation des entiers

Exemple

Soit le calcul des coefficients binomiaux C_n^k (ou $\binom{n}{k}$) tel que :

$$C_n^k = \frac{n!}{k!(n-k)!}$$

On rappelle que $n! = \prod_{k=1}^{n} k$. Donc, on sait que $n! > \prod_{k=1}^{n} 2^{i-1}$ avec i tel que $2^{i-1} \leq k < 2^i$.

On cherche le plus petit p tel que $\sum_{k=1}^{p} \lfloor \log_2(k) \rfloor \geq 64$.

On a $\sum_{k=1}^{23} \lfloor \log_2(k) \rfloor = 66$.

Dès lors, on est sûr que $23! > 2^{64} - 1$.

Représentation des entiers

Comment résoudre le dépassement de capacité ?

1. Utilisation d'un type ayant une plus grande capacité de représentation.
2. Utilisation d'un langage Objet implémentant des bibliothèques pour la gestion des grands nombres. On peut citer la librairie *BigInteger* en Java et C++.
   $\Rightarrow$ Ne fait que retarder l'inévitable...
3. Réfléchir à une meilleure implémentation...

Arithmétique binaire

Et pour la multiplication, comment on fait ? Et la division ? ?

Toutes les opérations arithmétiques se ramènent à des opérateurs logiques.
$\Rightarrow$ On le verra dans le Cours 2 : Logique booléenne et circuits combinatoires

Pour rappelle, on dispose des opérateurs logiques principaux ET, OU, XOR et NON.

x	y	$x \,\&\, y$	$x \mid y$	$x \wedge y$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

x	$\sim x$
0	1
1	0

Exemple

$$\begin{array}{c|cccc} & 1 & 0 & 0 & 1 \\ \& & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 & 1 \end{array} \qquad \begin{array}{ccccc} & 1 & 0 & 0 & 1 \\ | & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 1 & 1 \end{array} \qquad \begin{array}{ccccc} & 1 & 0 & 0 & 1 \\ \wedge & 1 & 0 & 1 & 1 \\ \hline & 0 & 0 & 1 & 0 \end{array}$$

$$\begin{array}{ccccc} \sim & 1 & 0 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

Arithmétique binaire

Soit une séquence de bits $x = \langle x_{n-1} ... x_0 \rangle_2$ et un entier $k \in [\![0, n-1]\!]$.

Définition (Opération de décalage)

On appelle *opérateur de décalage* à gauche $x \ll k$ (resp. à droite $x \gg k$) l'opération qui ajoute k 0 à gauche de x (resp. retire les k premiers bits de x).

$$y = x \ll k = \left\langle y_i \middle| \forall i \in [\![0, n-1+k]\!], \begin{cases} y_i = 0 & \text{si } i < k \\ y_i = x_{i-k} & \text{sinon} \end{cases} \right\rangle$$

et

$$y = x \gg k = \langle y_i \, | \forall i \in [\![0, n-1-k]\!], y_i = x_{i+k} \rangle$$

Il existe d'autres façons d'exécuter une division au sein d'un ordinateur.

Théorème (Algorithme de Newton)

Soit une fonction $f : \mathbb{R}^+ \to \mathbb{R}^+$ *(supposée de classe* $\mathcal{C}^1$*) telle que* $f(\alpha) = 0$*. On pose la suite* $(x_n)_{n\in\mathbb{N}}$ *telle que :*

$$\begin{cases} x_0 \in \mathbb{R}^+ & \\ x_{n+1} & = x_n - \frac{f(x_n)}{f'(x_n)} \end{cases}$$

Où $x_0 \approx \alpha$*, Alors, on a* $\lim_{n\to+\infty} x_n = \alpha$ *et la convergence de* x_n *est quadratique.*

Si α est issu d'une fonction $g(a)$. Alors $f(x) = g^{-1}(x) - a$.

Humm... C'est encore un peu compliqué. Et ça n'explique pas comment on fait une division.

Calcul de $1/a$ par l'algorithme de Newton

On veut calculer $\alpha = g(a) = \frac{1}{a}$. Dès lors, on $f(x) = \frac{1}{x} - a$ et $f'(x) = -\frac{1}{x^2}$.
On calcule :

$$\begin{aligned} x - \frac{f(x)}{f'(x)} &= x + x^2(\frac{1}{x} - a) \\ &= x(2 - xa) \end{aligned}$$

L'itération de Newton associée à notre problème est donc :

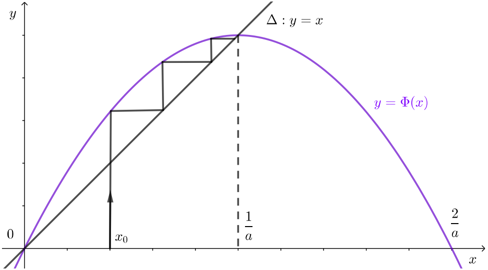
$$x_{n+1} = \Phi(x_n) = x_n(2 - x_n a)$$

Pour savoir quel x_0 choisir, on doit étudier la fonction Φ.

Arithmétique binaire

On voit clairement que pour que la suite converge x_0 doit appartenir à $]0, {}^2/_a[$.

C'est un problème... On ne connait pas $\frac{2}{a}$.



- Heureusement, les ordinateurs travaillent avec la notation en *Virgule Flottante* pour les réels.
- Et là miracle, on peut démontrer que l'on peut choisir $x_0 < 2$ et ça suffit pour converger.

Arithmétique binaire

Exemple

On cherche à calculer :

$$Q = \frac{0.42 \times 10^{56}}{0.37 \times 10^{-21}}$$

Mais combien vaut $\frac{1}{0.37}$? ?
On applique la méthode de Newton avec $x_{n+1} = x_n(2 - 0.37x_n)$.

Arithmétique binaire

Exemple

On cherche à calculer

$$Q = \frac{0.42 \times 10^{56}}{0.37 \times 10^{-21}}$$

- $x_0 = 1$
- $x_1 = 2,276947$
- $x_2 = 2,635633573$
- $x_3 = 2,701038343$
- $x_5 = 2,702701678$
- $x_6 = 2,702702702$

Dès lors

$$Q \approx 2,702702702 \times 10^{21} \times 0.42 \times 10^{56} \approx 1.135135135 \times 10^{77}$$

Représentation des flottants

Et donc, si $x \in \mathbb{R}$? Comment fait-on ? On a dit Virgule Flottante ?

Une première solution est de considérer un couple $(\langle E(x)\rangle_b, \langle F(x)\rangle_b)$, avec $E(x) \in \mathbb{Z}$ la partie entière de x et et $F(x) \in [0, 1[$, sa partie fractionnaire.

La représentation binaire de $F(x)$ est telle que :

$$\langle F(x)\rangle_b = \left\langle x_i \middle| x_i \in [\![0, b-1]\!], F(x) = \sum_{i=1}^{\infty} x_i \times \frac{1}{b^i} \right\rangle$$

On dénote abusivement $x \in \mathbb{R}$, mais l'arithmétique flottante forme un ensemble $\mathbb{F}$ fini et discret qui comporte :

- Des représentations correctes de certains éléments de $\mathbb{Q}$ et des approximations d'éléments de $\mathbb{R}$.
- Ne respecte pas toutes les règles du corps $(\mathbb{R}, +, \times)$

L'écriture d'un réel $x \in [0,1[$ dans une base b avec une précision de n est donnée par l'algorithme suivant :

Data : $b > 1, x \in [0,1[, n \geq 1$
Result : $(x_1, ..., x_n)$
$i \leftarrow 1$;
while $i \leq n$ **do**
  $x \leftarrow x \times b$;
  $x_i \leftarrow \lfloor x \rfloor$;
  $x \leftarrow x - x_i$;
  $i \leftarrow i + 1$;
end

Algorithme - Calcul de $x \in [0,1[$ selon une précision n

Exemple

Soit $x = 1,625$.
On a $\langle x \rangle_2 = (\langle 1 \rangle_2, \langle 101 \rangle_2)$ car :
$0,625 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$

$0,625$	$1,25$	1
$0,25$	$0,5$	0
$0,5 \times 2$	1	1

Représentation des flottants

Une représentation selon deux entités $\langle E(x)\rangle_b$ et $\langle F(x)\rangle_b$est assez gourmande en mémoire, pour condenser l'écriture, on utilise la notation scientifique normalisée.

Définition (Notation scientifique normalisée de base b)

On peut représenter tout nombre non nul selon la notation scientifique normalisée.

$$\forall x \in \mathbb{R}, x = s \times m \times b^e$$

avec :

- $s \in \{-1, 1\}$,
- $m = x_0, x_1 x_2 ...$ tel que tout $x_i \in [\![0, b[\![$ (et $x_0 \neq 0$),
- $e \in \mathbb{Z}$.

Représentation des flottants

Ainsi, un nombre flottant $x \in \mathbb{F}_n^*$ (de précision $n \in \mathbb{N}^*$) est codé par trois nombres représentés tel que :

Signe s

	Exposant E	Mantisse M

Le coefficient M est appelé la *mantisse complétée*, E est appelé l'*exposant decalé* et s représente le signe (0 positif, 1 négatif).

On a alors : $|s| + |E| + |M| = n$.

Ce mode de représentation est appelé Standard *IEEE 754*.

Représentation des flottants

Le standard inclu deux représentations : simple précision et double précision.

Type	Nb bits	$\lvert s \rvert$	$\lvert E \rvert$	$e_{\min}$	$e_{\max}$	$\lvert M \rvert$	ε
`float`	32	1	8	-126	127	23	127
`double`	64	1	11	-1022	1023	52	1023

L'exposant peut être positif ou négatif. Cependant, la représentation habituelle des nombres signés en complément à 2 rendrait la comparaison entre les nombres flottants un peu plus difficile.

Pour régler ce problème l'exposant est *décalé*, afin de le stocker sous forme d'un nombre non signé. Ce décalage est de ε.

Représentation des flottants

Codage IEEE 754 (Kahan, 1985)

Soit $x \in \mathbb{F}$, tel que $\langle x \rangle_b = (\langle E(x) \rangle_b, \langle F(x) \rangle_b .)$.

- La mantisse m est obtenue en concaténant $\langle E(x) \rangle_b$ et $\langle F(x) \rangle_b$ tels que $(\langle E(x) \rangle_b, \langle F(x) \rangle_b)$ puis en décalant e bits vers la droite (ou la gauche) de manière à ce qu'il n'y ait qu'un seul chiffre $\nu \neq 0$ avant la virgule. M est obtenu en complétant m par une série de 0.
- L'exposant E est égal au nombre de bits décalés (compté positivement si l'on décale vers la droite et négativement vers la gauche) plus le biais ε. Au final $E = e + \varepsilon$.
- Le signe s est 1 si le nombre est négatif, 0 sinon.

Si $b = 2$, on omet ν dans la représentation car on sait qu'il est toujours égale à 1, dès lors, il n'est pas nécessaire de le prendre en compte. On l'appelle alors *hidden bit*.
Si $b \neq 2$, il faut le conserver dans la mantisse pour le décodage.

Décodage IEEE 754

La valeur d'un nombre x sous la représentation IEEE 754 en base $b = 2$ est donnée par la formule :

$$x = (-1)^s \times \left(1 + \sum_{i=1}^{|M|} m_i \times \frac{1}{2^i}\right) \times 2^{E-\varepsilon}$$

Pour $b \neq 2$, on compte le bit de tête ν, on a :

$x = (-1)^s \times \left(\sum_{i=0}^{|M|} m_i \times \frac{1}{b^i}\right) \times b^{E-\varepsilon}$

où $f_0 = \nu$.

Représentation des flottants

Exemple

On considère une architecture 32 bits. On considère $b = 2$.
Soit $x = -123,5$. On a $\langle x \rangle_2 = (\langle 1111011 \rangle_2, \langle 1 \rangle_2)$
On code alors $(1111011, 1)$ puis on décale la virgule de 6 rangs vers la droite, on obtient $(1,1110111) \times 2^6$. On en déduit que :

- Le bit de signe $s = 1$,
- La mantisse $m = 1110111$ qu'on complète pour obtenir m sur 23 bits, soit $M = 1110\ 1110\ 000\ 0000\ 0000\ 000$,
- L'exposant $e = 6$. Dès lors $E = e + \varepsilon = 6 + 127 = 133 = \langle 1000\ 0101 \rangle_2$.

On a x tel que :

1	1000 0101	$\cancel{1}$1110 1110 0000 0000 0000 000

Représentation des flottants

Il existe certaines valeurs réservées pour gérer les exceptions mathématiques. La philosophie par défaut : *le calcul doit toujours continuer* !

-0	1	00000000	00000000000000000000000
0	0	00000000	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
`NaN`	0	11111111	`Chaîne non nulle`

Soit $x \in \mathbb{F}^*$ et $y \in \mathbb{F}^*_+$:

- Des choses intuitives : $\frac{x}{0} = +\infty$, $x + (-\infty) = -\infty$
- Des choses plus... exotiques : $\sqrt{-0} = -0$
- Une valeur absorbante `NaN` pour les calculs incohérents ou formes indéterminées : $\sqrt{-y}$, $(\pm\infty) \pm \infty$ $\frac{\pm\infty}{\pm\infty}$, $(\pm 0) \times \pm\infty, \texttt{Nan} \pm x$, etc

D'autres choses un peu bizarres :

$$\frac{x^2}{\sqrt{x^3+1}} \underset{+\infty}{\sim} \sqrt{x}$$

mais :

- Si x^3 “dépasse“ mais pas x^2, on obtiendra 0
- Si les deux dépassent, on obtientra `NaN`.

La suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0 & = \frac{11}{2} \\ u_1 & = \frac{61}{11} \\ u_{n+1} & = 111 - \frac{1130}{u_n} + \frac{3000}{u_n u_{n-1}} \end{cases}$$

On a, en théorie, $\lim\limits_{n\to+\infty} u_n = 6$. Mais sur toutes les machines, on observe une convergence rapide de u_n vers 100...

Arithmétique flottante

- Compatibilité avec le complément à 2,
- Représentation importante avec peu de mémoire,

- Précision finie :
  Il est impossible de représenter parfaitement certains nombres dans l'arithmétique flottante en représentation IEEE 754.
- Perte de certaines bonnes propriétés de l'arithmétique :
  - $\times$ n'est plus associative : $\exists x, y, z \in \mathbb{F} | x \times (y \times z) \neq (x \times y) \times z$,
  - $\times$ n'est plus distributive par rapport à $+$:
    $\exists x, y, z \in \mathbb{F} | x \times (y + z) \neq x \times y + x \times z$,
  - $\times$ n'est plus une loi de composition interne : $\exists x, y \in \mathbb{F} | x \times y \notin \mathbb{F}$.

Arithmétique flottante

Soit $x \in \mathbb{F}$ tel que $x = m \times 2^e$.

On note ses deux plus proches voisins x^+ et x^- s'obtiennent en incrémentant ou décrémentant la mantisse de $2^{-|M|}$.

Répartition des flottants dans $\mathbb{F}$

Le corps des flottants $\mathbb{F}$ n'est pas linéairement réparti et le pas entre x et $x^\pm$ vaut $2^{e-|M|}$.

Démonstration.

Soit $x = m \times 2^e$. On note $m^\pm = m \pm 2^{-|M|}$. Dès lors, on a :

$$\begin{aligned} x^\pm &= m^\pm \times 2^e \\ &= (m \pm 2^{-|M|}) \times 2^e \\ &= m \times 2^e \pm 2^{e-|M|} = x \pm 2^{e-|M|} \end{aligned}$$

$\square$

Arithmétique flottante

Exemple

On considère le programme C suivant :

```
#include <stdio.h>

  int main(void) {
     float i;
     for (int j = 0; j < 1000; j++) {
         i += 0.1;
     }
     printf("i = %f\n", i);
     return 0;
}
---------
i = 99,999046
```

D'où vient cette erreur ?

Résolution

En binaire, on a : $\langle 0,1\rangle_2 = \left(\langle 0\rangle_2, \left\langle 0001\ \underbrace{1001\ 1001\ 1001\ 1001\ 1001\ \cdots}_{\text{récurrence}}\right\rangle_2\right)$. Dès lors, le nombre $0,1 \notin \mathbb{F}$ et ne peut être représenté de manière finie à l'aide de la représentation IEEE 754 car il est égale à la série infinie :

$$0,1 = \frac{1}{2^4} + \frac{9}{2^8}\sum_{i=0}^{\infty}\frac{1}{2^{4i}}$$

Démonstration.

Par observation, on suppose que $0,1 = \frac{1}{2^4} + \sum_{i=0}^{\infty} \left(\frac{1}{2^{4i+5}} + \frac{1}{2^{4i+8}}\right)$.

$$
\begin{aligned}
\frac{1}{2^4} + \sum_{i=0}^{\infty} \left(\frac{1}{2^{4i+5}} + \frac{1}{2^{4i+8}}\right) &= \frac{1}{2^4} + \sum_{i=0}^{\infty} \frac{1}{2^{4i+5}} \left(1 + \frac{1}{2^3}\right) \\
&= \frac{1}{2^4} + \sum_{i=0}^{\infty} \frac{1}{2^{4i}} \times \frac{1}{2^5} \times \frac{9}{2^3} \\
&= \frac{1}{2^4} + \frac{9}{2^8} \sum_{i=0}^{\infty} \frac{1}{2^{4i}} \\
&= \frac{1}{2^4} + \frac{9}{2^8} \times \left(\lim_{n \to +\infty} \sum_{i=0}^{n} \left(\frac{1}{2^4}\right)^i \right) \\
&= \frac{1}{2^4} + \frac{9}{2^8} \times \left(\lim_{n \to +\infty} \frac{2^4}{15} \times \left(1 - \frac{1}{2^{n+1}}\right) \right) \\
&= \frac{1}{2^4} + \frac{9}{2^4 \times 15} = \frac{1}{16} \times \frac{24}{15} = \frac{3}{30} = 0,1
\end{aligned}
$$

□

Arithmétique flottante

On considère le programme C suivant qui permet d'approximer l'erreur absolue commise lors du calcul :

```
#include <stdio.h>
#include <math.h>
int main(void) {
    float i;
    int iter = 0;
    for (iter = 0; iter < 1000; iter++) {
       i += 0.1;
       printf("%e\n", abs(i - (iter + 1) / 10.0));
    }
   /* Comme (iter + 1) / 10.0 est un double, cette valeur est proche de la
   valeur théorique */
   return 0;
}
```

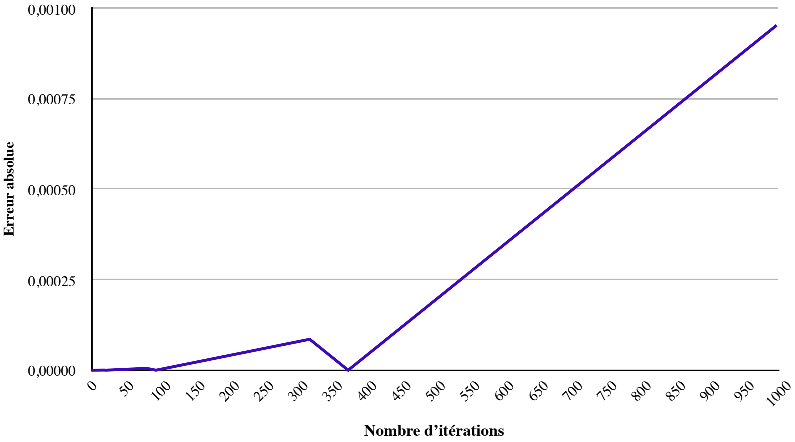


Figure - Erreur de calcul en fonction du nombre d'itérations

Arithmétique flottante

Comment contrôle t-on la précision de calcul ?

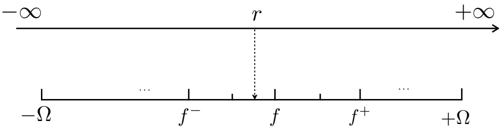
Arrondi (correct)

Une fonction d'arrondi au plus prés $\circ : \mathbb{R} \rightarrow \mathbb{F}$ definie telle que :

$$\circ(r) = f, \forall x \in \mathbb{F}, |r - f| \leq |r - x|$$

$\Rightarrow$ Permet de récupérer la commutativité !

Ainsi, si $r \in \mathbb{R}$ mais que $r \notin \mathbb{F}$, il est remplacé par l'ordinateur par le nombre $f \in \mathbb{F}$ qui minimise l'écart.



En cas d'égalité de distance, on prend le flottant avec le dernier bit de la mantisse nul.

Erreur d'arrondi

Soit $r \in \mathbb{R}$ et $f \in \mathbb{F}$ tel que $\circ(r) = f$. L'erreur δ d'arrondi est définie telle que

$$r = f + \delta$$

L'erreur d'arrondi exacte dépend du réel subissant l'arrondi, mais il est en général possible d'en donner un majorant.
En outre, on montre que

$$|\delta_{\max}| = \left| \frac{f^{\pm} - f}{2} \right| = 2^{e-|M|-1}$$

Un corrolaire immédiat de la répartition des nombres flottants est que plus r est grand, plus l'erreur d'arrondi est importante.

Arithmétique flottante

Erreur d'approximation

- Des *erreurs d'approximation* peuvent être occasionnées selon le type utilisé. En particulier attention au type `float` qui possède une précision minimale.
  Attention aux accumulations d'erreurs infinitésimales qui peuvent devenir importantes !
- Au sein de contextes ultra-précis (finance, sciences), il est conseillé de ne pas utiliser de calcul en virgule flottante. On préfèrera à la place les décimaux ou virgule fixe.
- Imposer l'ordre d'exécution des opérations.

Quelques erreurs d'approximation qui ont mal fini...

- Déviation du missile Patriot (1991) : 28 soldats tués.
- Explosion de la fusée Ariane (1996) : Coût d'1 Milliard $.
- Krach boursier de 1985 à la Bank de New-York : Coût 20 Milliards de $

Représentation des caractères

Et pour l'information textuelle ?

- Première représentation en 1961 par le codage *ASCII*.
  Représentations sur 7 bits, soit 128 caractères (en hexadécimal de 00 à $7F$).
- ASCII étendu par *ISO/CEI 8859-i* (avec $i \in [\![1, 16]\!]$) en utilisant le 8ème bit.
  i permet de fournir un ensemble adéquat de caractères selon la langue.
  1 (latin), 5 (cyrillique), 6(arabe), 7(grecque), ...
- Représentation actuelle sur 32 bits selon le standard *Unicode* et la norme *ISO/CEI 10646*.
  Unicode et ISO/CEI 10646 acceptent plusieurs formes de transformation universelle pour représenter un point de code valide : UTF-$\{8, 16, 32\}$.
  `http://www.unicode.org`

Représentation des caractères

Une constante représentant un caractère (de type `char`) est délimitée par des apostrophes, comme par exemple '`a`'.

En C et en Java, pour récupérer le code ASCII décimal d'un `char`, il suffit de le caster en `int`.

Exemple

```c
#include <stdio.h>
int main(int argc, char * argv[]) {

    char c = 'A';
    int x = (int) (c);
    printf("Sur votre machine, le caractère \'%c\' a
        pour code %d.\n", c, x);
    return 0;

}
---
Sur votre machine, le caractère 'A' a pour code 65.
```

Il est légale d'écrire `c+32`, dans ce cas, `c` vaut `97`, si `c` est recaster en `char`, il vaut la constante ASCII `97`, soit `'a'`.

Représentation des caractères

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	‘	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	’	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	ˆ	94	0136	0x5e	˜	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

FIGURE – Table de codage ASCII

Représentation des caractères

Au delà de l'ASCII, la valeur représentée par cette constante est néanmoins dépendante des conventions de codage de caractères employées.

Par exemple le caractère « œ » (ligature du o et du e) a pour valeur :

- 189 dans le jeu de caractères ISO-8859-15 (sous Unix),
- 156 sur certaines variantes du jeu de caractères Windows 1252 (sous Windows),
- 207 avec l'encodage Mac Roman (Mac OS 9 et antérieur),
- 0xc5, 0x93 (deux octets, donc deux `char`) en UTF-8,
- Pas d'équivalent en ASCII. (Sous ISO/CEI 8859-1, le caractère 189 est le symbole « ½ », le 207 est le « Ï » et le 156 n'est pas utilisé).

Des problèmes peuvent apparaitre lors de la transmission ou de la compression des données.

Conclusion

Pour terminer cette partie, j'aimerais revenir sur la différence entre une valeur et son représentant. Si on considère la séquence hexadécimal ci-dessous :

$$\langle 626F6E6A6F757221 \rangle_{16}$$

Elle peut représenter plusieurs valeurs :

- La chaîne de caractères "bonjour !" si l'on considère que la séquence représente des caractères codés en ASCII.
- La valeur 7 093 009 341 547 377 185 si l'on considère que la séquence représente un entier codé sur 64 bits en complément à 2.
- L'image , si l'on considère que chaque bloc de 8 bits code le niveau de gris d'un pixel considéré comme noir pour 0 et blanc pour 255.

Dans tous les cas, c'est au programme de dire ce que la séquence représente.

Conclusion

- En informatique, fondamentalement, tout est codé selon le *système binaire,*
- Pour modéliser les nombres entiers $\mathbb{Z}$, on utilise la représentation en *complément à 2*,
  - On utilise n bits pour représenter tout nombre $x \in [\![-2^{n-1}, 2^{n-1} - 1]\!]$.
  - Attention au dépassement de capacité !
- Pour modéliser les nombres flottants $\mathbb{F}$, on utilise la représentation *IEEE 754*,
  - Représentation riche des nombres à l'aide la notation scientifique.
  - Attention aux approximations effectuées dans l'arithmétique flottante !
- Pour modéliser les caractères, de nombreux standards existent. Les plus connus sont les standards *ASCII*, son étendu *ISO/CEI 8859* et *Unicode*.
- L'*information est contextuelle* et est spécifiée selon le contexte du programme (`int`, `char`, `double`, codage d'un pixel, etc).