Chapitre 4

Représentation d'un nombre réel en informatique

On va aborder la représentation en mémoire d'un nombre réel selon la norme IEEE 754.

1 Introduction

On a vu précédemment comment représenter en binaire des entiers naturels $(0, 1, 2, 3, \dots)$ et des entiers relatifs $(\dots, -3, -2, -1, 0, 1, 2, 3, \dots)$. On va maintenant regarder comment faire pour des nombres réels (les nombres à virgule).

Exemple décimal : 132,34

1	centaine	10^2
3	dizaine	10^1
2	unité	10^0
,		
3	dixième	10^{-1}
4	centième	10^{-2}

En informatique, la virgule est représentée par un point « . » ! Ce sont des anglais et des américains qui ont développés l'informatique à ses débuts, ils ont codés le point plutôt que la virgule.

$$132,34 = 1 \times 10^2 + 3 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2}$$

Peut-on faire la même chose en binaire ? Voyons cela sur l'exemple 101.01 :

1		2^2
0		2^1
1	unité	2^0
.		
0	demi	2^{-1}
1	quart	2^{-2}

$$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 4 + 1 + \frac{1}{4} = 5.25$$

2 Arithmétique en virgule flottante

L'arithmétique est cette partie des mathématiques qui s'occupe de comment faire des calculs.
La structure de la mémoire d'un ordinateur est organisée en morceaux de 64 bits (actuellement). Il n'y a pas longtemps, elle était en 32 bits. On a donc 64 bits pour représenter un nombre réel.
Si **on garde une position fixe pour la virgule**, alors on va placer la virgule au milieu des 64 bits. On sera alors limité pour exprimer les petits et les grands nombres. En outre, ceux-ci ne seront pas représentés avec une grande précision.
Pour cela, on choisit de laisser flotter la virgule comme dans la représentation scientifique : c'est pourquoi les nombres réels sont appelés en informatique des nombres flottants. D'où le terme `float` pour désigner en informatique un nombre réel. On va donc représenter un `float` sous la forme

$$\text{s } 1.\text{m} \times 2^n \tag{4.1}$$

- où `s` désigne le signe
- `m` est appelé la `mantisse`

– et `n` est l'exposant

Le signe

Un signe est positif ou négatif. On a donc besoin de 2 valeurs pour le coder, soit 1 bit. Par convention, il a été choisi `0 = positif` et `1 = négatif`.

L'exposant

Il a été choisi de coder l'exposant sur 11 bits. On verra en exercice que c'est suffisant pour exprimer les plus petits et les plus grands nombres de la physique.
Sur 11 bits, on peut coder $2^{11} = 2048$ valeurs. L'exposant peut être positif ou négatif, on le représente par un entier relatif compris entre -1022 et 1023.

$$-1022 \leq n \leq 1023$$

On a vu, dans le cours sur la représentation des entiers relatifs, qu'on représentait les entiers relatifs par un entier naturel qui sera appelé `exposant` :

$$\underset{n+1023}{1 \leq \text{exposant} \leq 2046}$$

On peut remarquer que la valeur 0 et 2047 ne sont pas prises en compte. Elles sont réservées. `0` avec le bit de signe permet de désigner $\pm\infty$. De son côté, `2047` permet de désigner `NaN` (Not a Number), qui est renvoyé comme erreur quand on essaye de diviser par zéro par exemple.

La mantisse

Sur 64 bits, avec 1 bit de signe et 11 bits pour l'exposant, la mantisse est représentée sur $64-1-11 = 52$ bits.
Une propriété de la mantisse est supérieure ou égale à 1, mais strictement inférieure à 2 : si elle est égale à 2, alors on augmente la puissance de 2 dans la définition (4.1) :

$$1 \leq \text{mantisse} < 2$$

Une autre manière de dire la même chose est de dire que la mantisse s'écrit forcément sous la forme `1.xxxxx`. Comme elle commence toujours par un 1, on ne le représente pas en mémoire pour gagner en précision. On ne va donc représenter en mémoire uniquement la partie encadrée de la mantisse `1.`$\boxed{\texttt{xxxxx}}$ où les `xxxxx` représentent 52 bits !

La représentation en mémoire

Il a été fait le choix suivant dans la norme `IEEE 754`. En mémoire, un flottant sera représentée sous la forme suivante (sans les « / » qui ne sont là que pour faciliter la lecture) :

Norme IEEE 754

Le nombre représenté par la notation scientifique en base 2 par s $1.\text{m} \times 2^n$ est représenté sur 64 bits par

signe s	/	exposant $(n + 1023)$	/	mantisse m
1 bit		11 bits		52 bits

3 Exercices

Photocopier les deux pages suivantes en recto-verso et faire les exercices.

Tableau de conversion pour FLOAT

n	2^n										
13	8192										
12	4096										
11	2048										
10	1024										
9	512										
8	256										
7	128										
6	64										
5	32										
4	16										
3	8										
2	4										
1	2										
0	1										
-1	0.5										
-2	0.25										
-3	0.125										
-4	0.0625										
-5	0.03125										
-6	0.015625										
-7	0.0078125										
-8	0.00390625										
-9	0.00195312										
-10	0.000976562										
-11	0.000488281										

n	10	9	8	7	6	5	4	3	2	1	0
2^n	1024	512	256	128	64	32	16	8	4	2	1

Voici la représentation du nombre 132.34 :

`01000000011000001000101011100001010001111010111000010100011110 11`

que l'on comprend mieux sous cette forme :

`0 / 10000000110 / 0000100010101110000101000111101011100001010001111011`

où l'on reconnaît

- un nombre positif (bit de signe `0`)
- un exposant `10000000110 = 1030 = n + 1023`, soit `n = 7`
- la partie représentée de la mantisse commence par `000010001010111` sur ces 15 premiers bits. Il ne faut pas oublier le 1 qui vient devant !
- Si on se limite à ces 15 premiers bits, la mantisse est alors `1.`$\boxed{\texttt{000010001010111}}$ qui représente le nombre

$$1 + 2^{-5} + 2^{-9} + 2^{-11} + 2^{-13} + 2^{-14} + 2^{-15} = 1.033905029$$

ce qui donne finalement

$$+\left(1 + 2^{-5} + 2^{-9} + 2^{-11} + 2^{-13} + 2^{-14} + 2^{-15}\right) \times 2^{7} = 132.3398438$$

Comme on s'est limité aux 15 premiers chiffres de la partie représentée de la mantisse, on n'a qu'une approximation de la valeur qu'on voulait représentée, à savoir 132.34.

Algorithme de Malcolm

```
def algo_malcolm() :
    x = 1.0
    y = x + 1.0
    i = 0
    while y-x == 1.0 :
        x = 2.0 * x
        y = x + 1.0
        i = i + 1
    return i
```

Exercices

1. A quel nombre réel correspond la représentation binaire :

   `1 / 10010100110 / 1101000 ... 0` ?
2. Trouver la représentation binaire des nombres suivants : 128.412, 0.1 et 0.3.
3. Comment peut-on qualifier la mantisse pour 0.1 et 0.3 ?
4. Que vaut le test `0.1+0.1+0.1-0.3 == 0` ? Pourquoi ?
5. Que vaut le test `1+1+1-3 == 0` ? Pourquoi ?
6. La fonction `algo_malcolm` code-t-elle une boucle infinie ?
7. Quelle valeur sera effectivement renvoyée par l'algorithme de Malcolm ? A vérifier !