# Représentation en binaire des nombres réels (Float)

– Lycée Vaugelas –

## 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 : 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.

Comment faire flotter la virgule ? En utilisant un modèle similaire à la notation scientifique ! On va donc représenter un `float` sous la forme

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

- 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 (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) :

| signe | / | exposant | / | mantisse |
|---|---|---|---|---|
| 1 bit | | 11 bits | | 52 bits |

Voici la représentation du nombre 132.34 :

`0100000001100000100010101110000101000111101011100001010001111011`

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.

# 3 Exemples

### Exercice 1

A quel nombre réel correspond la représentation binaire suivante ?

`1 / 10010100110 / 1101000 ... 0`

La solution est donnée en fin de document.

### Comment faire dans l'autre sens ?

Comment trouver la représentation binaire d'un réel ? On va représenter en binaire la valeur de `0.40625`. Pour cela, on va utiliser l'annexe qui s'intitule `Tableaux de conversion pour Float`.

On parcourt la colonne en partant du haut vers le bas : on chercher la première plus petite que la valeur à convertir (ici 0.40625) : on s'arrête donc à 0.25 et on met un 1 dans la première colonne vide à droite.

On continue en recommençant avec le reste : $0.40625 - 0.25 = 0.15625$.

Ce qui donne :

| | | | |
|---|---|---|---|
| -2 | 0.25 | 1 | $0.40625 - 0.25 = 0.15625$ |
| -3 | 0.125 | 1 | $0.15625 - 0.125 = 0.03125$ |
| -4 | 0.0625 | 0 | on ne peut pas car $0.03125 - 0.0625 < 0$ |
| -5 | 0.03125 | 1 | $0.03125 - 0.03125 = 0$ |
| -6 | 0.015625 | 0 | reste nul |
| -7 | 0.0078125 | 0 | reste nul |

Dans cet exemple, on s'arrête au rang de $2^{-5}$ car le reste est nul. Les deux lignes suivantes ont quand même étaient représentées.

Si le reste n'est pas nul, on continue. On s'arrête, soit parce que le reste devient nul plus tard, soit parce qu'on a rempli les 52 bits de la partie représentée de la mantisse.

La partie représentée de la mantisse est celle qui est en vert dans le tableau.

Pour trouver la valeur de l'exposant, on regarde la première valeur non nulle en descendant la colonne depuis le haut est la case jaune qui correspond à la valeur `n=-2`. Le codage binaire de 1024 et 1023 est indiqué pour comprendre comment trouver rapidement le code binaire de -2+1023 : il faut seulement retirer 2 à 1023.

| | $n$ | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $2^n$ | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| | 1024 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1023 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| -2 +1023 | 1021 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

Le signe est positif, donc le bit de signe sera 0.

La représentation binaire recherchée est donc :

`0 / 01111111101 / 10100... 0`

### Exercice 2

Trouver la représentation binaire des nombres suivants : 128.412, 0.1 et 0.3.

On pourra s'aider des tableaux de conversion.

# 4 Conséquences

Est-ce que nous avons manipulé une représentation `exacte` d'un nombre réel ?

La réponse est évidente : la partie représentée de la mantisse étant coupée à 52 bits, dans de nombreux cas, on fait une approximation.

Ce qui se traduit par le fait qu'un test `0.1+0.1+0.1-0.3 == 0` est **faux**. Alors qu'il sera **vrai** si on fait `1+1+1-3 == 0`, car la représentation des entiers est exacte.

**Conclusion : Les nombres réels sont représentés de manière approchée en binaire.** Une conséquence est qu'il faut bannir les tests sur des réels (et notamment le test à zéro 0.0) dans vos programmes.

Sur calculatrice, le test `0.1+0.1+0.1 == 0.3` sera **vrai** car c'est un autre choix de représentation des réels qui a été choisi.

**Exercice 3 : Algorithme de Malcolm**

Vous avez l'algorithme de Malcolm au dos des `tableaux de conversion Float`.

(1) Montrer qu'on doit s'attendre à une boucle infinie.

(2) En fait, ce n'est le cas. L'algorithme va s'arrêter et afficher une valeur. Pouvez-vous prédire laquelle ?

(3) Passer sur machine et tester cet algorithme. Aviez-vous raison ?

### Exercice 1 : correction

A quel nombre réel correspond la représentation binaire suivante ?

`1 / 10010100110 / 1101000 ... 0`

Le bit de signe est à 1, le nombre est donc négatif.

L'exposant est `10010100110 = 1024+128+32+4+2 = 1190`.

La mantisse est donc $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = \frac{16+8+4+1}{16} = \frac{29}{16} = \frac{29}{2^4}$

Le nombre réel est donc $-\frac{29}{2^4} \times 2^{1190-1023} = -29 \times 2^{163} = 3.390683799 \times 10^{50}$

### Exercice 2 : correction

| | |
|---|---|
| 128.412 | `0 / 10000000110 / 0000000011010010111100011010100111111011111001110111` |
| 0.1 | `0 / 01111111011 / 1001100110011001100110011001100110011001100110011010` |
| 0.3 | `0 / 01111111101 / 0011001100110011001100110011001100110011001100110011` |

Voir aussi le scan des tableaux de conversion page suivante.

### Exercice 3 : correction

L'algorithme de Malcolm s'arrêtera en affichant 53 car la mantisse est codée sur 52 bits et elle devient approximative sur le 53ème bit.

# Tableaux de conversions pour FLOAT

| $n$ | $2^n$ | 0,40625 | 128,412 | 12,94 | 1,294 | 0,1 | 0,2 | 0,3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 8192 | | | | | | | | | | |
| 12 | 4096 | | | | | | | | | | |
| 11 | 2048 | | | | | | | | | | |
| 10 | 1024 | | | | | | | | | | |
| 9 | 512 | | | | | | | | | | |
| 8 | 256 | | | | | | | | | | |
| 7 | 128 | | 1 | | | | | | | | |
| 6 | 64 | | 0 | | | | | | | | |
| 5 | 32 | | 0 | | | | | | | | |
| 4 | 16 | | 0 | | | | | | | | |
| 3 | 8 | | 0 | 1 | | | | | | | |
| 2 | 4 | | 0 | 0 | | | | | | | |
| 1 | 2 | | 0 | 0 | | | | | | | |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | | | | | |
| -1 | 0.5 | 0 | 0 | 1 | 0 | 0 | | | | | |
| -2 | 0.25 | 1 | 1 | 1 | 1 | 0 | | 1 | | | |
| -3 | 0.125 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | | | |
| -4 | 0.0625 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | | | |
| -5 | 0.03125 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | | | |
| -6 | 0.015625 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | | | |
| -7 | 0.0078125 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | | |
| -8 | 0.00390625 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | | | |
| -9 | 0.00195312 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | | | |
| -10 | 0.000976562 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | | | |
| -11 | 0.000488281 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | | | |

mantisse

| | $n$ | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $2^n$ | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
| -2+1023 | 1021 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | (1024-3) |
| 7+1023 | 1030 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | (1024+6) |
| 3+1023 | 1026 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | (1024+2) |
| 0+1023 | 1023 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | (1024-1) |
| -4+1023 | 1019 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | (1024-5) |
| | | | | | | | | | | | | | | |

11 bits de l'exposant