1 Nombres binaires fractionnaires

Pour un nombre binaire, la partie fractionnaire (les bits "à droite de la virgule") représentent les puissances de 2 négatives.

Exemple : $(0,1101)_2 = 2^{-1} + 2^{-2} + 0 + 2^{-4} = \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = 0,5 + 0,25 + 0,0625 = (0,8125)_{10}$.

> « Décaler » la virgule d'un cran vers la droite revient à multiplier le nombre par 2, et vers la gauche à diviser le nombre par 2.

Exemple : $0,1101 = 1,101.2^{-1} = 11,01.2^{-2} = 0,0001101.2^3$
(on s'est autorisé à écrire les puissances de 2 en base décimale)
On retrouve l'équivalent de la base 10 en remplaçant les puissances de 10 par des puissances de 2.

2 Conversion des nombres décimaux en nombres binaires

On souhaite convertir la partie décimale d'un nombre en base 10 vers sa représentation binaire (on sait déjà convertir la partie entière : voir chap. P1-1).

Méthode :
- Multiplier la partie décimale par 2 et relever la valeur de la partie entière (0 ou 1).
- Recommencer avec la partie décimale du résultat précédent.
- Poursuivre ainsi jusqu'à ce que la partie décimale soit nulle.

La partie fractionnaire du nombre binaire est alors la suite des parties entières (0 ou 1) obtenus dans l'ordre des opérations.

Algorithme : Convertir le décimal N en binaire

```
partie_decimale ← N
liste ← creerListeVide()
tant que partie_decimale ≠ 0 faire
    nombre ← partie_decimale × 2
    partie_entiere ← PartieEntiere(nombre)
    partie_decimale ← nombre − partie_entiere
    ajouter partie_entiere à liste
fin tq
renvoyer liste
```

Exemple : Convertir 0,828125 en binaire :
$0,828125 \times 2 = 1,65625 = \mathbf{1} + 0,65625$
$0,65625 \times 2 = 1,3125 = \mathbf{1} + 0,3125$
$0,3125 \times 2 = 0,625 = \mathbf{0} + 0,625$
$0,625 \times 2 = 1,25 = \mathbf{1} + 0,25$
$0,25 \times 2 = 0,5 = \mathbf{0} + 0,5$
$0,5 \times 2 = 1,0 = \mathbf{1} + 0$

Conclusion : $(0,828125)_{10} = (0,110101)_2$

3 Nombres à virgule fixe

On cherche une représentation en machine des nombres décimaux, forcément avec un nombre de bits limités (supposons sur 1 octet).
Une première idée consiste à utiliser n bits pour la partie entière et p bits pour la partie décimale (la virgule, de position fixe, se situe au milieu). Mais combien de bits consacrer à chaque partie ?
Si on garde beaucoup de bits pour la partie décimale, on sera précis sur les nombres représentés, mais ils resteront forcément petits (car peu de bits pour la partie entière). Au contraire si on garde beaucoup de bits pour la partie entière, les nombres pourront être grands, mais jamais précis.

Exemples extrêmes sur un octet :

1. On utilise 7 bits pour la partie entière et 1 seul bit pour la partie décimale.
   La partie entière permet de représenter les nombres de 0 à $(2^7 - 1) = 127$, et la partie décimale seulement 0,0 ou 0,5 (2^{-1}).
2. On utilise 1 seul bit pour la partie entière et 7 bits pour la partie décimale.
   La partie entière permet de représenter seulement les nombres 0 ou 1, et la partie décimale de 0,0 à 0,9921875 ($2^{-7} + 2^{-6} + 2^{-5} + 2^{-4} + 2^{-3} + 2^{-2} + 2^{-1}$), par pas de 0,0078125 (2^{-7}).

Autres exemples (6 bits de partie entière 2 bits de partie décimale) :

0	1	0	0	1	1	,	0	0
					19	,	0	
1	1	1	1	1	1	,	0	1
					63	,	25	
1	0	0	0	0	1	,	1	0
					33	,	5	
0	0	1	1	0	0	,	1	1
					12	,	75	

La solution qui a été trouvée est de ne pas donner de position fixe à la virgule pour pouvoir représenter une plus grande gamme de nombres avec plus ou moins de précision.

4 Nombres à virgule flottante

Pour les nombres à virgule flottante, on réserve un nombre de bits pour coder les chiffres significatifs du nombre, et d'autres bits pour coder sa puissance de 2 (comme en notation scientifique).

4.1 Définition

Un nombre flottant est la représentation informatique d'un nombre réel décimal. La norme IEEE-754 spécifie la manière de représenter ces nombres flottants (*en simple ou double précision*).

Un **nombre flottant** s'exprime sous la forme : $\boldsymbol{N = s \times m \times 2^e}$.
$\boldsymbol{s}$ est le signe, $\boldsymbol{m}$ la mantisse, $\boldsymbol{e}$ l'exposant.

Représentation en simple précision :
Le nombre est représenté sur 4 octets (32 bits) avec 1 bit pour le signe, 8 bits pour l'exposant, et 23 bits pour la mantisse normalisée.
La mantisse **normalisée** doit être sous la forme 1,**xx...xx** et les 23 bits coderont sa partie fractionnaire **xx...xx**.

4.2 Exemple : représentation en simple précision du nombre $(159.828125)_{10}$

Partie entière : $(159)_{10} = (1001\ 1111)_2$
Partie décimale : $(0,828125)_{10} = (0,1101\ 01)_2$
Donc $(159,828125)_{10} = (1001\ 1111,1101\ 01)_2$

Écriture en mantisse normalisée : $1001\ 1111,1101\ 01 = 1,0011\ 1111\ 1010\ 1 \times 2^7$
La mantisse se code sur 23 bits, donc on rajoute autant de 0 à droite que nécessaire. Ici 10 zéros pour obtenir $m = 0011\ 1111\ 1010\ 1000\ 0000\ 000$.

L'exposant est codé sur 8 bits et peut a priori varier de -127 à +128. Mais la norme IEEE-754 impose un exposant positif, donc la norme impose d'ajouter 127 à l'exposant avant de le convertir en binaire.

Ici on a donc $e = 7 + 127 = 134 = 1000\ 0110_2$.
(Si l'exposant a moins de 8 bits, on le complète avec des 0 à gauche.)

Le signe est positif donc le 1er bit est $s = 0$ (ce serait 1 pour un nombre négatif).

Au final, la représentation est :

s	e	m
0	1000 0110	0011 1111 1010 1000 0000 000

qu'on peut réécrire (en groupes de 4 bits) :

0100	0011	0001	1111	1101	0100	0000	0000

ou même en valeur hexadécimale pour plus de concision : $(43\ 1F\ D4\ 00)_{16}$

Remarque : les exposants -127 et +128 (donc après décalage : 0 ou 255) sont réservés pour représenter des nombres **dénormalisés** : zéro, l'infini ou **NaN** (Not a Number : pour traiter des cas particuliers).

Bonus : Il existe des convertisseurs en ligne pour s'exercer : `https://www.h-schmidt.net/FloatConverter/IEEE754.html` ou `http://www.binaryconvert.com/convert_float.html`

5 Problème des nombres non dyadiques

5.1 Mise en évidence du problème

Les **nombres dyadiques** sont des quotients d'un nombre entier par une puissance de 2.

Exemples : $0,5 = \frac{1}{2}$ $\quad 0,3125 = \frac{5}{16}$ $\quad 159,828125 = \frac{10229}{2^6}$

Ces nombres **s'écrivent avec un nombre de chiffres finis en binaire**. Donc si le nombre de bits est suffisant en machine, on pourra les représenter de manière exacte.
Mais qu'en est-il des nombres non dyadiques comme 0,1 par exemple ?

Remarque : on a le même problème en base 10 avec les nombres qui ne sont pas des quotients de puissance de 10.
Exemple : 1/3 s'écrit avec un nombre infinis de chiffres en base 10 (0,333333...).

5.2 Représentation en simple précision du nombre $(0,1)_{10}$

1ère étape : Convertir 0,1 en binaire :
$0,1 \times 2 = 0 + 0,2$
$0,2 \times 2 = 0 + 0,4$
$0,4 \times 2 = 0 + 0,8$
$0,8 \times 2 = 1 + 0,6$
$0,6 \times 2 = 1 + 0,2$
$0,2 \times 2 = 0 + 0,4$
$\cdots etc \cdots$
On constate à ce stade qu'on va retrouver à la suite la même répétition.
Conclusion : $(0,1)_{10} = (0,0\ 0011\ 0011\ \cdots)_2$

Cette suite est **infinie** et on va donc devoir la tronquer (mantisse limitée à 23 bits).

2ème étape : écriture de la mantisse normalisée :
$0,0\ 0011\ 0011\ \cdots = 1,1001\ 1001\ 1001\ 1001\ \cdots \times 2^{-4}$
$m = 1001\ 1001\ 1001\ 1001\ 1001\ 100$

3ème étape : écriture de l'exposant (+127) :
$2^{-4} \longmapsto 2^{123}$ $\quad (123)_{10} = (111\ 1011)_2$
Ajusté sur 8 bits : $e = 0111\ 1011$

Bilan :
Au final, la représentation est (signe positif $s = 0$) :

s	e	m
0	0111 1011	1001 1001 1001 1001 1001 100

qu'on peut réécrire (en groupes de 4 bits) :

0011	1101	1100	1100	1100	1100	1100	1100

et en valeur hexadécimale pour plus de concision $(3D\ CC\ CC\ CC)_{16}$

Si on « retraduit » 0011 1101 1100 1100 1100 1100 1100 1100 vers la base 10, on ne retrouve pas 0,1 ! ! !
Ceci est dû à la "troncature" qu'on a effectué sur la mantisse.

Exercice : Chercher de même la représentation de 0,25 et 1/3.

0011	1110	1000	0000	0000	0000	0000	0000	$\Leftarrow 0,25$
0011	1110	1010	1010	1010	1010	1010	1010	$\Leftarrow 1/3$

5.3 Précision et arrondi

> Ces arrondis rendent délicate et dangereuse l'utilisation de flottants en programmation. On retiendra que tout test d'égalité sur les flottants est à proscrire.

.
Exemple classique : En Python, tester : $0.1 + 0.2 == 0.3$
De façon qui peut sembler surprenante, la valeur de cette expression est *False*.

Anecdotes malheureuses :
- Crash de la fusée Ariane 5 en 1996 pour avoir tronqué un nombre sur 16 bits (au lieu de 64).
- Missile SCUD non intercepté en 1991 par un missile Patriot (anti-balistique) à cause d'arrondis de 1/10 (horloge interne).