NSI	**Représentation numérique de l'information** Nombres relatifs et nombres flottants	**Cours 7**

1. Nombres relatifs

Pour représenter des entiers, nous utilisions un nombre quelconque de bits.
A partir de maintenant, il est important de définir le **nombre de bits** qui seront utilisés pour pour représenter les nombres (4, 8, 16, 32 ou 64 bits).

1.1. Le bit de signe

La première possibilité pour représenter un nombre relatif de **n bits** serait de réserver un bit (le bit de poids fort : le plus à gauche) au signe (0 : positif, 1 : négatif). Un nombre de **n bits** se décomposerait de la manière suivante :

- **1 bit** pour représenter le signe
- **n-1 bit** pour représenter la valeur absolue du nombre à représenter.

Exemple : Représentons sur 4 bits en binaire, les entiers 5 et -5.

5 : 0101
-5 : 1101

Inconvénients de cette méthode :

L'existence de deux zéros : un zéro positif (0000) et un zéro négatif (1000)
L'algorithme d'addition classique ne fonctionne plus avec les nombres négatifs :
5 + (-5) = 0010 = 2

1.2. Méthode du complément à 2^n :

Pour cette méthode, nous allons chercher à représenter l'opposé d'un nombre **r** tel que :

r + (-r) = 0

Methode : déterminons la représentation de -12 sur 8 bits

- Représentons 12 sur 8 bits : 0000 1100
- **Inversons** tous les bits (les bits à 1 passent à 0 et vice versa) : 1111 0011
- **Ajoutons 1** au nombre obtenu à l'étape précédente : 1111 0100

- Vérification : 12 + (-12) : 0000 1100 + 1111 0100 = (1) 0000 0000 = 0

Astuce : Pour aller plus vite, an partant de la droite, je garde le premier 1 et inverse tout le reste.

Exercice : En utilisant le complément à 2, représentez -15 (sur 8 bits)

15 en binaire : 0000 1111
Inversion : 1111 0000
Ajout 1 : 1111 0001

Exercice : Représentez sur 8 bits l'entier **4** puis, toujours sur 8 bits, l'entier **-5**. Additionnez ces 2 nombres (en utilisant les représentations binaires bien évidemment) et vérifiez que vous obtenez bien **-1**.

4 : 0000 0100
-5 : 1111 1011
1111 1111 : -1

Exercice : Quel est le plus petit entier négatif que l'on peut représenter sur 8 bits ?

1000 0000 soit -128

Exercice : Quel est le plus grand entier positif que l'on peut représenter sur 8 bits ?

0111 1111 soit 127

Exercice : Quelles sont les bornes inférieure et supérieure d'un entier relatif codé sur 16 bits ?

-2^{n-1} et $+2^{n-1}- 1$

Exercice : En utilisant le complément à 2, représentez -25 (sur 8 bits)

1110 0111

2. Nombres flottants

En base 10, les chiffres à gauche de la virgule sont les puissances positives de 10 et les chiffres à droite de la virgule sont les puissances négatives de 10 :

10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}
1000	*100*	*10*	*1*	*1/10*	*1/100*	*1/1000*	*1/10000*
		1	5	1	8	7	5

Pour la base 2, on peut faire de même :

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	*4*	*2*	*1*	*1/2*	*1/4*	*1/8*	*1/16*
1	1	1	1	0	0	1	1

Représentons **15,1875** en binaire :

- Pour la partie de gauche, on fait comme pour les entiers, par divisions successives par 2 :
  15 : **1111**
- Pour la partie de droite, nous allons procéder par multiplications successives par 2 et garder les parties entières:
  0,1875 x 2 = **0**,375
  0,375 x 2 = **0**,75
  0,75 x 2 = **1**,5
  0,5 x 2 = **1**
  Et cette fois nous gardons dans l'ordre : **0011**

$(15,1875)_{10} = (1111,0011)_2$

Représentons 4,125 en binaire

$(100,001)_2$

Trouvons la représentation décimale de $(110,1011)_2$

1x4 + **1**x2 +**0**x1 = 6

1 x 1/2 + **0** x 1/4 + **1** x 1/8 + **1** x 1/16 = **1** x 0,5 + **0** x 0,25 + **1** x 0,125 + **1** x 0,0625 = 0,6875

$(110,1011)_2 = (6,6875)_{10}$

Maintenant, trouvons la représentation binaire de $(0,1)_{10}$

Pour la partie entière pas de problème $(0)_{10} = (0)_2$
Pour la partie décimale :
0,1 x 2 = **0**,2
0,2 x 2 = **0**,4
0,4 x 2 = **0**,8
0,8 x 2 = **1**,6
0,6 x 2 = **1**,2
0,2 x 2 = **0**,4 -> à partir d'ici, on recommence sans fin …
0,4 x 2 = **0**,8
0,8 x 2 = **1**,6
0,6 x 2 = **1**,2

$(0,1)_{10} = (0,0001100110011001100110011001100110011…)_2$

On constate ici que le nombre 0,1 décimal à une écriture infinie en binaire.
Nous connaissons déjà ce phénomène en base 10, par exemple :

1/3 = 0,3333333333333…

3. Représentation des flottants

Ecriture Scientifique
En base dix, pour représenter de très grands nombres flottants, nous utilisons l'écriture scientifique avec les puissances positives ou négatives de 10, par exemple :

15467890,34 = 1,546789034 x 10 000 000 = 1,546789034 x 10^{7}
0,0000787 = 7,87 x 0,00001 = 7,87 x 10^{-5}
13,5625 = 1,35625 x 10 = 1,35625 x 10^{1}

Il est possible de faire la même chose en binaire avec les puissances de 2 :

1101,1001 = 1,1011001 x 2^{11}

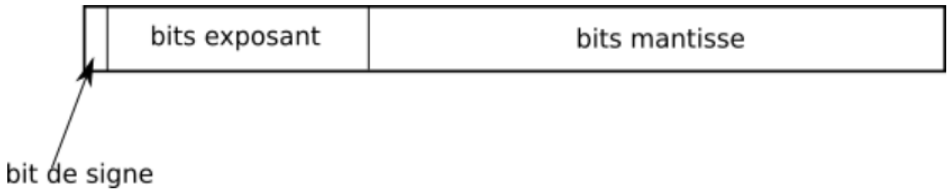
Pourquoi 2^{11} ? : Ici nous avons décalé de 3 rangs vers la gauche soit donc 11 en binaire

Dans un ordinateur
La norme **IEEE 754** est la norme la plus employée pour la représentation des nombres à virgule flottante dans le domaine informatique. La première version de cette norme date de **1985**.
Nous allons étudier deux formats associés à cette norme :

- Le format "**simple précision**" qui utilise **32 bits** pour écrire un nombre flottant
- Le format "**double précision**" utilise **64 bits**.

Que cela soit en simple précision ou en double précision, la norme **IEEE754** utilise :

- 1 bit de signe (1 si le nombre est négatif et 0 si le nombre est positif)
- des bits consacrés à l'exposant (8 bits pour la simple précision et 11 bits pour la double précision)
- des bits consacrés à la mantisse (23 bits pour la simple précision et 52 bits pour la double précision)



Nous pouvons vérifier que l'on a bien :

- *1 + 8 + 23 = 32 bits pour la simple précision*
- *1 + 11 + 52 = 64 bits pour la double précision*

Ces trois parties sont codées en binaire et concaténées pour former un nombre de 32 ou 64 bits.

En simple précision sur 32 bits, voyons comment coder notre nombre $(13,5625)_{10} = (1101,1001)_2$:

- Tout d'abord, on l'écrit de manière scientifique avec les puissances de 2 (voir ci-dessus):
  1,1011001 x 2^{11}
- Notez qu'en binaire, le 1er chiffre avant la virgule sera toujours 1, donc on ne le codera pas. La mantisse sera donc la partie soulignée 1,<u>1011001</u> x 2^{11}
  On la codera sur 23 bits en complétant avec des 0 à droite, ce qui donnera :
  10110010000000000000000
- Notre exposant est 11, mais comme vous le remarquez, il n'y a pas de bit de signe pour celui-ci, pour éviter de le coder, on ajoutera toujours 127 (1023 pour la double précision) à la valeur de l'exposant (on l'appelera l'exposant décalé) pour simplifier, faire le calcul en décimal et repasser en binaire :
  $(11)_2 = (3)_{10}$ => $(3)_{10} + (127)_{10} = (130)_{10} = (10000010)_2$
  Si nécessaire on compètera notre exposant par des 0 à gauche pour atteindre 8 bits (inutile dans notre cas)

Maintenant, recollons les morceaux : **1**bit de Signe + 8bits d'exposant et 23bits de mantisse :

010000010**10110010000000000000000**

Le site http://www.binaryconvert.com/ permet de faire les conversions et vérifier.

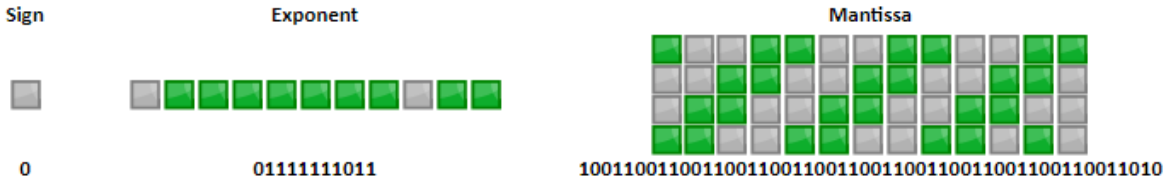
Note :
En simple précision, le plus grand nombre flottant représenté est :
011111111111111111111111111111111 soit **3.4028 x 10^{38}**

En double précision le plus grand nombre flottant a une mantisse égale en binaire à 1,111……..1 avec 52 chiffres après la virgule et un exposant 111 1111 1111. Soit **1.7977 x 10^{308}**

Un nombre réel compris entre ce plus grand nombre et son opposé est alors représenté par un nombre décimal, un flottant qui est la meilleure approximation possible.

En double précision pour $(0,1)_{10} = (0,0001100110011\ldots)_2 = 1,1001100110011001100110011\ldots \times 2^{-4}$
Le signe est **+**, donc le premier bit vaut **0**. L'exposant décalé vaut -4 + 1023 = 1019, donc l'exposant est codé par 011 1111 1011.

La mantisse s'écrit en binaire : 1001 1001 …1001 1001… sans fin, il faut donc l'arrondir. Comme le 53ème chiffre est un 1, on arrondit par valeur supérieure (1001 est arrondi à 1010), et donc on code : 1001100110011001100110011001100110011001100110011010 1001100110011001100110011001100110011001100110011010



Exemples

Voyons deux exemples de représentation en machine en double précision

Le réel 20 s'écrit $+1,25x2^4$ et le réel -0,375 s'écrit-1,5 2^{-2}.

Exemple 1 : Codage du réel **20** :
$(20)_{10} = (10100)_2$ => $1,0100 \times 2^4$

- Le signe est + donc le premier bit vaut **0**
- L'exposant est 4 donc l'exposant décalé est 4 + 1023 = 1027 soit **100 0000 0011** en binaire sur 11 bits.
- La mantisse est 1,01. On garde donc la partie décimale 01 et on complète avec des zéros à droite.

Le codage de 20 est donc : 0 100 0000 0011 0100 0000 . . . 0000.

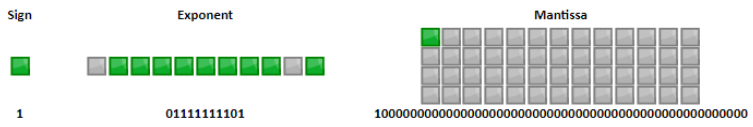
binaryconvert.com
(convertisseur en ligne)

Exemple 2 : Codage du réel **-0.375** :
$(-0,375)_{10} = (-0,011)_2$ => $1,1 \times 2^{-2}$

- Le signe est - donc le premier bit vaut **1**
- L'exposant est -2 donc l'exposant décalé est -2 + 1023 = 1021, soit 11 1111 1101 en binaire qu'on complète à gauche sur 11 bits par des zéros : **011 1111 1101**.
- La mantisse est 1,1. On garde la partie décimale 1 et on complète avec des zéros à droite.

Le codage de -0,375 est donc : 1 011 1111 1101 1000 0000 . . . 0000.

binaryconvert.com
(convertisseur en ligne)



4. Calculs

En Python, les nombres réels sont représentés par des nombres en virgule flottante qui sont du type ***float***. L'écriture permet à Python de considérer un nombre comme étant du type ***float***. Par exemple les écritures contenant un point décimal ou une puissance de 10 comme 3.14 ou 1. ou .0001 ou 1e12, définissent un nombre de type ***float***. Ce peut être aussi le résultat d'un calcul comme une division a/b.

Problème d'arrondi
Comme nous l'avons vu précédemment, certains nombres comme 0,1 sont représentés de manière arrondie en machine, ceci entraine des résultats parfois étranges :

```
>>> a = 0.1
>>> b = 0.2
>>> c = 0.3
>>> d = a + b
>>> c
0.3
>>> d
0.30000000000000004
>>> c==d
False
```

On évitera donc toujours de faire des comparaisons entre réels.

Overfow
Nous avons vu que même en double précision nous sommes limités, lorsque dans un calcul la valeur maximale est dépassée, le programme s'arrête et une erreur est signalée : **Overflow**

Le programme doit être modifié.

Ces deux problèmes peuvent avoir et ont eu dans le passé des conséquences graves :

- Pendant la première guerre du Golfe en **1991**, les anti-missiles américains (***Patriot***) chargés d'intercepter les missiles Irakiens (*Scud*) avaient une horloge interne émettant un signal toutes les **0,1 secondes**, le codage de ce nombre n'étant pas exact en machine, un décalage de l'horloge interne du missile a entrainé un décalage du missile de **550m** de sa cible provoquant la mort de 28 militaires américains.

- L'autodestruction de la fusée **Ariane 5**, lancée le 4 juin **1996**, quelques secondes après le décollage a été causée par un problème d'**overflow** ! La reprise du code d'Ariane 4 utilisait une variable contenant l'accélération horizontale codée sur **8 bits**. Cependant Ariane 5 était beaucoup plus puissante et la valeur a dépassé la valeur maximale (sur 8 bits : 256)