A.III. Représentation des nombres en informatique

Comprendre comment sont représentés les nombres en informatique vous permettra peut-être un jour d'éviter de porter la responsabilité de l'explosion d'une fusée comme Ariane 5, dont l'explosion fût d'origine numérique… Ce paragraphe devrait vous en apprendre plus sur ce qu'une machine peut faire et ne pas faire avec les nombres.

Prenons un exemple simple. Calculons avec python :

$$A = 0{,}2 + 0{,}1$$

Le résultat est très simple, n'est-ce pas ?

$$A = 0{,}2 + 0{,}1 = 0{,}3$$

Voyons de résultat sous Python :

```
>>> 0.2+0.1
0.30000000000000004
```

OUPS

Ou encore :

```
>>> 0.3==0.1+0.2
False
```

A.III.1 Code binaire

A.III.1.a Introduction

Un ordinateur manipule des informations binaires, c'est-à-dire à deux états : 0 ou 1

Il est donc nécessaire de traduire un nombre du système en base 10 en un nombre binaire afin de permettre à un système informatique de le manipuler.

Par exemple, le 10 de notre système en base 10 est représenté par le « nombre » 1010 en binaire.

On écrira :

$$10_{(10)} = 1010_{(2)}$$

Le principe est très simple. En base 10, on ajoute un nouveau chiffre à chaque fois que l'on dépasse la valeur 9, 99, 999 etc. puis on l'incrémente. En binaire, on ajoute cette retenue dès que l'on dépasse la valeur 1.

Ainsi, pour les 5 premiers entiers :

Base 10	Base 2
0	0
1	1
2	10
3	11
4	100
5	101

On appelle chaque information valant 1 ou 0 un « **bit** ». Un « **mot »** est une suite de « bits ». Un **octet** est un mot de 8 bits.

Un nombre entier naturel codé sur n bits permettra de représenter 2^n valeurs différentes et pourra donc au maximum correspondre à la valeur en base 10 de $2^n - 1$, le 0 étant inclus. Par exemple, un nombre entier codé sur 8 bits donnera 256 valeurs différentes et ne pourra excéder $2^8 - 1 = 255$. Ce nombre s'écrira en base 2 :

$$255_{(10)} = 11111111_{(2)}$$

A.III.1.b Principe de l'écriture d'un entier dans les bases 2 et 10

Prenons un exemple dans la base 10 :

$$352_{(10)} = 3 * 10^2 + 5 * 10^1 + 2 * 10^0$$

Chaque chiffres (digit) 3, 5 et 2 correspond à une puissance de 10.

En binaire, on respecte le même principe mais avec des puissances de 2 :

$$1010_{(2)} = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

A.III.1.c Transcodage Entier ↔ Binaire

Soient :

- Un nombre en base 10 de digits d_i tel qu'il s'écrive $(d_n d_{n-1} \dots d_1 d_0)_{10}$
- Un nombre binaire de digits b_i tel qu'il s'écrive $(b_m b_{m-1} \dots b_1 b_0)_2$

Ecrivons :

$$d_n 10^n + d_{n-1} 10^{n-1} + \cdots + d_1 10^1 + d_0 10^0 = b_m 2^m + b_{m-1} 2^{m-1} + \cdots + b_1 2^1 + b_0 2^0$$

Le transcodage d'un entier

- Binaire/Base 10 consiste à trouver les digits d_i connaissant les digits b_i
- Base 10/binaire consiste à trouver les digits b_i connaissant les digits d_i

A.III.1.c.i Binaire → Entier

Rien de plus simple, connaissant le nombre en binaire, il suffit de procéder à la somme de chacun de ses digits pris de droite à gauche en multipliant par des puissances de 2 croissantes.

Exemple : $1111101000_{(2)}$

$$1111101000_{(2)} = 1.2^9 + 1.2^8 + 1.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 0.2^1 + 0.2^0$$

$$1111101000_{(2)} = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^3 = 512 + 256 + 128 + 64 + 32 + 8$$

$$1111101000_{(2)} = 1.10^3 + 0.10^2 + 0.10^1 + 0.10^0$$

$$1111101000_{(2)} = 1000_{(10)}$$

A.III.1.c.ii Entier → Binaire

Supposons qu'un nombre entier N s'écrive :

$$N = b_m 2^m + b_{m-1} 2^{m-1} + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

Par définition, les digits b_i ne sont pas divisibles par 2.

Divisons ce nombre par 2 :

$$\frac{N}{2} = (b_m 2^{m-1} + b_{m-1} 2^{m-2} + \cdots + b_2 2^1 + b_1) + \frac{b_0}{2}$$

On a donc :

$$N = 2q_0 + r_0 = 2 * (b_m 2^{m-1} + b_{m-1} 2^{m-2} + \cdots + b_2 2^1 + b_1) + b_0$$

On trouve donc le dernier digit b_0 comme reste de la division euclidienne de N par 2.

De même, on a :

$$q_0 = 2q_1 + r_1 = 2(b_m 2^{m-1} + b_{m-1} 2^{m-2} + \cdots + b_2) + b_1$$

On trouve donc l'avant dernier digit b_1 comme reste de la division euclidienne de q_0 par 2.

On procède alors ainsi jusqu'à ce que :

$$q_{m-1} = 2q_m + r_m = b_m$$

Avec $q_m = 0$, il reste alors b_m.

Récapitulons :

$N = 2q_0 + r_0$	$r_0 = b_0$
$q_0 = 2q_1 + r_1$	$r_1 = b_1$
…	…
$q_{m-1} = 2q_m + r_m$ $q_{m-1} = q_m$	$r_m = b_m$

Exemple : $1000_{(10)}$

	1000	2									
$n_0 =$	0	500	2								
	$n_1 =$	0	250	2							
		$n_2 =$	0	125	2						
			$n_3 =$	1	62	2					
				$n_4 =$	0	31	2				
					$n_5 =$	1	15	2			
						$n_6 =$	1	7	2		
							$n_7 =$	1	3	2	
								$n_8 =$	1	1	2
									$n_9 =$	1	**0**

$$1000_{(10)} = 1111101000_{(2)}$$

A.III.1.d Transcodage Réel base 10 ↔ Réel base 2

En base 10, lorsque nous manipulons des nombres à virgule :

- Les chiffres avant la virgule sont des puissances de 10
- Les chiffres après la virgule sont des puissances de 1/10

Exemple :

$$5{,}375 = 5.10^0 + 3.10^{-1} + 7.10^{-2} + 5.10^{-3}$$

On peut exprimer ce nombre réel comme la somme de sa partie entière et de sa partie décimale :

$$5{,}375 = 5 + 0{,}375$$

En binaire, on va appliquer le même principe. Ainsi, le nombre $101{,}011$ représente le nombre :

$$1.2^2 + 0.2^1 + 1.2^0 + 0.2^{-1} + 1.2^{-2} + 1.2^{-3} = 4 + 0 + 1 + 0 + \frac{1}{4} + \frac{1}{8} = 5{,}375$$

Là aussi, on peut exprimer ce nombre comme somme de sa partie entière et sa partie décimale :

$$101{,}011 = 101 + 0{,}011$$

On pourra remarquer qu'il y a concordance entre partie entière et partie décimale entre les deux écritures :

$$(101)_2 = (5)_{10} \quad ; \quad (0{,}011)_2 = (0{,}375)_{10}$$

A.III.1.d.i Réel base 2 → Réel Base 10

Pour transcoder un nombre binaire réel en nombre réel en base 10, c'est très simple maintenant que l'on a compris le principe de l'écriture d'un nombre binaire réel.

Ecrivons un nombre binaire réel sous la forme :

$$x = (b_n \dots b_0, c_1 \dots c_m)_2$$

Avec b_i et c_i les bits du mot binaire associé au nombre représenté.

On a alors :

$$(x)_{10} = b_n.2^n + \cdots + b_0.2^0 + c_1.2^{-1} + \cdots + c_m.2^{-m}$$

A.III.1.d.ii Réel base 10 → Réel base 2

Soit un nombre réel en base 10 de la forme : $x = (e_n \dots e_0, f_1 \dots f_m)_{10}$

On sait qu'il doit s'écrire sous la forme : $b_{n'}.2^{n'} + \cdots + b_0.2^0 + d_1.2^{-1} + \cdots + d_{m'}.2^{-m'}$

Séparons partie entière et partie décimale : $Ent = e_n \dots e_0$; $Dec = f_1 \dots f_m$

- **<u>Partie entière binaire</u>**

Pour trouver la partie entière du nombre binaire associé, il suffit de transcrire le nombre entier Ent en binaire.

- **<u>Partie décimale binaire</u>**

Concernant sa partie décimale, prenons un exemple pour comprendre :

$$x = (0{,}375)_{10}$$

Pour obtenir le chiffre des dixièmes, on prend la partie entière de $10x$, soit la partie entière de $3{,}75$. On obtient le chiffre 3 et on définit un nouveau nombre correspondant à la partie décimale de $3{,}75$, soit $0{,}75$. On répète alors l'opération jusqu'à ce que la partie décimale soit nulle.

On va en binaire procéder de même mais en multipliant par 2. Ainsi :

Principe	Présentation améliorée
$0{,}375 * 2 = 0{,}750 \rightarrow \begin{cases} Bit\ 0 \\ Nouvelle\ partie\ décimale\ 0{,}75 \end{cases}$ $0{,}750 * 2 = 1{,}5 \rightarrow \begin{cases} Bit\ 1 \\ Nouvelle\ partie\ décimale\ 0{,}5 \end{cases}$ $0{,}50 * 2 = 1 \rightarrow \begin{cases} Bit\ 1 \\ Nouvelle\ partie\ décimale\ 0 \end{cases}$	0,375 | $*2=$ | 0, | 750 0,750 | $*2=$ | 1, | 5 0,50 | $*2=$ | 1, | **0**

C'est terminé :

$$(0{,}375)_{10} = (0{,}011)_2$$

Remarque : on verra dans une remarque dans la suite qu'en passant par une écriture scientifique binaire, ce transcodage est aussi possible. Calculez $0{,}011 = (11)_2.2^{-3}$

A.III.2 Représentation des nombres en machine

Le programme d'IPT limite ce chapitre à la représentation des nombres réels en virgule flottante normalisée sans traiter de cas particuliers. Après avoir vu les limites de la représentation des entiers naturels et une possibilité de représenter des entiers relatifs, nous aborderons donc la représentation associée à la **norme IEEE 754** des nombres à virgule flottante pour représenter les réels en binaire.

A.III.2.a Nombres entiers naturels

Comme nous l'avons vu, un entier naturel codé sur n bits peut avoir une valeur décimale comprise entre 0 et $2^n - 1$.

Les systèmes 32 bits présentent $2^{32} - 1 = 4\,294\,967\,295$ valeurs entières différentes.

Les systèmes 64 bits présentent $2^{64} - 1 = 1{,}8446744007371 * 10^{19}$ valeurs entières différentes.

Comment manipuler des entiers plus grands ? Négatifs ? Ou encore des nombres réels ?

A.III.2.b Nombres entiers relatifs – Codage par excès

Il existe plusieurs manières de coder des entiers relatifs en binaire. Nous n'aborderons ici que le codage par excès car il va nous servir pour le codage des nombres à virgule flottante abordé ensuite.

Le principe consiste à utiliser un entier N entre 0 et $2^n - 1$ et d'associer ses valeurs à des nombres entiers relatifs Z décalés d'un « biais » tel que l'on ait environ la moitié des termes négatifs, et l'autre positifs (il est impossible d'en avoir autant de chaque côté car après avoir enlevé 0, il reste $2^n - 1$ termes à répartir, ce qui est une quantité impaire de chiffres).

Exemple sur 3 bits des deux solutions possibles :

N	0	1	2	3	4	5	6	7
Z	-4	-3	-2	-1	0	1	2	3
Z	-3	-2	-1	0	1	2	3	4

On aura donc environ $\frac{2^n}{2} = 2^{n-1}$ termes négatifs et 2^{n-1} positifs.

Il suffit donc de décaler le nombre entier naturel N en lui soustrayant environ 2^{n-1}. En fait, on soustraira $2^{n-1} - 1$, ce que l'on appelle le « biais » du codage par excès :

$$Z = N - (2^{n-1} - 1)$$

Exemple : Soit un entier naturel N codé sur 8 bits devant coder des entiers relatifs. N permet de coder les entiers naturels de 0 à $2^8 - 1 = 256 - 1 = 255$. En décalant chaque valeur du biais valant $2^{n-1} - 1 = 2^7 - 1 = 127$, on obtient la table d'association suivante :

Entier naturel N	Entier relatif Z associé
0	-127
255	128

Il est ainsi possible de coder sur 8 bits des entiers relatifs associés à des entiers relatifs positifs et négatifs codés sur 7 bits, et d'une manière plus générale sur n bits des entiers relatifs associés à des entiers positifs et négatifs codés sur $n - 1$ bits.

A.III.2.c Nombres à virgule flottante

A.III.2.c.i Principe

Le principe de la norme IEEE 754 qui permet de représenter en binaire des nombres à virgule flottante (on parle de flottants) est basé sur une représentation similaire à notre représentation scientifique.

En effet, en base 10, nous avons appris à représenter les nombres ainsi :

$$\begin{cases} 1200 = 1{,}2.10^3 \\ 0{,}000056 = 5{,}6.10^{-5} \\ -289456{,}5 = -2{,}894565.10^5 \end{cases}$$

Prenons l'exemple de $X = -289456{,}5 = -2{,}894565.10^5$

Pour stocker le moins d'informations possible de ce nombre, on écrit :

$-2{,}894565.10^5$			
$-$	2	894565	5
Signe	Caractéristique	Mantisse	Puissance
	Partie significative		

Remarques :

- Le signe vaut 0 ou 1
- La caractéristique est un chiffre de 1 à 9, elle ne peut pas être nulle car dans ce cas, la notation scientifique n'est pas respectée : $\mathbf{0},022 = \mathbf{0},22.10^{-1} = 2{,}2.10^{-2}$

En binaire, on va utiliser le même principe mais avec des puissances de 2 au lieu de puissance de 10.

A.III.2.c.ii Ecriture « scientifique binaire »

Nous admettrons que l'on peut transformer un nombre binaire sous forme scientifique, par exemple :

$$(1100)_2 = (1{,}1)_2 * 2^3$$

Prenons deux exemples pour montrer que cela fonctionne :

$(1000)_2 = 8$	$(1100)_2 = 12$
$(1000)_2 * 2^0 = 8 * 1 = 8$ $(100{,}0)_2 * 2^1 = 4 * 2 = 8$ $(10{,}00)_2 * 2^2 = 2 * 4 = 8$ $(1{,}000)_2 * 2^3 = 1 * 8 = 8$	$(1100)_2 * 2^0 = 12$ $(110)_2 * 2^1 = 6 * 2 = 12$ $(11)_2 * 2^2 = 3 * 4 = 12$ $(1{,}1)_2 * 2^3 = 1{,}5 * 8 = 12$
$(1000)_2 = 1.2^3$	$(1100)_2 = (1{,}1)_2 * 2^3$

Rappelons que : $(1{,}1)_2 = 1.2^0 + 1.2^{-1} = 1 + \frac{1}{2} = 1{,}5$

Remarque : Un pourra remarquer que la transcription de binaire à virgule à scientifique binaire peut se faire en passant par un entier multiplié par une puissance de 2. Dans l'exemple du paragraphe précédent : $(0{,}011)_2 = (11)_2.2^{-3} = 3.2^{-3} = (0{,}375)_{10}$

A.III.2.c.iii Ecriture binaire du codage à virgule flottante

Reprenons l'exemple précédent :

$-2{,}894565.10^5$			
$+$	2	894565	5
Signe	Caractéristique	Mantisse	Puissance
	Partie significative		

Voyons comment stocker ce nombre sous forme scientifique binaire :

$$(-2{,}894565.10^5)_{10} = (-289456{,}5)_{10} = (-289456 - 0{,}5)_{10}$$
$$= -(100011010101010000)_2 - (0{,}1)_2 = -(100011010101010000{,}1)_2$$

On écrit donc ce nombre binaire sous forme scientifique binaire :

$$-(100011010101010000{,}1)_2 = -(1{,}00011010101010100001)_2.2^{18}$$

Pour stocker ce nombre, on a besoin des informations suivantes :

$-(1{,}00011010101010100001)_2.2^{18}$			
$-$	1	00011010101010100001	18
Signe	Caractéristique	Mantisse	Puissance
	Partie significative		

Comme précédemment, la caractéristique ne peut être nulle. En base 10, elle pouvait valoir un chiffre de 1 à 9. Maintenant, elle ne peut plus être différente de 1. Chouette ! Plus besoin de la stocker, on gagne un bit.

Finalement, il suffit de stocker les informations suivantes :

$-$	$00011010101011000 01$	18
Signe	Mantisse	Puissance

Toutefois, ces informations ne peuvent être stockées sous forme binaire, avec des 0 et des 1.

Les choix suivants sont effectués :

- Le signe est stocké sur un bit : $\begin{cases} s = 0 \Leftrightarrow x > 0 \\ s = 1 \Leftrightarrow x < 0 \end{cases}$
- La mantisse est déjà sous forme binaire
- La puissance est un nombre qui peut être positif ou négatif. Le choix est fait de le coder en binaire par excès, c'est-à-dire en décalant la valeur décimale en une valeur entière par l'opération vue précédemment $Z = N - (2^{n-1} - 1)$ avec n le nombre de bits sur lequel la puissance va être codée.

Pour coder la puissance, nous devons donc maintenant choisir un format de stockage. Nous les aborderons plus tard, pour le moment admettons qu'en 32 bits, on réserve 8 bits pour la puissance, soit un biais de 127.

La puissance 18 se transforme en un entier naturel $18 + 127 = 145$ qui en binaire s'écrit :

$$(145)_{10} = (10010001)_2$$

Les informations binaires à stocker sont donc les suivantes :

1	$00011010101011000 01$	10010001
Signe	Mantisse	Puissance

Le choix est fait de les représenter dans le sens suivant :

1	10010001	$00011010101011000 01$
Signe	Puissance	Mantisse

Dernier détail, en 32 bits, 8 bits sont alloués à la puissance, 1 bit au signe, le reste à la mantisse. Il en reste donc 23. Il faut donc compléter la mantisse de 0 à droite pour avoir 23 bits (a droite, car la mantisse est la partie décimale, donc par exemple $0{,}1 = 0{,}10000$).

Finalement, en 32 bits, on aura une représentation de $-289456{,}5$ par :

Représentation sur 32 bits	1	10010001	00011010101011000010000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)

Ne jamais oublier le bit implicite non décrit dans ce format.

A.III.2.c.iv Formats de la norme

La norme propose différents formats de stockage de flottants, voyons les trois principaux :

	Nombre de bits	Bit signe	Bits exposant	Bits implicite	Bits mantisse	Décalage
Simple précision	32	1	8	1	23	127
Double précision	64	1	11	1	52	1023
Quadruple précision	128	1	15	1	112	16383

Dans les logiciels de programmation, on peut déclarer la manière avec laquelle on veut stocker des nombres, par exemple en écrivant « $float\ a$ » ou « $double\ a$ ».

A.III.2.c.v Exemples de transcodage

- **Réel base 10 – Binaire en virgule flottante**

Nombre entier :

Soit le nombre $x = (2100)_{10}$ que l'on souhaite exprimer en simple précision. Traduisons ce nombre en binaire et mettons le sous forme scientifique binaire :

$$(2100)_{10} = (100000110100)_2 = (1{,}000001101)_2.2^{11} = 1.025390625 * 2^{11}$$

La mantisse vaut donc 000001101 qu'il faut compléter de 0 « à droite » pour avoir 23 bits :

$$00000110100000000000000$$

La puissance 11 est le résultat d'un codage par excès, le nombre associé est donc $11 + 127 = 138 = (10001010)_2$. La puissance vaut donc 10001010 et elle a déjà une taille de 8 bits dans ce cas.

Finalement, on a x codé en binaire simple précision :

	$(2100)_{10}$		
Simple précision 32 bits	0	10001010	00000110100000000000000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)
	01000101000000110100000000000000		

Nombre à virgule :

Soit le nombre $x = (-10{,}125)_{10}$ que l'on souhaite exprimer en simple précision. Procédons comme précédemment : $(-10{,}125)_{10} = (-1010{,}001)_2 = (-1{,}010001)_2.2^3$

$$3 + 127 = 130 = (10000010)_2$$

	$(10{,}125)_{10}$		
Simple précision 32 bits	1	10000010	01000100000000000000000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)
	11000001001000100000000000000000		

- **<u>Binaire en virgule flottante – Réel base 10</u>**

Nombre entier :

Soit le mot binaire 32 bits suivant : $01001101100011101111001111000010$

Simple précision 32 bits	Identification des 3 parties		
	0	10011011	00011101111001111000010
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)

Alors :

- Le signe est positif
- Le nombre n associé à l'exposant vaut $n = 1.2^7 + 0.2^6 + 0.2^5 + 1.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = 155$. Avec le décalage de 127, on obtient l'exposant : $e = 155 - 127 = 28$
- Sans oublier d'ajouter le bit implicite, la partie significative vaut : $(\mathbf{1}, 00011101111001111000010)_2 = 1.1168138980865479$

Finalement, le nombre décimal associé vaut : $x = 1.1168138980865479.2^{28} = 299792448$

Nombre à virgule :

Soit le mot binaire 32 bits suivant : $11000001001001110000000000000000$

Simple précision 32 bits	Identification des 3 parties		
	1	10000010	01001110000000000000000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)

- Le signe est négatif
- Le nombre n associé à l'exposant vaut $n = 130$. Avec le décalage de 127, on obtient l'exposant : $e = 130 - 127 = 3$
- Sans oublier d'ajouter le bit implicite, la partie significative vaut : $(\mathbf{1}, 0100111)_2 = 1{,}3046875$

Finalement, le nombre décimal associé vaut : $x = -1{,}3046875.2^3 = -10.4375$

Remarques suite à ces exemples :

- dans le cas où la mantisse a une taille inférieure à la valeur de la puissance, on peut se ramener à un nombre binaire sans virgule, et la conversion se fait directement d'un binaire à un entier : $(\mathbf{1}, 00011101111001111000010010 1)_2 * 2^{28} = (\mathbf{1}0001110111100111100001001010)_2 = (299792458)_{10}$. On pourra toutefois dans tous les cas utiliser une procédure de conversion d'un binaire à virgule multiplié par une puissance de 2.
- Pour que le résultat en base 10 soit un nombre à virgule, il est nécessaire que la mentisse soit de longueur plus grande que la valeur de l'exposant

A.III.2.c.vi Quelques nombres réservés

Sans rentrer dans tous les détails, certains nombres de la représentation en virgule flottante sont réservés, par exemple en simple précision :

- L'exposant maximum et une mantisse remplie de 0 est réservé au nombre ∞ :

$$\pm\infty \Leftrightarrow \begin{matrix}0\\1\end{matrix} \quad 11111111 \quad 00000000000000000000000$$

- L'exposant maximum et une mantisse non nulle est réservé au nombre NaN dire « Not a Number ». C'est par exemple le résultat d'une division par 0 ou une racine d'un nombre négatif…

$$NaN \Leftrightarrow \begin{matrix}0\\1\end{matrix} \quad 11111111 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

A.III.2.c.vii Notion de représentation normalisée et dénormalisée

Il existe des représentation dites « normalisées » respectant ce que nous venons de voir, et des nombres en notation « dénormalisée ». Lorsqu'un nombre a un exposant binaire nul (plus petite puissance), au lieu d'écrire

$$Valeur = signe * 1,mantisse * 2^{-décallage}$$

On écrit

$$Valeur = signe * mantisse * 2^{-décallage+1}$$

En 32 bits, cela donne : $Valeur = signe * mantisse * 2^{-126}$.

Cela permet de représenter des nombres encore plus petits et d'assurer une certaine continuité de valeurs entre nombre normalisés et dénormalisés, mais……..

Vous n'avez pas besoin de retenir cela, mais cela vous permettra de comprendre les deux valeurs minimales proposées dans le tableau un peu plus bas.

Limites de la représentation en virgule flottante

• Justesse

La justesse (correspondance entre nombre théorique et nombre réel) d'un nombre est associée au nombre de chiffres significatifs qui le décrivent, c'est-à-dire au nombre de bits codant la mantisse auquel on ajoute le bit implicite. La justesse est parfaite si l'on dispose d'un nombre infini de bits ou si, par chance, le nombre représenté possède un nombre de bits de mantisse égale au nombre de bits stockables dans le format choisi.

La quantité de nombre différents représentables est :

32 bits	4 294 967 296
64 bits	18 446 744 073 710 000 000

On représente donc autant de nombre avec la notation en virgule flottante, mais ils ne sont plus limités aux entiers et peuvent dépasser le nombre maximum de $2^n - 1$!!!

• Nombre minimum et maximum

Pour une simple précision :

- La plus grande puissance codée par excès sur 8 bits vaut 255. Elle vaut donc en réalité $255 - (2^7 - 1) = 128$. Toutefois, cette puissance est réservée au nombre NaN. C'est donc au maximum 127. La plus grande partie significative vaut :
$$(1{,}11111111111111111111111)_2 = (1{,}99999988079071)_{10}$$
Le nombre le plus grand vaut donc :
$$1{,}9999998807907104.\, 2^{127} = 3{,}4028234663852886.\, 10^{38}$$
- Le nombre le plus petit en notation normalisée a la puissance la plus faible : -126, et la partie significative :
$$(1{,}00000000000000000000000)_2 = (1)_{10}$$
En notation normalisée, le nombre le plus petit vaut donc :
$$2^{-126} = 1{,}17549435082229.\, 10^{-38}$$
- En revanche, en notation dénormalisée, le plus petit nombre est :
$$(0{,}00000000000000000000001)_2 * 2^{-127+1} = (1)_2 * 2^{-23} * 2^{-126} = 2^{-149}$$

Valeurs normalisées	Valeur min normalisée	Valeur min dénormalisée	Valeur max
Simple précision	$1{,}2.\, 10^{-38}$	$1{,}4.\, 10^{-45}$	$3{,}4.\, 10^{38}$
Double précision	$2{,}2.\, 10^{-308}$	$4{,}9.\, 10^{-324}$	$1{,}8.\, 10^{308}$

Sur un système d'exploitation 32 bits, ouvrez Excel par exemple, et tapez dans une case le nombre : $3{,}5.\, 10^{38}$, vous verrez alors : 

. Par contre, entrez $3{,}4.\, 10^{38}$, vous verrez 

De même sur un système d'exploitation 64 bits, essayez $1{,}8.\, 10^{308}$ et $1{,}7.\, 10^{308}$.

Vous savez maintenant quelle valeur maximale vous êtes capable de traiter avec votre outil informatique.

- **<u>Précision</u>**

Définissons la précision comme l'écart entre deux valeurs représentables en virgule flottante.

Prenons l'exemple des deux valeurs les plus grandes représentables (pour les valeurs minimales, intervient la notion de dénormalisation que je préfère éviter).

Ainsi, les deux plus grands nombres sont :

$$A = (1{,}11111111111111111111111)_2 * 2^{127} = 3{,}4028234663852886.10^{38}$$

$$B = (1{,}11111111111111111111110)_2 * 2^{127} = 3{,}40282326356119.10^{38}$$

L'écart entre A et B vaut :

$$A - B = -2{,}02824096036517.10^{31}$$

Il n'existe pas de nombres en simple précision entre A et B, et ils sont… relativement éloignés.

A.III.3 Conséquences de la représentation des nombres en virgule flottante

A.III.3.a Dépassement de capacité - Overflow

Le dépassement de capacité est atteint lorsqu'un nombre dépasse la valeur limite représentable dans le système choisi.

Un entier naturel binaire codé sur un octet (8 bits) ne peut dépasser 255. Le calcul 255+1 consistant à calculer $11111111 + 00000001$ donne normalement le résultat 100000000 sur 9 bits. La machine retiendra les 8 derniers bits, soit 00000000 et donnera un résultat nul.

Comme nous l'avons vu avec Excel pour un système 64 bits, le dépassement de capacité consistant à écrire le nombre $1{,}8.10^{308}$ renvoie une erreur car ici aussi, il y a dépassement de capacité.

Ariane 5 s'est crashée à cause d'un dépassement de capacité…

A.III.3.b Erreurs d'arrondis

A.III.3.b.i Exemple

Souvenez-vous, au début de ce paragraphe, nous avons abordé l'exemple suivant :

```
>>> 0.2+0.1
0.30000000000000004
```

Sans rentrer dans les détails, les mantisses des nombres utilisés sont de dimension 52 bits (double précision). Les additions sur les nombre en virgule flottante consistent à faire une addition de mantisses après les avoir mises au même exposant. La mise au même exposant consiste à effectuer des décalages de mantisse, ce qui conduit à perdre les bits décalés vers la droite au-delà du 52°, ce qui est à l'origine d'erreurs d'arrondis.

A.III.3.b.ii Conséquence importante – Test « == »

Exécutez ce programme et vous aurez tout compris :

```
a = 0.2
b = 0.1
c = a + b

print('c = ',a,' + ',b,' = ', c)
if c==0.3:
    print('OUI: c==0.3')
else:
    print('NON: c==0.3')

if abs(c-0.3)<=10**(-10):
    print('OUI: (c-0.3)<=10**(-
n)')
else:
    print('NON: (c-0.3)<=10**(-
n)')
```

```
>>> (executing file "<tmp 1>")
c =  0.2  +  0.1  =  0.30000000000000004
NON: c==0.3
OUI: (c-0.3)<=10**(-n)
```

Lorsque l'on travaille avec des flottants, il ne faut pas effectuer de tests d'égalité mais comparer des différences à un nombre très petit. Un exemple à votre portée est de réaliser un programme qui calcul les solutions d'une équation du second degré avec discriminent nul. Le test « $Disc == 0$ » ne fonctionnera jamais à tous les coups…

Exemple souvent rencontré : On veut tester si un nombre est entier :

BIEN	PAS BIEN

`a = 0.1`
`b = 0.2`
`c = 0.3`
 	
`Test = (a+b)/c`
 	
`if abs(Test - int(Test)) < 1e-10:`
`  print("Cool")`
`else:`
`  print("Pas cool")`
 	
`>>> (executing file "<tmp 1>")`
`Cool`
	`a = 0.1`
	`b = 0.2`
	`c = 0.3`
	
	`Test = (a+b)/c`
	
	`if Test == int(Test):`
	`  print("Cool")`
	`else:`
	`  print("Pas cool")`
	
	`>>> (executing file "<tmp 1>")`
	`Pas cool`

L'idée d'utiliser le test « `type(Test) == int` » n'est pas mauvaise mais ne fonctionne pas à tous les coups. En effet, pour que la variable Test soit un entier, il faut qu'elle soit issue d'entiers. Or, ici, si le calcul fonctionnait, le résultat serait 1.0, c'est-à-dire une valeur entière stockée sous forme de flottant. Il faut donc faire attention à ce que l'on manipule.