**Nombres Réels  
NORME IEEE 754 sur 32 et 64 bits**

**Virgule flottante : schéma simple**

En informatique, les nombres réels n’existent pas dans le sens mathématique du terme. En 2024, on connaît 202 trillion de décimales du nombre pi. En mathématique, le nombre de décimales est infini.

La notation décimale ressemble à ceci :

123 400 000 s’écrit 1,234 × 108 en notation scientifique.

92 384 s’écrit 9,2384 × 104

0,007 s’écrit 7 × 10−3

0,000123 s’écrit 1,23 × 10−4

± *a* × 10*n*

*a*  est la mantisse, un nombre réel compris compris entre 1 (inclus) et 1.0 (exclu).

n la valeur de l’exposant

Les nombres à virgule flottante s’inspirent de la notation scientifique.

Pour comprendre la notion de nombre à virgule flottante, il suffit d’imagier qu’on garde maximum dix chiffres pour la mantisse et 1 chiffre pour l’exposant.

▯ ± ▯ ▯ ▯ ▯ ▯ ▯ ▯ ▯ ▯ ▯

e S (chiffres significatifs)

Schématiquement, selon ce modèle, le chiffre dans la case de gauche représente la position de la virgule et à droite on a 10 chiffres significatifs pour le nombre.

Quelque soit la valeur du nombre, selon ce schéma, on a toujours 10 chiffres de précision au maximum. Un nombre à un seul chiffre a une précision au millionième, un nombre à deux chiffres a une précision à l’unité près.

En pratique, il faut être attentif aux valeurs suivantes  :

* +infini (+n / 0 )
* -infini (-n / 0 )
* Nan (Not à Number) (saisie incorrecte)

Visualiser la norme IEEE 754  
<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

On encode le signe, la mantisse et l’exposant :

(-1) ^ *signe* \* 1,*mantisse* \* 2 ^ (*exposant biaisé*)

Exposant biaisé = Exposant - Décalage

Signe  : 1 bit 1 bit

Exposant biaisé  : 8 bits 11 bits

Mantisse   : 23 bits 52 bits

------- -------

32 bits 64 bits

------- -------

Décalage : 127 1023

Exposant : -126..+127 (\*) -1022..+1023 (\*)

(\*) ATTENTION les valeurs extrêmes pour l’exposant  
 (tous les bits à 0 ou bien tous les bits à 1)  
 ont une signification spéciale (ci-dessous).

Représentation du nombre 1 selon la norme IEEE754

**32 bits**

0 01111111 00000000000000000000000  
 - -------- -----------------------  
 signe exposant mantisse (23 bits)

biaisé

**64 bits**

0 01111111111 0000000000000000000000000000000000000000000000000000

- ----------- ----------------------------------------------------  
 s exposant mantisse (52 bits)

biaisé

**Exemple : conversion de décimal vers IEEE 754 32 bits**

Nombre = -118,375

1. Le bit de signe et ensuite on continue avec la valeur absolue  
    nombre négatif bit = 1
2. Convertir la partie entière 132 en binaire

132 = 128 + 4 = 2^7 + 2^2 = 10000100

1. Convertir la partie décimale 0,375 en binaire  
    0,375 = 0,25 + 0,125 = 2^(-2) + 2^(-3) = 0,011
2. Ecrire le nombre en binaire

132,375 (décimal)

= 10000100,011 (binaire)

1. Utiliser la notation scientifique, en base binaire

132,375  
 = 1,0000100011 \* 2^7

1. Calculer la mantisse en enlevant 1 et en ajoutant des 0 derrière pour avoir le bon nombre de bits

00001000110000000000000

1. Calculer l’exposant d’après le décalage  
    exposant - décalage = 7  
    exposant - 127 = 7  
    exposant = 127 + 7

exposant = 134

1. Encoder l’exposant en binaire

134 = 128 + 4 + 2

= 10000110

1. Vérifier le nombre de bits utilisés par l’exposant

10000101

1. Assembler le signe, l’exposant et la mantisse

1 10000110 00001000110000000000000  
 - -------- -----------------------  
 signe exposant mantisse

**Exemple : conversion de IEEE 754 32 bits vers décimal**

1 10000110 00001000110000000000000  
 - -------- -----------------------  
 signe exposant mantisse

1. Encoder le signe  
    1 => nombre négatif

1. Convertir l’exposant  
    10000110 = 128 + 4 + 2  
    = 134
2. Appliquer le décalage à l’exposant

exposant = 134 – 127

= 6

1. Ajouter 1 devant la mantisse et enlever les 0 derrière  
    1,0000100011
2. Ecrire le nombre

1,0000100011 \* 2^6

1. Décaler la virgule d’après l’exposant  
    1000010,011
2. Convertir la partie entière  
    1000010 = 128 + 4  
    = 133
3. Convertir la partie décimale

0,011 = 0,25 + 0,125

= 0,375

1. Ecrire le nombre en décimal

133,375

1. Combiner le signe, la partie entière et la partie décimale

-133,375

**Valeurs spéciales**

|  |  |  |  |
| --- | --- | --- | --- |
| **Signe** | **Exposant biaisé** | **Mantisse** | **Signification** |
| 0  1 | 0 | 0 | + 0  - 0 |
| 0  1 | 0 | > 0 | Nombre dénormalisé |
| 0  1 | Tous les bits = 1 | 0 | + infini  - infini |
| 0  1 | Tous les bits = 1 | > 0 | NaN = Not A Number |

Les nombres dénormalisés :

1. Peuvent permettre d’encoder davantage de valeurs proches de 0.
2. Peuvent ralentir les traitements et être source de problèmes.
3. Le calcul se fait ainsi pour les nombres dénormalisés :

(-1) ^ *signe* \* 0,*mantisse* \* 2 ^ (décalage - 1)

**Calculs spéciaux**

|  |  |  |
| --- | --- | --- |
| **Opération** | **Résultat** | **Remarque (Python)** |
| n ÷ ±∞ | 0 |  |
| ±∞ × ±∞ | ±∞ |  |
| ±*nonZéro* ÷ ±0 | ±∞ | ZeroDivisionError |
| ±*fini* × ±∞ | ±∞ |  |
| ∞ + ∞ ∞ − −∞ | +∞ |  |
| −∞ − ∞ −∞ + −∞ | −∞ |  |
| ∞ − ∞ −∞ + ∞ | NaN |  |
| ±0 ÷ ±0 | NaN | ZeroDivisionError |
| ±∞ ÷ ±∞ | NaN |  |
| ±∞ × 0 | NaN |  |

|  |  |
| --- | --- |
| **Opération** | **Python** |
| ∞ == ∞ | Vrai |
| -∞ == -∞ | Vrai |
| +∞ > -∞ | Vrai |
| +∞ > +∞ | Faux |
| +∞ > -∞ | Vrai |
| NaN == Nan | Faux |
| NaN != Nan | Faux |
| NaN == 1 | Faux |
| NaN != 1 | Faux |

**Traitement des NaN**

Il faut utiliser la fonction math.isnan().

from math import isnan

x = float(input("Entrez un nombre : ")) # essayer Nan inf -inf

y = float(input("Entrez un deuxième : ")) # essayer Nan inf -inf

if isnan(x) and isnan(y):

print("Deux nombres NaN.")

elif not isnan(x) and not isnan(y):

print("Aucun NaN. Est-ce que x == y ? ",x==y)

else:

print("Un nombre NaN, un autre pas.")

**Calculs approximatifs en virgule flottante**

Les calculs en virgule flottante ne sont pas très précis, surtout lors de calculs de nombre de grandeurs différentes   
(par exemple 1000000000 + 0,00001).

Exemple 1 :

print(1.1 + 2.2) # 3.3000000000000003

Exemple 2 :

n = 0  
x = 0.01

print("DEBUT DU WHILE")  
while(x != 1 and n < 110): # tester ensute while(x <= 1):

x += 0.01

n += 1

print(" n = " + str(n) + " x = " + str(x) + " " + x.hex())

print("FIN DU WHILE.")

print("n = " + str(n) + " x = " + str(x) + " " + x.hex())

print("FIN DU PROGRAMME.")

Solution(s) :

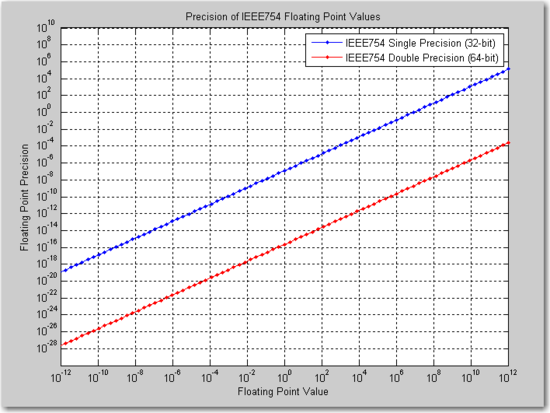
*S’assurer dès le départ que le niveau de précision est suffisant pour traiter le problème donné.*

*Utiliser un module spécialisé, par exemple decimal en Python*

*Éviter de faire des calculs entre deux nombres d’échelle différente*

*Effectuer les calculs avec un seuil de tolérance*

**Niveau de précision**



Source : <https://en.wikipedia.org/wiki/IEEE_floating_point>

Un nombre à 6 chiffres a un niveau de précision de 1 chiffre après la virgule en 32 bits et de 10 chiffres après la virgule en 64 bits.

Exemple de test de précision en Python :

a = 9999999999999.99999999999 # tester plusieurs valeurs

precision = -8 # on teste a + 10\*\*(precision)

b = 10\*\*(precision)

c = (a + b)

while c == a: # tant que ce n’est pas précis

precision += 1 # on diminue la precision

b \*= 10s

print("b = {:.8f}".format(b))

c = a + b

print("a = {}\nb = {:.8f}\nc = {}".format(a,b,c))

print("précision à 10^{}.".format(precision))