Binaire – Les nombres réels

I. Rappels sur les conversions d'entiers

Convertir en binaire (codé sur 1 octet) les nombres suivants écrits en base 10

$12 = 0b0000\ 1100$ car $12 = 8 + 4$	$150 = 0b1001\ 0110$ car $128 + 16 + 4 + 2$	$-15 = 0b1111\ 0001$ car $15 = 0b0000\ 1111$ on inverse : $0b1111\ 0000$ puis $+1$: $0b1111\ 0001$

Convertir en base 10 les nombres suivants, écrits en binaire ou en hexadécimal

$0b1100\ 0100 = 128 + 64 + 4 = 196$

$0x2A = 2 \times 16 + 10 \times 1 = 42$

Les deux nombres suivants sont des entiers négatifs

$0b1010\ 0011 = -93$
on enlève 1 : $0b1010\ 0010$
on inverse : $0b0101\ 1101$
on convertit : 93 donc c'est -93

$0xF1 = 0b1111\ 0001 = -15$
résolu à la question précédente

II. Une première manière de convertir des nombres à virgule

1) En base 10.

En base 10, les nombres décimaux sont notés dd,ddd (où chaque d est un chiffre entre 0 et 9).
Avec le nombre 12,345, il y a :

- une partie entière : 12 : 1 dizaine et 2 unités
- et une partie fractionnaire : 0,345 où :
  - le 3 est le chiffre des dixièmes : 3×10^{-1}
  - le 4, celui des centièmes : 4×10^{-2}

Ainsi : $12{,}345 = 12 + 0{,}345 = \underbrace{(1 \times 10^1 + 2 \times 10^0)}_{partie\ entière} + \underbrace{(3 \times 10^{-1} + 4 \times 10^{-2} + 5 \times 10^{-3})}_{partie\ fractionnaire}$.

2) En base 2

a) Une première représentation, dite "en virgule fixe"

Sur le même principe, un nombre se code en binaire sous la forme bb,bbb (où chaque b est égal à 0 ou 1).

Ainsi : $\underbrace{0b10{,}101}_{du\ binaire} = 0b10 + 0b0{,}101 = \underbrace{(1 \times 2^1 + 0 \times 2^0)}_{partie\ entière} + \underbrace{(1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})}_{partie\ fractionnaire}$

Donc $0b10{,}101 = 2 + 0{,}5 + 0{,}125 = 2{,}625$

Cette écriture est appelée "en virgule fixe" car la virgule est toujours placée juste après le chiffre des unités.

A vous de jouer :

- Convertir $0b0{,}1$ en décimal : $2^{-1} = 0.5$
- Convertir $0b110{,}01$ en décimal : $2^2 + 2^1 + 2^{-2} = 4 + 2 + 0.25 = 6.25$
- Convertir 4,75 en binaire : $4 + 0.5 + 0.25 = 2^2 + 2^{-1} + 2^{-2} = 0b100{,}11$

b) Deux méthodes pour convertir en binaire

Objectif : convertir $N = 0{,}6875$ en binaire (en virgule fixe bien évidemment !)
Si le nombre est supérieur à 1, on sépare partie entière et partie fractionnaire. On sait convertir la partie entière (c'est un entier !). Reste donc à convertir la partie fractionnaire.

Méthode n°1 :

- le nombre est plus petit que 1, donc il s'écrira $\mathbf{0b0}$, …
- y a-t-il du 2^{-1} ? (est-ce que $N \geq 0{,}5$?) : oui donc le nombre s'écrira $0b0$, **1** …
  et il reste $0{,}6875 - 0{,}5 = 0{,}1875$
- y a-t-il du 2^{-2} ? (est-ce que $0{,}1875 \geq 0{,}25$?) : non donc le nombre s'écrira $0b0{,}1$**0** …
  et il reste toujours 0,1875
- y a-t-il du 2^{-3} ? (est-ce que $0{,}1875 \geq 0{,}125$?) : oui donc le nombre s'écrira $0b0{,}10$**1** …
  et il reste $0{,}1875 - 0{,}125 = 0{,}0625$

➔ y a-t-il du 2^{-4} ? (est-ce que $0{,}0625 \geq 0{,}0625$?) : oui donc le nombre s'écrira $0b0{,}1011$ …
et il ne reste plus rien : donc $\boldsymbol{N = 0b0{,}1011}$

Méthode n°2 : plus pratique (elle ne nécessite pas de connaitre les puissances négatives de 2)
On multiple par 2 et on regarde si le résultat est supérieur à 1.

➔ le nombre est plus petit que 1, donc il s'écrira $\boldsymbol{0b0}$, …
➔ $0{,}6875 \times 2 = 1{,}375 \geq 1$ donc le nombre s'écrira $0b0,\mathbf{1}$ …
et on enlève 1 : il reste 0,375
➔ $0{,}375 \times 2 = 0{,}75 < 1$ donc le nombre s'écrira $0b0{,}1\mathbf{0}$ …
on reste à 0,75
➔ $0{,}75 \times 2 = 1{,}5 \geq 1$ donc le nombre s'écrira $0b0{,}10\mathbf{1}$ …
et on enlève 1 : il reste 0,5
➔ $0{,}5 \times 2 = 1 \geq 1$ donc le nombre s'écrira $0b0{,}101\mathbf{1}$ …
et on enlève 1 : il reste 0 donc on s'arrête donc $\boldsymbol{N = 0b0{,}1011}$

A vous de jouer :

▪ Convertir 6,3125 en binaire $6{,}3125 = 0b110{,}0101$	$6 = 4 + 2 = 0b110$ $0.3125 \times 2 = 0.625 \rightarrow 0b0{,}0$ … $0.625 \times 2 = 1.25 \rightarrow 0b0{,}01$ … $0.25 \times 2 = 0.5 \rightarrow 0b0{,}010$ … $0.5 \times 2 = 1 \rightarrow 0b0{,}0101$
▪ Convertir 0,1 en binaire $0{,}1 = 0b0{,}00011001100$ … 1100 se répète indéfiniment	$0.1 \times 2 = 0.2 \rightarrow 0b0{,}0$ … $0.2 \times 2 = 0.4 \rightarrow 0b0{,}00$ … $0.4 \times 2 = 0.8 \rightarrow 0b0{,}000$ … $0.8 \times 2 = 1.6 \rightarrow 0b0{,}0001$ … $0.6 \times 2 = 1.2 \rightarrow 0b0{,}00011$ … $0.2 \times 2 = 0.4 \rightarrow 0b0{,}000110$ … et on recommence
▪ Convertir $\frac{1}{3}$ en binaire $\frac{1}{3} = 0b0{,}0101010101$ … 10 se répète indéfiniment	$\frac{1}{3} \times 2 = \frac{2}{3} \rightarrow 0b0{,}0$ … $\frac{2}{3} \times 2 = \frac{4}{3} \rightarrow 0b0{,}01$ … et il reste $\frac{1}{3}$ $\frac{1}{3} \times 2 = \frac{2}{3} \rightarrow 0b0{,}010$ … et on recommence !

Quelle différence y a-t-il entre l'écriture binaire de ces nombres ?
Le premier se termine, les deux autres ont une infinité de chiffres après la virgule.

Vocabulaire :

- En base 10, certains nombres ont un nombre fini de chiffres après la virgule, on les appelle **nombres décimaux** (symbole $\mathbb{D}$) et d'autres ont une infinité de chiffres après la virgule comme $\frac{1}{3}$ ou $\sqrt{2}$.
- Il se passe la même chose en base 2 : certains nombres ont un nombre fini de chiffres après la virgule, on les appelle **nombres dyadiques** et d'autres ont une infinité de chiffres après la virgule.

A vous de jouer :

1) **Ecrire** une fonction en Python qui :
   - a comme nom ConversionNbreVirguleEnBinaire
   - a un paramètre un nombre à virgule écrit en base 10 (de type FLOAT)
   - les préconditions de cette fonction :
     - le nombre à convertir est dyadique
     - le nombre à convertir est strictement compris entre 0 et 1
   - les post-conditions de cette fonction :
     - la réponse doit être au format STR et doit commencer par "0b"
     - la virgule est représentée par une virgule

- les lignes suivantes ne doivent pas renvoyer de messages d'erreur :
  - assert(ConversionNbreVirguleEnBinaire(0.5)=="0b0,1")
  - assert(ConversionNbreVirguleEnBinaire(0.75)=="0b0,11")
  - assert(ConversionNbreVirguleEnBinaire(0.125)=="0b0,001")

```
def ConversionNbreVirguleEnBinaire(x):
    rep="0b0,"
    while (x>0):
        x=2*x
        if (x>=1):
            rep=rep+"1"
            x=x-1
        else:
            rep=rep+"0"
    return rep
```

2) **Modifier** la fonction précédente pour que les nombres supérieurs à 1 soient convertis correctement. Ainsi les lignes suivantes ne doivent générer de messages :
   - assert(ConversionNbreVirguleEnBinaire(1.5)=="0b1,1")
   - assert(ConversionNbreVirguleEnBinaire(5.75)=="0b101,11")
   - assert(ConversionNbreVirguleEnBinaire(0.125)=="0b0,001")
   - les 3 assertions précédentes doivent toujours être valables !
   - Facultatif : assert(ConversionNbreVirguleEnBinaire(4)=="0b100"

   On réutilise la fonction ConvertionEntierEnBinaire vue en début d'année.

```
def ConversionNbreVirguleEnBinaire(x):
    entier=int(x)
    x=x-entier
    if (x==0) :
        rep=ConversionEntierEnBinaire(entier)
    else :
        rep=ConversionEntierEnBinaire(entier)+","
    while (x>0) :
        x=2*x
        if (x>=1) :
            rep=rep+"1"
            x=x-1
        else:
            rep=rep+"0"
    return rep
```

```
def ConversionEntierEnBinaire(nombre):
    if (nombre==0):
        rep="0"
    else:
        rep=""
    while (nombre>0) :
        if (nombre%2==0) :
            rep="0"+rep
        else :
            rep="1"+rep
        nombre=nombre//2
    rep="0b"+rep
    return rep
```

3) **Modifier** la fonction précédente pour que, dans le cas où le nombre à convertir n'est pas dyadique (on considèrera qu'un nombre qui possède plus de 10 chiffres après la virgule dans son écriture en binaire n'est pas dyadique), la fonction renvoie "infinité de chiffres" et pas le nombre converti. Par contre, si le nombre est dyadique, la fonction doit renvoyer la conversion en binaire.
   Ainsi les lignes suivantes ne doivent générer de messages :
   - assert(ConversionNbreVirguleEnBinaire(0.25)=="0b0,01")
   - assert(ConversionNbreVirguleEnBinaire(0.1)==" infinité de chiffres ")
   - les 6 assertions précédentes doivent toujours être valables !

```
def ConversionNbreVirguleEnBinaire2(x):
    entier=int(x)
    x=x-entier
    if (x==0) :
        rep=ConversionEntierEnBinaire(entier)
    else :
        rep=ConversionEntierEnBinaire(entier)+","
    compteur=0
    while (x>0) and (compteur<=10) :
        x=2*x
        compteur+=1
        if (x>=1) :
            rep=rep+"1"
            x=x-1
        else:
            rep=rep+"0"
    if (compteur==11) :
        rep="infinité de chiffres"
    return rep
```

c) Multiplier par 2 en binaire, c'est facile !

Pour les entiers :

➔ en base 10, que se passe-t-il quand on multiplie par 10 un nombre entier ?

On rajoute un zéro à droite du nombre.

➔ en base 2, il se passe la même chose quand on multiplie un entier par 2 !

Exemples :

- $4 = 0b100$ et donc $8 = 2 \times 4 = 0b1000$ (on a rajouté un 0 à la fin)
- $9 = 0b1001$ et donc $18 = 0b10010$ (on a rajouté un 0 à la fin)

Pour les nombres à virgule :

➔ En base 10, que se passe-t-il quand on multiplie par 10 un nombre à virgule ?

On décale la virgule d'un cran vers la droite.

➔ En base 2, il se passe la même chose quand on multiplie un nombre à virgule par 2 !

Exemples :

- $0{,}25 = 0b0{,}01$ et donc $0{,}5 = 2 \times 0{,}25 = 0b0{,}1$ (on a décalé la virgule)
- $0{,}6875 = 0b0{,}1011$ donc $1{,}375 = 2 \times 0{,}6875 = 0b1{,}011$ (on a décalé la virgule)

Lorsqu'on divise un nombre binaire par 2, on décalera la virgule d'un cran vers la gauche.

d) Les arrondis en binaire

Lorsque, en base 10, un nombre a une infinité de décimales (comme $\frac{1}{3}$ par exemple), on applique la règle de l'arrondi : "si le premier chiffre oublié est entre 0 et 4, on arrondit par défaut (on ne modifie rien) et sinon, on arrondit par excès (on rajoute 1 au dernier chiffre conservé).

Exemples : $\frac{1}{3} \approx 0{,}333$ (1^{er} chiffre oublié = 3) et $\frac{2}{3} \approx 0{,}667$ (1^{er} chiffre oublié = 6)

Le principe est le même en binaire : "si le premier chiffre oublié est un 0, on arrondira par défaut, sinon, on arrondira par excès en rajoutant 1 au dernier chiffre conservé".

A vous de jouer :

- Arrondir les nombres binaires suivants avec 3 chiffres après la virgule

$0b0{,}10011 \approx 0b0{,}101$	$0b0{,}10110 \approx 0b0{,}110$	$0b0{,}011001 \approx 0b0{,}011$
$0b10{,}00101 \approx 0b10{,}001$	$0b0{,}111101 \approx 0b1{,}000$	$0b1{,}1111 \approx 0b10{,}000$

- Justifier que $0{,}41 \approx 0b0{,}011$

$0{,}41 < 1 \rightarrow 0b0, \ldots$
$0{,}41 \times 2 = 0{,}82 \rightarrow 0b0{,}0 \ldots$
$0{,}82 \times 2 = 1{,}64 \rightarrow 0b0{,}01 \ldots$
$0{,}64 \times 2 = 1{,}28 \rightarrow 0b0{,}011 \ldots$
$0{,}28 \times 2 = 0{,}56 \rightarrow 0b0{,}0110 \ldots$ donc si on garde 3 chiffres après la virgule : $0{,}41 \approx 0b0{,}011$

- Justifier que $0{,}3 \approx 0b0{,}0100110$

$0{,}3 < 1 \rightarrow 0b0, \ldots$
$0{,}3 \times 2 = 0.6 \rightarrow 0b0{,}0 \ldots$
$0{,}6 \times 2 = 1.2 \rightarrow 0b0{,}01 \ldots$
$0{,}2 \times 2 = 0.4 \rightarrow 0b0{,}010 \ldots$
$0{,}4 \times 2 = 0.8 \rightarrow 0b0{,}0100 \ldots$
$0{,}8 \times 2 = 1.6 \rightarrow 0b0{,}01001 \ldots$
et on recommence : $0{,}3 = 0b0{,}010011001\ldots$ (1001 se répète indéfiniment)
si on garde 7 chiffres après la virgule, le 1er chiffre oublié est 0, donc $0{,}3 \approx 0b0{,}0100110$

A vous de jouer :

➔ Donner, la valeur des booléens ci-dessous, **SANS UTILISER D'ORDINATEUR** :

Booléen	True / False ?	Booléen	True / False ?
$(0.1 + 0.1 < 0.2)$	False	$(0.1 + 0.2 < 0.3)$	False
$(0.1 + 0.1 == 0.2)$	True	$(0.1 + 0.2 == 0.3)$	True
$(0.1 + 0.1 > 0.2)$	False	$(0.1 + 0.2 > 0.3)$	False

➔ Dans la console de Python, vérifier les résultats précédents. Les réponses données par Python sont-elles cohérentes ?
- Pour $0.1 + 0.1$: OUI, on a bien $0.1 + 0.1 == 0.2$ qui est True
- Pour $0.1 + 0.2$: NON ! $0.1 + 0.2 == 0.3$ est False mais $0.1 + 0.2 > 0.3$ est True

➔ Avec Python :
- combien vaut $0.1 + 0.1$? 0.2
- combien vaut $0.1 + 0.2$? 0.30000000000000004
- d'après vous, quelle pourrait en être l'explication ?
  Il s'agit certainement d'un problème d'arrondi puisque 0.1 n'est pas dyadique.

➔ **Conversion de 0.1 en binaire** (déjà fait page 2). Puisque ce nombre n'est pas dyadique, donner sa valeur arrondie 7 chiffres après la virgule (donc à 2^{-7} près)
$0{,}1 = 0b0{,}00011001100 \ldots$
donc $0{,}1 \approx 0b0{,}0001101$ (arrondi par excès)

➔ **Conversion de 0.2 en binaire**. Si le nombre est dyadique, donner sa valeur exacte, sinon arrondir 7 chiffres après la virgule (donc à 2^{-7} près)
explications : $0.2 = 0.1 \times 2$ donc on décale la virgule d'un cran vers la droite : $0{,}1 = 0b0{,}00011001100 \ldots$ donc $0{,}2 = 0b0{,}0011001100 \ldots$
Le 8ème chiffre après la virgule est un 1, donc on arrondit par excès.
donc $0{,}2 \approx 0b0{,}0011010$

➔ **Conversion de 0.3 en binaire** (déjà fait page 5). Puisque ce nombre n'est pas dyadique, donner sa valeur arrondie 7 chiffres après la virgule (donc à 2^{-7} près)

$0{,}3 \approx 0b0{,}0100110$

➔ Combien vaut, en binaire, 0,1 + 0,1 ? Poser l'addition ci-contre

réponse : $0b0{,}0001101$

Est-ce le même résultat que 0,2 ? OUI

➔ Combien vaut, en binaire, 0,1 + 0,2 ? Poser l'addition ci-contre

réponse : $0b0{,}0100111$

Est-ce le même résultat que 0,3 ? NON, le 7ème chiffre est différent

```
  0b0,0001101
+ 0b0,0001101
  0b0,0011010
```

```
  0b0,0001101
+ 0b0,0011010
  0b0,0100111
```

Il se passe la même chose en base 10. Les additions entre arrondis sont parfois fausses :
Avec $\frac{1}{3} \approx 0{,}333$ on obtient $\frac{1}{3}+\frac{1}{3} \approx 0{,}333 + 0{,}333$ donc $\frac{1}{3}+\frac{1}{3} \approx 0{,}666$ alors que $\frac{2}{3} \approx 0{,}667$

A vous de jouer :

- Exécuter ce programme **à la main**.

Compléter le tableau ci-dessous avec les valeurs des variables.

i		1	2	3	4	...	10
a	0	0.1	0.2	0.3	0.4	...	1.0
b		0.1	0.2	0.3	0.4	...	1.0

```
a=0
for i in range(1,11):
    a=a+0.1
    b=0.1*i
    print(i,a,b)
```

Les valeurs de a et de b sont-elles égales à chaque tour de boucle ? OUI

- Taper ce programme puis vérifier les résultats. Obtient-on ce qui était prévu ? NON

Quand $i = 2$: on obtient 0.2 et 0.2
Quand $i = 3$: on obtient 0.30000000000000004 et 0.30000000000000004
Quand $i = 6$: on obtient 0.6 et 0.6000000000000001
Quand $i = 8$: on obtient 0.7999999999999999 et 0.8
Quand $i = 10$: on obtient 0.9999999999999999 et 1.0

- Bilan : Les opérations avec des nombres à virgule sont difficiles à prévoir !

A vous de jouer :

Sans utiliser d'ordinateur (mais en se servant de ce qui précède), répondre à ces questions.

```
def fct1(n):
    a=0.0
    for i in range(n):
        a=a+0.1
    return a
```

Si on tape fct1(2) dans la console, on obtient :

0.1	0.2	0,2	Autre valeur

Si on tape fct1(10) dans la console, on obtient :

1	1.0	Une erreur	Autre valeur

```
def fct2(n):
    a=1.0
    for i in range(n):
        a=a-0.1
    return a
```

Si on tape fct2(10) dans la console, on obtient :

0	0.0	−0.1	Autre valeur

```
def fct3():
    x=0.0
    while (x<1):
        x=x+0.1
    return x
```

Si on tape fct3( ) dans la console, on obtient :

1.0	Rien car c'est une boucle infinie	Un nombre proche de 1, inférieur à 1	Un nombre proche de 1, supérieur à 1

```
def fct4():
    x=1.0
    while (x!=0):
        x=x-0.1
    return x
```

Si on tape fct4( ) dans la console, on obtient :

0.0	Rien car c'est une boucle infinie	Un message d'erreur	Un nombre proche de 0

e) Inconvénient de cette écriture en virgule fixe

Lorsque l'on considère les nombres 0,000 000 000 123 4 ou 123 000 000 000, on utilise une grande quantité de chiffres. Si on doit convertir ces nombres en binaire, le nombre de zéros sera encore plus important.
Il existe une autre manière d'écrire ces nombres qui utilise moins de chiffres : l'écriture scientifique. Ainsi, $0{,}0000000001234 = 1{,}234 \times 10^{-10}$ et $123\,000\,000\,000 = 1{,}23 \times 10^{11}$.
Cette écriture est aussi appelée "écriture en virgule flottante" car la virgule n'est pas toujours après le chiffre des unités. C'est cette écriture qui est utilisée en informatique pour stocker des nombres à virgule (d'où le type FLOAT pour les nombres en virgule en Python).
La norme IEEE 754 (mise au point par l'Institute of Electrical and Electronics Engineers) définit les formats de représentation des nombres à virgule flottante en binaire.

III. Nombres écrits en virgule flottante (IEEE 754)

La notation scientifique en base 10 est de la forme : $\boldsymbol{signe\ \ mantisse \times 10^{exposant}}$.

où $signe \in \{+;-\}$ $\quad 1 \leq mantisse < 10$ $\quad$ et $\quad exposant \in \mathbb{Z}$

Dans $-1{,}23 \times 10^2$: le signe est $-$, l'exposant est 2 et la mantisse est 1,23
Dans $1{,}234 \times 10^{-3}$: le signe est $+$, l'exposant est -3 et la mantisse est 1,234

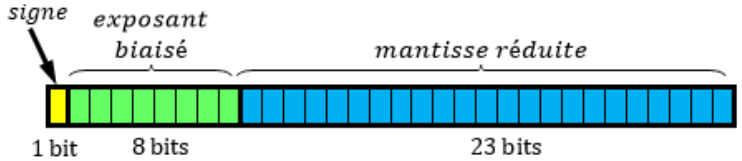
A vous de jouer : en binaire, cette écriture "scientifique" est : $signe\ \ mantisse \times 2^{exposant}$
Cette écriture n'est pas la réelle notation puisqu'elle mélange du binaire (la mantisse) et du décimal (exposant). On verra un peu plus tard, comment l'écrire intégralement en binaire !

On considère le nombre (écrit en binaire) $1{,}011 \times 2^2$ Donner son écriture binaire en virgule fixe.	On décale la virgule de 2 crans vers la droite : 101,1
On considère le nombre (écrit en binaire) : $N = 1010{,}011$ Déterminer la mantisse et l'exposant (en base 10).	Mantisse : 1,010011 Exposant : 3 (on a décalé la virgule de 3 crans)
On considère le nombre (écrit en binaire) : $N = 0{,}0101$ Déterminer la mantisse et l'exposant (en base 10)	Mantisse : 1,01 Exposant : -2 (on a décalé la virgule de 2 crans)
Un nombre négatif, écrit en binaire, a une mantisse de 1,11 et un exposant de 4. Donner la valeur, en base 10, de ce nombre.	Explications : c'est $-1{,}11 \times 2^4$ donc -11100 donc $-(16+8+4)$ Réponse : -28

Remarque : la méthode pour coder les nombres négatifs (en FLOAT) n'est pas la même que pour coder les entiers (on utilisait le complément à 2).

Le format IEEE–754–1985 Réels Normalisés permet de stocker les nombres sur 32 bits (on parle de simple précision) ou sur 64 bits (on parle alors de double précision).
Sur 32 bits :

- le signe est codé sur 1 bit (0 pour positif, 1 pour négatif)
- l'exposant est codé sur 8 bits. Il a été décidé de décaler l'exposant de 127 : on parle alors d'exposant biaisé. Quand les exposants sont compris entre -127 et 128, l'exposant biaisé est entre 0 et 255 et c'est lui qu'on code en binaire !
- la mantisse est codée sur le reste (donc sur 23 bits). Comme la mantisse commence obligatoirement par un 1, il a été décidé de ne pas le coder : on gagne ainsi 1 bit !



$$Valeur = signe \times mantisse \times 2^{exposant}$$

$$exposant\ biaisé = exposant + 127$$

Exemple détaillé pour bien comprendre comment ce codage fonctionne :
Convertir 12,25 en binaire (et en virgule flottante selon la norme IEEE – 754)

- $12{,}25 = 12 + 0{,}25$
- $12 = 0b1100$ (car $12 = 8 + 4$) et $0{,}25 = 0b0{,}01$ (car $0{,}25 = 0 \times \frac{1}{2} + 1 \times \frac{1}{4}$)
- Ainsi : $12{,}25 = 0b1100{,}01$ (écriture binaire en virgule fixe)
- Pour que la mantisse commence par 1, … il faut décaler la virgule de 3 crans vers la gauche donc l'exposant vaut 3 et la mantisse vaut donc 1,10001
- Passons maintenant au codage :
  - Le signe est +, donc le 1er bit sera égal à 0 (1 si le nombre avait été négatif)
  - L'exposant vaut 3, donc l'exposant biaisé vaut $3 + 127 = 130$ que l'on convertit en binaire sur 8 bits : $130 = 128 + 2 = 0b10000010$
  - La mantisse vaut 1,10001 mais, vu que dans tous les cas elle commence par 1, il a été décidé de ne pas coder ce 1 (on gagne ainsi 1 bit pour coder) : $0b10001$ et on complète à droite par des 0 pour avoir 23 bits
- Bilan : 12,25 se code sur 32 bits en : $0b$ 0 10000010 10001000000000000000000
- Si on l'écrit en hexadécimal, on regroupe les bits par paquets de 4 : $0x41440000$

A vous de jouer :

➔ Ecrire 0,25 en virgule flottante sur 32 bits
Réponse : $0b$ 0 01111101 00000000000000000000000

➔ Vérifier que, en virgule flottante, le nombre décimal 0,1 se code $0x3DCCCCCD$
Réponse : 0.1 se code $0b$ 0 01111011 10011001100110011001101 donc $0x3DCCCCCD$

➔ Quel nombre décimal est codé en 10111101110000000000000000000000 ?
Réponse : -0.09375

➔ Le nombre $0b01000010110101100000000000000000$ codé selon la norme IEEE–754 est-il entier ?
Réponse : oui, c'est 107. En réalité, c'est un float, donc c'est 107.0

Pour aller plus loin

➢ Plus d'informations sur la norme IEEE–754 : https://fr.wikipedia.org/wiki/IEEE_754

➢ Certaines valeurs de l'exposant biaisé (celui qui est codé en binaire) sont interdites pour coder les nombres en virgule flottante.

Nombre	Exposant biaisé	Mantisse réduite	Valeurs
Zéro	0000 0000	000 0000 0000 0000 0000 0000	$+0$ et -0
Nombres dénormalisés	0000 0000	autre que 0	
Infinis	1111 1111	000 0000 0000 0000 0000 0000	$+\infty$ et $-\infty$
Not a Number	1111 1111	Autre que 0	NaN
Nombres normalisés	Autres valeurs	Toutes valeurs	

- Les nombres "normalisés" (qui respectent la notation scientifique) permettent de coder des nombres entre $1{,}2 \times 10^{-38}$ et $3{,}4 \times 10^{38}$.
- Attention, tous les nombres entre ces valeurs ne sont pas codés ! L'écart entre 2 nombres (petits) est de l'ordre de 10^{-45} mais d'environ 10^{31} pour des grands nombres.
- Les nombres "dénormalisés" permettent de coder des nombres très proches de 0 (entre 10^{-45} et 10^{-38}) : on parle de "dénormalisés" car la notation scientifique ne s'applique pas car on accepte que la mantisse ne commence par 1. C'est de cette manière qu'on peut atteindre des nombres encore plus proches de 0 : on atteint des nombres d'environ 10^{-45}

➢ Si on code sur 64 bits (donc 8 octets), le principe est exactement le même.
Le signe est codé sur 1 bit, l'exposant (auquel on ajoute 1023) est codé sur 11 bits et la mantisse est codée sur 52 bits.
Cela permet de coder des nombres entre 5×10^{-324} et $1{,}8 \times 10^{308}$.