

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

Informatique

10

Représentation des nombres

Cours

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

Représentation des nombres 4

1.I. Code binaire 4

| | |
|--|---|
| 1.I.1 Introduction..... | 4 |
| 1.I.2 Principe de l'écriture d'un entier dans les bases 2 et 10..... | 5 |
| 1.I.3 Transcodage Entier \leftrightarrow Binaire..... | 5 |
| 1.I.3.a Binaire \rightarrow Entier | 6 |
| 1.I.3.b Entier \rightarrow Binaire | 6 |
| 1.I.3.c En Python | 6 |
| 1.I.4 Transcodage Réel base 10 \leftrightarrow Réel base 2 | 8 |
| 1.I.4.a Réel base 2 \rightarrow Réel Base 10 | 9 |
| 1.I.4.b Réel base 10 \rightarrow Réel base 2 | 9 |
| 1.I.4.b.i Partie entière binaire | 9 |
| 1.I.4.b.ii Partie décimale binaire | 9 |

1.II. Représentation des nombres en machine 10

| | |
|---|----|
| 1.II.1 Nombres entiers naturels..... | 10 |
| 1.II.2 Entiers multi-précision | 11 |
| 1.II.2.a Introduction | 11 |
| 1.II.2.b Principe | 11 |
| 1.II.2.c Remarques | 11 |
| 1.II.3 Nombres entiers relatifs..... | 12 |
| 1.II.3.a Complément à 2 | 12 |
| 1.II.3.a.i Principe | 12 |
| 1.II.3.a.ii Exemple..... | 13 |
| 1.II.3.b Codage par excès | 14 |
| 1.II.3.b.i Principe | 14 |
| 1.II.3.b.ii Exemple | 14 |
| 1.II.4 Nombres à virgule flottante | 15 |
| 1.II.4.a Introduction | 15 |
| 1.II.4.b Principe | 15 |
| 1.II.4.c Ecriture « scientifique binaire » | 16 |
| 1.II.4.d Ecriture binaire du codage à virgule flottante | 16 |
| 1.II.4.e Formats de la norme | 18 |
| 1.II.4.f Exemples de transcodage | 18 |
| 1.II.4.f.i Réel base 10 – Binaire en virgule flottante..... | 18 |
| • Nombre entier | 18 |
| • Nombre à virgule | 18 |
| 1.II.4.f.ii Binaire en virgule flottante – Réel base 10..... | 19 |
| • Nombre entier | 19 |
| • Nombre à virgule | 19 |
| • Remarques à la suite de ces exemples | 19 |
| 1.II.4.g Quelques nombres réservés | 20 |
| 1.II.4.g.i Puissance max..... | 20 |
| • Infini..... | 20 |
| • NaN | 20 |
| 1.II.4.g.ii Puissance min | 20 |
| • Exception du zéro | 20 |
| • Notation dénormalisée (pour info)..... | 20 |

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

| | |
|---|-----------|
| 1.II.4.h Limites de la représentation en virgule flottante | 21 |
| 1.II.4.h.i Justesse | 21 |
| 1.II.4.h.ii Nombre minimum et maximum | 21 |
| 1.II.4.h.iii Ecart entre deux nombres successifs et conséquences | 22 |
| • Expression de l'écart – Un exemple | 22 |
| • Expression de l'écart – De manière générale | 22 |
| • Applications | 24 |
| 1.II.4.i Conséquences de la représentation des nombres en virgule flottante | 26 |
| 1.II.4.i.i Dépassement de capacité - Overflow | 26 |
| • Principe | 26 |
| • Exemple : Ariane 5 – Juin 1996 | 27 |
| 1.II.4.i.ii Erreurs d'arrondis | 28 |
| • Principe | 28 |
| • Cumul d'erreur d'arrondi : Missile Patriot – Février 1991 | 28 |
| • Conséquence importante – Test « == » | 29 |
| • Approximation d'une dérivée | 30 |
| 1.III. Les types sous Python | 31 |

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

Représentation des nombres

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
```

1.I. Code binaire

1.I.1 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.

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

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 :

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

1.I.2 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 :

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

1.I.3 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 :

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

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

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.1.3.a 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 : $1111101000_{(2)}$

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

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

$$1111101000_{(2)} = 1 \cdot 10^3 + 0 \cdot 10^2 + 0 \cdot 10^1 + 0 \cdot 10^0$$

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

1.1.3.b Entier → Binaire

Supposons qu'un nombre entier N s'écrive :

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

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

Divisons ce nombre par 2 :

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

On a donc :

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

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

De même, on a :

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

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

1.1.3.c En Python

| Binaire → Base 10 | Base 10 → Binaire |
|---|--|
| <pre>>>> 0b1111101000 1000</pre> | <pre>>>> bin(1000) '0b1111101000'</pre> |

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

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 | |

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

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.I.4 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 :

$$5,375 = 5 \cdot 10^0 + 3 \cdot 10^{-1} + 7 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

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 :

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

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 :

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

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.1.4.a 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 : $(x)_{10} = b_n \cdot 2^n + \dots + b_0 \cdot 2^0 + c_1 \cdot 2^{-1} + \dots + c_m \cdot 2^{-m}$

1.1.4.b 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 : $b_n \cdot 2^n + \dots + b_0 \cdot 2^0 + d_1 \cdot 2^{-1} + \dots + d_m \cdot 2^{-m}$

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

1.1.4.b.i Partie entière binaire

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

1.1.4.b.ii Partie décimale binaire

Concernant sa partie décimale, prenons un exemple pour comprendre : $x = (0,375)_{10}$

$$x = ?_0 \cdot 10^{-1} + ?_1 \cdot 10^{-2} + ?_2 \cdot 10^{-3} \dots ; 0,375 \cdot 10 = 3,75 = ?_0 + ?_1 \cdot 10^{-1} + ?_2 \cdot 10^{-2} \dots$$

$$0,75 \cdot 10 = 7,5 = ?_1 + ?_2 \cdot 10^{-1} \dots ; 0,5 \cdot 10 = 5 + 0 ; (?_0, ?_1, ?_2) = (3, 7, 5)$$

| | | | |
|-------|--------|----|----|
| 0,375 | * 10 = | 3, | 75 |
| 0,750 | * 10 = | 7, | 5 |
| 0,50 | * 10 = | 5, | 0 |

x étant inférieur à 1, chaque reste est inférieur à 1

Pour les obtenir en binaire, on va procéder de la même manière mais en multipliant par 2 :

$$0,375 = ?_0 \cdot 2^{-1} + ?_1 \cdot 2^{-2} + ?_2 \cdot 2^{-3} \dots ; 0,375 \cdot 2 = 0,750 = ?_0 + ?_1 \cdot 2^{-1} + ?_2 \cdot 2^{-2} \dots$$

$$0,750 \cdot 2 = 1,5 = ?_1 + ?_2 \cdot 2^{-1} \dots ; 0,5 \cdot 2 = 1,0 = ?_2 + 0 ; (?_0, ?_1, ?_2) = (0, 1, 1)$$

| Principe | Présentation améliorée | | | | | | | | | | | | |
|--|--|-------|-------|----|-----|-------|-------|----|---|------|-------|----|---|
| $0,375 * 2 = 0,750 \rightarrow \begin{cases} \text{Bit 0} \\ \text{Nouvelle partie décimale } 0,75 \end{cases}$ $0,750 * 2 = 1,5 \rightarrow \begin{cases} \text{Bit 1} \\ \text{Nouvelle partie décimale } 0,5 \end{cases}$ $0,50 * 2 = 1 \rightarrow \begin{cases} \text{Bit 1} \\ \text{Nouvelle partie décimale } 0 \end{cases}$ | <table><tr><td>0,375</td><td>* 2 =</td><td>0,</td><td>750</td></tr><tr><td>0,750</td><td>* 2 =</td><td>1,</td><td>5</td></tr><tr><td>0,50</td><td>* 2 =</td><td>1,</td><td>0</td></tr></table> | 0,375 | * 2 = | 0, | 750 | 0,750 | * 2 = | 1, | 5 | 0,50 | * 2 = | 1, | 0 |
| 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 : Tout comme on pourrait dire que $0,375 = 375 \cdot 10^{-3}$, on verra dans la suite qu'en passant par une écriture scientifique binaire, ce transcodage est aussi possible : $(0,011)_2 = (11)_2 \cdot 2^{-3} = \frac{3}{8} = 0,375$

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II. 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.

1.II.1 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 $2^{64} - 1 = 1,8446744073709551615 * 10^{19}$ valeurs entières différentes.

Les ordinateurs ont la capacité de réaliser les opérations arithmétiques sur ces entiers (arithmétique matérielle) de manière très rapide.

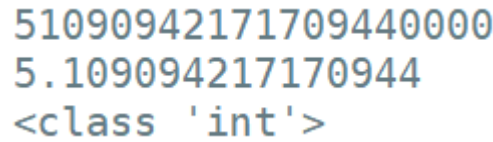
Mais, comment manipuler des entiers plus grands ? Négatifs ? Ou encore des nombres réels ?

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.2 Entiers multi-précision

1.II.2.a Introduction

En utilisant un ordinateur en 64 bits, je ne devrais pas pouvoir obtenir le résultat de 21 comme entier. En effet, $20! \approx 2,4 \cdot 10^{18}$ et $21! \approx 5,1 \cdot 10^{19}$. Et pourtant :

| | |
|---|--|
| <pre>from math import factorial N = factorial(21) print(N) print(N/(10**19)) print(type(N))</pre> |  |
|---|--|

Le besoin en grand nombres se retrouve en cryptographie, par exemple.

1.II.2.b Principe

Dans les grandes lignes, lorsqu'un entier N est trop grand pour être stocké « normalement », on stocke les chiffres qui le composent, par exemple :

$$N_1 = 111 \quad ; \quad L_1 = [1,1,1] \quad ; \quad N_2 = 22 \quad ; \quad L_2 = [2,2]$$

On remarque que l'on peut stocker tout nombre quelle que soit sa valeur, en une liste, la limitation sera la taille mémoire disponible pour stocker ses chiffres. On voit alors simplement arriver des opérations arithmétiques comme la somme ou le produit, en appliquant basiquement les méthodes de calculs apprises dans votre jeunesse :

$$\begin{array}{r}
 1 \ 1 \ 1 \\
 + \quad 2 \ 2 \\
 \hline
 = 1 \ 3 \ 3
 \end{array}
 \quad ; \quad
 \begin{array}{r}
 \ 0 \ 1 \ 1 \ 1 \\
 * \ 0 \ 0 \ 2 \ 2 \\
 \hline
 = 0 \ 2 \ 2 \ 2 \\
 + \ 2 \ 2 \ 2 \ 0 \\
 \hline
 = 2 \ 4 \ 4 \ 2
 \end{array}$$

Il suffit donc de créer des algorithmes qui réalisent des opérations spécifiques avec les retenues en vue d'obtenir le résultat souhaité. Toutefois, il a fallu développer des algorithmes efficaces afin de réaliser les opérations arithmétiques sur ces entiers :

- Les opérations d'addition et soustraction sont en $O(N)$
- Les multiplications sont réalisées par des algorithmes particuliers (ex : algorithme de Schönhage-Strassen en $O(n \log n \log \log n)$)
- Les divisions transforment le problème pour utiliser un petit nombre de multiplications et se reposent donc sur les algorithmes de multiplication

1.II.2.c Remarques

| Avantages | Inconvénients |
|---|---|
| Pas de limites de représentation des entiers | Complexité des opérations arithmétiques difficile à estimer |
| Possibilité de travailler avec de grands nombres pour diverses applications | Taille des entiers limitée à la mémoire machine utilisée N'utilisant plus l'arithmétique matérielle mais des algorithmes de calcul, la réalisation des opérations arithmétiques est considérablement plus lente |

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.3 Nombres entiers relatifs

Nous allons maintenant voir comment représenter des entiers relatifs (notés Z dans ce paragraphe) sur n bits, soit 2^n valeurs distinctes possibles, à l'aide d'entiers naturels (notés N dans ce paragraphe) dont les valeurs vont de 0 à $2^n - 1$.

Le principe du codage consiste à associer à chaque entier naturel N une valeur d'un entier relatif Z selon un calcul précis $N = f(Z)$ ou $N = f^{-1}(Z)$. Nous aborderons le complément à 2 et le codage par excès. Dans les deux cas, il y aura environ autant d'entiers relatifs Z négatifs et positifs, soit environ $\frac{n}{2}$ de chaque côté de 0. Environ ? 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.

1.II.3.a Complément à 2

1.II.3.a.i Principe

Le principe du codage par complément à 2 consiste à associer à tout entier N entre 0 et $2^n - 1$ un entier relatif Z dans l'intervalle $Z \in [-2^{n-1}, 2^{n-1} - 1]$.

Le principe consiste à choisir un bit qui représentera le signe de l'entier relatif codé Z et les $n - 1$ bits restants coderont pour la valeur absolue de Z .

Il est alors possible, par exemple, de choisir le premier bit (poids fort) comme bit de signe $\begin{cases} 0 & \text{si } Z > 0 \\ 1 & \text{si } Z < 0 \end{cases}$ et de prendre le reste pour coder $|Z|$:

$$n = 4 \rightarrow \begin{cases} (0001)_2 \leftrightarrow +(001)_2 = +(1)_{10} \\ (1001)_2 \leftrightarrow -(001)_2 = (-1)_{10} \end{cases}$$

Ce choix conduit, d'une part à deux représentations de 0 ($0^+ = 0000$ et $0^- = 1000$), mais surtout, ne permet pas de garder les propriétés de la somme en binaire. En effet, en restant dans cette représentation, voici ce que donnerait la somme 1-1 :

$$(0001)_2 + (1001)_2 = (1010)_2$$

$$(1)_{10} + (-1)_{10} \leftrightarrow (-2)_{10}$$

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

Un autre choix a donc été réalisé, il consiste à écrire :

$$N = \begin{cases} Z \text{ si } Z \geq 0 \\ 2^n - |Z| = 2^n + Z \text{ si } Z < 0 \end{cases} \Leftrightarrow Z = \begin{cases} N \text{ si } N \leq 2^{n-1} - 1 \\ N - 2^n \text{ si } N > 2^{n-1} - 1 \end{cases}$$

$$N \in [0, 2^n] \quad ; \quad Z \in [-2^{n-1}, 2^{n-1} - 1]$$

On obtient alors le codage en complément à 2 en traduisant l'entier $(N)_{10}$ en binaire naturel $(N)_2$.

Ainsi :

$$Z = (-2)_{10} \rightarrow N = (6)_{10} \leftrightarrow (110)_2$$

$$Z = (-7)_{10} \rightarrow N = (1)_{10} \leftrightarrow (001)_2$$

Sur 3 bits, on a donc la correspondance suivante :

| | | | | | | | | |
|------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $(Z)_{10}$ | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
| $(N)_{10}$ | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 |
| $(N)_2$ | $(100)_2$ | $(101)_2$ | $(110)_2$ | $(111)_2$ | $(000)_2$ | $(001)_2$ | $(010)_2$ | $(011)_2$ |

On remarquera que le premier bit est toujours un bit de signe et qu'il n'y a qu'une représentation de 0.

Le gros avantage de ce choix tient dans le fait que la somme binaire d'entiers fonctionne toujours entre des entiers relatifs, en restant sur n bits (overflow si on dépasse une taille de n , c'est-à-dire ne pas tenir compte du bit qui apparaît à gauche) :

$$(-3)_{10} + (3)_{10} \leftrightarrow (101)_2 + (011)_2 = (1000)_2 \leftrightarrow (000)_2 = 0$$

$$(-3)_{10} + (1)_{10} \leftrightarrow (101)_2 + (001)_2 = (110)_2 \leftrightarrow (-2)_{10}$$

1.II.3.a.ii *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 $2^8 - 1 = 256 - 1 = 255$ et des entiers relatifs de -128 à 127.

On a par exemple :

| | | |
|----------------|--------------|--------------|
| $(N)_2$ | $(N)_{10}$ | $(Z)_{10}$ |
| $(11010111)_2$ | $(215)_{10}$ | $(-41)_{10}$ |

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.3.b Codage par excès

1.II.3.b.i Principe

Le principe du codage par excès consiste à associer à un tout entier N entre 0 et $2^n - 1$ un entier relatif Z décalé d'un « biais » constant. Deux choix sont possibles :

- Biais $B = -\frac{2^n}{2} = -2^{n-1}$, ce qui donne $Z \in [-2^{n-1}, 2^{n-1} - 1]$
- Biais $B = -\left(\frac{2^n}{2} - 1\right) = -(2^{n-1} - 1)$, ce qui donne $Z \in [-2^{n-1} + 1, 2^{n-1}]$

En utilisant 3 bits, soit $2^3 = 8$ entiers, on va donc décaler l'entier N entre 0 et 7 d'un biais de -4 ou -3 , ce qui donne les deux solutions suivantes :

| | | | | | | | | |
|------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $(N)_2$ | $(000)_2$ | $(001)_2$ | $(010)_2$ | $(011)_2$ | $(100)_2$ | $(101)_2$ | $(110)_2$ | $(111)_2$ |
| $(N)_{10}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $(Z)_{10}$ | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
| $(Z)_{10}$ | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 |

Le choix est fait de prendre un biais valant $B = 2^{n-1} - 1$, ce que l'on appelle le « biais » du codage par excès :

$$\begin{aligned}
 B &= 2^{n-1} - 1 \\
 Z &= N - B \Leftrightarrow N = Z + B \\
 N &\in [0, 2^n] \quad ; \quad Z \in [-2^{n-1} + 1, 2^{n-1}]
 \end{aligned}$$

Soit finalement la table :

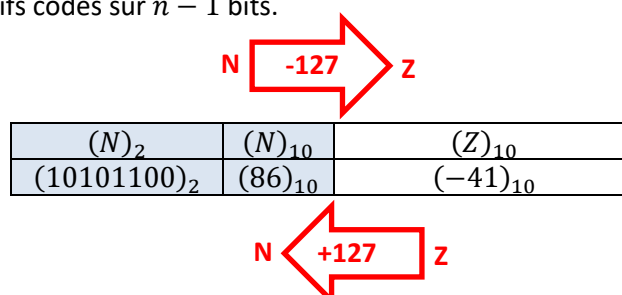
| | | | | | | | | |
|------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $(Z)_{10}$ | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 |
| $(N)_{10}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $(N)_2$ | $(000)_2$ | $(001)_2$ | $(010)_2$ | $(011)_2$ | $(100)_2$ | $(101)_2$ | $(110)_2$ | $(111)_2$ |

1.II.3.b.ii 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.



| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4 Nombres à virgule flottante

1.II.4.a Introduction

Comme pour les entiers, il n'est pas possible de représenter en machine l'infinité des nombres réels. Il a donc été choisi une normalisation permettant de représenter un nombre fini de réels à l'aide de « **flottants** », c'est-à-dire un **ensemble fini de nombres décimaux** qui seront associés à des réels.

Nous verrons un peu plus tard dans ce cours que 0.1 n'a par exemple aucune représentation informatique exacte, quel que soit le système d'exploitation utilisé (32 bits, 64 bits...). Ainsi, il ne sera jamais possible d'obtenir 0 dans le code suivant :

| | |
|--|--|
| <pre> S1 = 0 N = 100000000 for i in range(1,N): S1 += 0.1 S2 = i*0.1 Ecart = S2 - S1 print(Ecart) </pre> | <p>0.018870549276471138</p> <p>(Résultat en 64 bits)</p> |
|--|--|

1.II.4.b 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 :

$$\left\{ \begin{array}{l} 1200 = 1,2 \cdot 10^3 \\ 0,000056 = 5,6 \cdot 10^{-5} \\ -289456,5 = -2,894565 \cdot 10^5 \end{array} \right.$$

Prenons l'exemple de $X = -289456,5 = -2,894565 \cdot 10^5$

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

| $-2,894565 \cdot 10^5$ | | | |
|------------------------|-----------------|----------|-----------|
| — | 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 : $0,022 = 0,22 \cdot 10^{-1} = 2,2 \cdot 10^{-2}$

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

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.c 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 : $(0,011)_2 = (11)_2.2^{-3} = 3.2^{-3} = (0,375)_{10}$

1.II.4.d 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 :

$$\begin{aligned}
 (-2,894565.10^5)_{10} &= (-289456,5)_{10} = (-289456 - 0,5)_{10} \\
 &= -(10001101010110000)_2 - (0,1)_2 = -(10001101010110000,1)_2
 \end{aligned}$$

On écrit donc ce nombre binaire sous forme scientifique binaire :

$$-(10001101010110000,1)_2 = -(1,00011010101100001)_2.2^{18}$$

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

| $-(1,00011010101100001)_2.2^{18}$ | | | |
|-----------------------------------|-----------------|-------------------|-----------|
| — | 1 | 00011010101100001 | 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.

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

Finalement, il suffit de stocker les informations suivantes :

| | | |
|-------|---------------------|-----------|
| – | 0001101010101100001 | 18 |
| Signe | Mantisse | Puissance |

Toutefois, ces informations ne peuvent être stockées directement sous forme binaire, avec des 0 et des 1. (signe et puissance en base 10).

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 | 0001101010101100001 | 10010001 |
| Signe | Mantisse | Puissance |

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

| | | |
|-------|-----------|---------------------|
| 1 | 10010001 | 0001101010101100001 |
| 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 (à 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) |
| $(-289456,5)_{10} \leftrightarrow 11001000100011010101011000010000$ | | | |

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

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.e 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 = 10
>>> type(a)
<class 'int'>
>>> a = float(10)
>>> type(a)
<class 'float'>
>>> a = 10.0
>>> type(a)
<class 'float'>
```

1.II.4.f Exemples de transcodage

1.II.4.f.i 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 :

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

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

000001101000000000000000

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 (sinon la compléter de 0 à gauche : 100=0100).

Finalement, on a x codé en binaire simple précision :

| Simple précision 32 bits | $(2100)_{10}$ | | |
|-----------------------------|----------------------------------|--------------------|--------------------------|
| | 0 | 10001010 | 000001101000000000000000 |
| | 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 : $(-10,125)_{10} = (-1010,001)_2 = (-1,010001)_2 \cdot 2^3$

$$3 + 127 = (130)_{10} = (10000010)_2$$

| Simple précision 32 bits | $(10,125)_{10}$ | | |
|-----------------------------|----------------------------------|--------------------|--------------------------|
| | 1 | 10000010 | 010001000000000000000000 |
| | Signe (1 bit) | Puissance (8 bits) | Mantisse (23 bits) |
| | 11000001001000100000000000000000 | | |

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.f.ii Binaire en virgule flottante – Réel base 10

• Nombre entier

| | | | |
|--------------------------------|----------------------------------|--------------------|-------------------------|
| Mot 32 bits | 01001101100011101111001111000010 | | |
| Simple précision 32 bits | Identification des 3 parties | | |
| | 0 | 10011011 | 00011101111001111000010 |
| | Signe (1 bit) | Puissance (8 bits) | Mantisse (23 bits) |

- 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 : $(1,00011101111001111000010)_2 = (1,1168138980865479)_{10}$

Finalement, le nombre décimal associé vaut : $x = 1,11681389808654.2^{28} = 299792450$

Remarque : On ne peut pas coder la vitesse de la lumière en 32 bits... La preuve ? Essayer de transcoder 299792458, vous verrez que la mantisse dépasse 23 bits... Autrement dit :
 01001101100011101111001111000010 → 299792450 >>> `np.float32(299792458)`
 01001101100011101111001111000011 → 299792480 299792450.0

Attention : pour des problèmes d'arrondis des calculs en 64 bits (à priori), vous ne trouverez pas 299792450 mais 299792448 en faisant vous-même le calcul.

• Nombre à virgule

| | | | |
|--------------------------------|----------------------------------|--------------------|--------------------------|
| Mot 32 bits | 11000001001001110000000000000000 | | |
| Simple précision 32 bits | Identification des 3 parties | | |
| | 1 | 1000010 | 010011100000000000000000 |
| | 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 : $(1,0100111)_2 = (1,3046875)_{10}$

Finalement, le nombre décimal associé vaut : $x = -1,3046875.2^3 = -10,4375$

• Remarques à la suite de ces exemples

- Dans le cas où la mantisse possède une taille inférieure (ex 27) à la valeur de la puissance (ex 28), on peut se ramener à un nombre binaire sans virgule, et la conversion se fait directement d'un binaire à un entier : $(1,000111011110011110000100101)_2 * 2^{28} = (10001110111100111100001001010)_2 = (299792458)_{10}$
- Pour que le résultat en base 10 soit un nombre à virgule, il est nécessaire que la mantisse soit de longueur plus grande que la valeur de l'exposant :
 $(1,000111)_2 * 2^7 = (10001110)_2 = (142)_{10}$
 $(1,000111)_2 * 2^5 = (100011,1)_2 = (35,5)_{10}$

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.g Quelques nombres réservés

Les nombres associés aux puissances maximales positif et négative sont traités à part.

1.II.4.g.i Puissance max

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

• Infini

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

$$\pm\infty \Leftrightarrow \begin{matrix} 0 \\ 1 \end{matrix} \quad 11111111 \quad 000000000000000000000000$$

• NaN

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...

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

1.II.4.g.ii Puissance min

• Exception du zéro

On réserve le nombre rempli de zéros (exposant et mantisse) aux zéros, positif ou négatif :

$$\pm 0 \Leftrightarrow \begin{matrix} 0 \\ 1 \end{matrix} \quad 00000000 \quad 000000000000000000000000$$

Pourquoi ? Pour rappel, si la notation ci-dessus était normalisée, le nombre serait $1,000000000000000000000000 * 2^{-127}$. On ne pourrait donc jamais représenter le 0 puisque la caractéristique vaut 1.

• Notation dénormalisée (pour info)

Un nombre est dit dénormalisé lorsque l'exposant binaire est nul $(00000000)_2$ et la mantisse est non nulle. On est donc par exemple sur des nombres à puissance 2^{-127} en simple précision. Pour ces nombres, on devrait normalement écrire : $Valeur = signe * (1 + (0, mantisse)_2)_2 * 2^{-décalage}$. En notation dénormalisée, on écrit : $Valeur = signe * (0, mantisse)_2 * 2^{-décalage+1}$. En 32 bits, $décalage = 127$, cela donne : $Valeur = signe * (0, mantisse)_2 * 2^{-126}$. Ainsi, ce qui devrait être en notation normalisée $(1,1)_2 * 2^{-127}$ sera en dénormalisé $(0,1)_2 * 2^{-126} = (1)_2 * 2^{-127}$. Cela permet donc 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 selon notation normalisée ou dénormalisée proposées dans le tableau un peu plus bas.

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.h Limites de la représentation en virgule flottante

1.II.4.h.i 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 | 18 446 744 073 709 551 615 |

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 $2^n - 1$!!!

Exemple : on a vu précédemment que la vitesse de la lumière stockée en 32 bits donne 299792450 au lieu de 299792458. `>>> np.float32(299792458)`
299792450.0

1.II.4.h.ii 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 et ∞ . C'est donc au maximum 127. La plus grande partie significative vaut :

$$(1,11111111111111111111111111111111)_2 = (1,99999988079071)_{10}$$

Le nombre le plus grand vaut donc : $1,9999998807907104 \cdot 2^{127} \approx 3,4 \cdot 10^{38}$

- Le nombre le plus petit en notation normalisée a la puissance la plus faible. -127 étant réservé à la notation dénormalisée, c'est -126 , et la partie significative est nulle :

$$(1,00000000000000000000000000000000)_2 = (1)_{10}$$

En notation normalisée, le nombre le plus petit vaut donc : $2^{-126} \approx 1,2 \cdot 10^{-38}$

Si la dénormalisation n'existait pas, le plus petit nombre aurait été : $2^{-127} \approx 5,9 \cdot 10^{-39}$

- En revanche, en notation dénormalisée, le plus petit nombre est :

$$(0,00000000000000000000000000000001)_2 \cdot 2^{-127+1} = (1)_2 \cdot 2^{-23} \cdot 2^{-126} = 2^{-149} \approx 1,4 \cdot 10^{-45}$$

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

Sur un système d'exploitation 32 bits, ouvrez Excel par exemple, et tapez dans une case le nombre : $3,5 \cdot 10^{38}$, vous verrez alors : `#NOMBRE!`. Entrez $3,4 \cdot 10^{38}$, vous verrez `3,4E+38`. De même sur un système d'exploitation 64 bits, essayez $1,8 \cdot 10^{308}$ et $1,7 \cdot 10^{308}$.

Exemple sur iPhone 13 et « Calculator ∞ » :



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

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.h.iii *Ecart entre deux nombres successifs et conséquences*

• Expression de l'écart – Un exemple

Appelons ΔV l'écart entre deux valeurs successives représentables en virgule flottante. Les nombres vont se suivre de manière ordonnée ainsi :

$$\begin{aligned}
 &(1,000000000000000000000000)_2 * 2^n \\
 &(1,000000000000000000000001)_2 * 2^n \\
 &\vdots \\
 &(1,111111111111111111111111)_2 * 2^n \\
 &(1,000000000000000000000000)_2 * 2^{n+1}
 \end{aligned}$$

Soit Δm la variation de mantisse entre deux nombres successifs en 32 bits :

$$\Delta m = (0,000000000000000000000001)_2 = 2^{-23}$$

Prenons l'exemple des deux valeurs les plus grandes représentables en simple précision (128 réservé) :

$$\begin{aligned}
 A &= (1,111111111111111111111110)_2 * 2^{127} = 3,40282326356119 \cdot 10^{38} \\
 B &= (1,111111111111111111111111)_2 * 2^{127} = 3,4028234663852886 \cdot 10^{38}
 \end{aligned}$$

$$\Delta V = B - A = \Delta m * 2^{127} = 2^{-23} 2^{127} = 2^{127-23} = 2^{104} \approx 2,028240960365167 \cdot 10^{31}$$

Oui, il n'existe pas de nombres en simple précision entre A et B , et ils sont... relativement éloignés. Notons que ΔV grandit avec la puissance de 2 du nombre représenté. Il est constant pour une puissance de 2 donnée mais augmente avec n .

```
>>> a=3.40282326356119*10**38
>>> a+10**10-a
0.0
```

• Expression de l'écart – De manière générale

Objectif : Déterminer k tel que $\Delta V = 10^{(n-k)} = B - A$ avec $A = a \cdot 10^n, a \in [0 ; 10[$

Soit $A = a \cdot 10^n$ un réel représenté en virgule flottante, et B le réel lui succédant. Montrons que $\Delta V = B - A$ s'écrit $10^{(n-k)}, k \in \mathbb{R}$. On cherche la valeur de k . Posons :

- $A = \alpha \cdot 2^i = a \cdot 10^n$ avec $\alpha \in [0 ; 2[$ - $a \in [0 ; 10[$ - $n \in \mathbb{N}$
- $\Delta V = 10^{n-k}$ avec $k \in \mathbb{R}$ à déterminer
- p le nombre de bits de la mantisse (23 en simple précision)

$$\text{Alors : } \Delta V = \Delta m * 2^i = 2^{-p} 2^i = 2^{i-p}$$

$$\text{Trouvons } i \text{ tel que } \alpha \cdot 2^i = a \cdot 10^n : a \cdot 10^n = \alpha 2^i \Leftrightarrow \ln \frac{\alpha}{a} + n \ln 10 = i \ln 2 \Leftrightarrow i = n \frac{\ln 10}{\ln 2} + \frac{\ln \alpha}{\ln 2}$$

$$\text{Trouvons ensuite } k \text{ tel que } \Delta V = 2^{i-p} = b \cdot 10^{n-k} : 2^{i-p} = 10^{n-k} \Leftrightarrow (i-p) \ln 2 = (n-k) \ln 10 \Leftrightarrow (i-p) \log 2 = (n-k) \Leftrightarrow (i-p) \log 2 = n-k \Leftrightarrow k = n + (p-i) \log 2$$

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

Remplaçons l'expression de i dans k : $k = n + \left(p - n \frac{\ln 10}{\ln 2} + \frac{\ln \frac{a}{\alpha}}{\ln 2} \right) \log 2 = n + p \log 2 - n \frac{\ln 10}{\ln 2} \log 2 + \frac{\ln \frac{a}{\alpha}}{\ln 2} \log 2 = n + p \log 2 - n + \log \frac{a}{\alpha} = p \log 2 + \log \frac{a}{\alpha}$

On a :

- En simple précision ($p = 23$) : $p \log 2 \approx 6,92$
- En double précision ($p = 52$) : $p \log 2 \approx 15,65$

Par ailleurs, compte tenu des intervalles possibles de a et α : $1 \leq \frac{a}{\alpha} < 5$, soit $0 \leq \log \frac{a}{\alpha} < 0,7$ environ.

Finalement, en arrondissant légèrement :

- En simple précision : $6,9 < k < 7,6$
- En double précision : $15,6 < k < 16,3$

L'écart le plus grand sera obtenu pour la valeur de k la plus faible puisque $\Delta V = 10^{n-k}$.

On retiendra l'ordre de grandeur suivant :

| Soit $A = a \cdot 10^n$ | Simple précision | Double précision |
|--|------------------|------------------|
| ODG de l'écart max entre A et le suivant B | 10^{n-7} | 10^{n-16} |

$7 \approx 23 \log 2$
 $16 \approx 52 \log 2$

Exemple en 64 bits :

```
>>> 1+1e-15
1.0000000000000001
>>> 1+1e-16
1.0
```

Nous avons $A = 1 = 1 \cdot 2^0 = 1 \cdot 10^0$ ($i = n = 0, a = \alpha = 1$), soit :

- Avec la formule $\Delta V = 2^{i-p}$, on trouve $\Delta V = 2^{0-52} \approx 2,2 \cdot 10^{-16}$
- En passant par $\Delta V = 10^{n-k}$ et $k = p \log 2 + \log \frac{a}{\alpha}$, on trouve $k = 52 \log 2 + \log \frac{1}{1} \approx 15,65$.
 $\Delta V = 10^{n-k} \approx 10^{-15,65} \approx 2,2 \cdot 10^{-16}$

Ces résultats sont cohérents avec le tableau proposé.

Conclusion : Le nombre $1 + 10^{-16}$ ne peut pas être stocké de manière exacte en double précision. L'ajout 10^{-16} étant trop faible, l'erreur d'arrondi donne 1... Ajouter $2 \cdot 10^{-16}$ permet d'obtenir, via un arrondi aussi, le nombre suivant. Retrancher 1 à ce nombre donne l'écart de $2,2 \cdot 10^{-16}$...

```
>>> 1+10**-16
1.0
>>> 1+2*10**-16
1.0000000000000002
>>> 1+2*10**-16-1.0
2.220446049250313e-16
```

Remarques :

- Plus n grandit, plus l'écart entre valeurs successives grandit
- Si vous ne l'aviez jamais remarqué : $10^{-15,6} \approx 2,5 \cdot 10^{-16}$. Donc, si un écart est de l'ordre de $10^{-15,6}$, ne pas croire que 10^{-16} le majore

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

• Applications

Comparaison à 0

Pour vérifier que 2 nombres sont égaux, on écrit souvent : $a == b$. Cela revient à écrire : $a - b == 0$

Et, malheureusement, alors que c'est censé retourner True, cela ne fonctionne pas toujours :

```
>>> 1.2+1.4-2.6==0
False
```

```
>>> abs(1.2+1.4-2.6)<1e-16
False
```

Pourquoi ? Des erreurs d'arrondis ! Du fait d'une erreur d'arrondi, on peut se retrouver à avoir a et b décalés, par exemple, du plus petit écart étudié précédemment ΔV ! Alors, $a = b \pm \Delta V$ ou $|a - b| = \Delta V$ avec $\Delta V > 0$

```
>>> abs(1.2+1.4-2.6)<1e-15
True
```

On va donc étudier l'inégalité $|a - b| \leq \varepsilon$. Mais quelle valeur de ε faut-il prendre ? En effet, si on prend ε :

- Trop faible : Les nombres ne seront jamais supposés égaux alors que c'est une erreur d'arrondis qui a fait qu'ils ne sont plus identiques... Et qu'ils devraient être considérés comme tel...
- Trop grande : Les nombres seront faussement supposés égaux, par exemple $|a - b| = 10\Delta V$, ce n'est probablement pas une erreur d'arrondi qui a conduit à un tel écart... Il faut donc les considérer distincts

Comparer deux nombres, cela revient donc à voir si l'écart entre eux est de l'ordre de l'écart possible entre deux représentations successives en virgule flottante ΔV . Ainsi, on supposera en 32 simple précision que $3,40282326356119 \cdot 10^{38} = 3,4028234663852886 \cdot 10^{38}$ alors que ΔV vaut $2,03 \cdot 10^{31}$

En restant dans ce cours sur des ordres de grandeur, on peut résumer les choses ainsi :

| Soit $A = a \cdot 10^n$ | Simple précision | Double précision |
|-------------------------|------------------|------------------|
| ODG de ε | 10^{n-7} | 10^{n-16} |

Mais attention à garder en tête que ce ne sont que des ordres de grandeur. On pourra calculer précisément ΔV si on le souhaite... Exemple pour $n = 0$ en double précision :

```
>>> abs(1.01+1.02-2.03)<10**(-16)
False
```

```
>>> abs(1.01+1.02-2.03)<10**(-15)
True
```

Les nombres sont proches de 1, on a donc environ $k = 52 \log 2 \approx 15,65$, soit $\Delta V \approx 2,2 \cdot 10^{-16}$ comme calculé précédemment. On comprend que les tests se comportent différemment à 10^{-15} et 10^{-16} ...

Si on veut être sûr que le test passe, on prendra :

| Soit $A = a \cdot 10^n$ | Simple précision | Double précision |
|-------------------------|------------------|------------------|
| ODG de ε | 10^{n-6} | 10^{n-15} |

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.i Conséquences de la représentation des nombres en virgule flottante

1.II.4.i.i Dépassement de capacité - Overflow

• Principe

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.

| | |
|---|-----|
| <pre>import numpy as np A = np.array([255], dtype='uint8') A[0] += 1 print(A)</pre> | [0] |
|---|-----|

Comme nous l'avons vu avec Excel pour un système 64 bits, le dépassement de capacité consistant à écrire le nombre $1,8 \cdot 10^{308}$ renvoie une erreur car ici aussi, il y a dépassement de capacité.

Voici un exemple d'overflow que vous pourriez rencontrer. Sur un calcul de force d'attraction gravitationnelle entre la terre et la lune, on définit des paramètres dans un array, en particulier la distance Terre-Lune en mètres :

```
Donnees = np.array([384000000])
```

La force gravitationnelle fait apparaître le carré de cette distance. On le calcul donc, et voici ce que l'on obtient :

```
>>> Donnees[0]**2
__main__:1: RuntimeWarning: overflow encountered in long_scalars
1677721600
```

Effectivement :

```
>>> 384000000**2
1474560000000000000
```

```
>>> type(Donnees[0])
<class 'numpy.int32'>
```

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

Regardons de plus près le problème :

En créant un array avec un entier, le nombre 384000000 a été codé en entier sur 32 bits... Le résultat au carré de ce nombre ne se code pas sur 32 bits... Le plus grand nombre est 4294967296. Il y a « overflow » et le résultat renvoyé est faux...

Solutions :

- La meilleure solution consiste à diviser la distance par une distance caractéristique
- **Sinon, on peut travailler avec un flottant en ajoutant « .0 » à la fin de 384000000.0**

```
>>> Donnees = np.array([384000.0])
>>> Donnees[0]**2
147456000000.0
```
- Ou encore, dire à python que l'entier doit être codé sur 64 bits (si la machine le permet) :

```
np.array([384000000],dtype='int64')
```

• Exemple : Ariane 5 – Juin 1996

Les valeurs d'accélération horizontales de la fusée étaient mesurées et stockées sur une variable entière codée sur 8 bits. Si les valeurs d'accélération mesurées sur Ariane 4 étaient d'un ordre de grandeur de 64, Ariane 5 bien plus puissante a subi des accélérations de l'ordre de 300... pour un nombre maximum codable de 255. Le résultat obtenu par la mesure était donc un nombre erroné qui a conduit l'ordinateur de bord à choisir l'autodestruction.



```
import numpy as np
a = np.int8(100)
print(type(a))
b = np.int8(200)
Resultat = a+b
print(Resultat)
```

```
__main__:5: RuntimeWarning: overflow encountered in
n ubyte_scalars
44
```

Au lieu de 300...

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.II.4.i.ii Erreurs d'arrondis

• Principe

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

```
>>> 0.1+0.2
0.30000000000000004
>>> 2.0**60+1-2.0**60
0.0
```

Les erreurs d'arrondis viennent de la limite de la taille de la mantisse. Elles peuvent avoir deux origines :

- Nombre décimal à stocker de mantisse binaire théorique supérieure à la capacité stockable
- Nombre obtenu par opérations sur des nombres stockables, dont le résultat est non stockable... (on pourra se renseigner sur la réalisation des opérations entre nombres en virgule flottante)

• Cumul d'erreur d'arrondi : Missile Patriot – Février 1991

Le 25 février 1991, une batterie anti-missile a laissé passer un missile scud à cause d'une erreur de calcul. L'horloge interne du système de défense comptait le temps en dixièmes de secondes. Le temps était calculé en 24 bits. Malheureusement, le nombre 1/10 n'a pas de développement binaire fini...



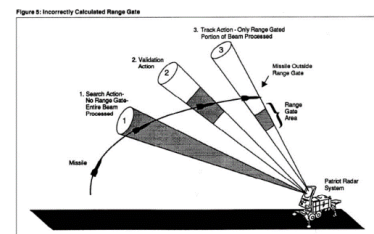
En effet, la partie décimale de 0,1 en binaire est infinie :

$$(0,1)_{10} = (0,0001100 \dots)_2$$

Le nombre réellement stocké était sur 24 bits, donc : $(0,00011001100110011001100)_2$

La valeur réellement codée était donc 0,0999999046325684 s et l'erreur d'arrondi valait 0,00000009536743164617610000000 s.

Comme le système anti-missile avait été réinitialisé environ 100h plus tôt, l'erreur d'arrondi sur chaque dixième d'heure s'est répétée 10 fois par seconde, soit environ 3,6 millions de fois... Le temps d'erreur cumulé était donc de 0,34 secondes... Le missile scud se déplaçant à 1676 m/s, il parcourt 569 m en 0,34s. Cette erreur de distance a fait sortir le missile scud de la zone d'acquisition du missile Patriot...



| | |
|---|---|
| <pre>t = 0 dt = 0.1 for i in range(10000): t+=dt print(t)</pre> | <pre>>>> (executing file "<tmp 1>") 1000.0000000001588</pre> |
|---|---|

Autre exemple pour une moyenne :

| | |
|--|--|
| <pre>N= 10000 L = [i for i in range(N)] S = 0 for t in L: S+=t M1 = S/N print(M1-M2)</pre> | <pre>S = 0 for t in L: S+=t/N M2 = S 9.094947017729282e-12</pre> |
|--|--|

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

• Conséquence importante - Test « == »

Soient $a = 0,1$, $b = 0,2$ et $c = a + b = 0,3$. Ils sont de l'ordre de $0,1 = 10^{-1}$, soit $n = -1$. L'écart entre nombres en virgule flottante proche de ces nombres est de l'ordre de $\Delta V \approx 10^{-1-16} = 10^{-17}$.

Exécutez ce programme (en 64 bits) et vous aurez tout compris :

| | |
|---|--|
| <pre> a = 0.2 b = 0.1 c = a + b print('c = ',a,' + ',b,' = ', c) if abs(c-0.3)<=10**(-17): print('OUI: (c-0.3)<=10**(-n) ') else: print('NON: (c-0.3)<=10**(-n) ') if abs(c-0.3)<=10**(-16): print('OUI: (c-0.3)<=10**(-n) ') else: print('NON: (c-0.3)<=10**(-n) ') </pre> | <pre> >>> (executing file "<tmp 1>") c = 0.2 + 0.1 = 0.30000000000000004 NON: c==0.3 OUI: (c-0.3)<=10**(-n) </pre> |
|---|--|

Comme on travaille sur des ordres de grandeur, et comme abordé plus haut, il sera prudent de prendre au minimum 10^{-15} au lieu de 10^{-16} pour être sûr que le test passe...

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 discriminant nul. Le test « *Disc* == 0 » ne fonctionnera pas à tous les coups...

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

| | |
|--|--|
| <pre> a,b,c = 0.1,0.2,0.3 Test = (a+b)/c print(Test) </pre> | 1.0000000000000002 |
| BIEN (en 64 bits) | JAMAIS BIEN |
| <pre> if abs(Test - int(Test)) < 1e-15: print("Cool") else: print("Pas cool") </pre> Cool | <pre> if Test == int(Test): print("Cool") else: print("Pas cool") </pre> Pas cool |
| | PAS BIEN (en 64 bits) |
| | <pre> if abs(Test - int(Test)) < 1e-16: print("Cool") else: print("Pas cool") </pre> Pas cool |
| | <pre> if type(Test) == int: print("Cool") else: print("Pas cool") </pre> Pas cool |

Remarque : « `type(Test) == int` » ne fonctionnerait même pas s'il n'y avait pas d'erreur d'arrondi car le résultat d'une division est un flottant :

```

>>> 1/1
1.0

```

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

• Approximation d'une dérivée

Soit le code suivant et ses résultats :

| | |
|---|-----------------------------------|
| <pre>def f(x): return x+1 def fp(f,x,h): print(x+h, f(x)-f(x+h)) return (f(x+h)-f(x))/h print(fp(f,1,1e-16)) print(fp(f,1,1e-15))</pre> | <pre>0.0 0.8881784197001251</pre> |
|---|-----------------------------------|

Il se produit des erreurs d'arrondis qui conduisent à obtenir une dérivée nulle si h trop petit et une dérivée fausse si h assez grand pour éviter le premier problème mais trop petit quand même, ce qui induit une évaluation de $f(x+h) - f(x)$ un peu trop arrondie. Regardons cela de plus près :

| | |
|---|---|
| <pre>def f(x): return x+1 def fp(f,x,h): print(x+h, f(x)-f(x+h)) return (f(x+h)-f(x))/h print(fp(f,1,1e-16)) print(fp(f,1,1e-15))</pre> | <pre>1.0 0.0 0.0 1.0000000000000001 -8.881784197001252e-16 0.8881784197001251</pre> |
|---|---|

On remarque que l'erreur d'arrondi se produit à la fois sur $x+h$ et $f(x+h) - f(x)$. En effet, ils sont du même ordre de grandeur proche de $x = f(x) = 1 = 10^0$. Comme vu précédemment, l'écart entre deux valeurs proches de 1 est très proche de 10^{-16} , l'ajout de 10^{-16} est insuffisant pour les 2 calculs. Soit maintenant l'exemple suivant et ses résultats :

| | |
|---|---|
| <pre>def f(x): return x+100 def fp(f,x,h): print(x+h, f(x)-f(x+h)) return (f(x+h)-f(x))/h print(fp(f,1,1e-15)) print(fp(f,1,1e-14))</pre> | <pre>1.0000000000000001 0.0 0.0 1.0000000000000001 -1.4210854715202004e-14 1.4210854715202004</pre> |
|---|---|

L'erreur d'arrondi ne se produit plus sur $x+h$ mais sur la différence $f(x+h) - f(x)$ car ils sont de l'ordre de $100 = 10^2$... On aurait pu s'attendre à ce que le calcul soit non nul pour 10^{-13} et non 10^{-14} mais nous travaillons sur des ordres de grandeur... Soit maintenant les exemples suivants :

| | |
|--|---|
| <pre>def f(x): return x+1 def fp(f,x,h): return (f(x+h)-f(x))/h print(fp(f,1,1e-16)) print(fp(f,1,1e-15)) print(fp(f,1,1e-14)) print(fp(f,1,1e-13)) print(fp(f,1,1e-12))</pre> | <pre>0.0 0.8881784197001251 0.9769962616701378 0.9992007221626409 1.000088900582341</pre> |
|--|---|

On remarque que plus h est grand, meilleure est la précision de la dérivée (elle vaut 1 normalement...). En effet, en augmentant h , on diminue l'effet de l'erreur d'arrondi de $f(x+h) - f(x)$ sur le calcul $\frac{f(x+h)-f(x)}{h}$. Mais attention, selon l'évolution de $f(x)$, qui ici est linéaire (par chance), l'augmentation de h va finir par fausser la dérivée renvoyée...

Conclusions : Veiller à choisir h assez grand pour ne pas avoir un résultat nul (selon ODG de x et $f(x)$), et assez petit ...mais pas trop pour bien évaluer f' . Pas si simple finalement

| | | |
|----------------------|---------------------------------|---|
| Dernière mise à jour | Informatique | Denis DEFAUCHY – Site web |
| 24/05/2022 | 10 - Représentation des nombres | Cours |

1.III. Les types sous Python

Finissons ce chapitre en abordant le module numpy qui permet de choisir les types des nombres utilisés sous Python.

Python attribue de manière automatique un type aux nombres entrés :

```
>>> a = 1
>>> type(a)
<class 'int'>
```

Et sur mon ordinateur, l'entier doit être en int64 puisque :

```
>>> a = 1.0
>>> type(a)
<class 'float'>
```

```
>>> a = 400000000
```

```
>>> a**2
1600000000000000000
```

```
>>> a = 400000000
```

Lorsque l'on crée un array avec numpy, d'autres types sont définis :

```
>>> A = np.array([a])
```

A l'aide de numpy, il est possible de préciser un type de nombre dans un array ou pour une variable :

```
>>> type(A[0])
<class 'numpy.int32'>
```

```
>>> import numpy as np
```

```
>>> A[0]**2
-66060288
```

```
>>> A = np.array([400000000], dtype='uint64')
```

```
>>> A[0]**2
1.6e+17
```

(u pour non « unsigned » ou « non signé », dans \mathbb{N})

```
>>> a = np.float64(0.1)
```

```
>>> b = np.float64(0.1)
```

```
>>> a-b
0.0
```

```
>>> b = np.float32(0.1)
```

```
>>> a-b
-1.4901161138336505e-09
```

Utiliser la commande « `dir(np)` » pour tous les types disponibles.