CH XI : Représentation de nombres en Python

Ce chapitre aborde la question de la représentation informatique des nombres en machine, à travers le langage **Python**. On y expose la manière dont les nombres entiers, puis les flottants sont représentés en machine, puis les approximations que cela induit dans le calcul avec flottants.

I. Représentation des entiers

I.1. Numération décimale, binaire, hexadécimale

I.1.a) Décomposition en base B de nombres entiers

Les ordinateurs fonctionnant, pour le moment, à l'aide de composants prenant uniquement deux états possibles (en attendant l'ordinateur quantique), l'information y est naturellement codée sur la base de **bits**. Un **bit** (contraction de *binary digit*, chiffre binaire) est un chiffre égal à 0 ou 1.
Ceci conduit naturellement à l'écriture des entiers et des nombres en base 2 (**binaire**), d'où on déduit presqu'immédiatement l'écriture en base 16 (**hexadécimale**) également beaucoup utilisée en informatique. Notre système de numération usuel est, quant à lui, exprimé en base 10.

On rappelle le théorème suivant.

Théorème 1 (Écriture d'un entier en base B)**.**
Soit B un entier naturel avec $B \geqslant 2$.
Pour tout $N \in \mathbb{N}$, il existe une unique suite presque nulle $(a_n)_{n\in\mathbb{N}}$ d'entiers de $[\![0, B-1]\!]$ telle que :

$$N = \sum_{k=0}^{+\infty} a_k B^k$$

Notation
Si N est non nul, on dispose alors d'un plus grand indice p tel que $a_p \neq 0$ et on écrit alors :

$$N_{(B)} = a_p a_{p-1} \cdots a_1 a_0$$

en juxtaposant les chiffres (comme c'est d'usage pour la base 10).

Algorithme de décomposition d'un entier en base B
On dispose d'un algorithme pour construire la suite $(a_n)_n$:

- on commence par effectuer la division euclidienne de N par B : le reste donne a_0, on récupère le quotient N_1,
- on effectue la division euclidienne de N_1 par B : le reste donne a_1, on récupère le quotient N_2,
- etc.
- on s'arrête lorsque le quotient est nul.

Remarque

- En base B, il faut donc B symboles pour représenter les nombres. Pour ce qui est des bases 2 (binaire), 10 (décimale) et 16 (hexadécimale) :
  - × dans le système binaire, on utilise les chiffres 0 et 1,
  - × dans le système décimal, on utilise les chiffres 0, 1, 2, ..., 9,
  - × dans le système hexadécimal, on utilise les chiffres 0, 1, ..., 9 auxquels on ajoute les lettres A, B, C, D, E, F.

  Par exemple, le nombre 503 (base 10) s'écrit
  - × en binaire : 111110111
  - × en hexadécimal : 1F7.

- Le passage de l'écriture binaire à hexadécimale est relativement simple : il suffit de regrouper les bits par 4 en partant de la droite, et d'associer le caractère idoine.
  On reprend l'exemple du nombre 503. On découpe son écriture en binaire en regroupant les bits par 4 en partant de la droite :

  $$1\ 1111\ 0111$$

  On remarque de plus que :
  - × le nombre binaire 0111 est égal à 7 (en décimal et en hexadécimal),
  - × le nombre binaire 1111 est égal à 15 (en décimal), donc F (en hexadécimal),
  - × le binaire 1 reste inchangé.

  On en déduit que l'écriture hexadécimal de 503 est 1F7.

I.1.b) Extension aux nombres non entiers

La représentation précédente s'étend également à certains nombres non entiers, des nombres possédant une représentation finie dans la base voulue.

- Pour une suite presque nulle $(a_n)_{n\in\mathbb{Z}}$ d'éléments de $[\![0, B-1]\!]$, le théorème précédent s'écrit :

  $$\sum_{k=-\infty}^{+\infty} a_k\, B^k \;=\; \sum_{k=0}^{+\infty} a_k\, B^k + \sum_{k=1}^{+\infty} a_{-k}\, B^{-k}$$

  Ceci s'écrit, en base B :

  $$a_p a_{p-1} \cdots a_0, a_{-1} a_{-2} \cdots a_{-q}$$

- De tels nombres, suivant la base B, permettent d'obtenir :
  - × les nombres décimaux pour $B = 10$, qui sont de la forme $\dfrac{p}{10^q}$ où $(p, q) \in \mathbb{N}^2$,
  - × les nombres dyadiques pour $B = 2$, qui sont de la forme $\dfrac{p}{2^q}$ où $(p, q) \in \mathbb{N}^2$.

  Remarquons que tout nombre dyadique est décimal, mais que la réciproque est fausse : pour des raisons arithmétiques, $\dfrac{1}{10}$ ne peut s'écrire sous la forme $\dfrac{p}{2^q}$ où $(p, q) \in \mathbb{N}^2$. En effet, si c'était le cas, on aurait $10\,p = 2^q$, ce qui est impossible car 5 divise $10\,p$, mais 5 ne divise pas 2^q.

Décomposition des réels en base B
On peut démontrer mathématiquement l'existence et l'unicité d'une telle décomposition d'un nombre réel en base B quelconque.
Néanmoins, d'un point de vue informatique, le stockage d'une représentation infinie est impossible : tous les objets sont de taille finie, et les représentations en base B de nombres réels échappant aux catégories décrites plus haut ne peuvent être qu'*approchées* (on détaille cela ultérieurement).

I.2. Représentation des entiers sur n bits

Sous Python 2.x et autres langages « anciens », les entiers relatifs sont représentés **sur n bits**, où n est un entier prédéterminé. Cet entier n, sur des systèmes anciens, était égal à 32, mais, aujourd'hui, presque tous les systèmes codent sur 64 bits. Ceci désormais permet de coder à peu près tous les entiers de l'intervalle $[\![-(2^n-1), 2^n-1]\!]$, sur la plupart des systèmes. Les variations dépendent de la convention choisie (cf. ci-dessous).

Sous Python 3.x, le mode de représentation sur n bits est toujours utilisé, et il est complété par un autre mode pour les entiers situés au-delà : il s'agit de la **représentation multi-entiers**, expliquée plus loin.

Il existe deux conventions pour représenter les entiers du n bits.

I.2.a) Convention de la valeur signée

- Le principe est le suivant. Chaque entier $p \in \mathbb{Z}$ est codé sur n bits $b_{n-1}\cdots b_1 b_0$, avec :
  - $b_{n-1} = 0$ si p est positif, $b_{n-1} = 1$ sinon,
  - $b_{n-2}\cdots b_1 b_0$ est l'écriture binaire de la valeur absolue $|p|$

  On code ainsi les entiers de l'ensemble $[\![-(2^{n-1}-1), 2^{n-1}-1]\!]$ et 0 possède une double écriture (commençant par un 1 ou un 0).
- L'addition de deux entiers codés suivant cette convention est un algorithme qui distingue 2 cas, suivant si les signes des deux entiers sont égaux ou opposés.

I.2.b) Convention du complément à deux

Un entier positif ou nul est représenté par la convention $0b_{n-2}\cdots b_1 b_0$ (sur n bits, donc). Pour un entier strictement négatif, la représentation commence par 1 et est suivie de la valeur absolue codée avec des bits inversés (1 pour 0 et 0 pour 1), incrémentée de 1.

Exemple
On cherche par exemple à coder $k = -13$ sur 6 bits :
- en écriture binaire, sur 5 bits, $13 = 8 + 4 + 1 = 01101_{(2)}$,
- on inverse les bits et on ajoute le signe en tête de représentation : 110010,
- on incrémente de 1 : 110011.

Le nombre obtenu est 51, c'est-à-dire $64 - 13 = 2^6 - 13$.

On résume la situation ainsi.

Définition
Soit $n \in \mathbb{N}^*$.
Soit $x \in [\![-2^{n-1}, 2^{n-1}-1]\!]$.
La représentation **complément à deux de x sur n bits** est la représentation binaire de l'entier :
- x lui-même si $x \geqslant 0$,
- $2^n - x$ si $x < 0$.

Remarque
Lorsqu'on dispose de n bits, les entiers codés vont de -2^{n-1} à $2^{n-1}-1$ (inclus). On a gagné un entier, par rapport à la convention de la valeur signée (c'est parce que 0 admet désormais une unique représentation), mais ce n'est pas là le plus grand avantage.
Par exemple, sur 8 bits :
- le plus grand entier codé est 0111 1111 qui est 127,
- le plus petit entier codé est 1100 0000 qui est -128.

Proposition 1.

Soit $n \in \mathbb{N}^*$.

- *Inversion de la représentation.*
  Étant donné une représentation $a_n a_{n-1} \cdots a_1 a_0$ sur n bits, on reconstitue l'entier x représenté en suivant la disjonction de cas suivante :
  - *si $a_n = 0$, il s'agit de $\sum_{k=0}^{n} a_k 2^k$,*
  - *si $a_n = 1$, il s'agit de $\sum_{k=0}^{n} a_k 2^k - 2^n$.*
- *Addition.*
  Soient x et y deux entiers représentés sur n bits, respectivement, par :

$$a_n a_{n-1} \cdots a_1 a_0 \qquad et \qquad b_n b_{n-1} \cdots b_1 b_0$$

L'addition binaire de ces représentations permet d'obtenir un entier $c_{n+1} c_n \cdots c_1 c_0$. La représentation en complément à deux de $x + y$ est alors donnée par $c_n c_{n-1} \cdots c_1 c_0$.

Exemple

Avec la convention de la valeur signée, il faut définit un algorithme pour l'addition de nombres de même signe et un autre pour l'addition de nombres de signes opposés.
Avec la convention des compléments à 2, il suffit de poser l'addition avec un système de retenue similaire à l'addition en base 10 pour conclure.

13	0	0	1	1	0	1
-11	1	1	0	1	0	1
2	0	0	0	0	1	0

Addition $13 + (-11)$

6	0	0	0	1	1	0
-11	1	1	0	1	0	1
-5	1	1	1	0	1	1

Addition $6 + (-11)$

Détaillons l'écriture des nombres négatifs :

- pour (-11) : sur 6 bits, $64 - 11 = 53$ qui s'écrit 110101. Ceci est la représentation de (-11) en complément à deux.
- le résultat de l'addition de droite est 111011 qui, en binaire, signifie 59. Pour retrouver quel nombre est ainsi représenté, on voit le chiffre 1 en tête : c'est un entier strictement négatif. C'est donc $59 - 2^6 = -5$.
- on confirme : $6 + (-11) = -5$.

Une économie de mémoire ?

Il peut paraître superflu d'immobiliser 64 bits pour représenter des entiers de petite taille (typiquement, 11 suffisent pour représenter les entiers compris entre -1024 et 1023). Avant Python 3.x, la taille des structures utilisées pour représenter les entiers variait suivant l'entier considéré. Cette économie de mémoire n'est désormais plus pratiquée sous Python.

I.3. Entiers multi-précisions

En faisant tourner l'affichage des entiers `2**p` pour `p` partant de 1, Python peut aller assez loin (au-delà de `p = 16000`), alors que la représentation sur 64 bits limite le sujet à `p=63`. La raison à cela est que la classe `int` de Python 3.x prend en charge les **entiers multi-précisions**.
Le calcul sur les entiers multi-précisions est systématique dans les logiciels de calcul formel ou de calcul symbolique (XCas, module SymPy de Python, SageMath (écrit en Python, d'ailleurs) ou, pour les versions payantes : Maple, Mathematica). Ce mode de représentation des entiers est capital dans certains domaines des mathématiques dites « expérimentales », comme l'arithmétique ou la cryptographie, ou l'utilisation de « très grands » entiers est monnaie courante (et préférée à celle des flottants où les approximations de calcul se cumulent et rendent les résultats peu fiables, cf. plus loin).

Représentation des entiers multi-précisions.
Le procédé de représentation des entiers multi-précisions n'est pas encore standardisé, nous détaillons donc *un* procédé parmi ceux utilisés.

1) L'idée est de représenter un nombre N en base b (et non plus forcément en base 2). La liste de ses chiffres est alors élaborée : pour $N = 1632$ et $b = 13$, la liste est `[7,8,9]` (puisque $1632 = 9 \times 13^2 + 8 \times 13 + 7$).

2) Chacun des chiffres est alors stocké dans un *registre* de taille imposée (par le gestionnaire de mémoire et par le processeur) : on choisit alors la base b la plus grande qui permette ce stockage, minimisant le nombre de registres utilisés.

Comme les registres ne sont pas de taille 2, ceci permet plus de possibilités en termes de base b qu'une représentation en base 2 comme pour la représentation sur n bits.

Opérations sur entiers multi-précisions.

- Les opérations d'addition et de multiplication sont confiées à des algorithmes spécifiques. Étant donné la potentielle (très) grande taille des entiers considérés, les opérations d'addition et de multiplication ne peuvent plus être considérées comme menées en temps constant. C'est pourquoi les algorithmes d'addition et de multiplication ont leur propre complexité, *dépendante de la taille des entiers considérés*.
- Pour illustrer tout cela, appelons *grands* entiers des entiers dépassant $2^{63} + 1$ en valeur absolue et *petits* entiers des entiers représentables sur 64 bits. Ainsi, les *grands* entiers sont représentés par des listes donnant leur décomposition en base b, où b est lui-même un *petit* entier. On suppose que les opérations arithmétiques (addition, multiplication) sur les petits entiers sont effectuées en temps constant.
- Soient N et M deux *grands* entiers, qu'on représente alors :
  - N par la liste $[c_0, c_1, \dots, c_p]$
  - M par la liste $[d_0, d_1, \dots, d_q]$.

  Les entiers p et q sont les *tailles* respectives des entiers M et N : on suppose que ce sont également de *petits* entiers (ce qui permet à N et M de valoir au plus $\alpha^{\alpha+1} - 1$ avec $\alpha = 2^{63}$, ce qui laisse de la marge).
- Mathématiquement, on a
$$N = \sum_{k=0}^{p} c_k b^k \quad \text{et} \quad M = \sum_{k=0}^{p} d_k b^k$$
  Ainsi, l'addition de N et M est un algorithme consistant à sommer les listes terme à terme, avec un système de retenue : le nombre d'opérations effectuées est d'une complexité en $O(\max(p, q))$.

- La multiplication « naïve » de N et M reproduit la formule suivante :

$$N \times M \;=\; \sum_{k=0}^{p+q} \left(\sum_{(i,j):i+j=k} c_i d_j \right) b^k$$

On effectue alors un nombre de multiplications de l'ordre de $\dfrac{p \times q}{2}$, ce qui donne une complexité en $O(p \times q)$.

Algorithmes de multiplication rapide

- Des algorithmes de multiplication plus efficaces sont connus depuis les années 60, et ont amélioré la complexité quadratique constatée dans le cas de la multiplication naïve. Citons par exemple l'algorithme de Karatsuba (1962, en $O(n^{1.58})$) et l'algorithme de Schönhage-Strassen par transformée de Fourier rapide (1971, en $O\big(n\ln(n)\ln\ln(n)\big)$).
- L'algorithme de Fürer (2007) possède l'un des plus faibles coûts de calcul à l'heure actuelle, avec une complexité en $O\left(n\ln(n)2^{\ln^*(n)}\right)$, où $\ln^*(n)$, *logarithme itéré*, désigne le nombre de fois que la fonction ln doit être appliquée à n pour que le résultat soit inférieur ou égal à 1. On a la croissance comparée suivante :

$$\frac{\ln^*(n)}{\ln\ln(n)} \underset{n\to+\infty}{\longrightarrow} 0$$

Ce qui donne l'avantage à l'algorithme de Fürer sur celui de Schönhage-Strassen.

II. Représentation des flottants sur n bits

Définition Un nombre **représenté en virgule flottante** est un nombre écrit de la manière suivante :

$$x \;=\; \varepsilon \times m \times b^k$$

où :

× $\varepsilon \in \{-1, 1\}$ est le signe du nombre x,

× b est une **base** entière $\geqslant 2$ (2 en binaire, 10 en décimal, 16 en hexadécimal, etc.)

× m est un nombre à virgule appelé **mantisse**, représenté en base b et ayant une représentation finie dans cette base,

× k est un entier relatif.

Exemple
Par exemple :

× en base 10, le nombre -32.75 possède comme représentations en virgule flottante -327.5×10^{-1} ou -3.275×10^{1}

× en base 2, le même nombre s'écrit -100000.11 en binaire et possède comme représentations en virgule flottante -10000011×2^{-2} ou -1.0000011×2^{5} en base 2.

Définition
Une **représentation en virgule flottante normalisée** est une représentation dans laquelle la mantisse est de la forme :

$$m \;=\; b_0.b_{-1}b_{-2}\cdots b_{-p}$$

avec, pour tout $k \in [\![0, p]\!]$, $b_k \in [\![0, b-1]\!]$ et $b_0 \neq 0$.

Proposition 2 (Existence et unicité des représentations)**.**

Soit $x \in \mathbb{R}$.

Le nombre x admet une représentation en virgule flottante si et seulement si il admet une représentation finie en base b. Dans ce cas, il admet une unique représentation en virgule flottante normalisée.

Remarque

- Par exemple, seuls les nombres décimaux (resp. les nombres dyadiques) admettent une représentation en virgule flottante en base 10 (resp. en base 2).
- Pour tout nombre dyadique *non nul*, la représentation en virgule normalisée a une mantisse qui commence nécessairement par un 1.

III. Calculs avec les flottants

Pour calculer avec des flottants en Python, on est habitués à entrer des flottants sous forme décimale et à lire des affichages également sous forme décimale.
Or, comme nous l'avons vu précédemment, les flottants sont représentés sous forme normalisée binaire. Ainsi, à moins d'entrer des nombres décimaux qui sont aussi dyadiques et à représentation binaire pas trop longue, un arrondi est effectué à deux moments :

× au moment de convertir les variables en entrée sous forme normalisée binaire,

× au moment de convertir, après opérations, le résultat d'une forme normalisée binaire à une forme décimale.

Ces opérations d'arrondies peuvent être résumées par une fonction Φ définie sur $\mathbb{R}$ et à valeurs dans l'ensemble des représentations normalisées binaires, bijective. On utilise alors Φ en entrée et Φ^{-1} en sortie.
Un objectif de cette partie du cours est de saisir l'importance des erreurs d'arrondi dans les calculs avec flottants : comment s'additionnent, se multiplient, s'accumulent les approximations consenties au moment d'arrondir, dans les calculs avec flottants.

III.1. Sommer des flottants

III.1.a) Exemple 1 : $\sum_{k=1}^{n} \frac{1}{n}$ peut ne pas donner 1

Soit $n \in \mathbb{N}^*$.
Prenons l'exemple du calcul de $\sum_{k=1}^{n} \frac{1}{n}$ qui, mathématiquement, doit donner 1.

```
n = int(input('Entrer un entier n : '))
S = 0
for k in range(n) :
    S = S + 1/n
print(S)
```

- Par exemple, pour $n = 10$, la somme contient des nombres dont l'écriture dyadique n'est pas finie. En effet, si $\frac{1}{10}$ était un nombre dyadique, on pourrait écrire (ce qui est impossible) :

$$10p = 2^k \quad \text{avec } (k,p) \in \mathbb{N}^2$$

Ce nombre possède donc un développement infini.

- Plus précisément, on peut écrire :

$$\frac{1}{10} = 0.0001100110\underline{0011}$$

Il suffit de vérifier que $\frac{1}{5} = \sum_{k=1}^{+\infty} \left(\frac{1}{2^{4k-1}} + \frac{1}{2^{4k}} \right)$ ce qui est bien le cas.

- Ceci signifie qu'en machine la conversion dyadique de $\frac{1}{10}$ génère une erreur d'approximation qui se cumule dans le calcul. Et, en effet, le programme ci-dessus affiche :

```
Out [1] :   0.9999999999999999
```

Remarque
En revanche, lorsque n est une (petite) puissance de 2, le calcul est exact et renvoie `1.0`. C'est le cas, par exemple, pour `n=16`.

III.1.b) Exemple 2 : non associativité de l'addition

Si on suit le titre, cela commence à devenir inquiétant. Mais c'est pourtant bien ce qui a lieu.
Prenons par exemple :

```
1  a = 10**30 # type int
2  b = -a # type int
3  c = 1.0 # type float
```

Alors :

- si on effectue `s = (a+b)+c` : l'addition des deux entiers est faite (de manière exacte), renvoie l'entier `0`, qui est ensuite additionné au flottant `c` pour donner le flottant `1.0`
- si on effectue `s = a+(b+c)` : l'addition de l'entier `-10**30` avec le flottant `1.0` donne l'approximation flottante de `10**30` (la mantisse ne permettant pas d'absorber le trop faible choc que représente `+1.0`).
  Puis, l'addition de l'entier `a` avec l'approximation flottante de `-10**30` donne le flottant `0.0`.

Ainsi, dans un cas, on a `(a+b)+c` qui renvoie `1.0` et, dans l'autre, `a+(b+c)` qui renvoie `0.0`. Ceci provient du fait qu'en effectuant l'addition du l'entier `10**30` avec le flottant `1.0`, Python convertit l'entier `10**30` en flottant, puis *aligne les deux représentations en virgule normalisée sur la même valeur d'exposant décalé*. Ainsi, l'opération `+1.0` devient totalement invisible pour Python. On appelle cela un **phénomène d'absorption**.
En revanche, avec le code suivant :

```
1  a = 10**30 # type int
2  b = -a # type int
3  c = 1 # type int
```

Les calculs sont exacts, car le type `int` est géré avec une précision infinie.

III.2. Multiplier / diviser des flottants

Les mêmes problèmes sont à constater pour des multiplications et des divisions. On le constate sur l'exemple ci-dessous, consacré au calcul des solutions d'une équation du second degré.

III.2.a) Accumulation d'erreurs

Considérons l'équation $x^2 + \dfrac{1}{a^2}x - 1 = 0$ d'inconnue $x \in \mathbb{R}$. Le réel $a \in \mathbb{R}_+^*$ joue le rôle de paramètre. Cette équation possède un discriminant strictement positif et deux racines :

$$x_1 = \frac{-1+\sqrt{1+4a^4}}{2a^2} \quad \text{et} \quad x_1 = \frac{-1-\sqrt{1+4a^4}}{2a^2}$$

Mathématiquement, le produit x_1x_2 des racines doit valoir -1. On programme le calcul des racines et on vérifie cela.

```
def racinesNaif(a,b,c) :
    Delta = b**2 - 4*a*c
    delta = Delta**(1/2)
    x1 = (-b - delta) / (2*a)
    x2 = (-b + delta) / (2*a)
    return x1,x2

N = 7
listeA = []
for k in range(N) :
    listeA.append(0.7*10**(-k))

listeX1 = []
listeX2 = []
listeC = []
for a in listeA :
    x1,x2 = racinesNaif(1,1/a**2,-1)
    listeX1.append(x1)
    listeX2.append(x2)
    listeC.append(x1*x2)
print(listeC)
```

Le tableau de valeurs pour $c = x_1x_2$, donné par la liste `listeC` dans le code ci-dessous, ne doit théoriquement contenir que des 1. Or, ce n'est pas le cas :

a	x_1	x_2	$x_1 \times x_2$
0.8	-2.05	0.48	-1.0
0.08	-156.26	6.40×10^{-3}	-1.0
0.008	-1.56×10^{4}	6.40×10^{-5}	-1.0
0.0008	-1.56×10^{6}	6.40×10^{-7}	-1.0
0.00008	-1.56×10^{8}	1.49×10^{-8}	-2.33
0.000008	$-1.56 \times 10^{1}0$	0.0	-0.0
0.0000008	$-1.56 \times 10^{1}2$	0.0	-0.0

À noter que les « -1.0 » des quatre premières lignes font apparaître des décimales non nulles à des rangs de moins en moins éloignés (10^{-16} puis 10^{-12}, puis 10^{-8} et enfin 10^{-5}).

III.2.b) Effets d'annulation (*cancellation effects*)

Le calcul de x_2 dans le dernier exemple ci-dessus fait apparaître des 0 (alors que 0 n'est pas racine de l'équation). Ceci provient de ce qu'on effectue une somme entre deux « grands nombres » dont les mantisses s'annulent. On appelle cela les **effets d'annulation** (*cancellation effects*, en anglais) : les *bits de poids fort* se neutralisent et ne reste que 0.
Une manière de remédier à cela est d'orienter les calculs afin que les « grands » nombres ne fassent que s'additionner, en jouant sur le fait que les effets d'annulation ne concernent pas les produits ou les quotients.
Par exemple, on peut reprogrammer la fonction `racines_naif` en se concentrant sur la racine où le calcul de $\dfrac{-b \pm \delta}{2a}$ fait intervenir deux quantités de même signe au numérateur.

```
def racinesMalin(a,b,c) :
    Delta = b**2 - 4*a*c
    delta = Delta**(1/2)
    if b > 0 :
        x1 = (-b - delta) / (2*a)
        x2 = (a*c) / x1
    else :
        x2 = (-b + delta) / (2*a)
        x1 = (a*c) / x2
    return x1,x2
```

Les résultats du calcul sont alors les suivants :

a	x_1	x_2	$x_1 \times x_2$
0.8	-2.05	0.48	-1.0
0.08	-156.26	6.40×10^{-3}	-1.0
0.008	-1.56×10^{4}	6.40×10^{-5}	-1.0
0.0008	-1.56×10^{6}	6.40×10^{-7}	-1.0
0.00008	-1.56×10^{8}	6.40×10^{-9}	-1.0
0.000008	$-1.56 \times 10^{1}0$	6.40×10^{-11}	-1.0
0.0000008	$-1.56 \times 10^{1}2$	6.40×10^{-13}	-1.0

Les valeurs affichées pour x_2 sont plus cohérentes, de même que pour le produit $x_1 \times x_2$.