



Introduction à l'arithmétique flottante

25 décembre 2018

Table des matières

1. Introduction	3
2. Représentation des réels en machine	5
2.1. Exigences pour une représentation des réels en machine	5
2.1.1. Amplitude	5
2.1.2. Précision	6
2.1.3. Performance	6
2.2. Représentation des réels par les nombres à virgule flottante	6
2.2.1. Rappel sur la notation scientifique	6
2.2.2. Nombres à virgule flottante	7
2.2.3. Satisfaction des exigences de représentation des réels	8
2.3. Le format de nombres à virgule flottante binary64	8
2.4. Autres formats standards de nombres à virgule flottante	10
2.4.1. Formats binaires	10
2.4.2. Format décimaux	11
3. Approximation des réels par les flottants	12
3.1. L'ensemble des flottants	12
3.1.1. Bornes de l'ensemble des flottants	12
3.1.2. Précision des flottants	13
3.1.3. Répartition des flottants parmi les réels	14
3.2. Opérations d'arrondi	16
3.2.1. Modes d'arrondis	16
3.2.2. Opérations d'arrondi	17
3.3. Erreurs d'arrondi	19
3.3.1. Définition de l'erreur d'arrondi	19
3.3.2. Erreur d'arrondi maximale pour un flottant donné	19
3.3.3. Valeur de l'erreur d'arrondi maximale en fonction du flottant	20
3.3.4. Exemples d'application	21
4. Calcul avec les flottants	22
4.1. Opérations élémentaires	22
4.1.1. Déroulement d'une opération élémentaire	22
4.1.2. Erreurs d'arrondi	23
4.1.3. Absorption	24
4.1.4. Annulation catastrophique	24
4.2. Enchaînements d'opérations	25
4.2.1. Combinaison de plusieurs calculs	25
4.2.2. Quelques propriétés de l'arithmétique flottante	26
4.3. Exemples de situations problématiques	28
4.3.1. Accumulation d'erreurs	28

Table des matières

4.3.2. Perte massive de chiffres significatifs	29
4.3.3. Convergence de la série harmonique	30
4.3.4. Récurrence de Müller et Kahan	30
4.4. Exemple d'algorithme intelligent : la sommation compensée de Kahan	32
5. Conclusion	34

1. Introduction

De nos jours, de nombreux calculs sont effectués à l'aide d'ordinateurs. Il devient même difficile de trouver des domaines qui y soient étrangers, tant le calcul numérique est omniprésent. On l'utilise ainsi pour les simulations physiques, dans les calculateurs des avions, des voitures, ou des machines industrielles ou encore pour l'informatique financière.

Nombreux sont les gens qui effectuent des calculs sur ordinateur sans attention particulière. Pourtant, même le plus simple des programmes peut avoir un comportement surprenant, comme le montre l'exemple suivant, saisi dans l'interpréteur Python¹.

```
1 >>> x = 0.1
2 >>> x + 0.2 == 0.3
3 False
```

Cette inégalité un peu inattendue est causée par les arrondis effectués par la machine, qui rendent le comportement difficilement prédictible. Ce comportement n'est d'ailleurs pas propre à Python ; un programme plus complexe, peu importe le langage, regorge de cas similaires !

Les erreurs d'arrondi sont inhérentes au calcul sur ordinateur, et donc inévitables. Ainsi, il est important de connaître leur origine pour les identifier et les maîtriser. Les carences à ce niveau ont parfois des conséquences graves. Par exemple, pendant la première guerre du Golfe, un missile intercepteur *Patriot* rate sa cible à cause d'une erreur d'arrondi dans le calcul de trajectoire, causant la mort de 28 soldats américains.²

Ce tutoriel est une introduction à l'arithmétique des nombres flottants. Vous y apprendrez comment les nombres à virgule flottante représentent les nombres réels en machine, en quoi ils en sont une approximation, et enfin quelles sont les conséquences en termes de calcul. Au terme de votre lecture, vous serez en mesure d'identifier les situations induisant des erreurs numériques importantes et connaîtrez quelques parades élémentaires. Par ailleurs, vous aurez aussi un bagage minimal pour aborder les méthodes plus complexes.



Notions abordées

Ce tutoriel aborde les notions suivantes :

- exigences d'une représentation des réels en machine (gamme, précision, performance),
- notion générale de représentation en virgule flottante (significande, exposant, signe),
- format IEEE-754 *binary64* (gamme, précision, flottants normalisés et dénormalisés,

1. Tous les exemples du cours sont en Python, mais facilement transposables à votre langage préféré.
2. [Rapport officiel du GAO sur l'incident](#) [↗](#) .

1. Introduction

i

- infinis),
- arrondi des réels vers des flottants (modes d'arrondis, erreurs d'arrondi, erreurs d'arrondi maximales),
- calcul avec des flottants (arrondis des opérandes, propagation d'arrondis, cas pathologiques, exemple d'algorithme non-naïf).

!

Prérequis

Pour être compris, ce tutoriel nécessite les prérequis suivants :

- la notation scientifique des nombres,
- la numération en base 2,
- des connaissances de base en programmation, peu importe le langage.

2. Représentation des réels en machine

Cette partie explique tout d'abord les principales exigences que doit remplir une représentation des réels. Elle introduit ensuite le type de représentation des réels le plus classique, dit à *virgule flottante*. Elle présente enfin le format le plus courant de nombres à virgule flottante, le format *binary64*.

2.1. Exigences pour une représentation des réels en machine

Il existe des solutions pour représenter les réels de manière exacte en machine. Cependant, les contraintes de performance restreignent l'usage de ces méthodes au calcul formel. Les autres usages utilisent ainsi systématiquement des représentations *approximatives* des réels, qui sont les seules à remplir les exigences principales pour le calcul numérique, à savoir l'amplitude, la précision, et la performance.

2.1.1. Amplitude

L'amplitude d'une représentation correspond aux *ordres de grandeur des nombres représentables*. Il est nécessaire de pouvoir représenter à la fois des nombres très grands et très petits pour que les scientifiques s'y retrouvent.

En physique, par exemple, les ordres de grandeurs s'étalent entre 10^{80} (volume de l'univers observable, en mètres cubes) et 10^{-45} (volume d'un proton, en mètres cubes).

En mathématique, il est facile de générer des nombres encore plus extrêmes. Vous connaissez peut-être l'extrait de poème suivant.

Je fais souvent ce rêve étrange et pénétrant
D'une femme inconnue, et que j'aime, et qui m'aime,

« *Mon rêve familier* », Paul Verlaine

La probabilité de taper du premier coup ce texte de 105 caractères, espaces compris, en frappant aléatoirement les touches d'une machine à écrire, est de 45^{-105} , soit environ 10^{-174} . Si l'on considère le nombre de textes de 105 caractères que ce procédé peut générer, on trouve l'ordre de grandeur inverse, c'est-à-dire 10^{174} .

2. Représentation des réels en machine

2.1.2. Précision

La précision d'une représentation est encore plus importante que son amplitude. La précision correspond au *nombre de chiffres significatifs qui peuvent être stockés*.

En physique, les constantes connues avec le plus de précision ne dépassent pas la vingtaine de chiffres significatifs. Par exemple, la [constante de Coulomb](#) ϵ_0 est connue avec 17 chiffres et la [charge élémentaire](#) e avec environ 10 chiffres.

Ces deux constantes se côtoient dans les calculs d'électrostatique, mais n'ont pas du tout le même ordre de grandeur (10^9 pour la constante de Coulomb et 10^{-19} pour la charge élémentaire). Il est donc important que la représentation des réels utilisée garde la *même précision quel que soit l'ordre de grandeur*.

2.1.3. Performance

La performance correspond à la rapidité avec laquelle les calculs peuvent s'effectuer. Il est en effet hors de question d'attendre l'éternité pour obtenir un résultat. Il s'agit aussi d'être performant sur toutes les machines couramment utilisées.

La plupart des ordinateurs partagent la même architecture globale, conçue autour d'un processeur. Ce composant effectue des calculs en manipulant des groupes de *bits* de taille fixe appelés *mots machine*. La taille de ces mots est la caractéristique principale d'un processeur. On entend ainsi souvent parler de processeurs 64 *bits* ou 32 *bits*. Pour être le plus rapide possible, il est intéressant d'avoir un format de nombre pour lequel le câblage des opérations du processeur est performant. Cela passe par des compromis, notamment le fait de favoriser un format sur *un seul mot machine*.

Il existe différentes familles de représentation des réels qui remplissent les critères d'amplitude, précision et performance, mais la plus utilisée est sans conteste la représentation à virgule flottante.

2.2. Représentation des réels par les nombres à virgule flottante

L'idée générale des nombres à virgule flottante est de stocker les chiffres significatifs d'une part et l'ordre de grandeur d'autre part. Ils s'opposent aux nombres à virgule fixe, dont l'idée générale consiste à stocker tous les chiffres du nombre.

2.2.1. Rappel sur la notation scientifique

2.2.1.1. En base 10

En base 10, la notation scientifique d'un réel prend la forme suivante :

$$s \times 10^e$$

2. Représentation des réels en machine

avec :

- e un entier relatif, appelé *exposant*,
- s un réel, tel que $1 \leq |s| < 10$ ou $s = 0$, appelé *significande*.

Le significande correspond aux chiffres significatifs (et au signe), tandis que l'exposant correspond à l'ordre de grandeur.

L'exposant et le significande sont suffisants pour définir un nombre en notation scientifique. Plus précisément, il suffit du signe du significande, de ses chiffres et de l'exposant. Il n'y a pas besoin de s'encombrer du chiffre 10 une fois que le choix d'utiliser la notation scientifique a été fait.

2.2.1.2. En base 2

En base 2, la notation scientifique d'un réel prend la forme suivante :

$$s \times 2^e$$

avec :

- e un entier relatif, l'*exposant*,
- s un réel, tel que $1 \leq |s| < 2$ ou $s = 0$, le *significande*.

Tout ça est évidemment très similaire à la notation en base 10. Le 10 se transforme en 2, et le significande se retrouve entre 1 et 2, au lieu d'être entre 1 et 10. Similairement, l'exposant, le significande et le signe suffisent à définir le nombre, sans besoin de s'encombrer du chiffre 2 une fois que le choix de la notation scientifique a été fait.

2.2.2. Nombres à virgule flottante

En informatique, un nombre à virgule flottante est tout simplement la donnée d'un significande et d'un exposant. Comme, il n'est pas trivial de stocker efficacement des chiffres en base 10, les représentations à virgule flottante généralement utilisées sont fondées sur la notation scientifique des nombres en base 2.

Stocker un nombre à virgule flottante est très simple, car il suffit de stocker les chiffres (binaires) du significande, son signe et l'exposant. Un seul *bit* suffit pour le signe, auquel on adjoint un certain nombre de *bits* pour stocker les chiffres du significande et l'exposant. Il n'y a pas besoin d'utiliser d'astuces ou de structures de données compliquées.



Vocabulaire

Les *nombres à virgule flottante* sont aussi appelés nombres flottants ou tout simplement **flottants**. C'est ce mot qui sera le plus souvent utilisé dans la suite du tutoriel.

2.2.3. Satisfaction des exigences de représentation des réels

Combien de *bits* sont nécessaires pour obtenir une amplitude et une précision satisfaisantes ? Pour le savoir, revenons un peu sur les ordres de grandeur.

Pour satisfaire l'amplitude, il faut stocker l'exposant, qui est un entier relatif. En calculant avec largesse, il suffit d'avoir quelques centaines d'entiers, pour e entre -1000 et 1000 (dans la notation scientifique en base 2), on atteint des ordres de grandeurs entre 10^{-300} et 10^{300} . Une petite dizaine de *bits* suffit donc, car 10 *bits* permettent déjà de stocker $2^{10} = 1024$ valeurs.

Pour satisfaire la précision, il faut stocker une petite quinzaine de chiffres en base 10. Cela correspond à une précision de l'ordre de 10^{-15} (15 décimales) sur le significande, soit de l'ordre de 2^{-50} (50 *bits* de partie fractionnaire). Il faut donc quelques dizaines de *bits* pour stocker le significande.

Par ailleurs, comme l'exposant et le significande sont stockés indépendamment, l'ordre de grandeur n'a aucune influence sur la précision du significande, ce qui est parfait pour les calculs mêlant divers ordres de grandeur.

En somme, il suffit de quelques dizaines de *bits* pour stocker tout le nécessaire, c'est-à-dire la taille des mots machine usuels. Par conséquent, on arrive à obtenir assez facilement une représentation performante des réels.

Le nombre et l'organisation exacte des *bits* formant un flottant sont définis dans ce qu'on appelle des *formats* de nombres à virgule flottante. Il existe différents formats standards, dont le plus utilisé est un format binaire sur 64 *bits*, qui est décrit plus en détails dans la prochaine section.

2.3. Le format de nombres à virgule flottante *binary64*

Au début de l'ère informatique, de nombreux formats de flottants coexistaient et étaient bien sûr incompatibles. Pour mettre fin au chaos, quelques formats standards ont été définis dans la norme IEEE 754. Le plus utilisé est le format binaire double précision, aussi appelé *binary64*.

Dans ce format, les flottants sont des nombres écrits en notation scientifique en base 2, sous la forme suivante :

$$\pm s \times 2^e$$

avec les éléments suivants :

- le signe, positif ou négatif;
- le significande s , un réel positif tel que $1 \leq s < 2$;
- l'exposant e , un entier relatif.

Cette forme recouvre celle présentée dans la section précédente, sauf que j'ai séparé le signe, et donc le significande est toujours positif. Dans un cas très particulier, le significande pourra être inférieur à 1, mais nous reviendront dessus plus tard.

L'ensemble de ces éléments sont stockés sur 64 *bits*, dont :

- un champ de 1 *bit* pour le signe,

2. Représentation des réels en machine

- un champ de 52 *bits* pour la partie fractionnaire de s , appelée *mantisse*, c'est-à-dire les chiffres après la virgule uniquement,
- un champ de 11 *bits* pour l'exposant.



FIGURE 2.1. – Représentation des flottants 64 bits (schéma par GMjeanmatt, CC-BY-SA 3.0)



Notation

Dans la suite de ce cours, les nombres en base 2 seront écrits entre crochets pour éviter la confusion avec la base 10. Par exemple, $[10010100]$ est une représentation binaire et 10010100 une représentation décimale.

Les notations littérales, telles que s et e ne sont pas concernées, puisque la base n'a aucune influence sur la valeur d'un nombre. Les nombres 0 et 1 ne sont pas non plus concernés, car identiques en base 10 et en base 2.

Le *bit* de signe vaut 0 pour le signe *plus* et 1 pour le signe *moins*.

Le champ de 52 *bits* stocke uniquement la mantisse, c'est-à-dire la partie fractionnaire du significande. Pourquoi seulement la partie fractionnaire ? Eh bien, parce que la partie entière du significande est constante et égale à 1. En effet, le significande étant entre 1 et 2 exclu, il est de la forme $[1, \dots]$. Comme le champ fait 52 *bits*, cela signifie qu'on stocke 52 chiffres binaires après la virgule, auquel s'ajoute le *bit* fantôme de la partie entière.

Le champ de 11 *bits* restant code un entier relatif compris entre -1023 et 1024^3 . Les valeurs entre -1022 et 1023 sont des valeurs d'exposant. Autrement dit, e est entre -1022 et 1023. Les deux valeurs restantes sont des valeurs spéciales, utilisées pour deux cas particuliers :

- Si le champ prend la valeur spéciale +1024, le flottant représente $\pm\infty$ pour une mantisse nulle et un *NaN* sinon. *NaN* est l'acronyme de *Not a Number*, c'est-à-dire qu'il s'agit d'une valeur indéterminée, comme le résultat de $0 \times \infty$.
- Si le champ prend la valeur spéciale -1023, alors on rentre dans le domaine des *nombres dénormalisés* (voir encadré ci-dessous). De tous les dénormalisés, le seul important est zéro. Il est représenté avec le signe positif ou négatif (il y a deux zéros!), une mantisse nulle, et bien sûr l'exposant -1023.



Que sont les nombres dénormalisés ?

Lorsque l'exposant prend la valeur spéciale -1023, alors le *bit* implicite de la partie entière vaut 0 (au lieu de 1 habituellement), et l'exposant vaut -1022. Autrement dit, les nombres dénormalisés ont la forme suivante :

$$\pm[0, a_1 \dots a_{52}] \times 2^{-1022}$$

3. La représentation biaisée des entiers relatifs est utilisée : les 11 *bits* codent un entier naturel entre 0 et 2047, auquel on soustrait 1023 pour obtenir le nombre relatif représenté.

?

où les a_n sont les *bits* de la mantisse qui prennent chacun la valeur zéro ou un.

La norme présente là une entorse astucieuse à la notation scientifique, qui permet de représenter des nombres beaucoup plus petits sans *bit* supplémentaire, mais avec une précision dégradée.

Si cela vous dépasse, reprenez simplement qu'il y a des nombres «bizarres» en dessous de 2^{-1022} , qui est le dernier nombre normal. Vous n'en rencontrerez quasiment jamais au cours de calculs usuels, si ce n'est 0.

Le tableau suivant contient quelques exemples de flottants.

Bit de signe	Signe	Bits d'exposant	Valeur	Bits de mantisse	Signifi-cande	Interpré-tation	Com-mentaire
0	+	000 0000 0000	-1023	0...0	[0,0]	+0	avec l'autre signe, on code -0
1	-	111 1111 1111	+1024	0...0	[0,0]	$-\infty$	avec l'autre signe, on code $+\infty$
0	+	111 1111 1110	+1023	10...0	[1,1]	$+1,5 \times 2^{1023}$	nombre normalisé
0	+	111 1111 1110	+1023	010...0	[1,01]	$+1,25 \times 2^{1023}$	nombre normalisé

2.4. Autres formats standards de nombres à virgule flottante

La norme IEEE 754 définit de nombreux autres formats à virgule flottante. Ils permettent d'avoir des flottants pour différentes tailles de mots machine (16, 32, 64, 128), voire carrément d'utiliser une représentation décimale, c'est-à-dire une véritable notation scientifique en base 10.

2.4.1. Formats binaires

Les formats binaires sont des formats où la mantisse est stockée sous forme binaire, c'est-à-dire la méthode classique. On peut remarquer que la mantisse prend une place toujours plus importante par rapport à l'exposant.

Nom	Nom commun	Bits de mantisse	Bits d'exposants
binary16	Demie précision	10	5
binary32	Simple précision	23	8
binary64	Double précision	52	11

2. Représentation des réels en machine

binary128	Quadruple précision	112	15
-----------	---------------------	-----	----

2.4.2. Format décimaux

Pour les formats décimaux, la mantisse est codée sous forme décimale binaire, c'est-à-dire que chaque chiffre décimal est représenté directement en binaire. Cette technique offre des avantages pour effectuer de manière exacte certains calculs (financiers par exemple), mais ce n'est pas l'objet de ce cours.

Nom	Décimales	Bits d'exposant ⁴
decimal32	7	7,58
decimal64	16	9,58
decimal128	34	13,58

Vous avez vu dans cette partie ce que sont les flottants et comment ils répondent aux impératifs d'amplitude, de précision et de performance pour le calcul scientifique.

Dans la partie suivante, vous verrez en quoi les flottants ne sont qu'une approximation des réels, ce qui vaut bien sûr aussi pour le format *binary64*.

4. Le nombre de *bits* de l'exposant est non entier, car le nombre de valeurs possibles n'est pas une puissance de deux. Par exemple, il y a 768 valeurs possibles pour le *decimal64*, soit $\log_2(768) = 9,58$ *bits*.

3. Approximation des réels par les flottants

Les flottants étant une approximation des réels, il n'est pas possible de tous les représenter exactement. En quoi l'ensemble des flottants se distingue de l'ensemble des réels ? Comment approxime-t-on un réel par un flottant ? Quelle erreur est faite à cause de cette approximation ?

3.1. L'ensemble des flottants

L'ensemble des réels présente quelques propriétés sympathiques : il est non borné, on trouve toujours un réel entre deux autres réels, et il contient un nombre infini d'éléments. Rien de tout cela n'est vrai pour les flottants *binary64* :

- il existe un nombre fini de mantisses différentes, car elles sont codées sur un nombre fini de *bits* ;
- il existe pour la même raison un nombre fini d'exposants, et donc un nombre fini de flottants ;
- un nombre fini d'exposant signifie un exposant maximal et minimal, c'est-à-dire des bornes à l'ensemble des flottants en les associant respectivement aux mantisses maximales et minimales.⁵

3.1.1. Bornes de l'ensemble des flottants

La représentation des flottants est parfaitement symétrique grâce au *bit* de signe. Les bornes dans les négatifs s'obtiennent donc en changeant le signe des bornes obtenues pour les positifs. Dans la suite, chaque nombre sera ainsi précédé du symbole \pm .

3.1.1.1. Plus grand flottant non infini (en valeur absolue)

Les plus grands flottants en valeur absolue sont les infinis (négatifs et positifs), qu'on représente avec la mantisse nulle et l'exposant à la valeur spéciale $+1024$. Cependant, les infinis ne sont pas des réels. Il vient alors la question suivante : quelle est le plus grand flottant non infini (en valeur absolue) ?

Pour obtenir la plus grande valeur non infinie, il faut prendre naturellement le plus grand exposant ($+1023$) et la plus grande mantisse possible pour cet exposant (c'est-à-dire une suite de 52 « un », en binaire). La partie entière, implicite, vaut 1. On obtient donc le flottant suivant, qui est la plus grande valeur représentable avec un flottant binaire double précision (*binary64*).

5. Les infinis ne sont pas des flottants à proprement parler.

3. Approximation des réels par les flottants

$$\pm\Omega = \pm[1, 11\dots1] \times 2^{1023} = \pm(2 - 2^{-52}) \times 2^{1023} \approx \pm 1,798 \times 10^{308}$$

3.1.1.2. Plus petit flottant non-nul (en valeur absolue)

Le plus petit flottant est évidemment zéro, qui est représenté par la mantisse nulle et un exposant à la valeur spéciale -1023. Quelle est la valeur directement supérieure, c'est-à-dire le plus petit flottant non-nul (en valeur absolue) ?

Nous laissons de côté les flottants dénormalisés, pour nous intéresser uniquement aux flottants normaux. Le plus petit flottant non-nul est alors celui dont l'exposant est le plus petit exposant (-1022), la partie entière implicite du significande vaut 1 et la mantisse est la plus petite mantisse normalisée (que des zéros).

$$\pm\alpha = \pm[1, 0\dots0] \times 2^{-1022} = \pm 2^{-1022} \approx \pm 2,225 \times 10^{-308}$$

i

Le véritable plus petit flottant non-nul

En vérité, le plus petit flottant est un flottant dénormalisé. On l'obtient pour un champ d'exposant à -1023, et une mantisse non nulle minimale, c'est-à-dire avec tous les *bits* à zéros sauf le dernier (51 zéros après la virgule, suivis d'un *un*).

$$\pm\varepsilon = \pm[0, 0\dots01] \times 2^{-1022} = \pm 2^{-52} \times 2^{-1022} \approx \pm 4,941 \times 10^{-324}$$

Il est rare de faire des calculs dans la zone des dénormalisés, entre α et ε , c'est pourquoi j'insiste peu sur leurs particularités dans ce cours.

3.1.2. Précision des flottants

La précision correspond au nombre de chiffres du significande qu'il est possible de stocker. Par exemple, un format dans lequel on ne peut stocker que les dixièmes a une précision plus faible qu'un format où l'on peut stocker les dixièmes et les centièmes.

Pour les flottants, il s'agit donc du nombre de *bits* disponibles pour stocker la mantisse plus le *bit* implicite. Par exemple, la précision du format *binary32* (23 *bits* de mantisse) est plus faible que celle du *binary64* (52 *bits* de mantisse).

Le fait que le format *binary64* stocke toujours le même nombre de *bits* pour la mantisse fait que la précision est constante quel que soit l'exposant.

?

Combien de décimales pour 52 *bits* ?

Depuis le début, non parlons de *bits*, mais c'est la base 10 dont on se sert tout le temps. Comme $2^{-52} \approx 2,2 \times 10^{-16}$, on peut stocker environ 15 à 16 décimales pour le significande. Ceci est très approximatif, puisque le passage du décimal vers le binaire force à faire des arrondis, et que certains nombres s'arrondissent mieux que d'autres. Nous en parlerons



dans la prochaine section.



Précision des flottants dénormalisés

Puisque les dénormalisés ont toujours une partie entière implicitement nulle, cela signifie que plus le nombre est petit, plus il y a de zéros en tête et moins il y a de *bits* codant véritablement la mantisse. Ainsi, les flottants dénormalisés ont une précision d'autant plus mauvaise qu'ils sont petits. Par exemple, le flottant dénormalisé suivant a seulement 51 *bits* de précision :

$$[0, 01\dots 1] \times 2^{-1022}$$

La précision est liée directement à la répartition des flottants parmi les réels.

3.1.3. Répartition des flottants parmi les réels

Si un flottant normalisé s'écrit :

$$x = s \times 2^e$$

alors les mantisses de ses deux plus proches voisins x^+ et x^- s'obtiennent en incrémentant ou décrémentant la mantisse de 2^{-52} :

$$s^\pm = s \pm 2^{-52}$$

Ceci est vrai pour toutes les mantisses, l'écart reste constant. Par contre, l'écart entre deux flottants consécutifs augmente avec l'exposant. En effet,

$$x^\pm = s^\pm \times 2^e = (s \pm 2^{-52}) \times 2^e = s \times 2^e \pm 2^{e-52} = x \pm 2^{e-52}$$

Cela signifie que l'écart absolu entre deux flottants consécutifs (avec le même exposant e) vaut 2^{e-52} .

Les flottants avec un grand exposant seront ainsi plus écartés les uns des autres, car le changement du dernier *bit* de la mantisse correspondra alors à un bond en avant de plusieurs milliers, millions, voire plus. Au contraire, pour les petits flottants, le changement du dernier chiffre après la virgule correspondra à un petit saut de quelques dixièmes, millièmes, *etc.* En somme, la densité des flottants diminue à mesure qu'on s'éloigne de zéro pour progresser vers l'infini.

Les deux figures ci-dessous montrent ce changement de densité, en montrant les flottants autour de 1 et de 2^{64} .

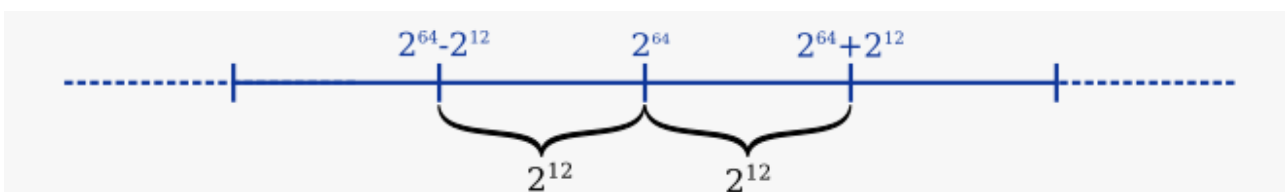


FIGURE 3.1. – Flottants autour de $2^{64} \approx 1,84 \times 10^{19}$.

3. Approximation des réels par les flottants

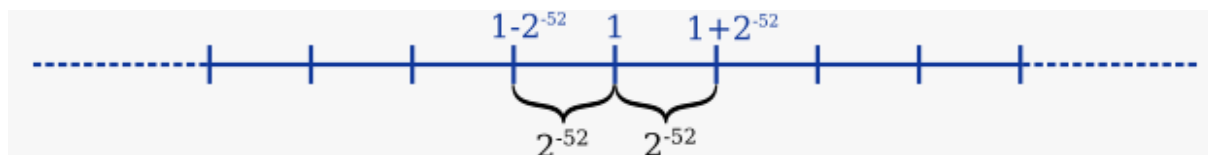


FIGURE 3.2. – Flottants autour de 1.

Pour une vision plus complète, voici les écarts entre flottants consécutifs sur un large intervalle de valeurs.

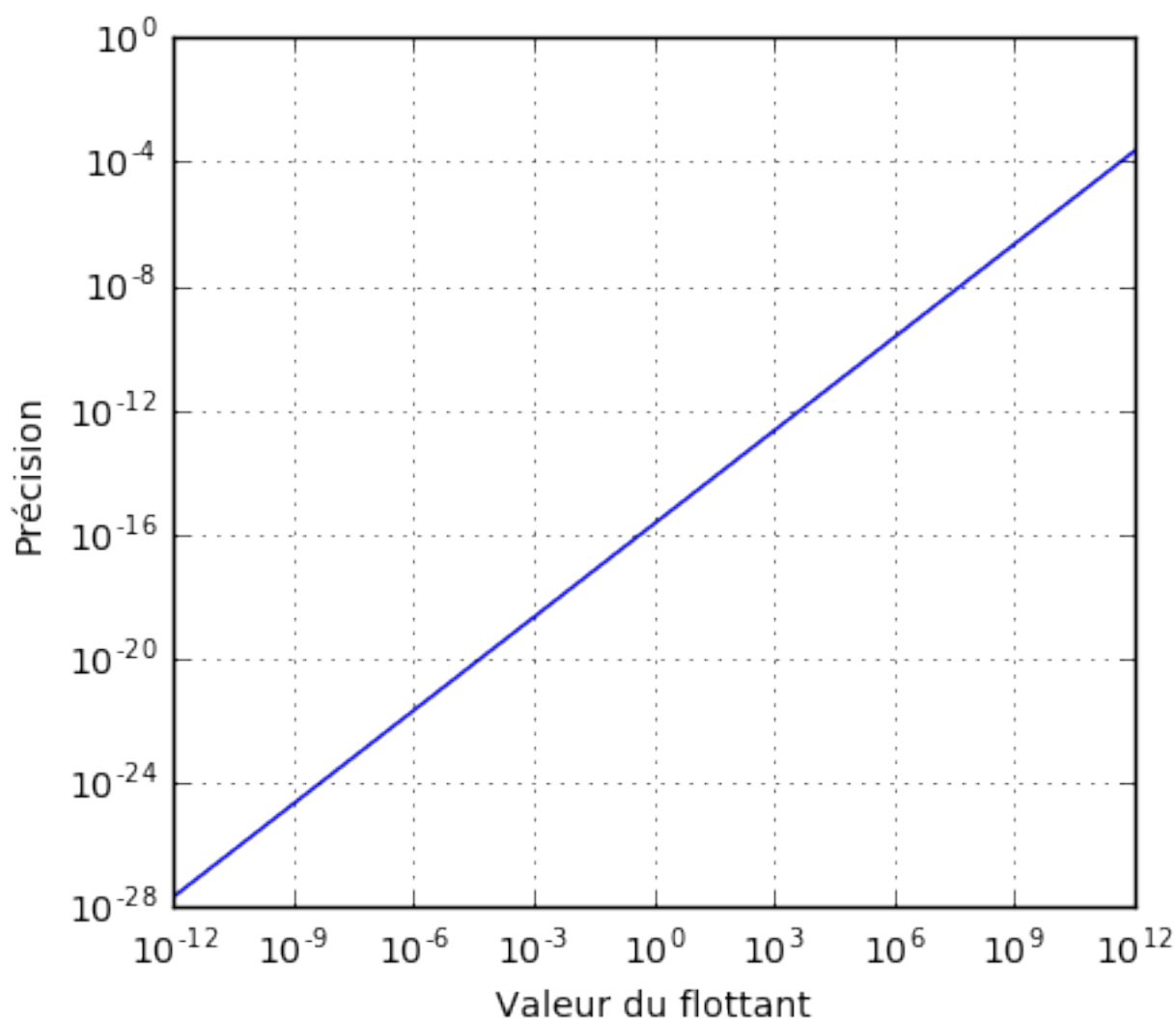


FIGURE 3.3. – Écart entre flottants binary64 consécutifs.

i

On peut noter que la différence entre deux flottants dénormalisés consécutifs est constante, étant donné qu'une seule valeur d'exposant est définie et que seule la mantisse varie.

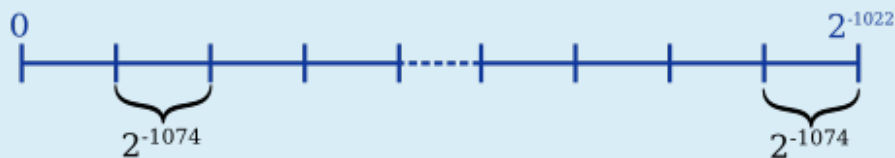


FIGURE 3.4. – Flottants entre 0 et ε

3.2. Opérations d'arrondi

Comme les flottants sont codés sur un nombre fini de *bits*, il y en a un nombre fini. Pour faire correspondre chaque réel, qui existent en nombre infini, à un flottant, il est donc nécessaire de faire des arrondis.

3.2.1. Modes d'arrondis

3.2.1.1. Liste des modes d'arrondis

La norme IEEE 754 définit deux familles d'arrondis et différents modes d'arrondi pour les formats binaires, tels que *binary64* :

- les arrondis au plus proche :
 - au chiffre pair le plus proche,
 - au flottant le plus proche avec rupture des égalités au profit de la plus grande valeur absolue.
- les arrondis orientés :
 - vers zéro,
 - vers $+\infty$,
 - vers $-\infty$.

Chacun de ces modes présente différentes propriétés qui les rendent appropriés pour différents cas d'application.

3.2.1.2. Mode d'arrondi par défaut : l'arrondi au chiffre pair le plus proche

Bien qu'il soit possible de configurer les modes d'arrondi, le mode par défaut est utilisé dans la grande majorité des cas : il s'agit de l'arrondi au chiffre pair le plus proche. Ce mode présente le comportement suivant :

- Pour les réels plus grands que Ω en valeur absolue, on arrondit vers l'infini du bon signe (dépassement).
- Pour les réels plus petits que α en valeur absolue, la norme garantit le signalement d'un *souppassement*, mais rien de plus (voir plus loin).
- Pour le reste, on arrondit au plus proche, et en cas d'égalité, on arrondit vers le flottant avec le dernier *bit* de mantisse égal à zéro (arrondi au chiffre pair le plus proche).

3.2.2. Opérations d'arrondi

3.2.2.1. Arrondi des nombres très grands : dépassement

Tous les nombres supérieurs à Ω (en valeur absolue) seront arrondis vers un infini : on parle de dépassement, ou plus communément d'*overflow*. Par exemple, le nombre $-1,8 \times 10^{308}$ est hors-limite, il sera arrondi vers l'infini.

```
1 >>> -1.8e308 # inférieur à -1.798e308
2 -inf
```



Pour les besoins de ce tutoriel, j'ai configuré l'interpréteur Python de manière à afficher plus de décimales. Gardez cela en tête si vous souhaitez tester chez vous : vous n'obtiendrez pas forcément exactement le même affichage, même si le résultat numérique sera identique. Notamment, il est possible que Python vous cache des décimales si beaucoup des premières décimales sont nulles.

3.2.2.2. Arrondi des nombres très petits : soupassement

Quand un nombre est inférieur à α , on parle de soupassement, ou plus communément d'*underflow*. Le fonctionnement est un peu plus délicat que pour le dépassement. En effet, une certaine liberté est laissée au langage qui implémente la norme.

En général, lorsqu'un nombre est inférieur à α , il va être stocké sous forme dénormalisée, par arrondi au chiffre pair le plus proche (voir plus loin). Les nombres inférieurs à ε seront arrondis vers zéro. Par exemple, 10^{-324} est arrondi vers zéro.

```
1 >>> 1e-324 # Exactement zéro pour l'ordinateur
2 0.000000000000000000e+00
```



Finalement, la droite des réels est coupée au deux bouts, et tout ce qui dépasse est arrondi vers l'infini. Au milieu, même combat, le zéro s'étale sur les nombres trop petits qui l'entourent.

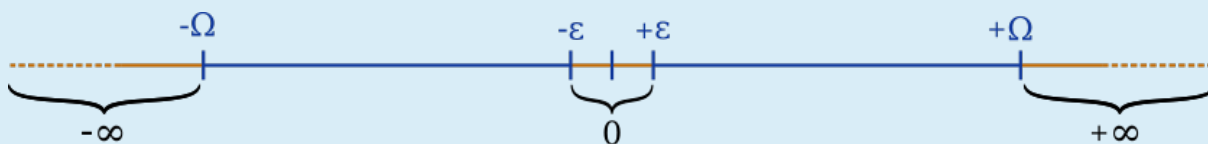


FIGURE 3.5. – Intervalles contenant des flottants sur la droite des réels (bleu) et arrondis vers zéro ou vers les infinis (orange).

3. Approximation des réels par les flottants

3.2.2.3. Arrondi usuel : arrondi au chiffre pair le plus proche

Lorsqu'un réel ne coïncide pas directement avec un flottant, mais qu'il n'est pas hors limite, il est arrondi au flottant le plus proche, c'est-à-dire qu'il sera arrondi vers le flottant le moins éloigné de lui sur la droite des réels. En cas d'égalité de distance, on prend le flottant avec le dernier *bit* de la mantisse nul (arrondi au chiffre pair le plus proche). En pratique, chaque flottant représente tout un intervalle de réels.

La figure ci-dessous illustre ce phénomène. Trois flottants consécutifs a , b et c sont représentés. Tous les réels entre $\frac{a+b}{2}$ et $\frac{b+c}{2}$ sont arrondis vers le même flottant b .

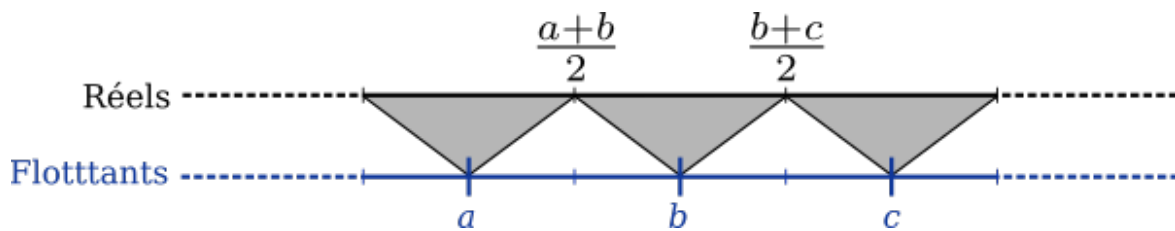


FIGURE 3.6. – Arrondi des réels au flottant le plus proche.

En programmation, un arrondi est fait lors de l'affectation d'une valeur littérale à une variable. La valeur littérale représente un réel, et elle doit être arrondie pour être stockée dans la variable sous forme de flottant. Par exemple, regardez le code ci-dessous.

```
1 >>> x = 0.9999999999999999
2 >>> y = 1.0000000000000001
3 >>> x == y
4 True
```

Les valeurs littérales 0,9999999999999999 et 1,0000000000000001 sont converties en flottants lors de l'affectation. Pour ces deux valeurs, le flottant le plus proche est 1, donc un arrondi est fait. On a en fin de compte $x = y = 1$.

Autre exemple plus gênant, certains nombres « simples » subissent des arrondis, visibles lorsqu'on affiche suffisamment de décimales. Par exemple, le réel 0,1 n'est pas représentable de manière exacte (aucun flottant n'est égal à 0,1) et est arrondi.

```
1 >>> 0.1
2 1.0000000000000001e-01
```

Quelle est l'erreur d'arrondi faite dans des cas comme ça ? La réponse se trouve dans la partie suivante.

3.3. Erreurs d'arrondi

Dans la section précédente, nous avons vu que les réels étaient arrondis par défaut au flottant le plus proche. Au cours de ce processus, une erreur d'arrondi est faite.

3.3.1. Définition de l'erreur d'arrondi

L'erreur d'arrondi est l'écart entre le réel de départ et le flottant vers lequel il est converti lors de l'opération d'arrondi. Si on note r le réel de départ, f le flottant résultant de l'arrondi et δ l'écart entre les deux, cette définition se traduit par la relation suivante :

$$r = f + \delta$$

Le schéma ci-dessous met en image la relation précédente.

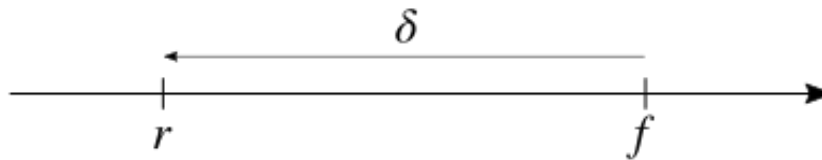


FIGURE 3.7. – Erreur d'arrondi

L'erreur d'arrondi exacte dépend du réel subissant l'arrondi, mais il est en général possible d'en donner un majorant.

3.3.2. Erreur d'arrondi maximale pour un flottant donné

Sur l'intervalle compris entre le plus grand flottant positif $+\Omega$ et le plus grand flottant négatif $-\Omega$, il est possible d'estimer l'erreur d'arrondi maximale, qui est fonction du réel arrondi. En dehors de cet intervalle, l'erreur peut être arbitrairement grande, puisqu'on a affaire au phénomène de dépassement.

Chaque flottant de l'intervalle $]-\Omega; +\Omega[$ est encadré par un prédécesseur direct et un successeur direct. Pour un flottant f , si on note f^- son prédécesseur direct et f^+ son successeur direct, on a la relation suivante :

$$f^- < f < f^+$$

Dans le mode d'arrondi au plus proche, les réels compris entre $\frac{f^-+f}{2}$ et $\frac{f+f^+}{2}$ seront arrondis vers f .

3. Approximation des réels par les flottants

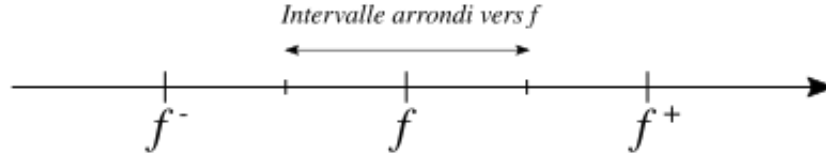


FIGURE 3.8. – Intervalle arrondi vers f .

On peut ainsi estimer l'erreur d'arrondi maximale. Si un réel r est arrondi vers un flottant f , alors l'erreur d'arrondi est au plus en valeur absolue :

$$|\delta_{max}| = \left| \frac{f^+ - f}{2} \right| = \left| \frac{f - f^-}{2} \right|$$

Il ne reste plus qu'à expliciter f^+ et f^- en fonction de f pour calculer une valeur numérique de cette erreur.

3.3.3. Valeur de l'erreur d'arrondi maximale en fonction du flottant

Prenons un flottant f de la forme :

$$f = s \times 2^e$$

Le successeur direct de f , f^+ , est obtenu en incrémentant le dernier *bit* du significande de f . Son significande s^+ est ainsi égal à :

$$s^+ = s + 2^{-52}$$

Le prédécesseur direct de f , f^- , est obtenu de manière similaire, mais en décrémentant le dernier *bit* du significande de f . Son significande s^- vaut alors :

$$s^- = s - 2^{-52}$$

On peut alors utiliser la formule du paragraphe précédent pour calculer l'erreur d'arrondi maximale :

$$|\delta_{max}| = \frac{s - s^-}{2} \times 2^e = \frac{s^+ - s}{2} \times 2^e = 2^{-53} \times 2^e$$

On a donc une erreur d'arrondi d'au pire 2^{-53} sur le significande. L'erreur sur le flottant dans son ensemble dépend de l'exposant e et vaut au plus 2^{e-53} .

3.3.4. Exemples d'application

3.3.4.1. Estimer l'erreur d'arrondi pour un réel donné

Si l'on a un nombre, qui n'est potentiellement pas représentable de manière exacte, quelle est l'erreur maximale d'arrondi qui sera faite ? Prenons pour exemple le réel 2,2, qui subira un arrondi lors de sa conversion en flottant.

La première étape pour estimer l'erreur d'arrondi maximale consiste à trouver l'exposant. Pour cela, il suffit d'encadrer le nombre entre deux flottants d'exposants consécutifs :

$$2^1 \leq 2,2 < 2^2$$

L'exposant qu'aura 2,2 une fois converti en flottant est donc 1. Il suffit maintenant d'appliquer la formule vue précédemment pour obtenir l'erreur maximale :

$$|\delta_{max}| = 2^{e-53} = 2^{1-53} = 2^{-52} \approx 2,22 \times 10^{-16}$$

Il est également possible de calculer la véritable erreur d'arrondi, par exemple à l'aide d'un [calculateur en précision arbitraire](#) [↗](#). On obtient approximativement la valeur suivante :

$$|\delta| = 1,78 \times 10^{-16} < \delta_{max}$$

L'erreur exacte est donc bien inférieure à l'erreur maximale. Tout va bien !

3.3.4.2. Encadrer un résultat

À l'issue d'une opération quelconque entre deux flottants, on obtient comme résultat un certain flottant f , issu de l'arrondi du résultat exact. Dans quel intervalle se trouvait nécessairement le résultat exact, avant d'être arrondi ?

Supposons qu'on ait obtenu pour résultat la valeur arrondie 0,375. Pour trouver l'intervalle où se trouvait la valeur exacte, il faut d'abord convertir ce nombre en notation scientifique. On obtient $1,5 \times 2^{-2}$.

Comme l'exposant est désormais connu, il suffit d'appliquer la formule vue plus haut donnant la valeur maximale de l'erreur d'arrondi. On trouve $\delta_{max} = 2^{-2-53} = 2^{-55}$. Ainsi, le résultat exact se trouvait nécessairement entre $0,375 - 2^{-55}$ et $0,375 + 2^{-55}$.

Vous avez appris dans ce chapitre que les flottants approximent les réels, avec en particulier des bornes et une précision limitée, ce qui conduit à des arrondis.

4. Calcul avec les flottants

À première vue, le calcul avec les flottants est très similaire à celui avec les réels. Cependant, de nombreuses différences apparaissent en regardant de plus près, causées notamment par la nécessité de faire des arrondis.

Dans cette partie, nous verrons tout d'abord ce qu'il se passe pour les opérations élémentaires, effectuées individuellement, avant de voir les phénomènes apparaissant quand de multiples opérations s'enchaînent. Ensuite, nous étudierons quelques cas problématiques, et nous présenterons enfin un exemple d'algorithme de sommation astucieux.

4.1. Opérations élémentaires

La plupart des réels ne sont pas représentables exactement par un flottant, et subissent donc un arrondi lors de leur conversion. La nécessité d'arrondir rend les opérations, même les plus simples comme l'addition, non triviales. La manière précise de les effectuer est définie dans la norme IEEE 754.

4.1.1. Déroulement d'une opération élémentaire

Pour de nombreuses opérations élémentaires (listées plus loin), la norme dicte comment il faut les effectuer. Dans le cas où ces opérations sont effectuées *individuellement*, la procédure est la suivante :

- convertir les opérandes en flottant, comme expliqué dans la partie précédente ;
- effectuer l'opération sur les opérandes *arrondies* ;
- convertir le résultat en flottant, comme si le calcul avait été effectué de manière *exacte* ;
- stocker le résultat (affectation à une variable par exemple).

Ainsi, lorsqu'une opération simple est effectuée, on n'obtient pas simplement le résultat, mais l'arrondi du résultat exact de l'opération effectuée sur les opérandes arrondis. Il y a donc déjà trois opérations d'arrondi pour la moindre addition !

Les opérations élémentaires devant nécessairement arrondir correctement sont les suivantes : addition, soustraction, multiplication, division, *multiply-accumulate*⁶ et racine carrée.

La norme recommande aussi aux langages de proposer de nombreuses autres fonctions arrondissant correctement, telles que l'exponentielle, le logarithme, la puissance, la racine n-ième, les fonctions trigonométriques, hyperboliques et leurs réciproques.

6. Il s'agit du calcul direct, sans arrondi intermédiaire, de l'opération $a \times b + c$. C'est une opération très utilisée en traitement du signal, et qui est souvent optimisée dans les processeurs.

4. Calcul avec les flottants

Les comportements d'arrondis provoquent différents phénomènes indésirables lors des calculs avec les flottants.

4.1.2. Erreurs d'arrondi

Lors d'un calcul, le résultat n'est généralement pas l'arrondi du résultat effectué avec les opérandes réels, à cause des opérations d'arrondi qu'elles subissent. À cette erreur d'arrondi s'ajoute l'erreur d'arrondi sur le résultat.

Pour voir un effet concret de ces phénomènes, reprenons l'exemple de l'introduction, où nous avons vu que $0,1 + 0,2$ n'était pour l'ordinateur pas égal à $0,3$.

```
1 >>> a = 0.1
2 >>> a
3 1.00000000000000006e-01
4 >>> b = 0.2
5 >>> b
6 2.00000000000000011e-01
7 >>> c = 0.1 + 0.2
8 >>> c
9 3.00000000000000044e-01
10 >>> d = 0.3
11 >>> d
12 2.99999999999999989e-01
```

En regardant les résultats ci-dessus, on voit tout de suite pourquoi l'ordinateur annonçait **False** lors de la comparaison ! L'arrondi de $0,3$ n'est pas le même que l'arrondi de l'addition des arrondis de $0,1$ et $0,2$.

Il est possible de majorer l'erreur pour les opérations élémentaires. Nous allons le faire à titre d'exemple pour l'addition (qui est pareil que la soustraction). Prenons deux opérandes réelles a et b , qui subissent une erreur d'arrondi δ_a et δ_b respectivement. Après l'addition des opérandes arrondies, le résultat est arrondi au flottant le plus proche avec une erreur δ_c . Ainsi, la distance δ entre le résultat sur les réels c_r et le résultat sur les flottants c_f peut être majorée de la manière suivante :

$$|c_r - c_f| = |\delta| \leq |\delta_a| + |\delta_b| + |\delta_c|$$

En faisant l'application numérique avec les variables du code Python vu ci-avant, on trouve :

$$|0,3 - c| = |\delta| \leq |\delta_a| + |\delta_b| + |\delta_c| \leq |\delta_{a,max}| + |\delta_{b,max}| + |\delta_{c,max}| = 2^{-57} + 2^{-56} + 2^{-55} \approx 4,86 \times 10^{-17}$$

Ce qui est cohérent avec l'erreur de calcul effectivement obtenue, qui vaut environ $4,4 \times 10^{-17}$.

4. Calcul avec les flottants

4.1.3. Absorption

Les phénomènes d'absorption se produisent lorsque l'on additionne ou soustrait un nombre relativement petit à un nombre relativement grand. Dans ces conditions, un des opérandes absorbera l'autre : tout se passe comme si l'opération n'avait pas eu lieu !

On peut voir le phénomène à l'œuvre dans l'exemple suivant, où l'on additionne un nombre de l'ordre du milliard avec un nombre de l'ordre du milliardième.

```
1 >>> a = 2**30 # a est grand
2 >>> b = 2**-27 # b est petit relativement à a
3 >>> a + b == a # absorption
4 True
```

Dans cet exemple, les deux flottants à additionner n'ont pas les mêmes exposants. Dans ce cas-là, l'addition s'effectue comme suit : les deux nombres sont d'abord mis au même exposant (le plus grand), puis seulement ensuite leurs significandes sont additionnées. La mise au même exposant consiste à multiplier le nombre par la puissance de deux nécessaire pour avoir le même exposant, et diviser dans le même temps le significande par le même nombre. Mathématiquement, l'opération laisse le nombre inchangé, mais c'est sans compter sur la précision limitée des flottants. En effet, diviser un nombre par une puissance de deux revient à décaler ses *bits* vers la droite. Si on décale trop loin, des *bits* vont être perdus.

C'est exactement ce qu'il se passe dans l'exemple. Lors de la mise au même exposant, le significande de b va être divisé par 2^{57} , ce qui vide entièrement les 52 *bits* du significande, et il ne reste plus rien à additionner !

4.1.4. Annulation catastrophique

L'annulation catastrophique est le nom donné à la perte de précision qui se produit lors de la soustraction de deux flottants très proches. En effet, quand on soustrait deux flottants très proches, il est possible de perdre de nombreux chiffres utiles, alors même que le résultat théorique peut être stocké dans un flottant.

Regardons ce phénomène sur un exemple : la soustraction de $M = \sqrt{2}(1 + 10^{-14})$ et $m = \sqrt{2}$. Le résultat de ce calcul est évidemment $M - m = \sqrt{2} \times 10^{-14}$, mais la machine fait une erreur grossière :

```
1 >>> import math
2 >>> M = math.sqrt(2) * (1 + 10**-14)
3 >>> m = math.sqrt(2)
4 >>> M - m
5 1.42108547152020037e-14
```


La troisième décimale est déjà fautive puisque le résultat correctement arrondi est $1,414\,213\,562\,373\,095 \times 10^{-14}$, qu'il est tout fait possible de stocker dans un flottant.

4. Calcul avec les flottants

Les flottants utilisés par Python ici sont binaires, mais le phénomène est similaire en décimal et plus facile à comprendre. Si l'on stockait exactement 16 chiffres décimaux pour faire ce calcul, voici ce qu'il se passerait :

$$M - m = 1,414\,213\,562\,373\,109 - 1,414\,213\,562\,373\,095 = 0,000\,000\,000\,000\,014 = 1,40 \times 10^{-14}$$

La troisième décimale est fausse ici aussi. On voit dans ce calcul que la plupart des chiffres sont identiques et s'annulent en masse. En fin de compte, ils n'apportent rien au calcul. On se retrouve donc avec seulement quelques chiffres utiles et très peu de précision ! C'est là l'essence du phénomène d'annulation catastrophique.

Pour un autre exemple moins artificiel et plus développé, je vous invite à lire [cet article](#) , qui traite d'un phénomène d'annulation catastrophique dans le calcul des racines de trinômes du second degré.

Les phénomènes d'arrondi, d'absorption et d'annulation catastrophique sont inhérents aux opérations avec les flottants et ne peuvent être évités que par des calculs astucieux, quand cela est possible, ou en utilisant des nombres ayant une plus grande précision, ce qui se fait souvent au détriment de la performance.

4.2. Enchaînements d'opérations

4.2.1. Combinaison de plusieurs calculs

La norme définit précisément le comportement pour une unique opération, mais laisse une certaine liberté aux langages quant au comportement pour une combinaison d'opérations. Les règles exactes diffèrent donc d'un langage à l'autre, mais les règles générales sont tout de même conservées. On traite d'abord :

- les calculs entre parenthèses,
- puis les multiplications et divisions,
- enfin les additions et soustractions.

Quand plusieurs calculs de priorités identiques s'enchaînent, ils sont généralement effectués dans l'ordre, de gauche à droite. En somme, les priorités sont tout à fait identiques à celles avec les réels.

Le point le plus subtil concerne les arrondis intermédiaires. En effet, certains langages ou processeurs sont capables d'enchaîner les calculs en utilisant une précision étendue (80 *bits* au lieu de 64, par exemple). Dans ce cas-là, les résultats intermédiaires ne sont pas arrondis comme un résultat final, mais avec plus de décimales. Les calculs s'enchaînent ensuite avec ces opérandes en précision étendue, avant un arrondi final lors de l'affectation. Un calcul dont les résultats intermédiaires seraient affectés à des variables peut alors ne pas donner le même résultat qu'un calcul effectué tout d'un coup !

En pratique, cela a assez peu d'influence, mais si vous souhaitez avoir des détails, référez-vous au manuel de votre langage préféré.



Des limites de la liberté

Cette liberté laissée au langage par la norme mène à des problèmes de reproductibilité des résultats. Par exemple, une même simulation effectuée sur deux machines différentes peut donner des résultats différents ; qui croire dans des cas comme ça ?

Pour les calculs dont la reproductibilité compte, par exemple pour valider le résultat d'une simulation, la norme donne carrément une méthode pour y parvenir ! Elle consiste principalement à éviter soigneusement toute une série d'opérations dont les résultats diffèrent d'une machine conforme à l'autre.

Les calculs avec les flottants ressemblent de loin à ceux avec les réels. Pourtant une grande partie des propriétés des réels sont absentes. En particulier, l'arithmétique flottante :

- présente des phénomènes d'absorption, où un opérande est « mangé » par un autre ;
- est non distributive, autrement dit, factoriser ou développer affecte généralement le résultat ;
- est non associative, c'est-à-dire que l'ordre des calculs importe.

4.2.2. Quelques propriétés de l'arithmétique flottante

4.2.2.1. Commutativité

Puisque quand on effectue une opération sur des flottants, le résultat est l'arrondi correct du résultat exact, alors l'arithmétique sur les flottants est commutative : l'ordre des termes dans une addition ou une multiplication ne change pas le résultat.

Que l'on effectue $a + b$ ou $b + a$, il se passera exactement la même chose, à savoir l'arrondi des opérandes, le calcul de l'addition et l'arrondi correct du résultat.

On retrouve là une propriété des réels, ce qui est plutôt agréable, car d'autres propriétés des réels ne sont pas applicables aux flottants.

4.2.2.2. Non associativité

Les opérations dans les réels sont associatives. Par exemple, effectuer les opérations $(a + b) + c$ ou $a + (b + c)$ donne un résultat identique quand on travaille avec des réels.

Dans les flottants, cela ne marche pas dans le cas général. En effet, l'ordre des opérations pour $(a + b) + c$ est le suivant :

- arrondi des opérandes a et b ,
- calcul de $a + b$ avec les opérandes arrondis,
- arrondi du résultat $a + b$,
- calcul de l'arrondi de c ,
- addition du résultat arrondi de $(a + b)$ avec c ,
- arrondi du résultat final.

4. Calcul avec les flottants

Les différentes opérations d'arrondi peuvent emmener le résultat dans une direction différente en fonction de l'ordre dans lequel les opérations sont effectuées, et on perd alors l'associativité dont disposent les réels.

Ce phénomène est assez simple à voir en cas de dépassement. Par exemple, si l'opération $a + b$ provoque un dépassement, c'est-à-dire un résultat infini, alors l'addition de c conservera cet infini. Pourtant, il existe des cas où l'addition de $b + c$ d'abord et ensuite de a ne provoque pas de dépassement. C'est le cas par exemple quand $a + b$ est très positif, mais que c est très négatif.

Voyons la non-associativité sur un exemple :

```
1 >>> a = 9e307
2 >>> b = 9e307
3 >>> c = -2e306
4 >>> (a + b) + c # dépassement
5 inf
6 >>> a + (b + c) # pas de dépassement
7 1.780000000000000023e+308
```

4.2.2.3. Non distributivité

Dans les réels, l'addition est distributive sur la multiplication. On peut écrire la chose suivante :

$$a \times (b + c) = a \times b + a \times c$$

Dans les flottants, cela n'est plus vrai, encore une fois à cause des erreurs d'arrondis. Il n'y a pas de raison que les arrondis soient toujours exacts avec les deux opérations, et on perd alors la distributivité.

Ce phénomène arrive par exemple en cas de dépassement de la somme $b + c$. Le calcul effectué en développant évite alors le dépassement, pour peu que a soit relativement petit, et on obtient un autre résultat. Ce phénomène existe aussi avec des nombres normaux, notamment si dans une des deux formes, l'addition provoque une annulation catastrophique.

Voyons la non-distributivité sur un exemple :

```
1 >>> a = 1e-10
2 >>> b = 9e307
3 >>> c = 9e307
4 >>> a * (b + c) # on obtient l'infini
5 inf
6 >>> a * b + a * c # on obtient un autre résultat et on évite
  l'infini
7 1.800000000000000021e+298
```

4.3. Exemples de situations problématiques

4.3.1. Accumulation d'erreurs

Lorsqu'on effectue des additions, ou des multiplications, les unes à la suite des autres, les erreurs s'accumulent, de sorte qu'après un nombre important d'opérations, la précision obtenue n'est pas au niveau attendu d'un calcul avec des flottants, c'est-à-dire de l'ordre de 10^{-15} .

Prenons l'exemple, assez artificiel suivant : sommer n fois la valeur $\frac{1}{n}$ dans le but d'obtenir le résultat $\frac{n}{n} = 1$. En prenant n de plus en plus grand, et donc $\frac{1}{n}$ de plus en plus petit, on fera des calculs de plus en plus longs pour aboutir au même résultat.

Le programme ci-dessous montre comment on effectue le calcul pour $n = 10$.

```
1 >>> v = 0.
2     for i in range(10):
3         v = v + .1
4     1-v
5 1.1102230246251565404236316680908203125e-16
```

En effectuant ce calcul pour différentes valeurs de n , on obtient le résultat du tableau ci-dessous.

Additions	Incrément	Écart à 1
10^1	10^{-1}	1.1×10^{-16}
10^2	10^{-2}	-6.7×10^{-16}
10^3	10^{-3}	-6.7×10^{-16}
10^4	10^{-4}	9.4×10^{-14}
10^5	10^{-5}	1.9×10^{-12}
10^6	10^{-6}	-7.9×10^{-12}
10^7	10^{-7}	2.5×10^{-10}

On voit que plus le nombre d'opérations est important, moins la précision obtenue est élevée. À partir de quelques dizaines de milliers d'opérations, on commence à perdre plusieurs chiffres significatifs, et à partir d'un million d'opérations, on commence à disposer que d'un tiers de la précision disponible !

Les exemples de la vie réelle ne sont pas forcément aussi désavantageux que celui-ci, mais ce phénomène arrive nécessairement et est difficile à contrôler dans les calculs longs, comme les simulations physiques utilisant un pas de temps très court. Il s'agit d'ailleurs d'un phénomène limitant la précision temporelle des méthodes de simulation les plus simples, comme la [méthode d'Euler](#) ↗ .

4.3.2. Perte massive de chiffres significatifs

Pour effectuer un calcul, il y a parfois plusieurs formes possibles, qui sont équivalentes lorsqu'on travaille avec les réels. Par exemple, pour évaluer un polynôme, il est possible de travailler à partir de la forme factorisée ou de la forme développée, sans changer le résultat.

Avec les flottants, le choix de la forme du calcul peut influencer de manière importante sur le résultat. En effet, en fonction des valeurs prises par les variables, la propagation des erreurs d'arrondi ne se fait pas de la même manière, donc il est possible de perdre grandement en précision avec les opérations dans un ordre désavantageux.

Prenons l'exemple du calcul de la fonction polynomiale suivante :

$$f(x) = (x - 2)^9 = \sum_{i=0}^9 \binom{9}{i} (-2)^{9-i} x^i$$

Deux formes s'offrent à nous : la forme factorisée $(x-2)^9$ et la forme développée $\sum_{i=0}^9 \binom{9}{i} (-2)^{9-i} x^i$.⁷ Voyons ce qu'il se passe quand on calcule la fonction pour des valeurs de x proches de 2 avec la figure suivante.

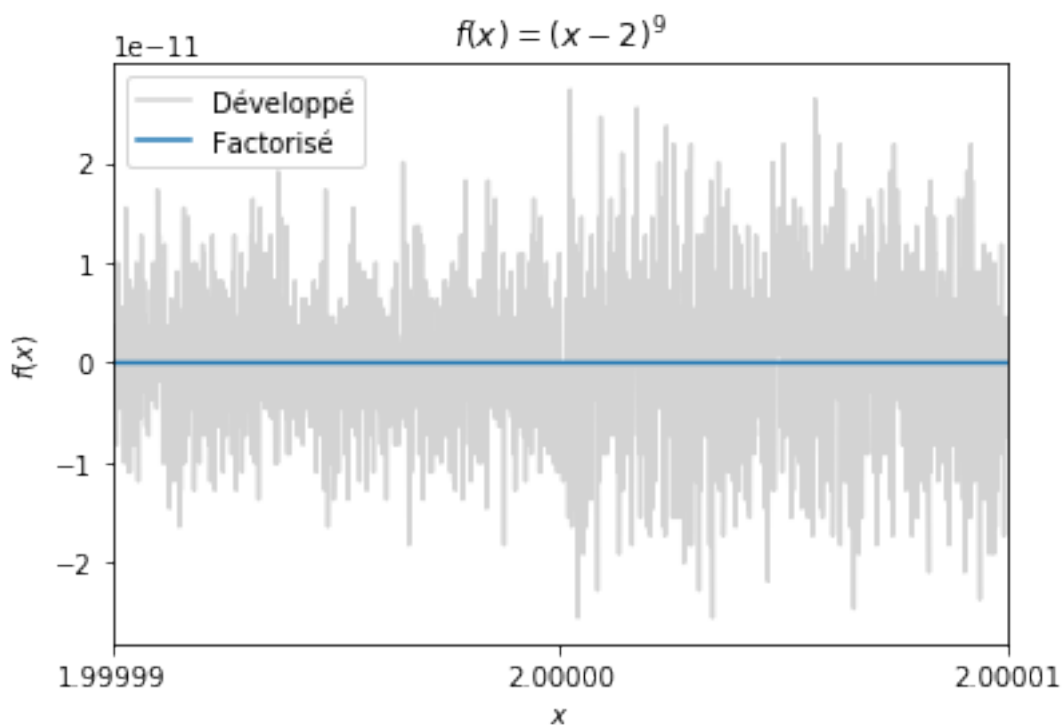


FIGURE 4.1. – Différences de précision entre le calcul développé ou factorisé.

On voit immédiatement que la forme factorisée est plus précise. En fait, la forme développée présente une alternance du signe des termes, ce qui fait qu'en pratique, on effectue des soustractions entre des termes d'ordres de grandeur similaires, et on a alors un phénomène d'annulation massive.

7. On passe de la forme factorisée à la forme développée à l'aide de la [formule du binôme de Newton](#) [↗](#).

4. Calcul avec les flottants

4.3.3. Convergence de la série harmonique

Dans le monde des flottants, on ne peut pas calculer la série harmonique n'importe comment. La série harmonique désigne la somme suivante :

$$\sum_{k=1}^n \frac{1}{k}$$

Dans le monde des réels, cette somme diverge quand n tend vers l'infini, et évidemment, pour n fixé, l'ordre de sommation n'a pas d'importance. Cependant, avec les nombres flottants, quelques précautions sont à prendre. Regardez le tableau ci-dessous, qui montre les différences obtenues en faisant croître n ou en le faisant décroître.

n	Croissant	Décroissant
10^5	12.090146129863 335	12.090146129863 408
10^7	16.69531136585 7272	16.69531136585 9965
10^8	18.99789641385 2555	18.99789641385 3447

Le problème n'est pas seulement lié à l'ordre de sommation, car dans les flottants, la série harmonique converge ! En effet, pour n suffisamment grand, $\frac{1}{n}$ est extrêmement petit, et un phénomène de souppassement provoque un arrondi vers zéro. Il y a ainsi un nombre fini de termes non-nuls et la série converge. Il faut cependant atteindre des nombres de l'ordre de $n = 10^{324}$ en double précision.

4.3.4. Récurrence de Müller et Kahan

Avec les flottants, un calcul de suite tout simple peut présenter quelques surprises. Voici une petite récurrence proposée par Kahan et Muller, écrite ici en Python :

```
1 x0 = 11./2.
2 x1 = 61./11.
3
4 for i in range(100):
5     x2 = 111. - (1130. - 3000./x0) / x1
6     x0 = x1
7     x1 = x2
8     print(x2)
```

Cet algorithme est assez amusant. En effet, en calculant la limite mathématiquement, avec des réels, on converge vers 6. Cependant, lors de l'exécution de ce programme avec des *doubles*, on converge vers 100.

Ce qu'il se passe, c'est qu'en calculant $11/2$ et $61/11$, une petite erreur d'arrondi se glisse dans le résultat. Là où cette récurrence est subtile, c'est que cette erreur est commune à tous les

4. Calcul avec les flottants

types flottants binaires, et donc qu'augmenter la précision ne la résout pas. Une fois cette erreur faite, le changement des valeurs initiales suffit à transformer la suite, qui converge vers une autre valeur que celle attendue (à savoir 100 au lieu de 6).

Le tableau ci-dessous montre la différence entre le calcul correct de la suite et celui effectué par le programme vu ci-avant.

Calcul avec des flottants	Calcul avec des réels
5.5901...	5.5901...
5.6334...	5.6334...
5.6746...	5.6746...
5.7133...	5.7133...
5.7491...	5.7491...
5.7818...	5.7818...
5.8113...	5.8113...
5.8376...	5.8376...
5.8610...	5.8609...
5.8835...	5.8813...
5.9359...	5.8991...
6.5344...	5.9145...
15.4130...	5.9277...
67.4723...	5.9390...
97.1371...	5.9486...
99.8246...	5.9568...
99.9895...	5.9637...
99.9993...	5.9696...
99.9999...	5.9745...
99.9999...	5.9787...
99.9999...	5.9822...
99.9999...	5.9851...
99.9999...	5.9875...
99.9999...	5.9896...

4.4. Exemple d'algorithme intelligent : la sommation compensée de Kahan

L'exemple de la série harmonique montre que pour les flottants, l'ordre de la sommation compte, parce qu'on risque de perdre des petits morceaux en cours de route par absorption ou par annulation massive. La sommation compensée de Kahan est un algorithme qui résout astucieusement ce problème.

L'idée générale de l'algorithme est de ramasser les miettes (les *bits* de poids faible) qui seraient absorbées par la somme et de les ajouter aux éléments (plus petits) avant d'ajouter le tout à la somme courante (plus grande), ce qui limite l'absorption.

Le morceau de code ci-dessous implémente cet algorithme.

```
1 def sum_kahan(list):
2     s = 0 # accumulateur pour la somme
3     c = 0 # compensation pour les bits de poids
         faible
4     for e in list:
5         y = e - c # l'élément à sommer corrigé par la
                 compensation des bits de poids faible
6         t = s + y # on perd des bits de poids faible avec
                 cette addition
7         c = (t - s) - y # cœur de l'algo : récupération des bits
                 de poids faible (cas extrême : si y est entièrement
                 absorbé, il est inclus dans c pour être ajouté à
                 l'élément suivant de la liste)
8         s = t # on passe à l'étape suivante
9         print("s =", s, "| c =", c) # affichage pour la
                 démonstration
10    return s
```

L'intérêt de cet algorithme est essentiellement de sommer de très longues listes sans perdre trop de précision, mais on peut le voir fonctionner sur un exemple. On y somme une liste contenant un nombre relativement grand et des miettes.

```
1 >>> eps = 10**-17
2 >>> l = [1, eps, eps, eps, eps, eps, eps, eps, eps, eps, eps, eps, eps,
         eps, eps, eps, eps]
3 >>> sum_kahan(l)
4 1.00000000000000002
```

Le résultat 1.00000000000000002 est ce qu'on obtiendrait en arrondissant le résultat exact de la somme vers les flottants. On peut le comparer avec le résultat de l'algorithme naïf qui donne :

4. Calcul avec les flottants

```
1 >>> sum(l)
2 1.0
```

Ici, la somme n'a pas ramassé les miettes et on a seulement le premier élément de la liste.

En regardant l'affichage, on peut comprendre comment l'algorithme de Kahan opère pour arriver à faire mieux que l'algorithme naïf :

```
1 s = 1 | c = 0 # somme du premier élément, tout va bien
2 s = 1.0 | c = -1e-17 # l'élément est absorbé dans la somme, il est
   retenu dans la compensation
3 s = 1.0 | c = -2e-17 # idem
4 s = 1.0 | c = -3e-17 # idem
5 s = 1.0 | c = -4e-17 # idem
6 s = 1.0 | c = -5.0000000000000005e-17 # idem
7 s = 1.0 | c = -6e-17 # idem
8 s = 1.0 | c = -7e-17 # idem
9 s = 1.0 | c = -8e-17 # idem
10 s = 1.0 | c = -9.0000000000000001e-17 # idem
11 s = 1.0 | c = -1.0000000000000001e-16 # idem
12 s = 1.0 | c = -1.1000000000000001e-16 # idem
13 s = 1.0000000000000002 | c = 1.020446049250313e-16 # une partie de
   la compensation a pu être sommée ! La compensation accumulée
   diminue légèrement.
14 s = 1.0000000000000002 | c = 9.20446049250313e-17 # idem
15 s = 1.0000000000000002 | c = 8.20446049250313e-17 # idem
16 s = 1.0000000000000002 | c = 7.20446049250313e-17 # idem
17 s = 1.0000000000000002 | c = 6.20446049250313e-17 # idem
18 s = 1.0000000000000002 | c = 5.2044604925031294e-17 # idem
19 s = 1.0000000000000002 | c = 4.204460492503129e-17 # idem
```

5. Conclusion

Merci à @Taurre pour la validation et à @Vayel pour la relecture en bêta.

Illustration du tutoriel : artefacts numériques sur le détail d'un [diagramme de Voronoi](#) par Craig S. Kaplan, 2005 [↗](#) .