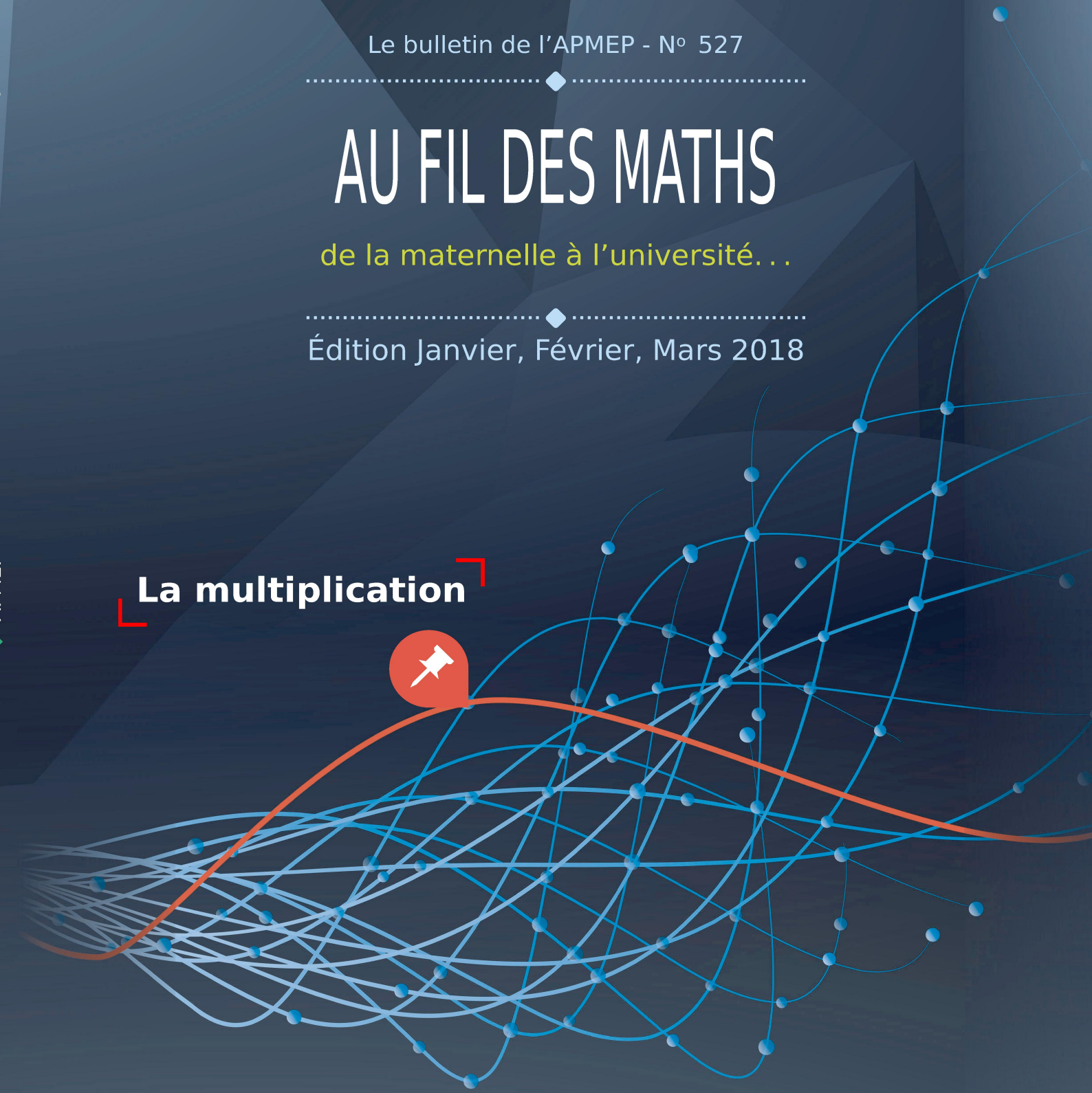
Le bulletin de l'APMEP - N° 527

AU FIL DES MATHS

de la maternelle à l'université. . .

Édition Janvier, Février, Mars 2018

La multiplication



APMEP

Association des Professeurs de Mathématiques de l'Enseignement Public

ASSOCIATION
DES PROFESSEURS DE MATHÉMATIQUES
DE L'ENSEIGNEMENT PUBLIC

26 rue Duméril, 75013 Paris
Tél. : 01 43 31 34 05 - Fax : 01 42 17 08 77
Courriel : secretariat-apmep@orange.fr - Site : https://www.apmep.fr
Présidente d'honneur : Christiane Zehren

Au fil des maths, c'est aussi une revue numérique augmentée :
https://afdm.apmep.fr
version réservée aux adhérents. Pour y accéder connectez-vous à votre compte *via* l'onglet *Au fil des maths* (page d'accueil du site) ou *via* le QRcode, ou suivez les logos.

Si vous désirez rejoindre l'équipe d'*Au fil des maths* ou bien proposer un article, écrivez à aufildesmaths@apmep.fr

Annonceurs : pour toute demande de publicité, contactez Mireille Génin mcgenin@wanadoo.fr

ÉQUIPE DE RÉDACTION

Votre adhésion à l'APMEP vous abonne automatiquement à *Au fil des maths*.
Pour les établissements, le prix de l'abonnement est de 60 € par an.
La revue peut être achetée au numéro au prix de 15 € sur la boutique en ligne de l'APMEP.

Mise en page : Olivier Reboux
Dépôt légal : Mars 2018
Impression : Imprimerie Horizon P.A. de la plaine de Jouques 200 avenue de Coulin
13420 GEMENOS
ISBN : en cours

AU FIL DES MATHS

Autour de la multiplication des flottants

François Boucher nous explique pourquoi le langage Python se met en faute sur des calculs aussi élémentaires que $3 \times 0{,}1$ (qu'il affiche différent de 0,3). Pour comprendre ce phénomène, il faut consentir à pénétrer dans les tréfonds de la machine. Au bout de ce voyage, les mathématiques sortent encore une fois victorieuses : ça s'explique...

François Boucher

L'interrogation commence (il y a quelques années) avec une surprise — classique mais réelle pour un débutant — provoquée par une réponse du langage Python : le booléen $3*0.1 == 0.3$ est évalué à faux mais $9*0.1 == 0.9$ est évalué à vrai. Les exemples étonnants du même acabit se multiplient alors au point de se demander comment il peut être possible de calculer avec Python dans ces conditions.

Toute personne qui écrit (ou utilise) des programmes de calculs numériques dans un langage quelconque est avertie des dangers de l'utilisation des décimaux informatiques, en particulier dans les tests. Mais la perception des causes reste, il faut bien l'avouer, floue et se réduit à attribuer les problèmes aux aléas du calcul sur machine, nécessairement approché.

Or le calcul numérique sur un système respectant certaines spécifications de représentations des nombres et des opérations est, au moins en théorie, déterministe[1] : les résultats des calculs flottants ne sont pas aléatoires et peuvent faire l'objet d'authentiques preuves formelles.

On suppose que le programme utilisé est le logiciel Python version 3.x qui semble avoir le même comportement sous les systèmes d'exploitation W et L ; le fait que, dans ce langage, les entiers ne sont pas limités peut induire des différences de comportement avec un logiciel dans lequel ce n'est pas le cas (java, C).

Quelques problèmes liés à la multiplication

Commençons par élargir, toujours avec Python (version 3.x) le problème posé en introduction ; on teste l'égalité $n \times 0{,}1 = n/10$ pour n entre 1 et 50[2] ; on obtient faux pour :

$$3,\ 6,\ 7,\ 12,\ 14,\ 17,\ 19,\ 23,\ 24,\ 28,$$
$$29,\ 33,\ 34,\ 38,\ 39,\ 41,\ 46,\ 48$$

et bien sûr vrai pour :

$$1,\ 2,\ 4,\ 5,\ 8,\ 9,\ 10,\ 11,\ 13,\ 15,\ 16,$$
$$18,\ 20,\ 21,\ 22,\ 25,\ 26,\ 27,\ 30,31,\ 32,$$
$$35,\ 36,\ 37,\ 40,\ 42,\ 43,\ 44,\ 45,\ 47,\ 49,\ 50$$

Je laisse le lecteur prolonger cette exploration, rechercher des régularités, des invariances dans ces deux suites[3] et formuler quelques conjectures.

Avec une calculatrice (TI, Casio, HP), et Maple ou Xcas mais aussi avec un tableur (LibreOffice), le phénomène ne se produit pas, ce qui s'explique par l'utilisation de la base dix[4] ; mais en Pascal (Turbo-Pascal 7), on obtient vrai pour tous les n entre 1 et 100.

Une autre surprise vient du test $10*(n/10) == n$ qui se révèle lui toujours vrai, mais pas $100*(n/100) == n$ et plus généralement $p*(n/p) == n$, en particulier pour $n = 1$. Le lecteur multipliera de lui-même les exemples.

1. Cette affirmation doit toutefois être tempérée ; voir l'article de David Monniaux [1].
2. Il y a une question naturelle avec l'utilisation du $n/10$ pour simuler le décalage de la virgule décimale ; voir l'annexe 2.
3. Non connues de l'encyclopédie en ligne des suites d'entiers [2].
4. L'utilisation du module Python Decimal permet de vérifier que le calcul en base dix résout les problèmes.

Nous allons essayer de comprendre (un peu) ce qui se passe, sans être en mesure de résoudre en toute généralité les problèmes posés ; par exemple, caractériser les entiers n pour lesquels $n * 0.1 \neq n/10$ semble une question ardue qui n'a sans doute pas de réponse agréable…

Pour cela, il est d'abord nécessaire d'entrer dans les détails de ce qu'on appelle l'« arithmétique flottante ». L'essentiel du contenu qui suit est tiré des publications de Jean-Michel Muller, surtout [3].

Savoir de quoi on parle

Il importe de distinguer un objet de pensée, par exemple un nombre ou une opération — défini d'une façon ou d'une autre —, de ses représentations, et des signes qui sont utilisés pour dénoter ces représentations, et ceci dans les deux contextes qui nous intéressent : celui des mathématiques (monde parfait des idéalités) et celui de l'informatique (pâle ombre du précédent ☺). Dans la pratique écrite, pas même dans cet article, on ne prend vraiment la peine de systématiquement faire la distinction entre tous ces objets ce qui peut être source de confusion ; il est vrai que trop de rigueur de langage finit par obscurcir le sens.

La représentation des entiers par le *système de numération de position à base avec zéro* est bien connue, au moins pour la base dix et au moins pour les entiers et les décimaux. Une fois les principes de ce système compris, le passage à la base deux ne pose pas de problème, tout en demandant un peu d'entraînement en particulier pour la manipulation des « nombres binaires à virgule » ; on sera aidé par la simplicité des tables d'opérations ou, au besoin, par une calculatrice binaire.

Dans la suite, on désignera par leur écriture décimale habituelle les nombres tels 17 ou 0,1 ou $\frac{1}{5}$; les écritures binaires utiliseront *systématiquement* un indice « deux » comme dans $0,1011_{\mathbf{b}}$; dans le cas d'une écriture décimale, on parlera de *chiffre* et dans celui d'une écriture binaire, on parlera de *bit*. On utilisera aussi la convention courante du surlignement pour les périodes des développements illimités des rationnels ; on désignera par ailleurs, si elle existe, par une écriture du type $\mathrm{fl}(x)$ (fl pour « flottant ») la représentation en machine du nombre réel x, représentation qui peut avoir plusieurs encodages ; enfin au besoin, on désignera par $\texttt{print}(x)$ l'écho console fourni par le système utilisé. D'autres écritures seront introduites dans le texte en fonction des besoins. Toutes ces valeurs sont déterminées par des algorithmes [5] précis dont certains vont être explicités.

Commençons par décrire un univers en général moins bien connu que celui des décimaux.

Les dyadiques

Les nombres dyadiques [6] sont à la base deux ce que les décimaux sont à la base dix. Ils sont définis comme un sous-ensemble des rationnels. Comme il n'y a pas d'écriture standard (à ma connaissance) pour désigner l'ensemble des dyadiques, on utilisera l'écriture $\mathbb{Z}\left[\frac{1}{2}\right]$ [7].

$$\mathbb{Z}\left[\frac{1}{2}\right] = \left\{\frac{p}{2^q} \mid (p\,;\,q) \in \mathbb{Z}\times\mathbb{N}\right\}$$

Il est utile de traduire cette définition globale sous une forme plus opératoire : un rationnel x est dyadique si et seulement s'il existe un entier naturel q tel que $2^q x$ soit entier.

En évidence, les entiers sont des dyadiques lesquels sont aussi des décimaux ; les réciproques sont fausses.

> Par exemple $a = 0{,}132\,812\,5$ est un dyadique non entier : on a $a = \frac{1\,328\,125}{10^7}$; puis, on regarde si 5^7 divise $1\,328\,125$ ce qui est le cas et donc $a = \frac{17}{2^7} \in \mathbb{Z}\left[\frac{1}{2}\right]$. Cet exemple est générique et fournit un algorithme pour tester le caractère dyadique d'un décimal donné par son écriture décimale.
> Mais $\frac{1}{10}$ n'est pas dyadique : en effet, s'il existait p, q tels que $\frac{1}{10} = \frac{p}{2^q}$; on aurait aussi $2^q = 10p$ et donc 5 diviserait 2^q ; d'où la contradiction.

On obtient l'unicité d'écriture dans l'écriture fractionnaire d'un dyadique en imposant q minimal. Cette hypothèse est implicite dans tout ce qui suit. L'ensemble des dyadiques possède de bonnes propriétés algébriques : c'est un sous-anneau de $\mathbb{Q}$, principal (et même euclidien [8]). On va voir qu'il possède aussi une propriété précieuse de densité.

Bien sûr, on ne va pas se contenter de l'écriture décimale pour écrire un dyadique.

Écriture binaire des dyadiques

Tout dyadique $d \in]0\,;1[$ s'écrit de façon unique sous la forme :

$$d = \sum_{k=1}^{q} a_k\, 2^{-k} \text{ avec } q \geqslant 1 \text{ et } a_k \in \{0\,;\,1\} \text{ et } a_q \neq 0$$

ce qui fournit l'écriture binaire de d :

$$d = (0, a_1 a_2 \cdots a_q)_{\mathbf{b}}$$

5. Pour la quasi-totalité des utilisateurs actuels du mot, un algorithme est en réalité un programme — qui ne veut pas dire son nom — écrit dans un langage non spécifié — ce qui est fort commode — donc qui utilise, outre la séquence, l'affectation, donc des variables informatiques donc de la mémoire. Personnellement, j'en suis resté, pour ce qui concerne nos niveaux d'enseignement, à la conception prônée en son temps par Jacques Arsac[4] : un algorithme est un ensemble de définitions effectives ; des égalités, des récurrences, mais pas l'affectation qui nous fait sortir du champ mathématique…

6. À ne pas confondre avec les nombres 2-adiques.

7. Si $a \in \mathbb{Q} \setminus \mathbb{Z}$, $\mathbb{Z}[a]$ désigne usuellement le plus petit sous-anneau de $\mathbb{Q}$ — au sens de l'inclusion — qui contient a. On imagine bien la construction naïve d'un tel sous-anneau en y mettant tous les entiers, puis tous les multiples entiers de a, puis tous ceux de a^2, puis tout ceux de a^3 etc. puis toutes les sommes finies des précédents ; on obtient ainsi tous les polynômes en a à coefficients entiers (d'où la dénotation $\mathbb{Z}[a]$).

8. L'application $\varphi = \frac{p}{2^q} \longmapsto |p|$ (q minimal) est un sthasme euclidien sur les dyadiques qui permet de définir une division ; ah ! Bourbaki…

L'unicité concerne à la fois q et la suite des a_k. **Démonstration :** Il suffit d'écrire en base deux le numérateur de la représentation rationnelle de d. Le numérateur étant impair, son bit de poids faible est 1. Il est intéressant de proposer une démonstration utilisant un algorithme.

Entrées : x un dyadique (strictement) entre 0 et 1
Sorties : N et la liste $(a_k)_{1\leqslant k\leqslant N}$ des bits de l'écriture binaire de x
Définitions : $x_0 = x$, $a_k = 2\lfloor x_{k-1}\rfloor$, $x_k = 2x_{k-1} - a_k$, pour tout $k \geqslant 1$, $N = \min\{h | x_h = 0\}$.

Un jeu d'essais pour comprendre le principe simple de cet algorithme compte-goutte avec $x = 0{,}687\,5$:

k	a_k	x_k
0	—	0,687 5
1	1	0,375
2	0	0,75
3	1	0,5
4	1	0

Donc $x = 0{,}101\,1_{\mathbf{b}}$
Pour fournir une preuve de l'algorithme, il convient d'abord de prouver l'existence de N, puis, que les a_k sont bien les bits de l'écriture binaire de x. On va passer sous silence la démonstration d'existence des suites (x_k) et (a_k). En évidence, $x_k \in [0\,;1[$ et $a_k \in \{0\,;\,1\}$; de plus, s'il existe h tel que $x_h = 0$ alors $x_k = 0$ pour tout $k > h$.
Puis, on résout la récurrence $x_k = 2x_{k-1} - a_k$ par la technique du facteur sommant : pour $k \geqslant 1$, on a $\frac{x_k}{2^k} - \frac{x_{k-1}}{2^{k-1}} = -\frac{a_k}{2^k}$, et par télescopage

$$\frac{x_k}{2^k} - x_0 = -\sum_{h=1}^{k}\frac{a_h}{2^h} \text{ donc } 2^k x - \sum_{h=1}^{k} a_h\, 2^{k-h} = x_k.$$

Comme x est dyadique, il existe N tel que $2^N x$ est entier. On a alors $2x_{N-1} = 2^N x - \sum_{h=1}^{N-1} a_h 2^{N-h}$, donc $2x_{N-1}$ est entier et donc $x_N = 0$. On en déduit que $x = \sum_{h=1}^{N}\frac{a_h}{2^h}$, ce qui achève la démonstration. ■

On peut alors représenter tout dyadique non nul d par un triplet (signe, entier, fraction) généralisant l'écriture décimale habituelle des décimaux.
Par exemple :

- $17 = 10001_{\mathbf{b}}$, $\frac{17}{64} = \frac{10\,001_{\mathbf{b}}}{10^7_{\mathbf{b}}} = 0{,}001\,000\,1_{\mathbf{b}}$: pour diviser en base deux par 2^q, on décale la virgule de q rang vers la gauche.
- $\frac{47}{16} = \frac{101\,111_{\mathbf{b}}}{10^4_{\mathbf{b}}} = 10{,}111\,1_{\mathbf{b}}$.
- Passage du binaire à l'écriture décimale :

$$\begin{aligned} &-1\,011{,}110\,011_{\mathbf{b}} \\ &= -\left[(2^3 + 2^1 + 1) + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^5} + \frac{1}{2^6}\right] \\ &= \frac{755}{64} = -11{,}796\,875. \end{aligned}$$

Les exemples précédents font apparaître une propriété *a posteriori* évidente : l'écriture décimale et l'écriture binaire de la fraction d'un dyadique ont le même nombre de chiffres après la virgule.

Densité et approximation

Comme les décimaux, les dyadiques sont denses dans les réels ; c'est une propriété de nature archimédienne.
Démonstration : Il suffit de montrer qu'ils le sont dans les décimaux. Comme deux décimaux distincts sont distants d'au moins un 10^{-p} ($p \geqslant 1$) et que $2^{-4p} < 10^{-p}$, il existe au moins un terme de la suite de dyadiques $(k2^{-4p})_k$ entre ces deux décimaux. ■
Il est utile d'avoir à l'esprit la représentation graphique des graduations dyadiques emboîtées : les entiers, les demi-entiers, les quart-entiers, etc., pendant de la représentation usuelle des graduations décimales : les entiers, les dixièmes, les centièmes, etc.
On en déduit [9] un théorème de représentation des réels (en se limitant à l'intervalle $]0\,;1[$ pour faire l'économie du traitement de la partie entière) : tout réel x de $]0\,;1[$ s'écrit sous la forme :

$$x = \sum_{n=1}^{+\infty} a_n 2^{-n} \text{ avec } a_n \in \{0\,;\,1\}.$$

On écrit symboliquement $x = 0, a_1a_2a_3\ldots_{\mathbf{b}}$ qu'on appellera développement dyadique illimité de x.
Remarque : il y a unicité dans le seul cas où x est non dyadique. Sinon il y a deux représentations ; un exemple suffira pour comprendre : $\frac{1}{2} = 0{,}1_{\mathbf{b}} = 0{,}0\overline{1}_{\mathbf{b}}$ (avec la convention sur la périodicité de la partie surlignée). En effet $\sum_{n=2}^{+\infty} 2^{-n} = 2^{-2}\frac{1}{1-\frac{1}{2}} = 2^{-1}$ (c'est la même histoire que $0{,}499\,99\cdots = 0{,}5$ en base dix).
La démonstration classique de la périodicité du développement décimal illimité des rationnels se transpose *mutatis mutandis* au développement dyadique.
Un exemple important pour la suite : quelle est l'écriture binaire de 0,1 ?

On peut utiliser l'algorithme donné pour l'écriture binaire d'un dyadique ; si x est non-dyadique, la suite des x_k ne stationne pas en zéro mais on la sait *a priori* périodique. On peut aussi ruser en cherchant à utiliser la série $\frac{1}{1-q} = \sum_{k=0}^{+\infty} q^k$ avec q une puissance de deux d'exposant négatif ; pour cela, on cherche un multiple de dix qui s'écrive comme différence de deux puissances de deux [10] : $30 = 32 - 2$ donc :

$$\begin{aligned} \frac{1}{10} &= \frac{3}{32-2} = \frac{3}{2^5}\frac{1}{1-2^{-4}} \\ &= \frac{2+1}{2^5}\sum_{n=0}^{+\infty} 2^{-4n} \\ &= \sum_{n=0}^{+\infty}\left(2^{-4n-4} + 2^{-4n-5}\right) \\ &= 0{,}000\,110\,011\ldots_{\mathbf{b}} = 0{,}000\overline{11}_{\mathbf{b}}, \end{aligned}$$

ce qui redémontre que $\frac{1}{10}$ n'est pas dyadique.
Avec un peu d'arithmétique, on peut exhiber un algorithme général pour déterminer le d.d.i. de $\frac{1}{n}$; voir l'annexe 1.

9. « La marge est trop étroite pour contenir la démonstration » ; mais l'annexe 3 y pourvoit.
10. Il y a beaucoup de petits entiers qui possèdent cette propriété, mais pas tous.

Représentation binaire en virgule flottante normalisée[11]

Tout nombre réel non nul x s'écrit de façon unique sous la forme :

$$x=(-1)^s \times m \times 2^e \text{ avec } s \in \{0,1\},\ 1 \leqslant m < 2,\ e \in \mathbb{Z}$$

$(-1)^s$ est le signe, m le significande et e l'exposant.
On perd l'unicité si on impose seulement $0 < m$. On parle alors de virgule flottante. On peut déjà observer une invariance d'échelle : x et $x2^k$ ont même significande.

Démonstration : Donnons une preuve algorithmique :

Entrée : $x \in \mathbb{R}^*$
Sorties : $s \in \{0,1\}$, $m \in [1\,;2[$, $e \in \mathbb{Z}$ vérifiant $x = (-1)^s m 2^e$
Définitions :
- si $x > 0$ alors $s = 0$ sinon $s = 1$;
- $e = \min\{h \in \mathbb{Z} \mid |x| < 2^{h+1}\}$;
- $m = |x|2^{-e}$;

Il suffit de démontrer l'existence de e ; si $|x| \geqslant 1$, cela résulte de la divergence vers $+\infty$ de la suite (2^h) et si $|x| < 1$ de la convergence vers 0 de la suite (2^{-h}).

L'unicité ne pose pas de problème particulier. ■

Exemple : comme $\frac{1}{2^4} < \frac{1}{10} < \frac{1}{2^3}$ on a : $\frac{1}{10} = \frac{8}{5}2^{-4}$ avec $1 \leqslant \frac{8}{5} < 2$; puis $\frac{8}{5} = 1,\overline{1001}_{\mathbf{b}}$.
Exprimés en base dix : $m = 1,6$ et en base deux : $m = 1,\overline{1001}_{\mathbf{b}}$.

On notera que le premier bit (avant la virgule) de m est toujours 1. À noter aussi le fait qu'il n'est pas possible d'inclure 0 dans cette représentation, sauf à accepter $m = 0$ pour le significande mais l'exposant est alors indéterminé et il y a deux signes possibles.

L'ensemble des flottants

Un ordinateur travaille-t-il nécessairement en base deux ? Au niveau de l'unité arithmétique et logique certainement, mais il peut exister une unité arithmétique matérielle utilisant une autre base, typiquement huit, seize ou dix (calculatrices). Pour le savoir, on peut utiliser l'algorithme de Malcom-Gentlemen ; nous renvoyons à l'exposé de J.-M. Muller [5] qui est très clair.
Un ordinateur utilisant des mots d'un nombre fini de bits, doit matériellement limiter le nombre de bits du significande et l'intervalle de variation de l'exposant, cela quand bien même on peut envisager par voie logicielle de manipuler des nombres en arithmétique non finie (comme les entiers dans Python) et en base quelconque.
L'ensemble des nombres informatiques obtenus sera dénoté génériquement $\mathbb{F}$ (bien qu'il dépende du format considéré) et ses éléments désignés par le terme « flottants ». C'est un ensemble fini, donc discret[12]. La représentation (qui va consister à projeter $\mathbb{R}$ sur $\mathbb{F}$) est nécessairement approchée.

Deux critères sont importants :
- l'amplitude de l'intervalle des nombres représentés ; en particulier la petitesse des nombres représentés ;
- la distribution des écarts entre deux nombres consécutifs représentés qui détermine la précision de la représentation.

Le deuxième critère est essentiel car un flottant représente tout un intervalle de réels.
Il s'agit donc de coder la représentation (signe, significande, exposant). Un bit suffit pour le signe ; soit p le nombre de bits utilisés pour le significande, et $[\![e_{\min}\,;e_{\max}]\!]$ l'intervalle d'entiers utilisé pour l'exposant lui-aussi codé en base deux avec q bits. On a donc besoin *a priori* de $p+q+1$ bits. Typiquement les mots informatiques sont formés de 16, 32, 64, 128 bits. Nous allons détailler le format sur 64 bits.

Le lecteur intéressé par les formats utilisés au cours de l'histoire du calcul flottant pourra consulter [6].

Encodage sur 64 bits

En 1985, après 8 années de concertations entre des chercheurs universitaires et des chercheurs du fondeur Intel sous la houlette de William Kahan, paraissait la « norme pour l'arithmétique binaire en virgule flottante », intitulée IEEE 754, aujourd'hui norme internationale, bien intégrée par les différents fondeurs et respectée par les logiciels de programmation (dont Python) ; cette norme a été complétée en 2008. On pourra consulter Wikipedia sur ce sujet (de préférence en anglais).
Dans le format *double*, un flottant dit *normal* est codé par un mot de 64 bits (les bits de poids forts à gauche) :

1. le bit de poids fort (poids 63) pour le signe : 0 pour $+$, 1 pour $-$;
2. les 11 bits suivants pour l'exposant, avec $e_{\min} = -1\,022$ et $e_{\max} = 1\,023$; l'exposant est dit *biaisé* car ces 11 bits codent en réalité non pas e mais $e + 1\,023 \in [\![1\,;2\,046]\!]$. On note que $2\,046 = 2^{11} - 2$; il y a donc deux valeurs de l'exposant *a priori* non utilisées pour coder les flottants normaux.
3. les 52 bits de poids faibles pour le significande. Le significande est dans l'intervalle $[1\,;2[$, donc le bit fort de son écriture binaire est toujours égal à 1, et il n'est pas nécessairement de le coder effectivement, ce qui porte en réalité à 53 les bits codés du significande. C'est la convention dite du « bit implicite ». La partie fractionnaire du significande est appelée la *fraction*. On dit que la précision du format double est 53.

11. Il y a du flottement, semble-t-il, sur la définition de « normalisé » ; la définition donnée ici est celle d'une partie de la communauté française. Mais une autre prend $\frac{1}{2} \leqslant m < 1$.
12. Chaque élément de $\mathbb{F}$ (qui est aussi un élément de $\mathbb{R}$) peut être isolé dans un intervalle non trivial.

4. le nombre 0 est représenté par un exposant biaisé et un significande nuls tous les deux et le bit de signe quelconque ; il y a donc deux zéros -0 et $+0$ confondus dans les calculs ordinaires mais pas dans le traitement des exceptions.

Un mot de 64 bits encode donc le nombre binaire $(-1)^s \times (1,m) \times 10_{\mathbf{b}}^{e-1\,111\,111\,111_{\mathbf{b}}}$ formule dans laquelle e est l'entier binaire codée par les 11 bits de l'exposant, et m la fraction codée par les 52 derniers bits ; il s'agit donc d'un dyadique.
Donnons quelques exemples de codage/décodage :

- un encodage exact $x = \frac{71}{512}$: on a $2^6 < 71 < 2^7$ donc $2^{-3} < x < 2^{-2}$ et $e = -3$ et donc $e + 1\,023 = 1\,020 = 1\,111\,111\,100_{\mathbf{b}}$.
Ensuite $m = x2^3 = \frac{71}{64} = \frac{64+4+2+1}{64} = 1{,}000\,111_{\mathbf{b}}$.
Le code de x est donc :

$$\begin{aligned}\textbf{binary64}(x) =& 0 && \text{signe}\\ & 01111111100 && \text{exposant}\\ & 000111\underbrace{0\cdots0}_{\text{46 zéros}} && \text{significande}\end{aligned}$$

- décodage de :

$$\begin{aligned}X =& 1\\ & 10000000001\\ & 0000000001\underbrace{0\cdots0}_{\text{42 zéros}}\end{aligned}$$

signe : $s = 1$ donc $(-1)^s = -1$.
exposant biaisé : $10000000001_{\mathbf{b}} = 1\,025$ puis $e = 1\,025 - 1\,023 = 2$.
significande avec bit implicite :

$$1,0000000001_{\mathbf{b}} = 1 + \frac{1}{2^{10}} = \frac{1\,025}{1\,024}$$

donc $x = -1 \times \frac{1\,025}{1\,024} \times 2^2 = -\frac{1\,025}{256} = -4{,}003\,906\,25$

Essayons de donner une représentation graphique des flottants *normaux* strictement positifs du format double.

- D'abord ceux de l'intervalle $[1\,;2[$; en posant $\varepsilon = 2^{-52}$ (l'epsilon-machine), on y trouve exactement les dyadiques de la suite arithmétique $1 + k\varepsilon$ avec $0 \leqslant k < 2^{52} - 1$; tout réel de l'intervalle $]1\,;2]$ est donc à une distance inférieure à $\frac{\varepsilon}{2}$ d'un flottant. On peut aussi représenter ces flottants par un quotient $\frac{h}{2^{52}}$ avec $2^{52} \leqslant h < 2^{53}$.
La représentation binaire de ces flottants est :

$$0 \mid 01111111111 \mid \text{ significande}$$

le champ significande variant de

$$\underbrace{0\cdots0}_{\text{52 zéros}}$$

(fraction de 1) à

$$\underbrace{1\cdots1}_{\text{52 un}}$$

(fraction de $2 - \varepsilon$).
On observe que les flottants normaux sont ordonnés selon l'ordre lexicographique de leur encodage (considéré comme un mot, ce qu'il est effectivement) et qu'on passe d'un flottant au suivant à ajoutant 1 à son encodage (considéré maintenant comme un nombre binaire).

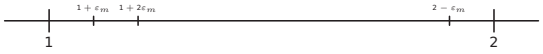


Figure 1. *Représentation des flottants* normaux.

- Ensuite ceux de l'intervalle $[2\,;2^2[$ qui sont exactement les doubles du précédent, soit les dyadiques de la forme $2 + k(2\varepsilon)$ avec $0 \leqslant k < 2^{52} - 1$; ce doublement se poursuit jusqu'à l'intervalle $[2^{e_{\max}}\,;2^{e_{\max}+1}[$, dans lequel la distance entre deux flottants normaux consécutifs est $2^{e_{\max}}\varepsilon$ ($= 2^{971} \approx 2{\cdot}10^{292}$! ! !). Le plus grand flottant représenté est donc $2^{e_{\max}}(2 - \varepsilon)(= 2^{1024} - 2^{971} \approx 1{,}8{\cdot}10^{308})$.
Il est amusant d'observer qu'à partir de l'exposant $e = 52$ tous les flottants supérieurs à 2^{52} sont (mathématiquement) des entiers, de plus en plus dispersés !
Revenons à l'intervalle $[2\,;2^2[$; la représentation binaire des flottants y résidant est :
La représentation binaire de ces flottants est :

$$0 \mid 100000000000 \mid \text{ significande}$$

le champ significande variant encore de

$$\underbrace{0\cdots0}_{\text{52 zéros}}$$

(fraction de 2) à

$$\underbrace{1\cdots1}_{\text{52 un}}$$

(fraction de $4 - 2\varepsilon$).
On peut faire la même observation que précédemment sur l'ordre de ces flottants ; de plus, le raccord entre les deux intervalles $[1\,;2[$ et $[2\,;2^2[$ se fait bien puisque l'encodage de $2 - \varepsilon + 1$ est :

$$\underbrace{0 \mid 01111111111 \mid \underbrace{1\cdots1}_{\text{52 un}}}_{\text{encodage de } 2-\varepsilon} + 1$$

$$= \underbrace{0 \mid 10000000000 \mid \underbrace{0\cdots0}_{\text{52 un}}}_{\text{encodage de 2}},$$

qui est bien l'encodage de 2.
(re-précisons : en considérant les mots de 64 bits comme des entiers binaires). En revanche, on observe une discontinuité dans la précision au passage des valeurs 2^e : à gauche $2^{e-1}\varepsilon$, à droite $2^e\varepsilon$.

- En allant vers 0, l'intervalle suivant est $[\frac{1}{2}\,;1[$ dans lequel les flottants normaux sont exactement les moi-

tiés de ceux de $[1\,;2[$, donc de la forme $\frac{1}{2}+k\frac{\varepsilon}{2}$ avec $0 \leqslant k < 2^{52}-1$. La distance entre deux flottants normaux est donc divisée par deux. Cette dimidiation se poursuit jusqu'à l'intervalle $[2^{e_{\min}}\,;2^{e_{\min}+1}[$ dans lequel la distance entre deux flottants normaux consécutifs est $2^{e_{\min}}\varepsilon (= 2^{-1074} \approx 5{\cdot}10^{-324})$.
Le plus petit flottant normal est donc

$$2^{e_{\min}} (= 2^{-1022} \approx 2{\cdot}10^{-308}).$$

Mais ce n'est pas tout à fait fini ; la norme introduit aussi des flottants dits *dénormalisés* (ou sous-normaux) qui peuplent l'intervalle $]0, 2^{e_{\min}}[$. Leur exposant réel est -1023 et leur significande, toujours codé sur 52 bits, est entre 0 et 1 (exclus) ; il s'agit donc des flottants $k\varepsilon 2^{-e_{\min}} (= k2^{-1074})$ pour $1 \leqslant k < 2^{52}$ dont la représentation est

$$0 \mid 00000000000 \mid \text{ significande}$$

le champ significande variant de

$$\underbrace{0\cdots0}_{51\text{ zéros}}1 \text{ à } \underbrace{1\cdots1}_{52\text{ un}};$$

le code exposant indique conventionnellement que le bit implicite est nul.
Enfin, la norme introduit des entités qui ne sont plus des nombres (`+inf`, `-inf`, `NaN`), toujours codés sur 64 bits, qui sont pleinement intégrées à l'arithmétique spécifiée[13], mais dont, dans la suite, nous ignorerons l'existence ; quand on parlera de flottant, il s'agira donc toujours de *nombre* flottant (normal ou sous-normal).
Résumons :

L'ensemble des flottants strictement positifs décrits précédemment est l'ensemble

$$\left\{N2^{E-52} \text{ avec } 1 \leqslant N \leqslant 2^{53}-1 \right.$$
$$\left. \text{et } -1022 \leqslant E \leqslant 1023\right\}.$$

On observera qu'à E fixé, les flottants sont tous des multiples *entiers* de 2^{E-52}. C'est bien une partie de l'ensemble des dyadiques $\mathbb{Z}\left[\frac{1}{2}\right]$. Cette représentation des éléments de $\mathbb{F}$ avec un significande entier s'avère très utile dans les démonstrations ; il n'y a pas unicité, mais on l'obtient en imposant à N d'être minimal.

Propriétés de la représentation

En parlant des éléments de $\mathbb{F}$, il faudrait distinguer à nouveau les nombres et leurs différentes représentations (binaire, décimale, quotient d'entiers, **binary64**…).

Ainsi l'inverse de 2 : $\frac{1}{2}$ qui est à la fois, un réel, un décimal (dont l'écriture est $0,5$), un dyadique (dont l'écriture binaire est $0,1_{\mathbf{b}}$ et un élément de $\mathbb{F}$ dont l'écriture binaire flottante est la chaîne

$$0 \mid 1111111110 \mid \underbrace{0\cdots0}_{52\text{ zéros}}.$$

Pour s'en sortir rigoureusement, il faudrait introduire des conventions d'écriture… Dans ce qui suit le contexte devrait permettre de distinguer les objets.

Il y a aussi les deux zéros qui embêtent un peu, mais heureusement, pour les calculs considérés, on a $0^+ = 0^-$, on peut donc oublier cette distinction.
Toute représentation des réels dans $\mathbb{F}$ s'appelle un *arrondi*. La norme IEEE 754 en définit plusieurs. Le plus intéressant (par défaut en Python) est *l'arrondi au plus près*, dénoté ici RN, qui doit vérifier la condition, pour tout réel x représentable :

$$\text{pour tout } y \in \mathbb{F}, |x - \mathrm{RN}(x)| \leqslant |x - y|.$$

Le fonctionnement pratique est simple : Soit x un réel non flottant mais dans la plage des nombres représentés ; soient x_1, x_2, $x_1 < x_2$, les deux flottants encadrant x.

- si $x_1 < x < \frac{1}{2}(x_1+x_2)$ alors $\mathbb{F}(x) = x_1$;
- si $x = \frac{1}{2}(x_1+x_2)$ alors $\mathbb{F}(x) = x_1$ ou x_2 selon une règle visant à l'équirépartition des choix[14] ;
- si $\frac{1}{2}(x_1+x_2) < x < x_2$ alors $\mathbb{F}(x) = x_2$.

Il y a une sorte de discontinuité dans la précision de l'arrondi au plus près au passage des puissances de 2 ; ainsi tout réel de l'intervalle $]2-\frac{\varepsilon}{2}\,;2[$ est représenté par 2 avec une erreur absolue d'arrondi inférieure à $\frac{\varepsilon}{2}$. Mais 2 représente aussi les réels de l'intervalle $]2\,;2+\varepsilon[$, cette fois avec une erreur absolue d'arrondi inférieure à ε.
En revanche, une propriété importante de l'arrondi au plus près est qu'il est à erreur relative constante : pour tout $x > 0$ de la plage de représentabilité, $|x - \mathrm{RN}(x)| \leqslant \frac{\varepsilon}{2}x$. On a d'ailleurs aussi $|x - \mathrm{RN}(x)| \leqslant \frac{\varepsilon}{2}\mathrm{RN}(x)$.

Démonstration : Il suffit de regarder le code binaire : si e est l'exposant de x, on a $|x - \mathrm{RN}(x)| \leqslant 2^{e-52} = \varepsilon \times 2^e \leqslant \frac{\varepsilon}{2}2^{e-1} \leqslant \frac{\varepsilon}{2}x$. ■

L'application de $\mathbb{R}$ dans $\mathbb{F}$, $x \mapsto \mathrm{RN}(x)$ possède, comme les autres arrondis, les propriétés suivantes :

1. elle est surjective (en oubliant les objets qui ne sont pas des nombres flottants) mais fortement non injective : l'image réciproque de chaque flottant est un intervalle de $\mathbb{R}$;
2. elle est croissante : si $-\max_{\mathbb{F}} \leqslant x < y \leqslant \max_{\mathbb{F}}$ alors $\mathrm{RN}(x) \leqslant \mathrm{RN}(y)$;
3. elle est impaire : $\mathrm{RN}(-x) = -\mathrm{RN}(x)$;
4. RN est l'identité sur les dyadiques non nuls de $\mathbb{F}$ qui sont les seuls nombres représentés exactement : si $x \in \mathbb{F} \setminus \{0^-\,;0^+\}$ alors $\mathrm{RN}(x) = x$;

13. « Pleinement intégrés » signifie que toutes les propriétés de ces objets sont spécifiées par la norme, la philosophie générale étant de faire en sorte d'éviter que les calculs s'interrompent ; le lecteur curieux pourra jouer dans la console Python avec float('inf') et float('nan').
14. C'est la règle du bit de parité : les flottants consécutifs x_1 et x_2 ont des significandes qui diffèrent par le bit de poids faible ; on choisit celui des deux flottants x_1 ou x_2 dont le significande se termine par 0. On observera que cette règle diffère de la coutume habituellement utilisée pour arrondir les décimaux.

5. la plage des nombres strictement positifs qui sont représentés est incluse dans l'intervalle $\left[2^{-1074}\,;2^{1023}\times(2-2^{-52})\right]$; par ailleurs, si x est compris entre 0 et 2^{-1074} alors $\mathrm{RN}(x)=0^+$ (sous-passement) et si $x>2^{1023}\times(2-2^{-52})$ alors $\mathrm{RN}(x)$ génère un surpassement[15] (+inf dans le standard IEEE 754).
6. une invariance d'échelle : on a
$$\mathrm{RN}(2^k x)=2^k\,\mathrm{RN}(x)$$
s'il n'y a pas sous ou surpassement.

Ces propriétés permettent de démontrer un certain nombre de théorèmes du calcul flottant.

Exemple de calcul d'arrondi en virgule flottante

Calculons l'arrondi de 0,1.
On a $2^{-4}<0,1<2^{-3}$; dans cet intervalle, les flottants sont de la forme $x_k=2^{-4}(1+k\varepsilon)$; il s'agit de trouver k entre 1 et $2^{52}-1$ minimisant[16] $|0{,}1-2^{-4}(1+k\varepsilon)|$.
Mathématiquement, la distance précédente s'annule pour $k=2\,702\,159\,776\,422\,297{,}6$ (qui bien sûr, n'est pas entier) mais donc, par monotonie linéaire, le k cherché est égal à 2 702 159 776 422 298 ; donc

$$\begin{aligned}\mathrm{RN}(0{,}1)&=2^{-4}(1+2\,702\,159\,776\,422\,298\varepsilon)\\&=\frac{2^{52}+2\,702\,159\,776\,422\,298}{2^{56}}\\&=\frac{3\,602\,879\,701\,896\,397}{36\,028\,797\,018\,963\,968}\\&=0,1000000000000000055511151\\&\quad\ldots 3125782702118158340454101562 5\end{aligned}$$

Procédons autrement : on a vu que
$$0,1=(0,\overline{0011}_{\mathbf{b}})=1,\overline{1001}_{\mathbf{b}}\times 2^{-4}$$
donc $m=1,\overline{1001}_{\mathbf{b}}$ et $e=-4$ donc
$$e+1023=1019=1111111011_{\mathbf{b}}.$$

Puisque m a une infinité de bits, on doit en retenir 52 (après la virgule) ; comme le bit d'arrondi (le 53e) est 1 et que le « sticky » bit (somme logique de tous les bits suivants) est aussi 1, la treizième quatraine 1001 est arrondie en 1010. Finalement (codé en 64 bits)
$$X=\mathrm{RN}(0,1)=0|01111111011|\underbrace{1001}_{12}1010$$

Or si on décode X (en décimal) on obtient
$$\begin{aligned}\mathrm{RN}(0,1)=2^{-4}\Bigg\{1+\sum_{k=0}^{11}\left(\frac{1}{2^{4k+1}}+\frac{1}{2^{4k+4}}\right)\\+\frac{1}{2^{49}}+\frac{1}{2^{51}}\Bigg\}\\=\frac{3602879701896397}{36028797018963968}\end{aligned}$$
et on retrouve le résultat précédent.
Ce calcul illustre le fait que pour obtenir un arrondi binaire au plus près, on a besoin de « bits de garde ».
On observe que $\mathrm{RN}(0,1)-0,1=0,1\times 2^{-54}$ (égalité dans $\mathbb{D}$).

Conversions décimale ↔ binaire

L'un des problèmes d'un système informatique réside dans la communication homme-machine, en particulier pour ce qui concerne les nombres, le passage d'une représentation décimale (préférée par les humains) à une représentation binaire et vice-versa. En Python, l'exécution de la ligne de commande « `print(0.1)` » fournit à l'écran « 0.1 ». Il s'agit d'un écho (un effet de bord) obtenu par un algorithme d'affichage (défini par Python) et non pas de la valeur exacte du flottant qui représente 0.1 (qui est pourtant bien encodé en **binary64**) ; ceci indique qu'une routine de conversion « binaire vers décimal » a été appelée et a fourni 0.1, ceci bien que le dyadique qui code 0.1 ne soit pas égal au décimal 0,1.
Une remarque technique : avec Python, on peut contourner la routine d'affichage avec la méthode format(). format(0.1,'.55f') renvoie

$$0.1000000000000000055511151\ldots$$
$$2312578270211815834045410156 25$$

et float.as_integer_ratio(0.1) renvoie :
(3602879701896397,36028797018963968)
qui est la représentation flottante avec significande entier, le dyadique
$$\frac{3\,602\,879\,701\,896\,397}{36\,028\,797\,018\,963\,968}$$
étant exactement égal à $\mathrm{RN}(0{,}1)$ comme on l'a vu ci-dessus.
Revenons au problème de conversion : dans [3], Muller donne un exemple surprenant au premier abord ; si $x=2^{-1022}-2^{-1074}$, x est un sous-normal de $\mathbb{F}$[17] ; sa représentation décimale exacte (que l'on peut obtenir en Python avec la méthode format) comporte 767 chiffres ; mais heureusement, `print`(x) n'af-

15. Cocorico ! Toute la planète informatique dit « underflow » et « overflow ».
16. Il est *a priori* exclu qu'il y ait deux tels entiers car $0,1$ n'est pas dyadique.
17. **binary64**$(x)=0\mid 00000000000\mid 1111111111111111111111111111111111111111111111111111$.

fiche que $2.225073858507201e-308$; bien que distinct (dans $\mathbb{D}$) de $2^{-1022}-2^{-1074}$, ces nombres sont égaux dans $\mathbb{F}$, c'est-à-dire $\mathrm{RN}(2^{-1022}-2^{-1074}) = \mathrm{RN}(2.225073858507201e-308)$. Toute l'information utile (pour le système) sur un décimal à 767 chiffres est contenue dans un décimal à 16 chiffres. Pourquoi 16 chiffres, qui en général ne suffisent pas pour distinguer deux flottants : ainsi 1,234 567 890 123 456 $\neq$ 1,234 567 890 123 456 2 dans $\mathbb{F}$.

Précisons le problème : on part d'un décimal positif x avec d chiffres de précision ; soit y le dyadique arrondi au plus près de x avec b chiffres de précision ; soit enfin z le décimal arrondi au plus proche de y avec d chiffres de précision (avec une règle de choix en cas d'équidistance quelconque). À quelle(s) condition(s) a-t-on $x = z$ (sous-entendu quelque soit x), contrainte imposée par le standard IEEE 754.
La réponse est fournie par le théorème de Matula : la condition nécessaire et suffisante est $10^d < 2^{b-1}$ (à d donné). Bien sûr, pour le cycle : binaire $\to$ décimal $\to$ binaire, la condition est $2^b < 10^{d-1}$ (à b donné).
Ainsi, si $d = 16$, comme $10^{16} > 2^{53-1}$, une précision en décimal égale à 16 est trop grande pour le format double ($b = 53$), il faut se limiter à 15. En revanche, comme $2^{53} < 10^{17-1}$, le système peut afficher 17 chiffres décimaux comme traduction d'un calcul dans le format double. Dans l'exemple ci-dessus, les 16 chiffres affichés de $2{,}225\,073\,858\,507\,201 \cdot 10^{-308}$ sont bien en réalité 17, le zéro terminal du significande n'étant pas affiché.

Démonstration : On trouvera une démonstration simple dans un article de Mark Dickinson [7]. ■

Les algorithmes derrière les conversions sont plutôt complexes ; ils sont décrits dans Muller [3]. Le passage du binaire au décimal est plus simple puisque le binaire est connu des systèmes ; c'est leur langage de calcul. Le passage du décimal au binaire est, lui, délicat car le décimal n'existe pas pour un système binaire ; c'est juste une suite de caractères.

Arithmétique flottante

La théorie est elle aussi plutôt compliquée ! Déjà $\mathbb{F}$ n'est pas stable par les opérations arithmétiques : par exemple, la somme de deux nombres représentables n'est pas nécessairement représentable et si c'est le cas, la représentation de la somme n'est pas nécessairement la somme (mathématique) des représentations. On doit donc redéfinir les opérations dans $\mathbb{F}$. Il est possible de le faire en respectant la règle suivante :

> le résultat d'une opération $\odot$ dans $\mathbb{F}$ de deux nombres représentables est l'arrondi au plus près dans $\mathbb{F}$ du résultat de l'opération correspondante dans $\mathbb{D}$.
> On dit alors que l'opération flottante $\odot$ est *correcte*.

Dans la règle précédente, chaque terme est important, et il convient de ne pas se méprendre sur son sens en particulier sur sa traduction en terme d'affichage ; ainsi `print`$(2 \oslash 3)$ affiche 0,666 666 666 666 666 6 alors qu'on attendrait la meilleure approximation 0,666 666 666 666 666 7 (comme sur une calculatrice travaillant en base dix). Mais le lecteur — maintenant plus au fait de l'arithmétique dans $\mathbb{F}$ — pourra vérifier la cohérence de cette réponse avec la règle ci-dessus.

La norme IEEE 754 impose que cette règle s'applique aux quatre opérations (addition, soustraction, multiplication et division) ainsi qu'à la racine carrée. Cette exigence nécessite l'utilisation de *bits de garde*. L'extension de cette règle aux fonctions usuelles fait encore l'objet de recherches [18].

Le problème majeur est que, même équipé d'une arithmétique flottante avec des opérations $\oplus, \ominus, \otimes, \oslash$ correctes, l'ensemble $\mathbb{F}$ ne possède pas les propriétés algébriques de $\mathbb{Z}\left[\frac{1}{2}\right]$, loin s'en faut. Associativité et distributivité, par exemple, font défaut.

Pourtant, l'arithmétique flottante reste — au moins en théorie — déterministe et peut faire l'objet de théorèmes démontrés selon les règles de l'art en usage en mathématique, au besoin avec l'aide de systèmes de preuves formelles. À titre d'exemple simple (et classique), on peut citer le *théorème de Sterbenz* :

Théorème 1

Si x et y sont deux flottants normaux vérifiant $\frac{y}{2} \leqslant x \leqslant 2y$ alors la différence (mathématique) $x-y$ est aussi un flottant, normal ou sous-normal, donc exactement représentable (et ceci quelque soit le mode d'arrondi) ; en particulier, $\mathrm{RN}(x-y) = x-y$.

Démonstration : On a nécessairement $y \geqslant 0$; le cas $y = 0$ peut être éliminé ; par raison de symétrie on peut supposer que $y \leqslant x \leqslant 2y$; utilisons la représentation en virgule flottante normalisée : $x = 1, x_1 \cdots x_{52} \times 2^{e_x}$ et $y = 1, y_1 \cdots y_{52} \times 2^{e_y}$ avec des exposants e_x et e_y entre $e_{\min}$ et $e_{\max}$ et des significandes écrits en base deux. Comme $2y$ est supposé implicitement flottant (en réalité, on n'a pas besoin de faire cette hypothèse, en gérant le flottant inf), on a $e_y + 1 \leqslant e_{\max}$. De l'encadrement $y \leqslant x \leqslant 2y$ on déduit facilement $e_x = e_y$ ou $e_x = e_y + 1$. Si $e_x = e_y$, on aura $x - y = 1, z_1 \cdots z_{52} \times 2^{e_x}$ ou $x - y = 0, z_1 \cdots z_{52} \times 2^{e_x}$; dans les deux cas, on obtient un flottant (éventuellement sous-normal).
Si $e_x = e_y + 1$, alors $x - y = (1, x_1 \cdots x_{52} - 1, y_1 \cdots y_{52}) \times 2^{e_y} = 1, z_1 \cdots z_{52} \times 2^{e_y}$ ou $x - y = 1, z_1 \cdots z_{52} \times 2^{e_y+1}$; dans les deux cas, on a bien un flottant (normal). ■

Calcul des sommes

L'addition n'étant pas le sujet du fil rouge de ce numéro, nous renvoyons par exemple à l'exposé de J.-M. Muller dans [5]. Ce qu'il faut savoir, c'est que, sous certaines conditions, la somme correcte $a \oplus b$ de deux flottants est un flottant avec une erreur sur la valeur exacte $a+b$ qui, si elle n'est pas nécessairement un flottant, peut être calculée exactement.

Calcul des produits

Le calcul du produit des flottants est de première importance car une grande partie du temps de fonctionnement des unités dédiées lui est consacré. On trouvera une revue dans [8].

18. À ma connaissance.

On suppose toujours que la base est $b =$ deux et que la précision est $p = 53$. Les flottants considérés appartiennent à $\mathbb{F}$; mais, comme souvent en mathématiques, il est plus commode de travailler avec des lettres b, p, $e_{\min}$ et $e_{\max}$ plutôt qu'avec des valeurs (et en réalité, les résultats sont valides quels que soient ces paramètres). Il peut aussi être commode de désigner l'ensemble des flottants correspondant à ces paramètres par $\mathbb{F}(b, p, e_{\min}, e_{\max})$.

Il est à peu près évident pour qui a pratiqué les produits de décimaux (ou même seulement d'entiers ou mieux de polynômes...), que, si l'écriture de x et y comporte chacune p chiffres, celle de xy en comporte exactement $2p-1$ ou $2p$. Donc, si l'on veut calculer exactement le produit de deux flottants dont la précision est p, on a *a priori* besoin de manipuler des flottants de précision $2p$ avant arrondi. Ce calcul est du ressort de l'unité de calcul flottant du microprocesseur ; tout système respectant les spécifications du standard IEEE 754 est en mesure de fournir la valeur exacte de $\mathrm{RN}(x \times y)$.

Une première idée est d'utiliser la représentation avec significande entier des flottants ; le produit de deux flottants se ramène alors au produit de deux entiers (et à la somme des exposants), produit pour lequel on dispose d'algorithmes performants.

Un autre principe possible en base deux est le suivant : on commence par décomposer chaque opérande x et y en une somme (exacte) de deux flottants dont les significandes comportent seulement $s = \left\lfloor \frac{p}{2} \right\rfloor$ bits (algorithme de Veltkamp) ; xy est alors somme de quatre produits exactement représentables avec la précision p.

Algorithme de Veltkamp :
Données : $x \in \mathbb{F}$, $s \in [\![2 ; p-2]\!]$
Sorties : x_1, x_2 tels que $x = x_1 + x_2$, $x_1, x_2 \in \mathbb{F}(2, s, e_{\min}, e_{\max})$
Définitions : $t = \left\lceil \frac{p}{2} \right\rceil$, $C = b^t + 1$, $u = \mathrm{RN}(Cx)$, $v = \mathrm{RN}(x-u)$, $x_1 = \mathrm{RN}(u+v)$, $x_2 = \mathrm{RN}(x - x_1)$

Un exemple est sans doute bienvenu : prenons une précision $p = 9$ pour simplifier ; donc $t = 5$. Soit $x = 0{,}000\,110\,011\,010$ (arrondi au plus près de $\frac{1}{10}$).
On mène les calculs en base deux ; on trouvera une calculatrice binaire à l'adresse [9].
On a successivement

- $C = 100\,001$ puis $Cx = 11{,}010\,011\,011\,01$ donc $u = 11{,}010\,011\,1$
- $x - u = -11{,}001\,101\,001\,1$ donc $v = -11{,}001\,101\,0$
- $u + v = 0{,}000\,110\,1$ donc $x_1 = 0{,}000\,110\,1$
- $x - x_1 = -0{,}000\,000\,000\,11$ donc $x_2 = -0{,}000\,000\,000\,11$

On peut vérifier que l'on a exactement $x = x_1 + x_2$; on observe aussi que x_1 et x_2 ne sont pas nécessairement positifs (ce n'est pas un hasard) et que la solution calculée n'est pas la solution humaine-évidente $x_1 = 0.00011$ et $x_2 = 0.00000001101$. L'utilisation d'un nombre négatif est impérative pour que les x_i soient toujours représentables sur $\left\lfloor \frac{p}{2} \right\rfloor$ bits, 26 bits si $p = 53$.

La validité de l'algorithme de Veltkamp peut être démontrée tout-à-fait rigoureusement dans le contexte du standard IEEE 754 donc en gérant tous les cas (base quelconque, surpassement, sous-passement). On trouvera une preuve simplifiée dans [3].

De la même façon que pour les sommes, sous certaines conditions, l'erreur sur un produit de deux flottants est elle-même un flottant qui peut être calculé exactement (algorithme de Dekker).

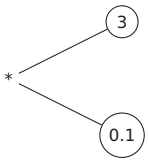
Retour sur le problème du test $n \times 0,1 == n/10$

On sait maintenant que deux réels x et y sont déclarés égaux si et seulement si $\mathrm{RN}(x) = \mathrm{RN}(y)$ dans $\mathbb{F}$.

Examinons tout d'abord le cas $n = 3$.

Lorsqu'on écrit à l'aide du clavier la suite de caractères Unicode « 3 * 0.1 » dans une console Python et qu'on en demande l'évaluation par l'interpréteur (en appuyant sur ⏎), on provoque une cascade d'appels de routines diverses — le tout restant transparent pour l'utilisateur — et d'écritures en mémoire[19] :

1. un analyseur syntaxique traduit la suite par un arbre syntaxique qui doit abstraitement ressembler à quelque chose comme



2. s'il n'y a pas d'erreur de syntaxe, un analyseur sémantique détermine les types des objets constituant l'expression[20], ici des flottants ;
3. l'évaluation de l'expression $3 * 0.1$ par l'interpréteur qui passe par la conversion des suites « 3 » et « 0.1 » en flottants binaires manipulables par le système, et l'appel de la routine de multiplication ;
4. pour terminer, une routine d'affichage qui va convertir le flottant binaire calculé à l'étape précédente en une suite de caractères qui, une fois affichée, *sera interprétée par le lecteur humain* comme un flottant décimal (le résultat du calcul).

On a donc 3 sources d'erreur :

- l'erreur de représentation $x \mapsto \mathrm{RN}(x)$;
- l'erreur de calcul : l'ordinateur ne calcule pas $3 * 0.1$ mais $\mathrm{RN}(3) \otimes \mathrm{RN}(0.1)$
- l'erreur d'affichage : le système n'affiche pas la valeur décimale du flottant calculé (ici un décimal à 52 chiffres qui va être déterminé dans la suite) mais une valeur psychologiquement lisible, en l'occurrence 0.30000000000000004 pour $3 * 0.1$, mais on a bien
$$\mathrm{RN}(3 * 0.1) = \mathrm{RN}(0.30000000000000004).$$

Les entrées et sorties dans une console ne sont donc pas vues par le système avec leur sémantique humaine mais uniquement comme des suites de caractères ; c'est l'utilisateur qui y met la couche sémantique. La surprise dont nous parlions dans l'introduction vient la plupart du temps de l'oubli de ce fait.

Reprenons avec l'évaluation plus complexe de la chaîne « 3 * 0.1 == 0.3 », interprétée comme un booléen entre deux flottants :

1. Encodage de 0,1 au format **binary64** : on donnera un algorithme général plus loin ; on va utiliser une technique très naturelle.

19. L'énumération qui suit est à vocation didactique et n'a aucune prétention à l'exactitude scientifique.
20. En Python, le type des objets n'est pas déclaré mais évalué dynamiquement.

On commence par un jeu de réécritures :
$0{,}1 = \frac{1}{10} = \frac{1}{1\,010_{\mathbf{b}}}$; il ne reste qu'à poser la division de 1 par $1\,010_{\mathbf{b}}$ en calculant en base deux, ce qui ne pose pas de problème vue la simplicité des tables dans cette base.

$$\begin{array}{r|l} 1\,0\,0\,0\,0 & 1\,0\,1\,0 \\ \cline{2-2} 1\,0\,1\,0 & 0,\,0\,0\,0\,1\,1 \\ \cline{1-1} 0\,1\,1\,0\,0 & \\ 1\,0\,1\,0 & \\ \cline{1-1} 0\,0\,1\,0 & \end{array}$$

Le retour du reste 10 signe la périodicité et la partie correspondante du dividende donne la période : $0{,}1 = 0{,}0\overline{0011}_{\mathbf{b}}$.
Puis on fait flotter la virgule pour obtenir un significande entre 1 et $10_{\mathbf{b}}$:

$$0{,}1 = (1, \overline{1001}_{\mathbf{b}}) \times 2^{-4}.$$

Enfin on arrondit au plus près le significande sur 52 bits :

$$(1, \overline{1001})_{\mathbf{b}} = 1, \underbrace{1001\ \cdots\ 1001}_{52\text{ bits}}\ \underbrace{1}_{\substack{\text{bit}\\ \text{d'arrondi}}}\ \underbrace{001001\cdots}_{\text{sticky bit}=1}$$

avec un bit d'arrondi et un sticky bit tous deux égaux à 1, l'arrondi au plus près correspondant à l'arrondi vers $+\infty$ donc en ajoutant 1 au bit de poids faible du significande :

$$\mathrm{RN}(0,1) = 1, \underbrace{1001\ \cdots\ 1001}_{48\text{ bits}}1010$$

L'exposant biaisé est $-4 + 1023 = 1019 = 1111111011_{\mathbf{b}}$ d'où finalement la chaîne de 64 bits représentant $0,1$

$$0|01111111011|\underbrace{1001\cdots1001}_{48\text{ bits}}1010$$

Bien sûr, un programme Python permet d'obtenir ceci, voir la figure 2.

float.hex(x) fournit la représentation interne de x sous la forme d'une chaîne de caractères selon le format (si $x > 0$) '0x1.sssssssssssssssp±eee' ;
les deux premiers caractères ('0x') indiquent que la chaîne est codée en hexadécimal ; les 15 suivants codent le significande ; les derniers codent l'exposant en décimal !

Le cas de 3 est simple :

$$3 = 1,5 \times 2^1 = (1,1)_{\mathbf{b}} \times 2^1$$

donc la chaîne de bits est :

$$0|10000000000|1\underbrace{0\cdots0}_{51\text{ zéros}}.$$

Bien sûr aussi, il existe sur la toile des outils faisant tout cela, par exemple [10].

2. le calcul du produit : le principe imposé pour le calcul du produit de deux flottants (représentables dans le format considéré) par le standard IEEE 754 est celui de toute opération dite correcte : tout doit se passer comme si le produit des significandes était calculé exactement en base deux, puis normalisé, puis le significande du produit est arrondi à la précision du format en utilisant l'arrondi considéré et éventuellement à nouveau normalisé (si l'arrondi provoque une propagation de retenues).
Ici le produit de $1, \underbrace{1001\ \cdots\ 1001}_{48\text{ bits}}1010_{\mathbf{b}}$ par $1,1_{\mathbf{b}}$ (produit des significandes de $0,1$ et de 3) donne un significande sur 55 bits :

$$1, \underbrace{0011\cdots0011}_{52\text{ bits}}10_{\mathbf{b}}$$

dont l'arrondi au plus près selon la règle de parité (sur 53 bits) est

$$1, \underbrace{0011\cdots0011}_{48\text{ bits}}0100_{\mathbf{b}}$$

avec un exposant -1 , soit en décimal

$$\mathrm{RN}(3 * 0{,}1) = 0{,}300\,000\,000\,000\,000\,044\,408$$
$$9\,209\,850\,062\,616\,169\,452\,667\,236\,328\,125$$

Calculons ensuite $\mathrm{RN}(0{,}3)$; on écrit $0{,}3 = \left(1 + \frac{1}{5}\right) \times 2^{-2}$; le d.d.i de $\frac{1}{5}$ s'obtient par exemple à partir de celui de $\frac{1}{10}$ par produit par $2 = 10_{\mathbf{b}}$: $\frac{1}{5} = 0,0\overline{1001}_{\mathbf{b}}$ donc $0{,}3 = 0{,}010\overline{1001}_{\mathbf{b}} = 1{,}01\overline{0011}_{\mathbf{b}} \times 2^{-2}$.
Donc $\mathrm{RN}(0{,}3) = 0{,}01\underbrace{0011\cdots0011}_{\substack{12\text{ quatraines}\\ \text{deux}}}00$ puisque le bit d'arrondi est 0, soit en décimal

$$\mathrm{RN}(0{,}3) = 0{,}299\,999\,999\,999\,999\,988\,897\,769$$
$$753\,748\,434\,595\,763\,683\,319\,091\,796\,875$$

Ainsi donc, on a bien $\mathrm{RN}(0{,}3) \neq \mathrm{RN}(3 * 0{,}1)$.
Pour illustrer l'utilisation de l'algorithme de Veltkamp, donnons le calcul de $3 * 0.1$ au format IEEE 754 simple (sur $32 = 1 + 8 + 23$ bits), en omettant les détails :

$x = \mathrm{fl32}(0,1) = 0,00011001100110011001101_{\mathbf{b}}$ (significande sur 24 bits)
$x_1 = 0,000110011001101$ et
$x_2 = -0,00000000000000000001100110011$
En évidence, pour $y = 3 = 11_{\mathbf{b}}$, on a $y_1 = 11$ et $y_2 = 0$.
On a alors

$$\mathrm{fl32}(x_1 \times y_1 + x_2 \times y_1) = 0,01001100110011001101_{\mathbf{b}} = 0,30000001192092895507812 5$$

et en reprenant le calcul ci-dessus,
$\mathrm{fl32}(0{,}3) = 0{,}300\,000\,011\,920\,928\,955\,078\,125$.
Donc en format simple, $3 * 0.1 == 0.3$ est évalué à vrai, ce que l'on peut vérifier sous Python en utilisant les float32 du module numpy. Mais en revanche $9 * 0.1 == 0.9$ est évalué à faux...

```python
def float_to_bin( x ) :
    """
    renvoie la représentation binary64 du flottant x, si elle existe
    """
    cf = '{0:b}{1:b}{2:b}' # chaîne de formatage pour la méthode format
    # le suffixe b pour la conversion en binaire
    if x == 0:
        return  str(x) + '0' *63
    else :
        signe , w = (0 , float.hex ( x )) if x > 0 else (1 , float.hex (x) [1:])
        exposant_biaise = int ( w [18:]) +1023
        # valeur en base dix
        significande = int ( w [4:17] , 16)
        # valeur en base dix
        return cf.format (signe , exposant_biaise , significande )

print(float_to_bin(0.1))
```

Figure 2. *Programme python3.*

On observera que le décimal qui représente exactement 0,1 est

0,100 000 000 000 000 005 551 115 123 125
7 827 021 181 583 404 541 015 625

dont le triple dans $\mathbb{D}$ est

0,300 000 000 000 000 016 653 345 369 377
3 481 063 544 750 213 623 046 875

et que, dans $\mathbb{D}$, $3 \times \mathrm{RN}(0{,}1) \neq \mathrm{RN}(3 \times 0{,}1)$ mais que, dans $\mathbb{F}$, on a bien l'égalité.
Autre observation : $\mathrm{RN}(3 * 0{,}1)$ est le flottant successeur de $\mathrm{RN}(0{,}3)$: $\mathrm{fl64}(3 * 0{,}1) = \mathrm{RN}(0{,}3) + \frac{\varepsilon}{4}$
On observera enfin que, $0{,}3 - \mathrm{RN}(0{,}3) = 0{,}1 \times \frac{\varepsilon}{2} \approx 10^{-17}$.

Ensuite, rapidement, le cas $n = 9$.
On a

$$\mathrm{RN}(0{,}9) = 0{,}900\,000\,000\,000\,000\,022\,204\,46 \\ 49\,250\,313\,080\,847\,263\,336\,181\,640\,625.$$

Puis on évalue le produit $9 \times 0{,}1$: le produit de $1,\underbrace{1001\cdots1001}_{\text{48 bits}}1010_{\mathbf{b}}$ par $1{,}001_{\mathbf{b}}$ (produit des significandes) donne $1,\underbrace{1100\cdots1100}_{\text{48 bits}}110101_{\mathbf{b}}$ sur 55 bits, qui arrondi au plus près sur 53 bits donne $1,\underbrace{1100\cdots1100}_{\text{48 bits}}1101_{\mathbf{b}}$ donc $\mathrm{fl64}(9 * 0{,}1) = 1,\underbrace{1100\cdots1100}_{\text{48 bits}}1101_{\mathbf{b}} \times 2^{-1}$ soit

$$\mathrm{RN}(9 * 0{,}1) = 0{,}900\,000\,000\,000\,000\,022\,204 \\ 460\,492\,503\,130\,808\,472\,633\,361\,816\,406\,25$$

d'où l'égalité. En revanche, en format simple,

$$9 * 0.1 == 0.9$$

est évalué à faux.
Caractériser les entiers flottants n pour lesquels $n * 0.1 == n/10$ est évalué à vrai reste un problème ouvert [21] ; certaines conjectures sont simples : stabilité par produit par 2, vrai si n est un multiple de 5.

Retour sur le problème $n \times \left(\frac{1}{n}\right) \neq 1$

Un programme Python simple permet de déterminer les premiers entiers n pour lesquels $n * (1/n) == 1$ est évalué à false : 49, 98, 103…. On cherche à déterminer une caractérisation de tels entiers.
On continue de travailler avec le format **binary64**. Soit toujours $\varepsilon = 2^{-52}$. On sait que les flottants entre 1 et 2 (exclus) sont de la forme $1 + k\varepsilon$ avec $k \in [\![1\,;2^{52}-1]\!]$ et ceux entre $\frac{1}{2}$ et 1 (exclus) sont de la forme $1 - m\frac{\varepsilon}{2}$ avec aussi $m \in [\![1\,;2^{52}-1]\!]$.
On va d'abord démontrer la propriété suivante : si x est un flottant non nul, alors

$$\mathrm{RN}\left(x \cdot \mathrm{RN}\left(\frac{1}{x}\right)\right) \in \left\{1 - \frac{\varepsilon}{2}, 1\right\}.$$

Démonstration : D'abord, x peut s'écrire sous la forme $m2^e$ avec m flottant appartenant à $[1\,;2[$ et e entier ($e = \lfloor \log_2(x) \rfloor$).
Or $\mathrm{RN}\left(\frac{1}{x}\right) = \mathrm{RN}\left(\frac{1}{m}\right)2^{-e}$ et donc

$$\mathrm{RN}\left(x \times \mathrm{RN}\left(\frac{1}{x}\right)\right) = \mathrm{RN}\left(m \times \mathrm{RN}\left(\frac{1}{m}\right)\right).$$

Si $m \in [1\,;2[$ alors $\frac{1}{m} \in \left]\frac{1}{2}\,;1\right]$; on peut mettre de côté le cas $m = 1$.
On a alors $\left|\frac{1}{m} - \mathrm{RN}\left(\frac{1}{m}\right)\right| \leqslant \frac{\varepsilon}{4}$ d'après la spécification de l'arrondi au plus près ; donc

$$\left|m\frac{1}{m} - m\,\mathrm{RN}\left(\frac{1}{m}\right)\right| \leqslant m\frac{\varepsilon}{4} < \frac{\varepsilon}{2}.$$

Ainsi $1 - \frac{\varepsilon}{2} < m\,\mathrm{RN}\left(\frac{1}{m}\right) < 1 + \frac{\varepsilon}{2}$ d'où la conclusion. ■

On vérifie que, en Python, on a bien $1 - 49 * (1/49) == 2^{**}(-53)$ évalué à true.
Peut-on maintenant trouver une caractérisation des entiers n tels que $\mathrm{RN}\left(n \times \mathrm{RN}\left(\frac{1}{n}\right)\right) \neq 1$? Je propose une réponse [22], mais assez peu satisfaisante d'un point de vue esthétique (la théorie des nombres nous a trop habitués à de beaux résultats).

21. Au moins pour l'auteur !
22. Inspirée d'un papier glané sur la toile dont je n'ai pas retrouvé la référence.

Soit donc n un entier qui n'est pas une puissance de deux (sinon le problème est réglé) ; en posant $e = \lfloor \log_2(n) \rfloor$, et $x = n2^{-e}$, on a $x \in]1\,;2[$ donc il existe $k \in [\![1\,;2^{52}-1]\!]$ tel que $x = 1+k\varepsilon$ donc $\frac{1}{x} = \frac{1}{1+k\varepsilon} \in \left]\frac{1}{2}\,;1\right[$ intervalle dans lequel la distance entre deux flottants est $\frac{\varepsilon}{2}$; l'idée est d'introduire l'entier $h \in [\![1\,;2^{52}-1]\!]$ *défini* par $\mathrm{RN}(\frac{1}{x}) = 1 - h\frac{\varepsilon}{2}$ qui est aussi caractérisé par la condition

$$\left|1 - h\frac{\varepsilon}{2} - \frac{1}{1+k\varepsilon}\right| < \frac{\varepsilon}{4} \qquad (1)$$

inégalité stricte car on voit facilement, à l'aide d'un argument de parité, que $\frac{1}{x}$ ne peut être au milieu de deux flottants consécutifs de l'intervalle $\left]\frac{1}{2}\,;1\right[$. On en déduit que

$$(1+k\varepsilon)\left(1-h\frac{\varepsilon}{2}\right) > 1 - \frac{\varepsilon}{4}(1+k\varepsilon) \qquad (2)$$

Comme $\mathrm{RN}\left(x \times \mathrm{RN}\left(\frac{1}{x}\right)\right) \neq 1$, on a $x\,\mathrm{RN}\left(\frac{1}{x}\right) < 1 - \frac{\varepsilon}{4}$ donc

$$(1+k\varepsilon)\left(1-h\frac{\varepsilon}{2}\right) < 1 - \frac{\varepsilon}{4} \qquad (3)$$

Comme (2) et (3) impliquent (1), on est amené à chercher les entiers k, h appartenant à $[\![1\,;2^{52}-1]\!]$ vérifiant (2) et (3). On obtient facilement l'équivalence : (2) et (2) si et seulement si

$$x_k = \frac{2k+\frac{1}{2}}{1+k\varepsilon} < h < \frac{1}{2} + \frac{2k}{1+k\varepsilon} = y_k \qquad (4)$$

On est donc ramené à chercher les entiers n, tel que l'intervalle $[x_k\,;y_k]$ avec $k = \frac{1}{\varepsilon}\left(n2^{-\lfloor \log_2(n)\rfloor} - 1\right)$ contienne un entier (il en contient au plus 1 puisque $y_k - x_k = \frac{1}{2}(1-\frac{1}{x}) < \frac{1}{2}$) ; le test $x_k < \lfloor y_k \rfloor$ est commode.

Un programme Python utilisant les flottants Python pour déterminer les n ne fonctionne pas car on est en limite de précision (k est très grand). En utilisant le module decimal, ou le module fraction, pour travailler en arithmétique non bornée, on retrouve la liste des échecs, par exemple entre 1 et 200 :

49 98 103 107 161 187 196 197

Annexe 1 : recherche du d.d.i. de $\frac{1}{n}$ pour n impair

On part d'un théorème d'Euler ; puisque 2 et n sont premier entre eux, $2^{\varphi(n)} \equiv 1 \mod n$ (φ est l'indicatrice d'Euler). Donc il existe un entier M_n tel que $2^{\varphi(n)} = 1 + nM_n$ qui donne $\frac{1}{n} = \frac{M_n}{2^{\varphi(n)}-1}$, puis

$$\frac{1}{n} = M_n 2^{-\varphi(n)} \frac{1}{1-2^{-\varphi(n)}} = M_n \sum_{k=1}^{+\infty} 2^{-k\varphi(n)}$$

avec $M_n = \frac{2^{\varphi(n)}-1}{n}$; comme $M_n < 2^{\varphi(n)}$, le nombre m_n de bits dans l'écriture binaire de M_n est plus petit que $\varphi(n)$ et donc les termes de la série

$$\sum_{k=1}^{+\infty} M_n 2^{-k\varphi(n)}$$

ne se chevauchent pas ; il n'y a pas de retenue, la sommation se réduisant à une concaténation ; on en déduit (en confondant M_n et la suite des bits de son écriture binaire) :

$$\frac{1}{n} = (0, \underbrace{\overline{0\ \cdots\ 0}}_{\varphi(n)-m_n}\ M_n)_{\mathbf{b}}$$

On voit tout de suite que cette technique ne fournit la période optimale que dans le cas où il n'existe pas de diviseur d de $\varphi(n)$ tel que $2^d \equiv 1 \mod n$.

Ainsi avec $n = 49 = 7^2$, on a $\varphi(n) = 7 \times \varphi(7) = 7 \times 6 = 42$, mais on a $49 \mid 2^{21} - 1$ donc on remplace $\varphi(n)$ par 21.

$$M_{49} = \frac{2^{21}-1}{49} = 42\,799 = (1010011100101111)_{\mathbf{b}}$$

comme il n'y que 16 bits, on ajoute 5 zéros en tête et

$$\frac{1}{49} = (0, \overline{00000101001110010111 1})_{\mathbf{b}}.$$

Pour $\frac{1}{10}$, on part de $\frac{1}{5}$; $\varphi(5) = 4$ et $\frac{2^4-1}{5} = 3 = (11)_{\mathbf{b}}$ donc $\frac{1}{5} = 0,\overline{0011}_{\mathbf{b}}$; puis $\frac{1}{10} = 0,\overline{00011}_{\mathbf{b}}$.

Annexe 2 : la division par dix

La division d'un flottant par dix est un vieux problème dans un contexte binaire, lié à celui de la conversion binaire vers décimal dont on a parlé plus haut. Il est amusant de constater que les premiers algorithmes utilisaient le fait que diviser par dix, c'est multiplier par $0,1$ ou du moins par une valeur approchée de $0,1$ puisque $0,1$ n'est pas dyadique. Knuth a proposé en 1969 un algorithme très simple n'utilisant qu'additions et décalages.
On donne l'algorithme pour un significande sur 64 bits.

Entrée : s significande sur 64 bits au plus
Sortie : $f \approx s/10$
Définitions : a = s+s/2 ; b = a+a/16 ; c = b+b/256 ; d = c+c/65536 ; e = d+d/4294967296 ; f = e/16

Pour faire fonctionner, notons d'abord que $16 = 2^4$, $256 = 2^8$, $65536 = 2^{16}$, $4294967296 = 2^{32}$; les divisions sont donc bien de simples décalages, donc calculées exactement ; ensuite, on a vu que $0,1 = 2^{-4} \times 1,\overline{1001}_{\mathbf{b}}$.
Déroulons les définitions : on peut prendre par homogénéité, $s = 1$.

Alors $b = 1 + 1/2 + 1/16 + 1/32 = 1,10011_{\mathbf{b}}$
puis

$$c = 1,10011_{\mathbf{b}} + 0,0000000110011_{\mathbf{b}} = 1,1001100110011_{\mathbf{b}},$$

puis

$$d = 1,1001100110011_{\mathbf{b}} + 0,00000000000000011001100110011_{\mathbf{b}}$$

donc
$d = 1,10011001100110011001100110011_{\mathbf{b}}$
et enfin $e = 1,\underbrace{1001\cdots1001}_{60\text{ bits}}1_{\mathbf{b}}$
et $f = 0,0001\underbrace{1001\cdots1001}_{60\text{ bits}}1_{\mathbf{b}}$
soit, en décimal :

$$f = 0,09999999999999999999945789891375724 \\ 77829962735995650291442871093 75$$

et (dans $\mathbb{D}$) $0,1 - f = 0,1 \times 2^{-64}$.
Le déroulé précédent permet de comprendre l'ingéniosité de l'algorithme.
Bien évidemment, l'algorithme s'applique parfaitement à une arithmétique flottante avec un significande à 53 bits. On vérifie que traduit en Python, cet algorithme divise par 10 comme Python, c.-à-d qu'appliqué à un flottant quelconque x, il donne le même résultat que $x/10$ ce que ne fait pas le produit par $0,1\ldots$

Le petit miracle de la multiplication par dix ($10 * (n/10) = n$) a été expliqué par William Kahan qui a montré que, plus généralement, si m est somme de deux puissances de deux, alors pour tout entier n, $n < 2^{52}$, on a (en arithmétique IEEE 754) $\mathrm{RN}\left(m * \mathrm{RN}(n/m)\right) = n$. Pour une démonstration, voir [11].

Annexe 3

On reprend l'algorithme donné à la page 57.

Soit $x \in]0\,;1[$; on définit deux suites récurrentes $x_0 = x$, $a_0 = 0$, $a_k = \lfloor 2x_{k-1} \rfloor$, $x_k = 2x_{k-1} - a_k$, pour tout $k \geqslant 1$. Le théorème de la suite récurrente permet de montrer que (a_k) est à valeurs dans $\{0\,;1\}$ et que (x_k) est à valeurs dans $[0\,;1[$.

On a vu que $\displaystyle\sum_{h=1}^{k} a_h 2^{-h} = x - x_k 2^{-k}$

donc $\displaystyle\sum_{h=1}^{+\infty} a_h 2^{-h} = x$.

L'auteur remercie François Bouyer pour sa relecture attentive.

François Boucher est tout jeune retraité de l'Éducation Nationale. Il a enseigné les mathématiques et l'informatique en CPGE au lycée Bellevue à Toulouse. boucherf@free.fr

Références

[1] David Monniaux. « The pitfalls of verifying floating-point computations ». In : *ACM Transactions on Programming Languages and Systems (TOPLAS)* Vol. 30.n° 3 (mai 2008). , p. 12.

[2] Neil J. A. Sloane. *On-Line Encyclopedia of Integer Sequences*. . 31 décembre 2017.

[3] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, p. 572.

[4] Jacques Arsac. *La construction de programmes structurés*. Paris : Dunod, 1977.

[5] Jean-Michel Muller. *Quels algorithmes ?* . 24 octobre 2011.

[6] Robert Munafo. *Survey of Floating-Point Formats*. .

[7] Mark Dickinson. .

[8] Richard Brent et Paul Zimmermann. « Modern Computer Arithmetic ». Version 0.5.9. In : *Cambridge Monographs on Computational and Applied Mathematics* n° 18 (octobre 2010).

[9] Rick Regan. *Exploring Binary*. .

[10] Christopher Vickery. *IEEE-754 Analysis*. .

[11] David Goldberg. « What Every Computer Scientist Should Know About Floating Point Arithmetic ». In : *ACM Computing Surveys* Vol. 23.n° 1 (1991). , pp. 5-48.

[12] Jacques Arsac. *Les bases de la programmation*. Dunod, 1983.

Sommaire du n° 527

La multiplication

APMEP

www.apmep.fr