Le calcul sur ordinateur

Frédéric Goualard, Christophe Jermann

Le calcul sur ordinateur

Frédéric Goualard et Christophe Jermann

Département d'Informatique, Nantes Université

LS2N UMR CNRS 6004

2023-06-15, v. 1.4

Je sais que deux et deux font quatre — et serais heureux de le prouver, aussi, si je le pouvais — bien que je dois dire que si, par un quelconque expédient, je pouvais transformer 2 *et* 2 *en cinq, cela me donnerait un bien plus grand plaisir.* — Lord Byron, In [1, p. 159], 1813–1814 (trad. de l'anglais par les auteurs).

Figure 1 : Poster de Yakov Guminer, 1931. Source : Wikimedia Commons.

[1] Lord Byron. *'Alas ! the love of Women !' Byron's Letters and Journals*. Sous la dir. de Leslie A. Marchand. T. 3. The Belknap Press of Harvard University Press, 1974

Peut-on tirer un 7 avec un dé à six faces ? Un nombre peut-il être différent de lui-même ? Si a et b sont deux entiers strictement positifs, leur produit peut-il être inférieur à a et b ? Autant de questions apparemment triviales qui se révèlent bien plus complexes lorsqu'un ordinateur entre en jeu.

Les nombres entiers et les nombres à virgule flottante — ou simplement, *flottants* — censés représenter les ensembles d'entiers naturels et relatifs ainsi que les nombres réels sur les ordinateurs n'en possèdent pas toutes les propriétés. Méconnaitre cet état de fait, c'est s'exposer à traduire servilement les méthodes mathématiques sur les entiers et les réels en des algorithmes aux résultats erronés et ainsi considérer l'arithmétique sur les ordinateurs comme de la magie, où chaque résultat ne découlerait pas logiquement du précédent. Nous décrivons dans ce fascicule les règles simples vérifiées par l'arithmétique entière et en virgule flottante sur les ordinateurs et montrons comment utiliser ces règles pour écrire de vraies traductions des méthodes mathématiques adaptées aux capacités des ordinateurs et établir des preuves sur les résultats.

Comment lire ce fascicule ? La section 1 justifie historiquement l'utilisation de la représentation binaire pour stocker et manipuler l'information dans un ordinateur. La section 2 présente la représentation des nombres entiers ; la section 2.1 rappelle la notion de représentation positionnelle des nombres ; elle est indispensable à la compréhension de la suite mais peut être omise par les lecteurs et lectrices qui la maîtrisent déjà. La section 3 présente les différentes représentations des nombres réels ; l'accent est mis sur la représentation à virgule flottante (section 3.2) suivant le standard IEEE 754. Les bonnes pratiques à adopter ainsi que la quantification *a priori* des erreurs de calcul sont présentées dans la section 3.2.2. L'annexe C présente une sélection d'exemples, réutilisables en classe, qui montrent les conséquences de l'utilisation des nombres flottants à la place des réels. Les exemples sont, pour l'essentiel, tirés des applications et outils potentiellement utilisés à l'école élémentaire, au collège, ou au lycée : calculatrice Numworks, langages Python 3 et Scratch, tableur LibreOffice Calc, … Sauf indication contraire, tous les exemples de code informatique sont écrits en Python 3. La version 20.4.0 du système Epsilon est utilisée pour tous les exemples sur la calculatrice Numworks. Le comportement de ces exemples peut changer sur des versions ultérieures du fait de la correction des erreurs qu'ils mettent en évidence.

Table des matières

1 Représentation de l'information

« […] le synchronisme des inventions atteste que le progrès d'un certain art a atteint un point où un pas donné devient inévitable […] cela montre que l'individu a moins d'importance dans l'invention que son environnement. » — William H. Babcock et P. B. Pierce, Cité *in* [2, p. 214], 1885 (trad. de l'anglais par les auteurs).

Notre histoire commence t-elle en 1937 dans la cuisine d'un mathématicien des *Bell Telephone Laboratories*[3] ? Ou la même année au *Massachusetts Institute of Technology* (MIT) ? Toujours en 1937, dans l'Illinois[4] ? Ou bien encore en 1936 en Allemagne ? Comme le remarque l'épigraphe de cette section, il peut être difficile d'associer une découverte à une seule personne, la science progressant souvent simultanément dans des lieux et par des chemins différents.

Jusqu'en 1936, le mot « *computer* » désignait le plus souvent un homme — ou plus justement, une femme[5] — chargé-e d'effectuer les nombreux calculs requis pour établir des trajectoires ballistiques, par exemple. Les calculateurs les plus sophistiqués étaient des ordinateurs analogiques mécaniques, comme l'analyseur différentiel de Vannevar Bush.

Intrigué par les parallèles possibles entre la logique booléenne et les relais électromécaniques (figure 2), George Stibitz (figure 3, droite), un mathématicien travaillant aux *Bell Labs* à New York, décida un soir de 1937 de ramener chez lui quelques relais et construisit dans sa cuisine sur une planche de bois un additionneur binaire (figure 4) : le *Model K* (*K*, pour *Kitchen*).

L'algèbre de Boole définit des fonctions sur deux quantités seulement : le 0 et le 1.

Table 1 – Quelques fonctions logiques atomiques.

x	non x
0	1
1	0

x	y	x ou y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x et y
0	0	0
0	1	0
1	0	0
1	1	1

À partir des fonctions logiques atomiques de la table 1, il est facile de définir des opérations arithmétiques sur des entiers représentés en base 2, ce que fit Stibitz pour des entiers de seulement 1 bit (*binary digit*).

Cette réalisation physique *ad hoc* par des relais électromécaniques d'une opération arithmétique *via* une traduction booléenne, un autre chercheur, Claude Shannon (figure 3, centre) l'avait théorisée et systématisée de manière indépendante dans une thèse de *Master*[6] du MIT la même année.

Relais électromécaniques et logique booléenne

Un relais électromécanique est composé d'un électro-aimant, piloté par un courant de contrôle, et deux lamelles reliées à un circuit électrique, le circuit contrôlé. La mise en contact de ces lamelles ferme ce circuit. Cette mise en contact est opérée en attirant une des lamelles mobile vers l'autre lamelle fixe grâce à l'électro-aimant.

En combinant les relais, il est possible de réaliser les opérations logiques de base :

[2] Robert K. Merton. *The Sociology of Science : Theoretical and Empirical Investigations*. Sous la dir. de Norman W. Storer. Chicago, IL : University of Chicago Press, sept. 1979

[3] Aujourd'hui renommé *Nokia Bell Labs*, après avoir été les *AT&T Bell Laboratories*, les *Bell Labs* sont à l'origine de centaines de découvertes et brevets dans de nombreux domaines de la science et de l'ingénierie.

[4] L'Atanasoff Berry Computer, bien que conceptualisé en 1937 sur une route menant à Rock Island dans l'Illinois, ne fut effectivement réalisé qu'en 1942.

[5] Jennifer S. Light. « When Computers Were Women ». In : *Technology and Culture* 40.3 (1999), p. 455-483

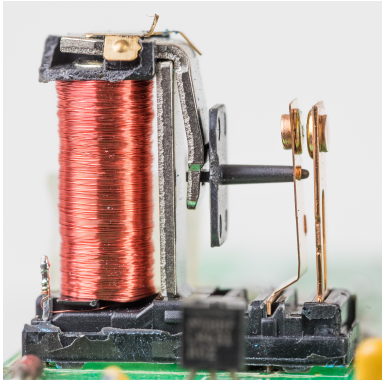


Figure 2 : Relais électromécanique (Henry, 1835). Source : Wikimedia.

Figure 3 : Konrad Zuse, 1910–1995 ; Claude Shannon, 1916–2001 ; George Stibitz, 1904–1995. Sources : Craftofcoding, Wikimedia, Wikimedia.

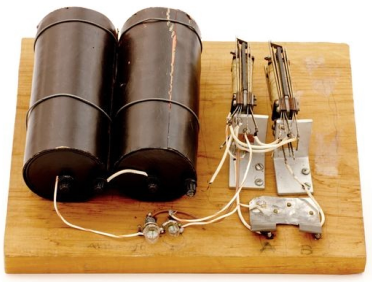


Figure 4 : Réplique du calculateur binaire de George Stibitz, 1937. Crédit photo : computerhistory.org.

[6] Claude Elwood Shannon. « A symbolic analysis of relay and switching circuits ». Mém. de mast. Massachusetts Institute of Technology, 1937

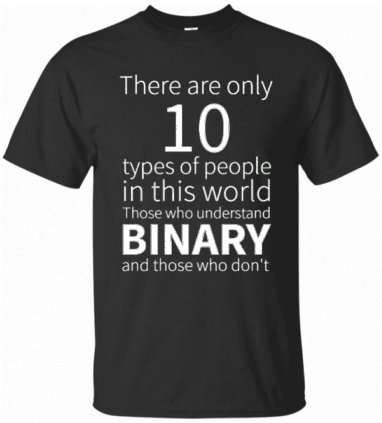
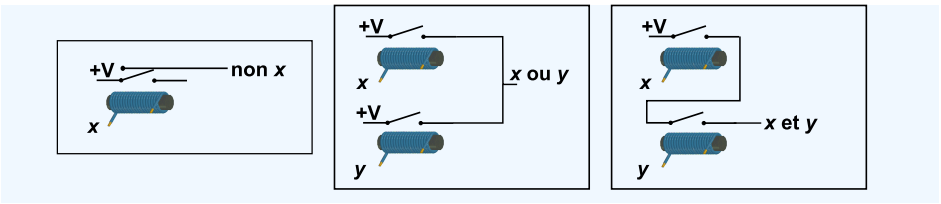


FIGURE 5 : Un tee-shirt jouant sur les ambiguïtés de la représentation positionnelle si l'on ne connait pas la base utilisée.



Trois ans auparavant, en 1934, Konrad ZUSE (figure 3, gauche) un ingénieur Berlinois, avait commencé à s'intéresser à la réalisation d'un calculateur automatique. Ses travaux étaient conduits en complète isolation, ce qui l'amena à mettre au point des solutions originales. En particulier, il inventa un système purement mécanique à base de plaques et de tiges de métal pour calculer physiquement des fonctions booléennes (figure 6). La combinaison de centaines de ces modules lui permit de créer le Z1, un ordinateur purement mécanique basé sur la logique booléenne et manipulant les nombres en binaire.

Le module mécanique de ZUSE, le relais électromécanique, puis, plus tard, le tube thermionique (« *lampes à vide* », DE FOREST, 1907), le transistor (SHOCKLEY, BARDEEN et BRATTAIN, 1947) et enfin, le circuit intégré (KILBY, 1958), opèrent tous suivant le même principe (figure 7) : un signal de contrôle détermine la présence ou l'absence d'un signal contrôlé (information binaire). L'homologie de ces dispositifs rend les travaux de SHANNON largement applicables quelle que soit la technologie utilisée et le choix d'une représentation de l'information en binaire, pratiquement inévitable sinon inéluctable. Certains ordinateurs « exotiques » ont cependant utilisé une représentation non binaire : le Setun[7] (1958) et le Setun 70 (1970), par exemple, étaient des ordinateurs de fabrication soviétique qui utilisaient une logique tri-valuée. Ils restent à ce jour des curiosités au regard de l'hégémonie des ordinateurs binaires.

Chaînes et interprétations

« *Il y a* 10 *types de personnes : ceux qui comprennent le binaire et les autres* » (figure 5). Cette blague d'informaticien perd évidemment beaucoup à être échangée oralement car elle repose sur l'ambiguïté possible de la représentation textuelle d'un nombre, le texte « 10 » pouvant être interprété comme dix en base 10 ou deux en base 2 (section 2.1).

Plus généralement, une *chaîne de caractères* (qu'elle soit limitée à une succession de chiffres ou pas) peut avoir — et aura, dans la suite de ce fascicule — de multiples interprétations. Afin de lever toute ambiguïté, nous utiliserons dans la suite trois notations :

- Une succession simple de chiffres décimaux représente une valeur exprimée en base 10, comme c'est habituellement l'usage. Exemple : 23 représente le nombre vingt-trois ;
- Une chaîne de caractères entre des guillemets anglais est un texte non évalué, sans valeur numérique définie. Exemple : “23” est la chaîne composée des symboles « 2 » et « 3 » ;
- Une chaîne de caractères suivie d'un indice en italique orange est interprétée numériquement en fonction de la convention définie par l'indice. Exemple : 0011_{XS-3} (voir plus loin le codage XS-3 par excès à 3) représente le nombre zéro. Lorsque l'indice est un entier b, la valeur calculée est obtenue par interprétation de la chaîne avec une représentation positionnelle en base b (voir section 2.1). Exemple : 23_{10} représente le nombre vingt-trois exprimé en base 10.

Ainsi, 23 représente la même *valeur* que 23_{10} mais est différent de “23”, qui n'a pas de valeur numérique définie. De même, “0011” est différent de “11”, même si la présence

[7] Boris Nikolaevitch MALINOVSKY. *Pioneers of Soviet Computing*. Sous la dir. d'Anne FITZPATRICK. Seconde éd. SIGCIS : The Special Interest Group for Computing, Information, et Society, 2010

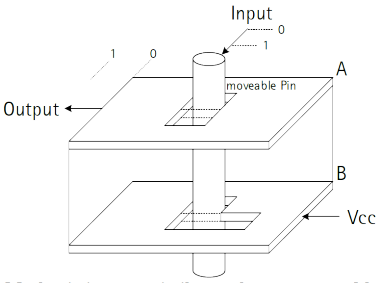


FIGURE 6 : Porte « *non* » mécanique de Konrad ZUSE (Source : DÜRRE *et al.*, WMSCI 2016[8]).

[8] Jan DÜRRE, Guillermo PAYÁ-VAYÁ et Holger BLUME. « Teaching Digital Logic Circuit Design via Experiment-Based Learning - Print your own Logic Circuit ». In : *Proceedings of the World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2016)*. Juill. 2016

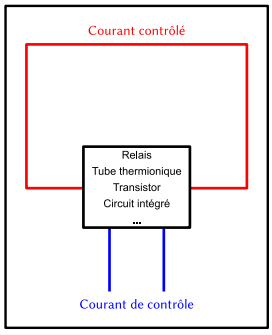


FIGURE 7 : Circuit pour l'mplémentation physique de la logique booléenne.

des zéros à gauche de la première chaîne n'a pas de sens dans une interprétation numérique de ces deux chaînes.

Malgré une utilisation quasi-universelle du binaire pour la manipulation de l'information, la représentation des nombres et le calcul dans les premiers ordinateurs étaient le plus souvent faits en décimal, cette base étant considérée comme plus naturelle. Les nombres en base 10 devaient alors être stockés dans un codage binaire adéquat pour être manipulés par des machines intrinsèquement binaires. Le *Complex Number Calculator*, construit par Stibitz en 1939 et employé en 1940 pour établir la première connexion à distance à un ordinateur, utilisait un codage *décimal-codé-binaire* par excès à 3, XS-3[9] : chaque chiffre décimal est représenté par une chaîne de quatre bits suivant la correspondance :

Chiffre décimal	XS-3	Chiffre décimal	XS-3
0	0011	5	1000
1	0100	6	1001
2	0101	7	1010
3	0110	8	1011
4	0111	9	1100

Un nombre décimal est représenté au format XS-3 par la chaîne binaire concaténant chacun des quadruplets de ses chiffres :

$$237_{10} = 0101\,0110\,1010_{\text{XS-3}}$$

Outre XS-3, il existe d'autres schéma de représentation *décimal-codé-binaire* (BCD, codage biquinaire, ...). Tous ont en commun une utilisation dispendieuse de l'espace mémoire et des algorithmes de calcul relativement compliqués.

En 1945, John von Neumann rédige un rapport[10] largement diffusé où il motive l'abandon du *décimal-codé-binaire*, quel que soit le schéma utilisé, pour une représentation en base 2 directe des nombres. Par la suite, l'écrasante majorité des ordinateurs adoptera ce principe, même si des instructions pour la manipulation des nombres en *décimal-codé-binaire* existent encore sur de nombreux processeurs pour des applications spécialisées.

1.1 Stockage et manipulation de l'information

Un ordinateur manipule des images, des sons, des vidéos, du texte, ... Toutes ces données sont stockées et manipulées sous forme de chaînes de bits de tailles variables.

Pour ce qui nous concerne, un ordinateur est essentiellement composé d'une mémoire (la mémoire RAM — *Random Access Memory*) contenant les données et les programmes sous forme de chaînes binaires, et d'un processeur chargé d'appliquer le programme sur les données (figure 8). Il existe différentes technologies de mise en œuvre de la mémoire RAM offrant des performances différentes. La modélisation reste cependant toujours la même : on considère la mémoire RAM comme un tableau dont chaque case peut contenir 8 bits[11] — un *octet* — (figure 9). Chaque case possède une *adresse* correspondant simplement à sa position dans le tableau à partir de l'indice 0. Dans la mémoire de la figure 9, chaque case possède une adresse sur 32 bits, ce qui fait 2^{32} adresses possibles, de la case 0 à la case $2^{32} - 1$.

Une mémoire dont chaque case contient huit bits est dite *adressable par octet*. L'octet est alors l'unité atomique : on ne peut pas lire ou écrire moins d'un octet à la fois. Une donnée représentable sur moins de 8 bits occupe une case mémoire entière. À l'inverse, une donnée requérant plus de 8 bits occupera plusieurs cases contiguës de la mémoire. Pour des raisons techniques, le nombre de cases occupées par un objet d'un type simple sera toujours une puissance de 2 : 1, 2, 4, ou 8 cases, en général. L'adresse associée à la donnée est alors celle de la case d'adresse la plus basse (En réalité, cette convention

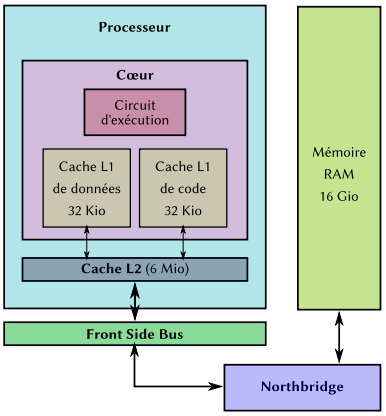


Figure 8 : Les éléments constitutifs d'un ordinateur mono-cœur : mémoire et processeur.

[9] Michael R. Williams. *A history of computing technology*. 2nd ed. Los Alamitos, Calif : IEEE Computer Society Press, 1997

[10] John von Neumann. *First draft of a report on the EDVAC*. Technical Report W-670-ORD-4926. Moore School of Electrical Engineering, University of Pennsylvania, juin 1945

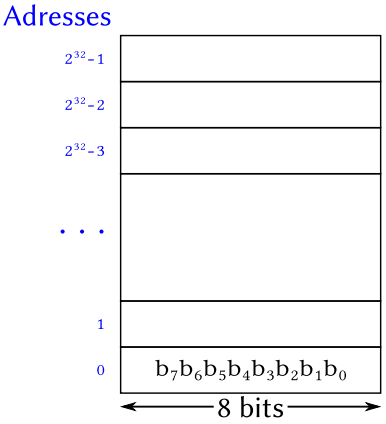


Figure 9 : Représentation d'une mémoire RAM de 4 GiB.

[11] Certains ordinateurs, très peu nombreux, possèdent une mémoire dont chaque case fait seulement 4 bits, ou au contraire 16 ou 32 bits. On ne les considérera pas ici.

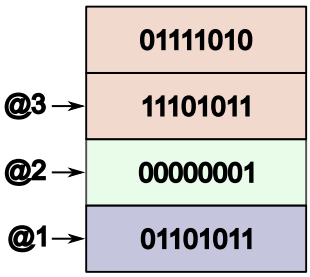


Figure 10 : Stockage en mémoire des données 01101011, 00000001 et 0111101011101011.

dépend de l'endianness du processeur utilisé. On adopte ici la convention en vigueur sur les processeurs de type Intel x86, largement répandus).

La figure 10 illustre le stockage en mémoire de trois données : une chaîne de 8 bits, à l'adresse `@1` ; un seul bit valant 1, à l'adresse `@2` (le reste de la case mémoire est rempli avec des zéros) ; une chaîne de 16 bits stockée à partir de l'adresse `@3`. Notez qu'avec les conventions choisies, cette dernière est découpée en deux blocs de 8 bits et le bloc de gauche (*octet de poids fort*) est stocké à l'adresse la plus haute alors que le bloc de droite (*octet de poids faible*) se trouve à l'adresse la plus basse.

Nous verrons dans les sections suivantes l'impact de ce découpage de l'information en blocs de tailles fixes sur la représentation des données et les calculs.

2 Les nombres entiers

En dépit de la longue tradition de construire des machines numériques dans le système décimal, nous sommes très fortement en faveur du système binaire pour notre appareil. Notre unité atomique de mémoire est naturellement adaptée au système binaire, comme nous ne cherchons pas à mesurer des charges graduées en un point particulier du sélectron mais nous contentons de distinguer deux états. — Arthur W. Burks, Herman H. Goldstine et John von Neumann, In [12], 1946 (trad. de l'anglais par les auteurs).

[12] Arthur W. Burks, Herman H. Goldstine et John von Neumann. *Preliminary Discussion of the logical design of an electronic computing instrument.* Technical Report. The Institute for Advanced Study, juin 1946

D'un trait supplémentaire de son calame sur la tablette d'argile, un berger sumérien de la troisième dynastie d'Ur (22e–21e s. AEC) pouvait facilement ajouter une nouvelle brebis à son maigre troupeau : un trait pour chaque brebis, cela ne faisait guère que quelques dizaines de traits pour les bergers les plus riches. Avec l'avènement de cités-États de plus en plus grandes et riches, cette manière de comptabiliser les ressources atteint vite ses limites.

Figure 11 : Écriture cunéiforme sur une tablette d'argile à l'aide d'un calame.
Crédit photo : Shevlin Sebastian.

Vers 2000 AEC, les babyloniens[13] mirent au point un moyen astucieux d'écrire des grands nombres sur leurs tablettes sans avoir besoin d'un nombre important de symboles. Ils utilisaient le symbole « 𒁹 » pour la valeur « 1 » et le symbole « 𒌋 » pour la valeur « 10 ». En combinant ces deux symboles, ils représentaient tous les entiers entre 1 et 59 (Table 2).

[13] Stephen Chrisomalis. *Numerical notation : a comparative history.* Cambridge, New York : Cambridge University Press, 2010

Table 2 – Représentation babylonienne des entiers de 1 à 59.

𒁹	1	𒌋𒁹	11	𒐐𒁹	21	𒐐𒐐𒁹	31	𒐖𒁹	41	𒐖𒌋𒁹	51
𒁹𒁹	2	𒌋𒁹𒁹	12	𒐐𒁹𒁹	22	𒐐𒐐𒁹𒁹	32	𒐖𒁹𒁹	42	𒐖𒌋𒁹𒁹	52
𒁹𒁹𒁹	3	𒌋𒁹𒁹𒁹	13	𒐐𒁹𒁹𒁹	23	𒐐𒐐𒁹𒁹𒁹	33	𒐖𒁹𒁹𒁹	43	𒐖𒌋𒁹𒁹𒁹	53
𒁹𒁹𒁹𒁹	4	𒌋𒁹𒁹𒁹𒁹	14	𒐐𒁹𒁹𒁹𒁹	24	𒐐𒐐𒁹𒁹𒁹𒁹	34	𒐖𒁹𒁹𒁹𒁹	44	𒐖𒌋𒁹𒁹𒁹𒁹	54
𒁹𒁹𒁹𒁹𒁹	5	𒌋𒁹𒁹𒁹𒁹𒁹	15	𒐐𒁹𒁹𒁹𒁹𒁹	25	𒐐𒐐𒁹𒁹𒁹𒁹𒁹	35	𒐖𒁹𒁹𒁹𒁹𒁹	45	𒐖𒌋𒁹𒁹𒁹𒁹𒁹	55
𒁹𒁹𒁹𒁹𒁹𒁹	6	𒌋𒁹𒁹𒁹𒁹𒁹𒁹	16	𒐐𒁹𒁹𒁹𒁹𒁹𒁹	26	𒐐𒐐𒁹𒁹𒁹𒁹𒁹𒁹	36	𒐖𒁹𒁹𒁹𒁹𒁹𒁹	46	𒐖𒌋𒁹𒁹𒁹𒁹𒁹𒁹	56
𒁹𒁹𒁹𒁹𒁹𒁹𒁹	7	𒌋𒁹𒁹𒁹𒁹𒁹𒁹𒁹	17	𒐐𒁹𒁹𒁹𒁹𒁹𒁹𒁹	27	𒐐𒐐𒁹𒁹𒁹𒁹𒁹𒁹𒁹	37	𒐖𒁹𒁹𒁹𒁹𒁹𒁹𒁹	47	𒐖𒌋𒁹𒁹𒁹𒁹𒁹𒁹𒁹	57
𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	8	𒌋𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	18	𒐐𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	28	𒐐𒐐𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	38	𒐖𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	48	𒐖𒌋𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	58
𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	9	𒌋𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	19	𒐐𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	29	𒐐𒐐𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	39	𒐖𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	49	𒐖𒌋𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	59
𒌋	10	𒐐	20	𒐐𒐐	30	𒐖	40	𒐖𒐐	50		

Pour représenter un nombre supérieur ou égal à 60, ils considéraient le nombre de paquets de 60 qu'il contenait. Par exemple, le nombre 63 contient un paquet de 60 et trois unités. Sa représentation était donc 𒁹 𒐐𒐐𒐐 : le nombre d'unités se trouve à droite et le nombre de paquets de 60 à gauche. En considérant chacun des cinquante-neuf graphismes de la table 2 comme un symbole unique, on peut ainsi représenter avec deux symboles tous les entiers entre 60 et $59 \times 60 + 59 = 3599$. La représentation de 3599 est

simple : 𒌋𒐗 𒌋𒐗. On constate qu'il y a cependant un problème pour représenter 60 car il se décompose en $1\times60+0$, or les babyloniens n'avaient pas de représentation du zéro. Ils le remplaçaient par un espace dans la notation ; 60 s'écrit donc 𒐗, soit exactement comme 1, et c'est le seul contexte qui permet de différencier les deux.

Pour représenter les nombres au-delà de 3599, il suffit de faire des paquets de $60\times60=3600$, puis des paquets de 60^3, 60^4, ... Le nombre $2\,808\,307 = 13\times60^3+0\times60^2+5\times60^1+7\times60^0$ s'écrit, par exemple, 𒌍𒐜 𒐖 𒐖. Notez l'espace plus important entre la représentation du 13 (𒌍𒐜) et celle du 5 (𒐖) pour représenter l'absence de paquet de 60×60.

L'absence de symbole spécifique pour représenter le zéro devait singulièrement compliquer la tâche des scribes chargés de relire les valeurs inscrites sur les tablettes : un espace entre les symboles représente t-il une absence de puissance ou est-ce seulement un décalage malheureux lors de l'impression de la tablette ? Malgré cela, le zéro semble être apparu très tardivement car les preuves matérielles les plus anciennes de son existence datent des alentours de 875 EC en Inde.

2.1 Représentation positionnelle

Si l'on fait abstraction de l'absence de zéro dans le système babylonien, ce système de représentation des nombres correspond à ce que l'on appelle aujourd'hui une *représentation positionnelle en base 60* : chaque nombre s'exprime sous la forme d'une chaîne de symboles, le symbole le plus à droite représentant le nombre d'unités (ou, le nombre de fois où 60^0 apparait dans le nombre), le symbole à sa gauche représente le nombre de fois où 60^1 apparait, le suivant le nombre de fois où 60^2 apparait, ... La valeur d'un même symbole dépend donc de sa position dans la chaîne.

Si l'on remplace la base 60 par la base 10 et que l'on introduit un symbole pour le zéro, de façon à ne plus devoir ajouter des espaces dans les représentations des nombres, on n'a plus besoin que de dix symboles pour représenter n'importe quel nombre. Dans la base 10, le symbole le plus à droite représente le nombre d'unités (le nombre de fois où 10^0 apparait), le symbole à sa gauche les *dizaines* (le nombre de fois où 10^1 apparait), etc.

En utilisant le symbole moderne « 0 » pour le zéro et les neuf premiers symboles de la table 2, on pourrait représenter le nombre $307 = 3\times10^2+0\times10^1+7\times10^0$ par :

𒌍	0	𒐖
10^2	10^1	10^0

En remplaçant les symboles babyloniens par les chiffres arabes 1, 2, 3, 4, 5, 6, 7, 8 et 9, on retrouve la notation des nombres dont nous avons l'habitude.

On peut choisir de représenter les nombres entiers dans n'importe quelle base. Pour une base b donnée, on doit posséder b symboles différents et la représentation d'un nombre dans la base se fait en déterminant le nombre de paquets de chaque puissance de la base.

Exemple — Représentation en base 7

Déterminons la représentation de 357_{10} en base 7. Pour cela, on va utiliser les sept symboles 0, 1, 2, 3, 4, 5 et 6. On a :

$$357_{10} = \mathbf{1}\times7^3+\mathbf{0}\times7^2+\mathbf{2}\times7^1+\mathbf{0}\times7^0$$

La représentation en base 7 de 357_{10} est donc 1020_7.

La base 10 est la base communément utilisée aujourd'hui, même si la base 60 est encore rencontrée dans certains domaines ; c'est par exemple le cas pour exprimer une

durée en heures, minutes et secondes : lorsque l'on écrit un temps sous la forme :

$$04 : 13 : 07$$

c'est à dire quatre heures, treize minutes et sept secondes, cela correspond à :

$$4 \times 60^2 + 13 \times 60^1 + 7 \times 60^0$$

secondes.

Comme vu dans la section précédente, les mécanismes internes des ordinateurs justifient l'emploi de la base 2 plutôt que de la base 10 en leur sein. Une autre base communément utilisée en informatique est la base 16, car 16 étant une puissance de 2, il est très facile de passer d'une base à l'autre et la représentation en base 16 d'un nombre requiert quatre fois moins de symboles qu'en base 2.

Table 3 – Correspondance symboles hexadécimaux / chaînes de quatre bits.

Chaîne binaire	Symbole hexadécimal	Valeur décimale	Chaîne binaire	Symbole hexadécimal	Valeur décimale
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Le passage de la base 2 à la base 16 se fait en considérant les chiffres binaires par paquets de quatre de la droite vers la gauche du nombre à recoder et en remplaçant chaque groupe par un des seize symboles hexadécimaux, conformément à la table 3. Notez que, la base 16 nécessitant seize symboles différents, on est obligé de trouver six symboles à rajouter aux dix chiffres arabes. Par convention, on utilise les lettres de « A » à « F », en minuscule ou majuscule indifféremment.

Exemple — Passage de la base 2 à la base 16

Le nombre binaire 1011010_2 se recode en base 16 en regroupant les bits par paquets de quatre *en partant de la droite* et en ajoutant des zéros à gauche si besoin, puis en appariant chaque groupe avec sa représentation hexadécimale :

$$\underbrace{0101}_{5}\ \underbrace{1010}_{\mathrm{A}}$$

La représentation hexadécimale de 1011010_2 est donc $5A_{16}$.

Le passage de la base 16 à la base 2 se fait de la même manière : chaque chiffre hexadécimal est remplacé par une chaîne de quatre bits conformément à la table 3.

Exemple — Passage de la base 16 à la base 2

Le nombre hexadécimal $3A7_{16}$ est représenté en binaire en concaténant les chaînes de quatre bits pour chaque symbole hexadécimal :

$$\overbrace{0011}^{3}\ \overbrace{1010}^{\mathrm{A}}\ \overbrace{0111}^{7}$$

La représentation binaire de $3A7_{16}$ est donc 001110100111_2.

Le passage d'une base b_1 à une base b_2 lorsqu'il n'existe pas de relation spéciale entre b_1 et b_2 demande un peu plus de calculs. En pratique, on s'intéressera seulement au passage de la base 10 à une base b ou d'une base b à la base 10. La maîtrise de ces deux techniques de recodage permet évidemment de passer indirectement d'une base b_1 à une base b_2 en transitant par la base 10.

2.1.1 Passage d'une base b à la base 10

Considérons un nombre quelconque x exprimé dans la base b. Soit :

$$d_k d_{k-1} \cdots d_1 d_0{}_b$$

la représentation de x en base b. Comme on l'a vu précédemment, cela veut dire que l'on a :

$$x = d_k \times b^k + d_{k-1} \times b^{k-1} + \cdots + d_1 \times b^1 + d_0 \times b^0 \tag{1}$$

En remplaçant chaque symbole d_i par sa valeur, il est possible d'effectuer le calcul de l'équation (1) en base 10 pour obtenir la valeur de x en base 10.

Exemple — Passage de la base 5 à la base 10

Soit le nombre x exprimé en base 5 par la chaîne 321014_5. On a :

$$\begin{aligned} x &= 3 \times 5^5 + 2 \times 5^4 + 1 \times 5^3 + 0 \times 5^2 + 1 \times 5^2 + 4 \times 5^0 \\ &= 10779_{10} \end{aligned}$$

Exemple — Passage de la base 31 à la base 10

Comme pour la base 16, on utilise les vingt-et-unes lettres de A à U en plus des chiffres de 0 à 9 pour avoir trente-et-un symboles. La lettre A vaut 10, la lettre B vaut 11, ...jusqu'à la lettre U qui vaut 30.

Le nombre x exprimé en base 31 par la chaîne HELLO_{31} s'exprime en base 10 par :

$$\begin{aligned} x &= \mathsf{H} \times 31^4 + \mathsf{E} \times 31^3 + \mathsf{L} \times 31^2 + \mathsf{L} \times 31^1 + \mathsf{O} \times 31^0 \\ &= 17 \times 31^4 + 14 \times 31^3 + 21 \times 31^2 + 21 \times 31^1 + 24 \times 31^0 \\ &= 16137787_{10} \end{aligned}$$

2.1.2 Passage de la base 10 à la base b

On connait désormais un nombre x en base 10 et on souhaite l'exprimer dans une base b. En base b, x s'écrit sous la forme :

$$x = \cdots d_k d_{k-1} \cdots d_1 d_0{}_b$$

On cherche à trouver les symboles $d_0, \cdots, d_{k-1}, d_k, \cdots$.

Par définition, on a :

$$\begin{aligned} x &= \cdots d_k d_{k-1} \cdots d_1 d_0{}_b \\ &= \cdots + d_k \times b^k + d_{k-1} \times b^{k-1} + \cdots + d_1 \times b^1 + d_0 \times b^0 \end{aligned}$$

En considérant la partie droite de la deuxième équation comme un polynôme en la variable b, on peut réécrire l'équation avec une forme de Horner :

$$\begin{aligned} x &= \cdots + d_k \times b^k + d_{k-1} \times b^{k-1} + \cdots + d_1 \times b^1 + d_0 \times b^0 \\ &= ((\cdots ((((\cdots + d_k) \times b + d_{k-1}) \times b + d_{k-2}) \times b + \cdots) \times b + d_1) \times b + d_0 \end{aligned}$$

Exemple — Représentation sous forme de Horner

Le nombre 72431_{10} vaut :

$$\begin{aligned} 72431_{10} &= 7 \times 10^4 + 2 \times 10^3 + 4 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 \\ &= (((7 \times 10 + 2) \times 10 + 4) \times 10 + 3) \times 10 + 1 \end{aligned}$$

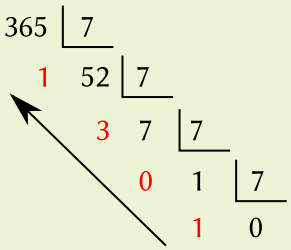
Considérons la forme factorisée :

$$x = ((\underline{\cdots ((((\cdots + d_k) \times b + d_{k-1}) \times b + d_{k-2}) \times b + \cdots)} \times b + d_1) \times b + d_0$$

On voit que si l'on divise x par b en suivant les règles de la base 10, on obtient un quotient correspondant à l'expressions soulignée en bleu et un reste égal à d_0. De la même façon, si l'on divise ce quotient par b de nouveau, on obtient un quotient correspondant à l'expression soulignée en rouge et un reste égal à d_1. En continuant à diviser chaque quotient successif par b jusqu'à obtenir un quotient nul, on peut ainsi obtenir en reste chacun des symboles servant à représenter x en base b. Notez cependant que l'on obtient les « chiffres » de x à l'envers : le premier « chiffre » trouvé est celui le plus à droite (les unités).

Exemple — Conversion de base 10 en base 7

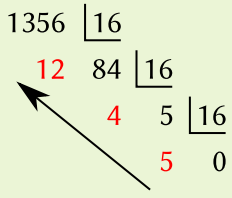
On cherche la représentation en base 7 de 365_{10}. On effectue des divisions successives par 7 :



On en déduit 365_{10}=1031_{7}.

Exemple — Conversion de base 10 en base 16

On veut convertir le nombre 1356_{10} en hexadécimal. On effectue les divisions successives par 16 :



On doit ensuite convertir tous les restes suivant la table 3 pour que chaque reste corresponde à un seul symbole. Il vient : 1356_{10}=$54C_{16}$.

Passage d'une base b_d à une base b_a

Le passage par la base 10 pour écrire en base b_a un nombre exprimé en base b_d n'est pas obligatoire et il est tout à fait possible de passer de la base b_d à la base b_a directement. Cependant, cela suppose de faire les calculs dans une base non décimale, ce qui n'est pas toujours aisé.

Ainsi, on peut passer de la base b_d à la base b_a :

— Par des multiplications itérées par b_d dans la base b_a :

$$\begin{aligned} 234_5 = 2\times5^2+3\times5^1+4\times5^0 &= (2_7\times5_7+3_7)\times5_7+4_7\\ &= (13_7+3_7)\times5_7+4_7\\ &= 16_7\times5_7+4_7\\ &= 122_7+4_7\\ &= 126_7 \end{aligned}$$

Pour le passage de la base 5 à la base 7, on doit faire les additions et les multiplications en base 7 ;

— Par des divisions par b_a itérées dans la base b_d :

$$\begin{array}{llll} 234_5 & \vert\ 12_5 & & \\ \underline{114_5} & 14_5 & \vert\ 12_5 & \\ 11_5 & \underline{2_5} & 1_5 & \vert\ 12_5 \\ & & \underline{1_5} & 0_5 \end{array}$$

Après avoir effectué toutes les divisions en base b_d (d'où l'utilisation de 12_5 plutôt que 7_{10}), il faut exprimer tous les restes en base b_a. Le nombre 11_5 devient donc 6_7 et l'on a comme avant :

$$234_5 = 126_7$$

2.2 Les entiers non signés

Comme on l'a évoqué dans la section 1, à partir des travaux de von Neumann[14], puis de Buchholz[15], on décide d'utiliser la représentation positionnelle en base 2 pour stocker et manipuler les nombres entiers en machine. Cela simplifie grandement la mise au point des *unités arithmétiques et logiques* (UAL) — les circuits spécialisés dans le calcul sur les entiers — car les opérations arithmétiques binaires sont facilement traduisibles en opérations logiques.

Les règles de base de l'addition binaire sont :

$$\begin{aligned} 0+0&=0\\ 0+1&=1\\ 1+0&=1\\ 1+1&=0, \text{ avec une retenue de } 1 \end{aligned}$$

Considérons alors, par exemple, la somme de deux nombres de 8 bits exprimés en base 2 :

[14] John von Neumann. *First draft of a report on the EDVAC*. Technical Report W-670-ORD-4926. Moore School of Electrical Engineering, University of Pennsylvania, juin 1945

[15] W. Buchholz. « Fingers or fists ? (the choice of decimal or binary representation) ». In : *Communications of the ACM* 2.12 (déc. 1959), p. 3-11

$$\begin{array}{rr} & 11010101_2 \\ + & 01110010_2 \\ & 111 \quad \textit{(retenues)} \\ \hline & 101000111_2 \end{array} \tag{2}$$

Soit $\mathbb{B}$ le domaine des booléens. La fonction logique :

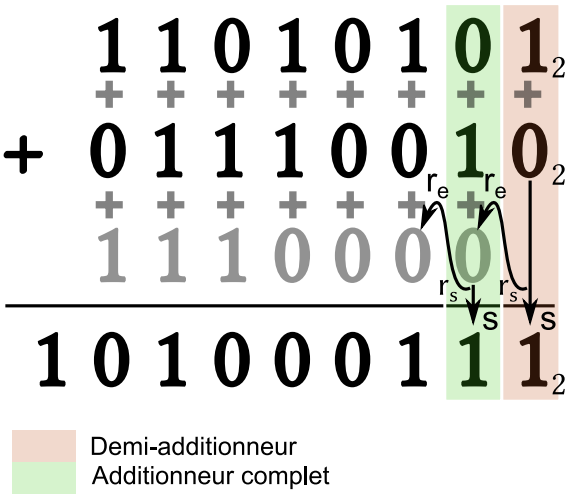
$$\begin{aligned} \text{sum} \colon \mathbb{B}^2 &\to \mathbb{B}^2 \\ (x, y) &\mapsto (s, r) \end{aligned}$$

avec s la somme des bits x et y, et r la retenue générée par la somme, peut se définir par :

$$\begin{aligned} s &= x \text{ xor } y \\ r &= x \text{ et } y \end{aligned}$$

où « xor » est le « *ou exclusif*[16] ».

Le circuit de la figure 12 montre une implémentation possible de la fonction « sum. » Il s'agit d'un *demi-additionneur*. Pour réaliser l'addition de deux nombres binaires, on a besoin d'un *additionneur complet* $\text{sum}_2(x, y, r_e)$ capable de faire la somme des bits x et y mais aussi du bit de retenue r_e provenant de la droite :



La colonne la plus à droite requiert seulement un demi-additionneur ; les colonnes suivantes demandent de faire la somme de trois bits ; pour la colonne i, on doit faire $x_i + y_i + r_{ei}$, où $r_{ei} = r_{si-1}$ (la retenue entrante correspond à la retenue sortante de la colonne immédiatement à droite).

Un additionneur complet s'obtient simplement à partir de deux demi-additionneurs (figure 13).

Ici, la somme de deux nombres de 8 bits présentée par l'équation (2) donne un résultat sur 9 bits. Plus généralement, la somme de deux nombres sur k bits peut donner un résultat sur $k + 1$ bits. On a vu dans la section 1.1 que les données sont stockées en mémoire et manipulées suivant des formats rigides dont la taille est une puissance de 2. Chaque donnée possède un *type* associé. Un *type* correspond à un ensemble de valeurs que peut prendre une variable. Il a une taille, qui correspond au nombre de bits (ou plus généralement d'octets) nécessaires à la représentation en mémoire d'un objet de l'ensemble. Deux valeurs entières de 8 bits ont un type d'une taille d'un octet. Si le résultat de leur somme est du même type, ce qui est habituellement le cas, on doit abandonner un bit du résultat de la somme (2). C'est le bit le plus à gauche (bit de poids fort) qui est supprimé[17]. Le résultat de la somme de 11010101_2 et 01110010_2 retourné à l'utilisateur sera donc la valeur 01000111_2. On a là un *dépassement de capacité*.

[16] La table de vérité du *ou exclusif* est :

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

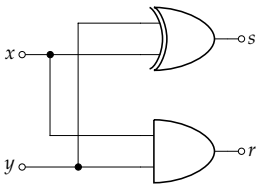


FIGURE 12 : Mise en œuvre de la fonction $\text{sum}(x, y)$.

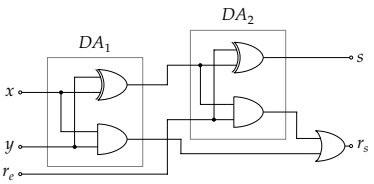


FIGURE 13 : Mise en œuvre de la fonction $\text{sum}_2(x, y, r_e)$.

[17] Pour être plus précis, le bit surnuméraire n'est pas supprimé mais déplacé dans un indicateur du processeur arithmétique, le *Carry Flag*, dont il est possible de connaître la valeur.

2.2.1 Les types

La notion de type est présente de manière plus ou moins explicite dans la plupart des langages de programmation.

Python, par exemple, est un langage fortement typé car chaque valeur possède un type qui détermine ce que l'on peut en faire :

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> 3+"a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Contrairement à de nombreux langages de programmation, les entiers manipulés en Python peuvent être représentés par un nombre variable d'octets en mémoire. Ainsi, lorsqu'une opération crée un résultat comportant plus de bits que les opérandes, Python ajuste la taille en conséquence pour pouvoir le représenter correctement. Cela facilite grandement la vie du programmeur mais a un coût important sur les performances par rapport à l'utilisation d'un type entier de taille fixe car tous les calculs sont désormais faits avec du logiciel, lorsque les calculs avec entiers de tailles fixes (8, 16, 32, ou 64 bits) sont faits directement par du matériel dans le processeur. La section A en annexe présente plus en détails l'implémentation des entiers en Python.

Lorsque de bonnes performances sont nécessaires, on peut utiliser la bibliothèque Python Numpy, qui offre des types entiers de tailles fixes comme ceux que l'on peut trouver dans d'autres langages :

- `uint8` : entier sur 8 bits ;
- `uint16` : entier sur 16 bits ;
- `uint32` : entier sur 32 bits ;
- `uint64` : entier sur 64 bits.

Ces types permettent de représenter des entiers *non signés*, c'est à dire seulement positifs ou nuls. La taille du type détermine le domaine des valeurs possibles. Comme la représentation interne est binaire, pour une taille de k bits, on peut représenter tous les entiers dans le domaine $[0, 2^k - 1]$.

À l'aide des types de Numpy, on peut mettre en évidence l'impact d'une représentation de taille fixe sur les calculs. Considérons, par exemple, le type `uint8`. Il permet de représenter tous les entiers dans le domaine $[0, 2^8 - 1] = [0, 255]$. Tant que l'on reste dans ce domaine, tous les calculs sont corrects :

```
>>> import numpy
>>> numpy.uint8(4) + numpy.uint8(5)
9
```

On peut sortir du domaine en calculant un nombre trop grand :

```
>>> numpy.uint8(255) + numpy.uint8(1)
<stdin>:1: RuntimeWarning: overflow encountered in ubyte_scalars
0
```

La somme de 255 et 1 donne 256, que l'on ne peut pas représenter sur 8 bits. Python nous en avertit par un message[18] et retourne la valeur 0. Cette valeur n'est pas arbitraire : comme on l'a vu précédemment, elle correspond au résultat de l'opération amputé de son bit de gauche surnuméraire. On a :

[18] Notez que, par défaut, Python n'affiche un avertissement que la première fois.

$$\begin{array}{rr} & 11111111_2 \quad (255_{10}) \\ + & 00000001_2 \quad (1_{10}) \\ \hline & 100000000_2 \end{array}$$

Ainsi, toutes les opérations sur 8 bits sont faites modulo 256 :

```
>>> numpy.uint8(235) + numpy.uint8(53)
32
```

Que se passe t-il si l'on effectue un calcul dont le résultat est négatif :

```
>>> numpy.uint8(4) - numpy.uint8(7)
253
```

La soustraction des opérandes est faite en binaire[19] :

$$\begin{array}{rr} & 00000100_2 \quad (4_{10}) \\ - & 00000111_2 \quad (7_{10}) \\ & 1111111 \quad \text{(retenues)} \\ \hline & 111111101_2 \quad (253_{10}) \end{array}$$

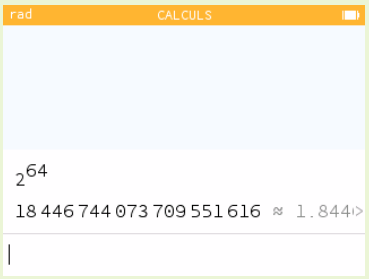
Le bit surnuméraire n'est pas considéré pour la valeur du résultat.

[19] Les règles de la soustraction binaire sont :

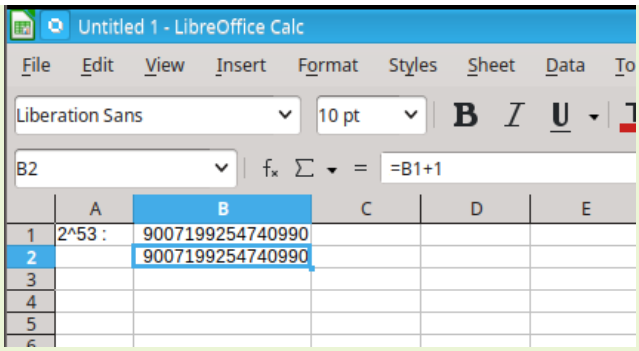
- $0 - 0 = 0$
- $0 - 1 = 1$, avec une retenue de 1
- $1 - 0 = 1$
- $1 - 1 = 0$

Exemple — Les entiers dans les applications

Comme Python, la calculatrice Numworks représente les entiers avec une précision variable, ce qui évite les problèmes de dépassement de capacité que l'on peut rencontrer avec les types entiers de Numpy :



En général, les tableurs représentent toutes les valeurs, même entières, avec des nombres en virgule flottante (voir Section 3.2). Contrairement à l'intuition, l'ensemble des nombres flottants censé représenter les nombres réels en machine, n'est pas un sur-ensemble des nombres entiers. On peut alors observer des comportements amusants lorsque l'on cherche à utiliser des entiers parfaitement représentables avec un type entier sur 64 bits natif de la machine utilisée mais non représentable avec des nombres flottants. Exemple avec LibreOffice 7.5.1 :



Notez les deux erreurs dans cette feuille de calcul :

— La valeur 2^{53} est un entier représentable qui vaut 9007199254740992. Elle est représentée par le nombre 9007199254740990. On verra dans la section 3.2 que cette valeur devrait aussi être représentable avec un nombre flottant — on a là un exemple d'erreur commune à plusieurs tableurs ;

— Si l'on ajoute 1 à cette valeur, on obtient un entier encore représentable. Pourtant la valeur affichée est la même que celle de la cellule du dessus.

Le même problème est visible avec Python :

```
>>> int(1.0 + 2**53)
9007199254740992
```

On demande de transformer en entier la valeur $1+2^{53}$. Cette valeur est représentable mais le calcul est fait avec les nombres flottants car on écrit 1.0 et pas 1. Comme $1+2^{53}$ n'est pas représentable en flottant, il est arrondi à 2^{53} avant d'être transformé en entier. On obtient donc 9007199254740992 au lieu de 9007199254740993. Si l'on fait le calcul entièrement avec des entiers, on obtient le bon résultat :

```
>>> 1 + 2**53
9007199254740993
```

2.3 Les entiers signés

Un nombre entier positif ou nul s'exprime naturellement avec une représentation positionnelle binaire et les calculs sont facilement implémentables avec des opérateurs logiques simples.

Pour pouvoir représenter aussi des nombres entiers négatifs, il faut décider d'un codage des chaînes binaires. Comme pour la représentation en décimal-codé-binaire, il existe plusieurs possibilités. Étant donné un ensemble de chaînes binaires de taille k, on pourrait choisir un codage arbitraire utilisant certaines des chaînes pour représenter des entiers négatifs, par exemple celui de la table 4. Notre codage arbitraire doit cependant respecter certaines règles raisonnables, comme de permettre la représentation d'un nombre d'entiers positifs et négatifs à peu près identique.

On voit rapidement la limite d'un choix arbitraire :

— Comment retrouver rapidement la valeur associée à une chaîne ?

— Comment connaître rapidement le signe d'un nombre ?

— Comment implémenter efficacement un circuit de calcul sur ces nombres ?

Les premières machines mécaniques à calculer opérationnelles, comme la *Pascaline* de Blaise Pascal (figure 14), ou plus tard le *comptomètre* de Dorr Eugene Felt (figure 15) étaient construites pour faire efficacement des additions ; pour faire une soustraction, on utilisait une représentation par *complément à 9* qui permettait de remplacer une soustraction par une addition et une soustraction sans retenue.

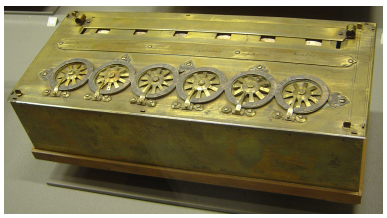


Figure 14 : Modèle de Pascaline de 1652. Source : Wikimedia.

Table 4 : Un codage arbitraire des entiers signés sur $k = 3$ bits.

Chaîne	Valeur
000	−3
001	2
010	1
011	0
100	−1
101	−2
110	3
111	4

Pour faire la soustraction $2357 - 1428$, par exemple, on calcule le complément à 9 de 1428 :

$$\begin{array}{rr} & 9999 \\ - & 1428 \\ \hline & 8571 \end{array}$$

puis l'on effectue l'addition :

$$\begin{array}{rr} & 2357 \\ + & 8571 \\ \hline & 10928 \end{array}$$

Il suffit alors d'ignorer le chiffre surnuméraire de gauche et d'ajouter 1 pour obtenir le résultat de la soustraction : 929.

Étant donnés deux entiers a et b de k chiffres décimaux, et la soustraction $a - b$, le calcul en complément à 9 revient à faire les opérations :

$$a - b = a + (\underbrace{99\cdots9}_{k} - b) - 10^k + 1$$

FIGURE 15 : Le comptomètre de FELT. Source : history-computer.com.

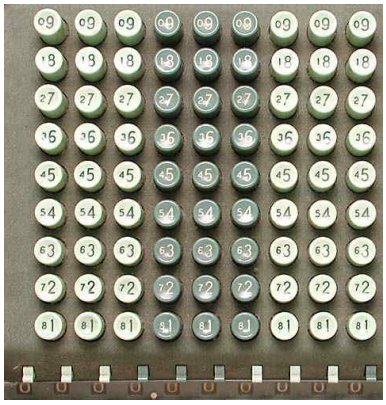


FIGURE 16 : Clavier de comptomètre avec le double affichage en complément à 9. Source : vintagecalculators.com.

TABLE 5 : Codage en complément à 1 sur 3 bits C_1^3.

Chaîne	Non-signé	C_1^3
000	0	0
001	1	1
010	2	2
011	3	3
100	4	−3
101	5	−2
110	6	−1
111	7	−0

[20] John von NEUMANN. *First draft of a report on the EDVAC*. Technical Report W-670-ORD-4926. Moore School of Electrical Engineering, University of Pennsylvania, juin 1945

Le *codage en complément à 1* est l'analogue pour la base 2 du complément à 9 pour la base 10. Le tableau 5 montre le codage des chaînes binaires associé. Le calcul du complément à 1 d'une chaîne binaire se fait de façon très efficace en inversant simplement tous les bits (opération « *non* » bit-à-bit). Comme les « zéros » sont remplacés par des « 1 », il est important de connaitre la taille du format binaire utilisé et de rajouter des « zéros » à gauche de la chaîne si nécessaire avant de faire l'inversion. La notation utilisée pour parler de représentation en complément à 1 indiquera donc aussi la taille t du format : C_1^t.

Exemple de calcul du complément à 1 sur huit bits de la chaîne "0101011" :

$$\text{“0101011”} \xrightarrow{\text{extension à 8 bits}} \text{“00101011”} \xrightarrow{\text{négation bit à bit}} 11010100_{C_1^8}$$

Cette opération est réversible.

On peut déterminer le signe d'un nombre en regardant son bit de poids fort : le nombre est positif si le bit de gauche vaut 0 et négatif sinon.

Un inconvénient de la représentation en complément à 1 est qu'elle admet deux chaînes pour 0, dont un zéro négatif. Cela complexifie en particulier les tests d'égalité.

La *représentation en complément à 2*, proposée dès 1945 par VON NEUMANN[20], élimine ce problème et permet aussi d'utiliser les mêmes circuits électroniques pour faire une opération signée en complément à 2 ou non-signée.

Pour obtenir le complément à 2 sur k bits C_2^k d'une chaîne binaire, on commence par prendre son complément à 1, puis on ajoute 1 :

$$\text{“0101011”} \to 11010100_{C_1^8} \xrightarrow{+1} 11010101_{C_2^8}$$

Cette opération est son propre inverse.

La table 6 montre le codage en complément à 2 des chaînes de trois bits. On constate que :

- Il n'existe plus qu'une représentation pour 0 ;
- Abstraction faite du 0, il y a plus de nombres négatifs représentables que de positifs.

Cette dernière propriété a des conséquences pratiques importantes car l'ensemble des entiers représentables en complément à 2 sur k bits n'est pas un groupe pour l'addition, la valeur -2^{k-1} ne possédant pas d'inverse. On peut ainsi obtenir des résultats surprenants si l'on calcule la valeur absolue de -2^{k-1} pour un entier signé sur k bits :

```
>>> abs(numpy.int8(-128))
-128
```

Malgré cet inconvénient, les avantages de la représentation en complément à 2 sont suffisamment importants pour que son utilisation se soit généralisée sur la plupart des machines pour représenter les entiers signés.

Les langages de programmation typés explicitement offrent généralement des types de différentes tailles pour les entiers signés en complément à 2 et les entiers non signés. On a vu dans la section 2.2 les types non signés `uint8`, `uint16`, `uint32` et `uint64` de la bibliothèque Numpy. Elle propose aussi les types signés `int8`, `int16`, `int32` et `int64`.

En règle générale, le choix d'un type signé ou non signé doit se faire en fonction de son utilisation : si l'on ne manipule que des entiers positifs ou nuls, on peut utiliser des nombres non signés ; sinon, il faut utiliser des nombres signés, en se rappelant que, à taille égale, la borne maximale est deux fois plus petite dans ce format :

```
>>> numpy.uint8(127)+numpy.uint8(1)
128
>>> numpy.int8(127)+numpy.int8(1)
<stdin>:1: RuntimeWarning: overflow encountered in byte_scalars
-128
>>> numpy.int8(126)+numpy.int8(3)
-127
>>> numpy.int8(127)+numpy.int8(2)
-127
>>> numpy.int8(-128)-numpy.int8(1)
127
```

Dans les langages explicitement typés largement utilisés aujourd'hui, comme C, C++ ou Java, le type entier employé par défaut par la plupart des programmeurs est le type entier sur 32 bits signé en complément à 2. La syntaxe même de ces langages, couplée à la paresse naturelle des programmeurs, encourage cela : pourquoi utiliser le type `unsigned int` (entier non signé sur 32 bits) alors que le type `int` ne prend que trois lettres à nommer ? Les développeurs de Youtube ont découvert à leurs dépens qu'on n'utilise pas impunément un type signé pour compter un nombre de vues, positif par nature : le 1er décembre 2014, le compteur de vues de la vidéo musicale « *Gangnam Style* » du chanteur coréen *Psy* se mit à afficher des valeurs négatives croissantes (figure 17). La vue qui suivait la $2^{31} - 1$e a incrémenté le compteur, passant de la chaîne

$$\text{“01111111111111111111111111111111”}$$

à la chaîne :

$$\text{“10000000000000000000000000000000”},$$

qui est interprétée en complément à 2 comme la valeur -2^{31} :

$$\begin{array}{rr} & 01111111111111111111111111111111_{C_2^{32}} \\ + & 00000000000000000000000000000001_{C_2^{32}} \\ & 11111111111111111111111111111111 \\ \hline & 10000000000000000000000000000000_{C_2^{32}} \end{array}$$

Une nouvelle incrémentation donne $10000000000000000000000000000001_{C_2^{32}}$, qui correspond à $-2^{31} + 1$, etc. Ce dépassement de capacité sur les entiers signés, aussi appelé *overflow* en anglais, est une source d'erreur commune dans les programmes.

Les machines utilisant le système d'exploitation Unix, ou ses clones tels Linux, calculent l'écoulement du temps à partir d'une date arbitraire, l'epoch, fixée au 1er janvier 1970. Le temps écoulé est stocké sous forme d'un nombre de secondes depuis l'*epoch* ; pendant longtemps, ce nombre était stocké comme un entier signé (!) en complément à 2 sur 32 bits. Le 19 janvier 2038, à trois heures, quatorze minutes et sept secondes, le

Table 6 : Codage en complément à 2 sur 3 bits C_1^3.

Chaîne binaire	Valeur non-signé	Valeur complément à 2
“000”	0	0
“001”	1	1
“010”	2	2
“011”	3	3
“100”	4	−4
“101”	5	−3
“110”	6	−2
“111”	7	−1

Figure 17 : De l'importance de choisir le bon type d'entier : *overflow* du compteur de vues de Youtube.

compteur dépassera $2^{31} - 1$ et les temps indiqués deviendront négatifs, au moins pour les machines qui n'auront pas été mises à jour avec une version corrigée du système d'exploitation.

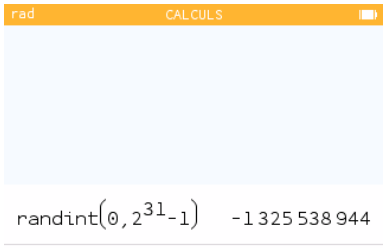


FIGURE 18 : Tirage d'un entier dans l'intervalle $[0, 2^{31} - 1]$ sur la calculatrice Numworks v. 20.4.0.

Par contraste avec les nombres en virgule flottante, qui trainent souvent une réputation sulfureuse du fait de leurs propriétés parfois « exotiques », les nombres entiers apparaissent comme un ensemble sûr pour faire des calculs sur machine : tant que l'on s'assure de rester dans l'intervalle représentable avec le format utilisé, tous les calculs sont corrects.

Ce n'est malheureusement pas toujours aussi simple et les applications font parfois appel aux nombres flottants même dans un contexte où l'on n'attendrait que des nombres entiers. On l'a déjà vu avec les tableurs (section 2.2.1). La calculatrice Numworks donne un autre exemple de cette perméabilité entre entiers et nombres flottants (figure 18) : si l'on demande à tirer un entier dans le domaine $[0, 2^{31} - 1]$, on obtient systématiquement des entiers négatifs. Quel que soit le domaine demandé, la fonction `randint()` peut aussi retourner des valeurs qui y sont extérieures. Il est ainsi possible d'obtenir la valeur 7 lors de la simulation de tirage d'un dé à six faces… Nous expliquerons dans la section 3.2 les raisons de cette erreur[21]. En attendant, cela justifie de s'intéresser aussi aux propriétés des nombres en virgule flottante, même lorsque l'on souhaite seulement travailler avec des nombres entiers.

[21] Scratch et Snap ! présentent le même type d'erreur de façon plus subtile.

3 Les nombres réels

Il y aura toujours une demande, faible mais contante pour des spécialistes de l'analyse d'erreur […] pour dénoncer les énormes erreurs des mauvais algorithmes et, plus important, remplacer les mauvais algorithmes par d'autres meilleurs. — William KAHAN, 1933– ????, In [22] (trad. de l'anglais par les auteurs).

[22] William M. KAHAN. « Interval Arithmetic Options in the Proposed IEEE Floating Point Arithmetic Standard ». In : *Interval Mathematics 1980*. Sous la dir. de KARL L. E. NICKEL. Academic Press, jan. 1980, p. 99-128

QUE LA RÉALITÉ PHYSIQUE QUI NOUS ENTOURE soit de nature discrète ou continue, la modélisation que nous en faisons et avec laquelle nous l'appréhendons est intrinsèquement continue et basée sur l'ensemble des réels.

Contrairement aux nombres entiers, où il est théoriquement possible de garantir la validité des calculs sur ordinateur dans un domaine fixé, la nature continue de l'ensemble des réels nous interdit d'imaginer pouvoir représenter tous les éléments de quelque intervalle que ce soit sur une machine dont les ressources sont forcément limitées.

La plupart des premiers ordinateurs — l'Atanasoff-Berry Computer (1942), le Harvard Mark I (1944), l'ENIAC (1945) — utilisaient une *représentation à virgule fixe* d'un sous-ensemble limité de nombres rationnels en lieu et place des nombres réels.

3.1 Les nombres à virgule fixe

Jusqu'à la sortie en 1989 du processeur Intel i486, les micro-ordinateurs ne possédaient généralement pas les capacités matérielles de faire du calcul sur les « réels. » Il était cependant possible d'acheter une puce dédiée, le *Floating-Point Unit* (FPU) à rajouter sur la carte-mère de l'ordinateur pour cela (Intel 8087 pour le processeur Intel 8086, Intel 80287 pour le processeur Intel 80286, …). Pour les utilisateurs qui n'avaient pas les moyens ou la nécessité d'acheter un FPU, les calculs « réels » étaient effectués en utilisant une représentation à virgule fixe.

Un nombre à virgule fixe est un rationnel stocké en machine sous la forme d'un entier à diviser par un facteur d'échelle, généralement implicite. Les calculs sont faits par du code logiciel s'appuyant sur les instructions sur les entiers du processeur.

Exemple — Une représentation binaire à virgule fixe sur 8 bits.

La définition d'un format à virgule fixe requiert deux paramètres : la taille totale utilisée et la position de la virgule (ou, alternativement, le facteur d'échelle). On peut choisir, par exemple de stocker des nombres sur 8 bits avec un facteur d'échelle égal à 2 : les six bits de gauche représentent un entier en complément à deux[23] et les deux bits de droite, une partie fractionnaire :

$$\text{“01001011”} \rightsquigarrow 010010.11_2 \quad (18.75_{10})$$
$$\text{“11011001”} \rightsquigarrow 110110.01_2 \quad (-10.25_{10})$$

La partie entière s'interprète comme un entier signé en complément à 2 sur six bits ; les bits de la partie fractionnaire correspondent aux coefficients des puissances négatives de 2 :

-2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}
b_5	b_4	b_3	b_2	b_1	b_0	.	b_{-1}	b_{-2}

Le plus petit nombre représentable dans ce format est -32.75 et le plus grand, 31.75. L'écart entre deux nombres consécutifs est constant et égal à 0.25.

La représentation des nombres à virgule fixe présente des avantages en terme de simplicité d'implémentation (elle est souvent privilégiée pour cela dans les processeurs embarqués de faible puissance). Pour une taille fixée, le domaine de variation est cependant modeste en comparaison de ce que l'on pourrait obtenir avec une représentation à virgule flottante, ce qui requiert un plus grand soin de la part de l'utilisateur pour ajuster la magnitude de ses données. C'est pourquoi cette dernière représentation est beaucoup plus utilisée, la représentation à virgule fixe occupant des niches d'application où ses propriétés simples en terme d'arrondi des calculs sont plus appropriées (par exemple, les programmes de comptabilité).

[23] Il est possible d'utiliser un autre codage pour le signe. Certaines représentations réservent un bit pour le signe et utilisent le reste pour la valeur absolue (*représentation en signe et magnitude*).

Exemple — L'erreur à 92 quadrillions de dollars de PayPal

Fin juin 2013, Chris Reynolds, un dirigeant d'entreprise de Pennsylvanie découvrit que le solde de son compte PayPal indiquait un débit de la somme faramineuse de \$92 233 720 368 547 800, soit 9 223 372 036 854 780 000 centimes (figure 19). Il s'agissait évidemment d'une erreur de PayPal qui fut promptement rectifiée. Il semble que PayPal stocke les informations monétaires en centimes sous la forme de nombres à virgule fixe dans des entiers signés de 64 bits. Le solde de M. Reynolds devient moins étrange lorsque l'on sait que $-2^{63} = -9\,223\,372\,036\,854\,775\,808$. Paypal n'a jamais indiqué la raison précise de l'erreur mais on peut supposer qu'une mauvaise manipulation des entiers signés en est la source.

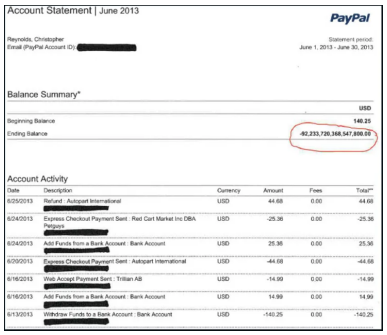


Figure 19 : Une erreur à 92 quadrillions de dollars de PayPal.

3.2 Les nombres à virgule flottante

L'utilisation de la notation à virgule flottante — ou *notation scientifique* — pour représenter et manipuler les nombres « réels » en machine n'est pas nouvelle. Elle est même antérieure à l'utilisation de la représentation en virgule fixe : tous les nombres manipulés par les machines Z1 (1934–1936) et surtout Z3 (1938–1941) de Konrad Zuse étaient dans un format à virgule flottante très proche de celui présent sur la grande majorité des ordinateurs[24] aujourd'hui.

« *Je pense qu'il y a un marché mondial pour, peut-être, cinq ordinateurs.* » Cette citation — probablement apocryphe[25] — attribuée à Thomas Watson, le président d'IBM, reflète la situation de l'époque à laquelle elle est censée avoir été prononcée (1943

[24] R. Rojas. « Konrad Zuse's legacy : the architecture of the Z1 and Z3 ». In : *IEEE Annals of the History of Computing* 19.2 (avr. 1997), p. 5-16

[25] En page 26 de la faq historique disponible sur le site d'IBM, cette citation est considérée comme une interprétation dans le mauvais contexte de paroles effectivement prononcées par Watson en 1953…

ou 1953, selon les sources). À cette époque, les ordinateurs étaient de monstrueuses machines hors de prix, aux performances limitées, prenant énormément de place et consommant beaucoup d'énergie. Il y en avait donc un nombre très réduit, essentiellement installé dans quelques grands centres militaires ou de recherche. La situation évolua lentement jusqu'au début des années soixante, où l'avènement de nouvelles technologies — en particulier, les transistors puis les circuits intégrés — permit une croissance de la construction de mini-ordinateurs par de nombreux fabricants, avec un coût les mettant à la portée des entreprises et des petits centres de calcul.

Chaque constructeur avait une idée différente de la manière d'implémenter une représentation à virgule flottante des nombres ; la précision et les propriétés garanties lors des calculs variaient donc d'une marque à l'autre et parfois d'un modèle à l'autre dans une même marque. La portabilité des codes de calcul étaient donc nulle et les applications devaient être réécrites presque de zéro pour chaque nouvelle machine en tenant compte des propriétés de son arithmétique[26]. Cela représentait un coût exhorbitant en temps et en moyens humains, qui n'était rendu acceptable que par le nombre encore relativement limité d'ordinateurs dans le monde.

L'avènement des micro-ordinateurs dans les années soixante-dix allait changer la donne : le coût très raisonnable de ces machines les mettaient à la portée de (presque) toutes les bourses et il n'était pas nécessaire d'être devin pour comprendre que leur nombre allait exploser.

En 1976, la société Intel décida le développement d'un coprocesseur arithmétique spécialisé dans le calcul en virgule flottante pour épauler leurs processeurs mis au point pour le marché des micro-ordinateurs. John Palmer, le responsable du programme chez Intel, contacta un spécialiste du calcul scientifique de l'*Université de Californie à Berkeley*, William Kahan (figure 20), qui s'attela à rédiger un rapport décrivant la représentation et les propriétés d'une arithmétique à virgule flottante implémentable avec les technologies de l'époque et qui permettrait d'établir aisément des preuves sur les calculs. Ce rapport, écrit avec Jerome Coonen et Harold Stone devint l'une des trois propositions soumises à partir de 1977 à l'*IEEE 754 Working Group for binary floating-point arithmetic* mis en place par l'*IEEE*[27] *Microprocessor Standards Committee* pour la définition d'un standard de calcul à virgule flottante visant à stopper la gabégie ayant marqué les époques précédentes.

Les deux autres rapports soumis à l'*IEEE Working Group* avaient été écrits par des personnels de *Hewlett-Packard*[28] (HP) et *Digital Equipment Corporation*[29] (DEC), deux géants de l'industrie des mini-ordinateurs. Comme il est habituel dans ce genre de situation, chacun des trois candidats à la standardisation proposait une solution qu'il implémentait déjà (HP et DEC) ou qu'il était en train d'implémenter (Intel, avec son processeur 8087). La bataille entre les trois propositions fit rage pendant plusieurs années car des intérêts économiques potentiellement immenses étaient en jeu. Finalement, ce n'est qu'en 1985 que le standard IEEE 754 pour l'arithmétique en virgule flottante binaire fut définitivement adopté[30]. Depuis lors, il a été revu et modifié à la marge plusieurs fois — en particulier, on a ajouté une partie sur l'arithmétique flottante décimale en 1987. La dernière version[31], sortie en 2019, est une révision mineure. Aujourd'hui, le standard IEEE 754 est mis en œuvre dans la quasi-totalité des processeurs possédant des capacités de calcul sur les nombres à virgule flottante.

3.2.1 Représentation interne des nombres flottants

La version la plus récente du standard IEEE 754 décrit la représentation interne des nombres à virgule flottante en base 2 et en base 10 (le standard original de 1985 ne s'intéressait qu'à la base 2). Dans la suite, nous ne considérerons cependant que la représentation en base 2, qui est de loin la plus utilisée.

Un nombre binaire à virgule flottante x est représenté par un triplet (s, E, m), avec :

$$x = (-1)^s \times m \times 2^E \tag{3}$$

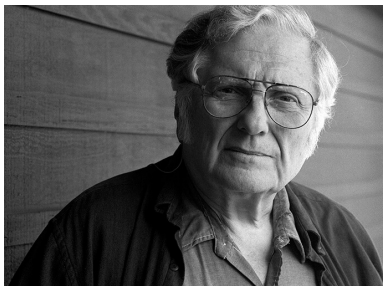


Figure 20 : William Kahan, 1933– ????. Source : Heidelberg Laureate Forum.

[26] David G. Hough. « The IEEE Standard 754 : One for the History Books ». In : *Computer* 52.12 (déc. 2019), p. 109-112

[27] *Institute of Electrical and Electronics Engineers*, un organisme professionnel définissant, entre autres, de nombreux standards relatif à l'électricité, l'électronique et l'informatique.

[28] Bob Fraley et Steve Walther. « Proposal to eliminate denormalized numbers ». In : *ACM SIGNUM Newsletter* 14.si-2 (oct. 1979), p. 22-23

[29] Mary Payne et William Strecker. « Draft proposal for a binary normalized floating point standard ». In : *ACM SIGNUM Newsletter* 14.si-2 (oct. 1979), p. 24-28

[30] *IEEE Standard for Binary Floating-Point Arithmetic*. American National Standard (ANSI) IEEE Std 754-1985. The Institute of Electrical et Electronics Engineers, Inc, 1985

[31] *IEEE Standard for Floating-Point Arithmetic*. Rapp. tech. IEEE Std 754-2019 (Revision of IEEE 754-2008). Institute of Electrical et Electronics Engineers (IEEE), juill. 2019

où s est le *bit de signe* ($s \in \mathbb{B}$), E est un *exposant* entier positif, négatif ou nul, et $m \in [0, 2)$ est le *signifiant*[32], de la forme :

$$m = b_{p-1}.b_{p-2}b_{p-3}\cdots b_1 b_0, \quad b_i \in \mathbb{B}.$$

La valeur de p détermine la précision du format de nombre flottant.

Comme on l'a vu dans le cas des nombres à virgule fixe, la valeur de m s'obtient par la formule :

$$m = b_{p-1}2^0 + b_{p-2}2^{-1} + b_{p-3}2^{-2} + \cdots + b_1 2^{-p+2} + b_0 2^{-p+1}$$

Exemple 1. *La valeur de* 3.5_{10} *peut être représentée en virgule fixe par* 11.1_2*. En forme flottante, cela donne* $1.11_2 \cdot 2^1$*. On a donc le triplet* ($s = 0$, $E = 1$, $m = 1.11_2$)*.*

La représentation en virgule flottante d'un nombre n'est pas unique. Dans l'exemple précédent, on aurait pu aussi choisir la représentation 0.111×2^2 ou 0.0111×2^3. Pour une taille fixée du signifiant, on voit facilement que la représentation où b_{p-1} vaut 1 conserve le plus de chiffres significatifs. Dans ce cas, on dit que le nombre est en *forme normalisée*. Comme on décide de s'attacher à avoir b_{p-1} toujours égal à 1, il est inutile de le stocker et il devient alors *implicite*. Au lieu de stocker m, il suffit désormais de stocker la *partie fractionnaire* f correspondant aux $p - 1$ bits à droite de la virgule.

L'exposant E peut être positif ou négatif (ou nul) en fonction du sens du décalage nécessaire pour obtenir une forme normalisée.

Exemple 2. *Le nombre* 0.1875_{10} *s'écrit* 0.0011_2*, ce qui donne* $1.1_2 \times 2^{-3}$ *en forme flottante normalisée.*

Pour représenter un exposant signé, le standard a choisi d'utiliser une *représentation biaisée* plutôt qu'une représentation en complément à 2 comme il est habituel pour les entiers. La représentation biaisée consiste à ajouter à l'entier signé une valeur, le *biais*, telle que leur somme devient positive et peut alors être stockée sous la forme de la chaîne binaire correspondant à sa représentation en base 2 (table 7).

Sur 8 bits, par exemple, on peut coder tous les entiers de 0 à 255. Comme on veut généralement pouvoir représenter à peu près autant de nombres positifs que de nombres négatifs, on choisit un biais en conséquence. Ici, on a deux possibilités :

- Choisir le biais $b = 127$, ce qui permet de représenter tous les entiers de -127 à $+128$;
- Choisir le biais $b = 128$, ce qui permet de représenter tous les entiers de -128 à $+127$.

La première solution privilégie les grands nombres (l'exposant peut être plus grand) alors que la deuxième solution privilégie les petits nombres (l'exposant peut être plus négatif). La proposition de Payne et Strecker[33] avait privilégié la deuxième solution. Le standard IEEE 754 a prévu un autre mécanisme pour bien gérer les petits nombres ; c'est pourquoi il a été décidé d'adopter la première solution pour limiter le risque de dépassement de capacité[34]. Ainsi, lorsqu'un exposant est représenté par un entier sur k bits, le biais est $b = 2^{k-1} - 1$.

Exemple 3. *Considérons un codage biaisé sur 8 bits avec un biais b de 127. Si l'on veut représenter la valeur* 13_{10}*, on commence par lui ajouter* 127_{10}*, puis l'on code en binaire la somme, qui est un nombre positif :*

$$13_{10} + 127_{10} = 140_{10} \xrightarrow{\text{Biais à 127}} 10001100_2$$

Pour représenter la valeur -15_{10}*, on fait de même :*

$$-15_{10} + 127_{10} = 112_{10} \xrightarrow{\text{Biais à 127}} 01110000_2$$

[32] Bien que le terme « mantisse » (ou « mantissa », en anglais) soit employé pour m dans de nombreux textes relatifs à la représentation des nombres flottants, il n'est jamais utilisé par le standard IEEE 754 lui-même, qui lui préfère le terme de « *significand* » (« significande » ou « signifiant », en français). La mantisse correspondant historiquement à la partie fractionnaire du logarithme en base 10 d'un nombre, on s'abstiendra d'utiliser le terme dans un contexte où un autre mot a *de facto* plus de légitimité.

Table 7 : Codage sur quatre bits avec un biais de 7.

Chaîne	Non-signé	Biaisé à 7
0000	0	−7
0001	1	−6
0010	2	−5
0011	3	−4
0100	4	−3
0101	5	−2
0110	6	−1
0111	7	0
1000	8	1
1001	9	2
1010	10	3
1011	11	4
1100	12	5
1101	13	6
1110	14	7
1111	15	8

[33] Mary Payne et William Strecker. « Draft proposal for a binary normalized floating point standard ». In : *ACM SIGNUM Newsletter* 14.si-2 (oct. 1979), p. 24-28

[34] J. T. Coonen. « An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic ». In : *Computer* 13.1 (jan. 1980), p. 68-79

Quelle que soit la valeur entière à représenter dans le domaine $[-127, 128]$*, l'ajout du biais donne une valeur positive que l'on peut stocker en binaire avec une représentation positionnelle classique.*

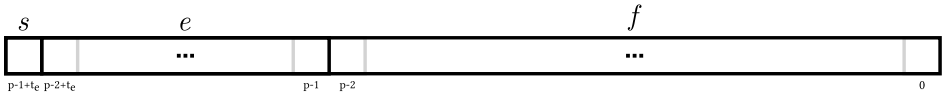
*Le décodage se fait par soustraction du biais au nombre binaire positif stocké. Par exemple, avec la chaîne binaire "*10011100*" :*

$$10011100_2 = 156_{10} \xrightarrow{\textit{Retrait du biais}} 156_{10} - 127_{10} = 29_{10}$$

*Pour la chaîne "*00011000*" :*

$$00011000_2 = 24_{10} \xrightarrow{\textit{Retrait du biais}} 24_{10} - 127_{10} = -103_{10}$$

Pourquoi avoir choisi une représentation biaisée plutôt qu'une représentation en complément à 2 pour l'exposant ? La raison est à chercher du côté du format choisi pour le stockage des nombres flottants : chaque nombre flottant au format IEEE 754 est encodé sous la forme d'une chaîne binaire en concaténant, dans l'ordre, le bit de signe s sur 1 bit, l'exposant biaisé $e = E + b$ sur t_e bits, puis la partie fractionnaire f sur $p - 1$ bits :



En utilisant une représentation biaisée, il est possible de comparer deux nombres flottants très rapidement avec les méthodes de comparaison des entiers signés par signe et magnitude car la représentation biaisée respecte l'ordre des entiers non signés, contrairement au complément à 2 (figure 21).

Le standard IEEE 754 définit plusieurs tailles de nombre flottant en fonction de la précision et du domaine souhaité. La taille totale d'un flottant est normalement une puissance de 2 de façon à pouvoir être manipulé facilement par le matériel[35]. Le standard définit en particulier deux formats (figure 22) :

- La *simple précision*, encodée sur 32 bits, avec 1 bit de signe s, 8 bits pour l'exposant biaisé e et 23 bits pour la partie fractionnaire f ;
- La *double précision*, encodée sur 64 bits, avec 1 bit de signe s, 11 bits pour l'exposant biaisé e et 52 bits pour la partie fractionnaire f.

Un format de nombres flottants suivant le standard IEEE 754 peut être entièrement défini par la donnée de la taille p du signifiant et la valeur de l'exposant non biaisé maximum emax[36]. On note $\mathbb{F}_p^{\text{emax}}$ un tel ensemble des nombres flottants. Les nombres flottants en simple précision sont dans l'ensemble $\mathbb{F}_{24}^{127}$ et ceux en double précision, dans l'ensemble $\mathbb{F}_{53}^{1023}$.

Pour simplifier la présentation, nous considérerons parfois dans la suite de ce fascicule des formats de nombres flottants respectant le standard IEEE 754 mais d'une taille suffisamment faible pour permettre de les appréhender graphiquement.

Représentation de la valeur zéro. La représentation d'un nombre binaire à virgule flottante est unique dès lors que l'on impose la représentation normalisée avec un bit à 1 à gauche de la virgule. Mais comment représenter zéro dans ce cas ? Aucun décalage du signifiant ne permettra jamais d'obtenir une représentation normalisée. Ce problème s'était présenté à Konrad Zuse lors du *design* de sa machine Z1 sans qu'il ne trouve de solution adéquate. C'est seulement avec la Z3 qu'il eut l'idée de réserver une valeur de l'exposant pour coder le zéro[37]. Le standard IEEE 754 fait de même : lorsque e et f sont tous les deux nuls (uniquement des bits à 0 dans leur représentation binaire), la valeur codée est 0. Comme le bit de signe est codé à part, il est alors possible d'avoir aussi la valeur -0 :

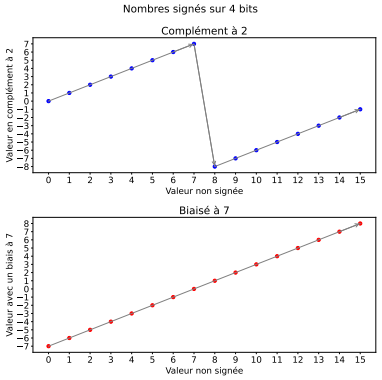


Figure 21 : Monotonie des représentations signées sur 4 bits.

[35] Une exception à cette règle : le coprocesseur arithmétique des processeurs Intel effectue les calculs en interne sur 80 bits (1 bit de signe, 15 bits pour l'exposant biaisé et 64 bits pour le signifiant — ce format n'a pas de bit implicite). Il s'agit d'une double précision étendue permise par le standard.

[36] Attention : comme on va le voir plus loin, étant donné un exposant sur t_e bits, emax ne vaut pas 2^{t_e-1} mais $2^{t_e-1} - 1$ car la valeur 2^{t_e-1} est réservée. Incidemment, cela implique que le biais b est égal à emax.

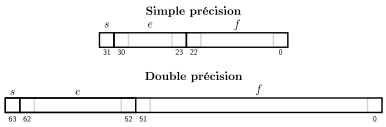


Figure 22 : Représentation au format IEEE 754 de nombres flottants en simple précision sur 32 bits et en double précision sur 64 bits.

[37] Raul Rojas. « The Z1 : Architecture and Algorithms of Konrad Zuse's First Computer ». Preprint arXiv :1406.1886 [cs]. Juin 2014

```
>>> 0.0
0.0
>>> -0.0
-0.0
```

Évidemment, les représentations de 0 et -0 sont considérées comme égales :

```
>>> 0.0 == -0.0
True
```

Le standard définit explicitement que la racine carrée de tout nombre supérieur ou égal à 0 est positive, à l'exception de $\sqrt{-0}$, qui vaut -0 :

```
>>> from math import sqrt
>>> sqrt(-0.0)
-0.0
```

Un des objectifs du standard était de garantir la continuité des calculs. Que retourner lorsque le résultat d'un calcul est trop grand pour être représenté ? Ou lorsqu'un calcul n'a pas de sens sur les réels (par exemple, $\sqrt{-1}$) ? La solution adoptée fut, encore une fois, de réserver une valeur d'exposant pour coder deux valeurs spéciales :

- L'infini, positif ou négatif (si $e = 2^{t_e} - 1$ et $f = 0$), pour les calculs de trop grande magnitude. On obtient ainsi une droite réelle achevée ;
- Un *Not a Number* ou NaN (si $e = 2^{t_e} - 1$ et $f \neq 0$), pour les calculs invalides.

Les infinis. L'infini est la valeur flottante que l'on peut utiliser à chaque fois qu'un calcul donne un résultat dont la magnitude est plus grande que le plus grand nombre flottant représentable. De plus, on a les règles :

- $1/0 = \infty$, $-1/0 = -\infty$;
- $1/\infty = 0$, $-1/\infty = -0$.

Ces règles permettent de manipuler plus naturellement certaines équations physiques. Par exemple, on sait que la résistance totale d'un circuit avec deux résistances R_1 et R_2 en parallèle (figure 23) est :

$$R_t = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}$$

Si l'une des deux résistances — par exemple R_1 — est nulle, la résistance totale est nulle. C'est bien ce que l'on obtient avec les règles de calcul avec un infini :

$$R_t = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0$$

On parle d'*overflow* lorsque le résultat d'un calcul a une magnitude supérieure à la plus grande valeur représentable et doit être remplacé par un infini.

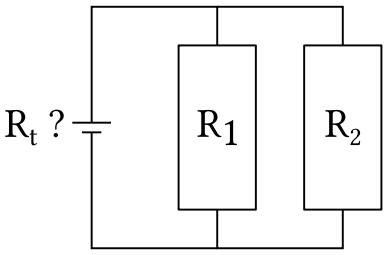


FIGURE 23 : Calcul de la résistance totale R_t d'un circuit.

Exemple — Gestion des *overflow*

Lors de l'écriture d'une expression à évaluer dans un code informatique, il est important de considérer le risque d'*overflow* : même si le résultat est parfaitement codable dans le format flottant choisi, il se peut qu'une expression intermédiaire génère un *overflow*. Dans ce cas, toute la précision d'évaluation est perdue.

Considérons par exemple le calcul de la longueur de l'hypothénuse d'un triangle

rectangle de côtés opposés de longueurs a et b. On a :

$$c = \sqrt{a^2 + b^2}.$$

Cette formule est utilisée dans de nombreux contextes (ex. : calcul de distance). Du fait de l'élévation au carré, on peut avoir un *overflow* lors du calcul alors que le résultat est représentable. Ce calcul est est tellement important qu'il possède généralement sa propre fonction d'évaluation dans les langages de programmation ; elle utilise habituellement un algorithme spécifiquement écrit pour éviter un *overflow* et garantir la meilleure précision. Une méthode simple, satisfaisant la première condition, consiste à diviser l'argument de plus petite magnitude par celui de plus grande magnitude et à calculer la racine carrée avec ce nouvel argument :

```
def hypothenuse2(a,b):
    mini = min(abs(a),abs(b))
    maxi = max(abs(a),abs(b))
    return maxi*sqrt(1+(mini/maxi)**2)
```

D'autres algorithmes plus sophistiqués encore existent[38]. Python, par exemple, en utilise un pour mettre en œuvre la fonction `hypot()`.

[38] Carlos F. Borges. « Algorithm 1014 : An Improved Algorithm for hypot(x,y) ». In : *ACM Transactions on Mathematical Software* 47.1 (déc. 2020), 9 :1-9 :12

Les NaNs. Les NaNs sont les valeurs associées à toutes les opérations pour lesquelles il n'existe pas de résultat sur les réels, par exemple $\sqrt{-1}$ ou $0/0$. Le NaN est une des rares instances du *reciprivers exclusion*[39] cher à Douglas Adams car le standard IEEE 754 le définit comme non-ordonné. Ainsi, toutes les comparaisons avec un NaN sont fausses, y compris lorsque l'on compare deux NaNs pour l'égalité :

```
>>> v=float("nan")
>>> v==v
False
```

Cela a un impact potentiellement important sur un certain nombre d'opérations lorsque l'on ne tient pas compte de cette propriété. Les fonctions `min()` et `max()` de Python, par exemple, ne sont pas commutatives à cause de cela :

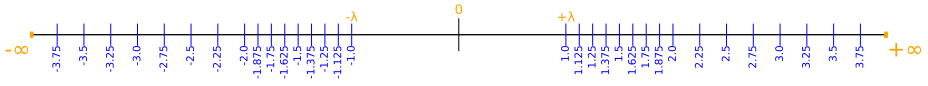
```
>>> a=float("nan"); b=3.0
>>> min(a,b)==min(b,a), max(a,b)==max(b,a)
(False, False)
```

[39] **Recipriversexcluson** · *Nombre dont l'existence ne peut se définir que comme étant tout sauf lui-même.* In Douglas Adams, Le guide du routard galactique, tome III, 1982 (trad. de l'anglais par les auteurs).

Figure 24 : Représentation interne des nombres flottants au format $\mathbb{F}_4^1$.

Les nombres dénormalisés. Considérons l'ensemble des nombres flottants représentables dans le format $\mathbb{F}_4^1$, c'est à dire 1 bit de signe, 2 bits d'exposant et 3 bits de partie fractionnaire (figure 24) — suivant la convention introduite par Higham[40], on appelle λ le plus petit nombre flottant normalisé positif :

[40] Nicholas J. Higham. *Accuracy and stability of numerical algorithms.* 2nd ed. Philadelphia : Society for Industrial et Applied Mathematics, 2002



Dans ce format, on voit qu'il existe un trou sans aucun flottant représentable autour de 0. Cela est dû à la contrainte de normalisation des nombres. En effet, pour un format $\mathbb{F}_p^{\text{emax}}$, le premier flottant normalisé après 0 est $\lambda = 1.0_2 \times 2^{1-\text{emax}}$; celui qui suit λ vaut $(1 + 2^{1-p}) \times 2^{1-\text{emax}}$. L'écart entre λ et son suivant[41], next (λ), est donc de $2^{2-p-\text{emax}}$

[41] Étant donné un nombre flottant $x \in \mathbb{F}_p^{\text{emax}}$, on note prev (x) le nombre flottant de $\mathbb{F}_p^{\text{emax}}$ précédant immédiatement x sur la ligne réelle et next (x) le nombre flottant suivant immédiatement x (avec prev $(-\infty) = -\infty$ et next $(+\infty) = +\infty$).

seulement.

Pour combler ce trou, Kahan, Coonen et Stone ont proposé à l'*IEEE 754 Working Group* d'autoriser une dégradation progressive de la précision : lorsque l'on cherche à normaliser un nombre, si l'exposant nécessaire pour faire apparaitre un 1 à gauche de la virgule devient trop négatif pour être représenté, on accepte de garder une *représentation dénormalisée* avec un 0 à gauche de la virgule. Pour signaler cette dénormalisation, on fixe, par exception, l'exposant biaisé à 0. L'exposant à utiliser pour calculer la valeur du nombre flottant est égal à l'exposant minimum : $1 - \text{emax}$. Par exemple, si l'on divise itérativement par 2 la valeur $1 \in \mathbb{F}_4^3$ (figure 25), passé λ, on ne peut plus décrémenter l'exposant, donc on commence à décaler le signifiant vers la droite (en base 2 comme en base 10, le décalage des chiffres vers la droite correspond à une division par la base). Le plus petit nombre dénormalisé positif — next (0) — est noté μ.

Pour le format $\mathbb{F}_4^1$, l'introduction des nombres dénormalisés remplit harmonieusement le trou autour de 0 (figure 26).

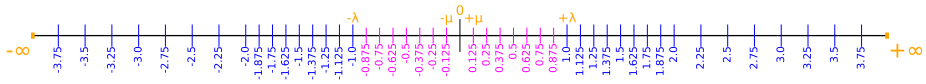


Figure 26 – Représentation de l'ensemble $\mathbb{F}_4^1$ sur la ligne réelle.

Par construction, tout nombre flottant est un multiple de μ. Ainsi, on a la garantie que la différence entre deux flottants est nulle uniquement s'ils sont égaux :

$$\forall (x, y) \in \left(\mathbb{F}_p^{\text{emax}}\right)^2 : x - y = 0 \iff x = y$$

Ce n'est pas le cas sans l'introduction des nombres dénormalisés. Un exemple fameux du genre de problème que l'absence de cette propriété peut entrainer se trouve dans le code Python suivant :

```
if x != y:
   y = 1/(x-y)
```

Le programmeur a essayé de se protéger d'une division par zéro. Malheureusement, sans les nombres dénormalisés, on peut avoir simultanément le test `x != y` vérifié et `x-y` qui vaut 0.

Malgré leurs avantages, les nombres dénormalisés sont en grande partie à l'origine du long délai entre le début du processus de normalisation de l'arithmétique flottante et son adoption définitive. Le rapport de Fraley et Walther[42] avait d'ailleurs essentiellement pour but leur élimination du futur standard. Les auteurs considéraient qu'ils étaient trop complexes à implémenter — ils n'existaient d'ailleurs pas sur les machines de Hewlett-Packard, leur employeur — et qu'ils rendraient le standard plus difficile à mettre en œuvre et à comprendre.

Pendant longtemps, les nombres dénormalisés ont été privés d'une implémentation matérielle efficace : dès qu'un nombre dénormalisé apparaissait dans un calcul, il était traité par des procédures logicielles plus lentes. Les architectures modernes ont commencé à faire des dénormalisés des *citoyens de première classe* et les pertes de performances dues à leur usage tendent à disparaitre. Le programme ci-dessous calcule le temps mis pour effectuer 2^{25} additions entre nombres dénormalisés et entre nombres normalisés :

```
import time

début = time.perf_counter()
```

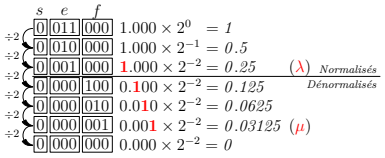


Figure 25 : Divisions itérées par deux de 1 à 0 sur $\mathbb{F}_4^3$.

[42] Bob Fraley et Steve Walther. « Proposal to eliminate denormalized numbers ». In : *ACM SIGNUM Newsletter* 14.si-2 (oct. 1979), p. 22-23

```
somme = 0.0
for i in range(2**25):
    somme += 5.0e-324 # Somme de nombres dénormalisés
fin = time.perf_counter()
print(somme, fin-début)

début = time.perf_counter()
somme = 0.0
for i in range(2**25):
    somme += 2.2250738585072014e-308 # Somme de nombres normalisés
fin = time.perf_counter()
print(somme, fin-début)
```

Sur une machine avec un processeur Intel Pentium T4500, sorti en 2010, le temps pour faire la somme des nombres dénormalisés est pratiquement double de celui pour les nombres normalisés. Sur une machine avec un processeur Intel Core i7-6700HQ, sorti fin 2015, les temps sont pratiquement identiques.

Lorsque le résultat d'un calcul est un nombre dénormalisé, on parle d'*underflow*.

Pour résumer, étant donné un nombre flottant v identifié par le triplet (s, e, f) dans un format $\mathbb{F}_p^{\text{emax}}$, on détermine la valeur de v suivant cinq cas :

$$\begin{cases} e = 0, f = 0 & : \quad v = (-1)^s \times 0 \\ e = 0, f \neq 0 & : \quad v = (-1)^s \times 0.f \times 2^{1-\text{emax}} \\ 0 < e < 2\text{emax} + 1 & : \quad v = (-1)^s \times 1.f \times 2^{e-\text{emax}} \\ e = 2\text{emax} + 1, f = 0 & : \quad v = (-1)^s \infty \\ e = 2\text{emax} + 1, f \neq 0 & : \quad v = \text{NaN} \end{cases}$$

La table 8 résume certaines des caractéristiques importantes des principaux formats de nombres flottants.

Table 8 – Caractéristiques des principaux formats de nombres flottants.

Format	t_e	t_f	emin	emax	$\varepsilon^\dagger$	$\lambda^\star$	$\mu^\diamond$
single	8	23	-126	$+127$	2^{-23}	2^{-126}	2^{-149}
double	11	52	-1022	$+1023$	2^{-52}	2^{-1022}	2^{-1074}

†) *epsilon de la machine*, égal à 2^{-t_f}
⋆) *realmin* (plus petit flottant normalisé positif), égal à 2^{emin}
⋄) *subrealmin* (plus petit flottant positif dénormalisé), égal à $\varepsilon\lambda$

Les exemples ci-dessous présentent l'interprétation d'une chaîne binaire sous la forme d'un nombre flottant au format IEEE 754 simple précision en fonction des différents cas qui peuvent se présenter.

Exemple 4. *On a la chaîne hexadécimale $v =$ "C0AC0000" représentant un nombre flottant au format* simple précision. *On souhaite connaître la valeur décimale codée.*

— *On commence par calculer la représentation binaire de v en séparant les champs s, e et f :*

$$v = \underbrace{1}_{s}\ \underbrace{10000001}_{e}\ \underbrace{01011000000000000000000}_{f}$$

— *On regarde ensuite la valeur de e pour déterminer si l'on a affaire à une exception (cas où $e = 0$ ou $e = 255$ pour le format simple précision). Ici, ce n'est pas le cas donc on peut continuer à interpréter le nombre comme une valeur au format normalisé ;*

— *Le signe s vaut 1 donc le nombre est négatif ;*

— *L'exposant biaisé e vaut* $10000001_2 = 129_{10}$. *On retire le biais de* 127 *pour retrouver l'exposant original* $E = 129 - 127 = 2$;
— *On ajoute le* 1 *implicite à la partie fractionnaire f pour obtenir :*

$$1.01011000000000000000000_2$$

— *On construit le nombre complet normalisé :* $v = -1.01011_2 \times 10^2$. *Comme l'exposant est petit, on peut s'en débarrasser en déplaçant le point flottant de deux positions vers la droite :*

$$v = -101.011_2$$

La partie entière vaut $101_2 = 5_{10}$. *La partie fractionnaire vaut* $2^{-2} + 2^{-3} = 0.375$. *On en déduit que* $v = -5.375$.

Exemple 5. *On a la chaîne hexadécimale "*7FA00000*" représentant un nombre flottant v au format simple précision.*
— *On calcule la représentation binaire de v :*

$$v = \underbrace{0}_{s}\ \underbrace{11111111}_{e}\ \underbrace{01000000000000000000000}_{f}$$

— *L'exposant biaisé e vaut* 255, *donc on est en présence d'une exception ; le champ f n'est pas nul, donc v est un* NaN.

Exemple 6. *On a la chaîne hexadécimale "*80100000*" représentant un nombre flottant v au format simple précision.*
— *On détermine la représentation binaire de v :*

$$v = \underbrace{1}_{s}\ \underbrace{00000000}_{e}\ \underbrace{00100000000000000000000}_{f}$$

— *L'exposant biaisé e est nul. On a donc affaire à une exception. La partie fractionnaire n'est pas nulle donc v est un nombre dénormalisé négatif (car s vaut 1) de la forme :*

$$v = -0.001_2 \times 10^{-126}$$

que l'on peut aussi écrire :

$$v = -2^{-3}2^{-126} = -2^{-129}$$

L'arrondi. Observons la figure 26. L'écart entre un nombre flottant $\phi \in \mathbb{F}_p^{\text{emax}}$ de la forme $\phi = m \times 2^E$ et le nombre flottant suivant vaut $2^{-p}2^E$. Il double donc d'une binade[43] à la suivante. Ce phénomène est encore plus visible si l'on considère un ensemble de nombres flottants plus grand, par exemple $\mathbb{F}_4^3$:

[43] Une *binade* correspond à l'ensemble des nombres flottants partageant le même exposant.

Ensemble des flottants de $\mathbb{F}_4^3$

Plus l'on va consacrer de bits au format flottant et plus l'on pourra représenter de valeurs de la ligne réelle. On ne pourra cependant jamais représenter tous les nombres réels du fait de la finitude des ressources de la machine. Que faire d'un nombre qui n'est pas représentable ? Cela peut arriver de trois façons :

1. Le nombre est irrationnel ;

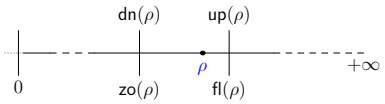


FIGURE 27 : Les différents arrondis d'un nombre réel ρ.

2. Le nombre est rationnel mais a une représentation infinie en base 2 (voir l'annexe B pour un rappel sur les critères de finitude de la représentation en base 2) ;
3. Le nombre est rationnel et a une représentation finie en base 2 requérant plus de bits que la précision du format utilisé n'en possède.

Le standard IEEE 754 original définit quatre fonctions d'arrondi d'un réel ρ, de l'ensemble des réels vers l'ensemble des nombres flottants $\mathbb{F}_p^{\text{emax}}$ (figure 27) :

— Arrondi vers $-\infty$ (par défaut) $\mathsf{dn}(\rho)$;
— Arrondi vers $+\infty$ (par excès) $\mathsf{up}(\rho)$;
— Arrondi vers 0 (par excès pour les nombres négatifs et par défaut pour les positifs) $\mathsf{zo}(\rho)$;
— Arrondi au plus proche-pair $\mathsf{fl}(\rho)$: l'arrondi au plus proche-pair $\mathsf{fl}(\rho)$ est le nombre flottant le plus proche de ρ. Si ρ est à équidistance de deux nombres flottants, il est arrondi à celui dont le bit de poids faible de la partie fractionnaire est nul.

Lorsque le sens de l'arrondi n'a pas d'importance, on utilise la notation $\mathsf{rnd}(\rho)$. L'arrondi au plus proche-pair est utilisé par défaut par la plupart des langages de programmation et des applications. C'est le seul que l'on va considérer dans la suite.

Pour faire des preuves, une propriété importante de la fonction d'arrondi, quelle que soit celle utilisée, est qu'elle préserve l'ordre (*monotonicité de l'arrondi*) :

$$\forall(\rho_1,\rho_2)\in\mathbb{R}:\rho_1\leqslant\rho_2\implies\mathsf{rnd}(\rho_1)\leqslant\mathsf{rnd}(\rho_2)$$

La fonction d'arrondi est aussi idempotente ($\mathsf{rnd}(\mathsf{rnd}(\rho))=\mathsf{rnd}(\rho)$) et ne modifie pas les nombres flottants d'un même format :

$$\forall v\in\mathbb{F}_p^{\text{emax}}:\mathsf{rnd}(v)=v$$

Arrondi et nombres entiers

L'ensemble des entiers relatifs $\mathbb{Z}$ est inclus dans l'ensemble des réels $\mathbb{R}$. On pourrait donc facilement penser que tous les nombres entiers inclus dans le domaine $[-2^{\text{emax}+1}+1, 2^{\text{emax}+1}-1]$ d'un ensemble de nombres flottants $\mathbb{F}_p^{\text{emax}}$ sont représentables. Ce n'est pas le cas et il suffit pour s'en convaincre de considérer l'écart entre deux nombres flottants consécutifs ; pour un nombre flottant de la forme $m\times 2^E$, l'écart au suivant est égal à $2^{1-p}2^E$. L'écart entre deux nombres flottants consécutifs devient supérieur à 1 dès lors que l'on a :

$$2^{1-p}2^E>1$$

C'est à dire dès que :

$$E>p-1$$

Pour les nombres flottants double précision, par exemple, le domaine de $\mathbb{F}_{53}^{1023}$ où ne manque aucun nombre entier est restreint à $[-2^{53},2^{53}]$. À partir de 2^{53} on ne peut représenter que les entiers pairs ; à partir de 2^{54}, que les entiers multiples de 4, etc.

Dans le livre « *C in a nutshell*[44] », les auteurs définissent la fonction factorielle sur les nombres flottants de précision étendue *long double* (1 bit de signe, 15 bits d'exposant et 64 bits de signifiant, en général) en arguant du fait que cela permet de calculer plus de valeurs de la fonction qu'avec des entiers sur 64 bits. En effet, si l'on choisit de calculer la fonction factorielle sur un type entier de 64 bits, on ne peut pas représenter le résultat au-delà de 20! Le plus grand entier représentable avec le type *long double* est largement supérieur à $2^{64}-1$ mais, malheureusement, à partir de 2^{64}, tous les entiers ne sont pas représentables. Ainsi, la fonction présentée dans

le livre peut retourner un résultat pour des valeurs bien supérieures à 20 mais la plupart d'entre elles sont fausses à partir de $n = 26$.

[44] Peter Prinz et Tony Crawford. *C in a nutshell : the definitive reference*. Second edition. O'Reilly Media, 2015

Les exemples ci-après présentent les différents cas de figure de codage de nombres rationnels au format IEEE 754 simple précision. Pour coder un nombre décimal dans un format IEEE 754, on commence par le mettre sous forme normalisée. On peut alors déterminer les champs s, e, et f.

Exemple 7 (Représentation finie). *On veut coder le nombre* 21.59375_{10} *dans le format* simple précision.

— *On a :*

$$21_{10} = 10101_2$$

Pour connaître la représentation binaire d'un nombre inférieur à 1 *(ici,* 0.59375*), on le multiplie par* 2 *pour obtenir un résultat sous la forme «* $0 + x$ *» ou «* $1 + x$ *» avec* x *un nombre inférieur à* 1*. Si* x *vaut* 0*, on s'arrête et l'on reprend chaque chiffre* 0 *ou* 1 *dans l'ordre dans lesquels on les a trouvés ; sinon, on multiple* x *par 2 comme au départ. Ici, on a :*

$$\begin{aligned} 0.59375 \times 2 &= 1 + 0.1875 \\ 0.1875 \times 2 &= 0 + 0.375 \\ 0.375 \times 2 &= 0 + 0.75 \\ 0.75 \times 2 &= 1 + 0.5 \\ 0.5 \times 2 &= 1 + 0.0 \end{aligned}$$

On a donc :

$$0.59375_{10} = 0.10011_2$$

En réunissant la partie entière et la partie fractionnaire, on obtient :

$$21.59375_{10} = 10101.10011_2$$

— *On met sous forme normalisée le résultat précédent en faisant attention à ce que l'exposant ne dépasse pas les bornes (pour le format simple précision :* $E \in [-126, 127]$*) :*

$$\begin{aligned} 21.59375_{10} &= 10101.10011_2 \\ &= 1.010110011_2 \times 2^4 \end{aligned}$$

Si nécessaire, on arrondit la partie fractionnaire au nombre de chiffres stockables dans f*. Ici, la partie fractionnaire ne contient que* 9 *chiffres alors que l'on peut en stocker* 23*. On complète avec des zéros ;*

— *On a :*

$$\begin{cases} s = 0 \text{ car } 21.59375 \text{ est positif,} \\ e = 4 + 127 = 131_{10} = 10000011_2, \\ f = 01011001100000000000000; \end{cases}$$

— *On construit la représentation IEEE 754 en insérant chaque champ à sa place :*

$$21.59375 \to \underbrace{0}_{s}\ \underbrace{10000011}_{e}\ \underbrace{01011001100000000000000}_{f}$$

Exemple 8 (Représentation infinie cyclique en base 2). *On souhaite connaître la représentation du nombre* 13.1 *dans le format* simple précision.

— *On commence par coder la partie entière* 13_{10} *et la partie fractionnaire* 0.1_{10} *en binaire. On obtient :*

$$13_{10} = 1101_2$$

et

$$
\begin{aligned}
0.1 \times 2 &= 0 + 0.2 \\
0.2 \times 2 &= 0 + 0.4 \\
0.4 \times 2 &= 0 + 0.8 \\
0.8 \times 2 &= 1 + 0.6 \\
0.6 \times 2 &= 1 + 0.2 \\
0.2 \times 2 &= \ldots \\
&\vdots
\end{aligned}
$$

Il vient[45] *:*

$$0.1_{10} = 0.0\overline{0011}_2$$

[45] **Notation.** On notera $\overline{xyz}$ la répétition à l'infini de la séquence xyz.

— *En combinant la partie entière et la partie fractionnaire, on obtient :*

$$13.1_{10} = 1101.0\overline{0011}$$

— *On représente le résultat sous forme normalisée :*

$$
\begin{aligned}
13.1_{10} &= 1101.0\overline{0011}_2 \\
&= 1.1010\overline{0011}_2 \times 2^3
\end{aligned}
$$

*Le nombre de bits nécessaire pour la partie fractionnaire dépasse celui alloué par le format. On doit donc l'arrondir. On choisit l'*arrondi au plus proche-pair *par défaut. On a la partie fractionnaire :*

$$
\begin{aligned}
f &= 1010\overline{0011} \\
&= \underbrace{10100011001100110011001}_{23\ bits}\,10011\ldots
\end{aligned}
$$

Le 24^e *bit est un* 1 *et il y a d'autres bits non nuls dans la suite, donc* 13.1 *est plus près de* $\mathsf{up}(13.1)$ *que de* $\mathsf{dn}(13.1)$*. On arrondit donc le résultat à :*

$$f = 10100011001100110011010$$

— *On a :*

$$
\begin{cases}
s = 0 \text{ car } 13.1 \text{ est positif} \\
e = 3 + 127 = 130_{10} = 10000010_2 \\
f = 10100011001100110011010
\end{cases}
$$

En concaténant les différents champs, il vient :

$$13.1 \rightarrow 0\,10000010\,10100011001100110011010$$

De façon peut-être surprenante pour qui n'aurait pas lu l'annexe B, un nombre comme 0.3_{10} n'admet pas de représentation finie en base 2 et doit donc systématiquement être arrondi. Dans l'interpréteur Python, on ne s'en rend pas forcément compte immédiatement car l'affichage est fait pour donner l'illusion :

```
>>> f"{0.3:f}"
'0.300000'
```

Si l'on demande explicitement plus de chiffres, la vérité est démasquée :

```
>>> f"{0.3:.17f}"
'0.29999999999999999'
```

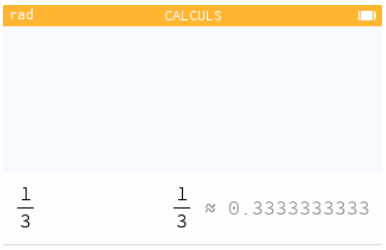


FIGURE 28 : Gestion des constantes sous forme rationnelle dans Numworks.

Dans la calculatrice Numworks, ce problème n'apparait généralement pas car elle est capable de stocker les valeurs sous forme rationnelle décimale (figure 28).

Exemple 9 (Représentation infinie sans cycle apparent en base 2)**.** *On veut coder au format simple précision le nombre* 18.13.

— *On code séparément en binaire la partie entière et la partie fractionnaire :*

$$18_{10} = 10010_2$$

et

$$\begin{aligned}
0.13 \times 2 &= 0 + 0.26 \\
0.26 \times 2 &= 0 + 0.52 \\
0.52 \times 2 &= 1 + 0.04 \\
0.04 \times 2 &= 0 + 0.08 \\
0.08 \times 2 &= 0 + 0.16 \\
0.16 \times 2 &= 0 + 0.32 \\
0.32 \times 2 &= 0 + 0.64 \\
0.64 \times 2 &= 1 + 0.28 \\
0.28 \times 2 &= 0 + 0.56 \\
0.56 \times 2 &= 1 + 0.12 \\
&\vdots
\end{aligned}$$

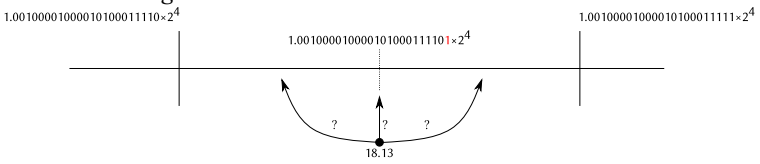
On a l'impression que le calcul de la représentation binaire de 0.13 *ne boucle pas mais ne va pas s'arrêter non plus. Comme on utilise le format* simple précision*, on peut stocker seulement* 23 *bits dans la partie fractionnaire. La représentation de la partie entière occupera déjà* 4 *bits (on exclut le bit de gauche de* 10010_2 *qui se trouvera à gauche de la virgule et ne sera donc pas codé). Il reste donc* 19 *bits à calculer pour la partie fractionnaire. On en a déjà* 11*. La suite est :*

$$\begin{aligned}
0.12 \times 2 &= 0 + 0.24 \\
0.24 \times 2 &= 0 + 0.48 \\
0.48 \times 2 &= 0 + 0.96 \\
0.96 \times 2 &= 1 + 0.92 \\
0.92 \times 2 &= 1 + 0.84 \\
0.84 \times 2 &= 1 + 0.68 \\
0.68 \times 2 &= 1 + 0.36 \\
0.36 \times 2 &= 0 + 0.72 \\
&\vdots
\end{aligned}$$

On a :

$$\begin{aligned}
18.13_{10} &= 10010.00010000101000111 10\cdots_2 \\
&= 1.00100001000010100011110\cdots_2 \times 2^4
\end{aligned} \tag{4}$$

Considérons la ligne réelle autour de la valeur 18.13 *:*



Les valeurs

$$1.00100001000010100011110 \times 10^{4}{}_{2}$$

et

$$1.00100001000010100011111 \times 10^{4}{}_{2}$$

sont représentables au format simple précision. Malheureusement, si l'on choisit d'arrondir au plus proche pair, *les bits de* 18.13 *de l'équation* (4) *ne permettent pas de savoir si ce nombre se trouve à gauche ou à droite du milieu :*

$$1.00100001000010100011110 1e4_{2}.$$

On doit donc calculer suffisamment de bits supplémentaires pour savoir comment arrondir. On a :

$$0.72 \times 2 = 1 + 0.44$$
$$0.44 \times 2 = 0 + 0.88$$
$$0.88 \times 2 = 1 + 0.76$$

On en déduit que 18.13 *se trouve à droite du milieu et qu'il faut donc l'arrondir à la valeur* $1.00100001000010100011111e4_{2}$. *Il vient donc :*

$$18.13_{10} = \underbrace{0}_{s} \underbrace{10000011}_{e} \underbrace{00100001000010100011111}_{f}$$

Quelle précision choisir ?

Lorsque le langage de programmation en laisse le choix, faut-il choisir la simple précision $\mathbb{F}_{24}^{127}$ ou la double précision $\mathbb{F}_{53}^{1022}$? Si les nombres manipulés sont susceptibles de sortir du domaine de définition de la simple précision, soit $(-2^{128}, 2^{128})$, la double précision devient indispensable. Jusqu'à l'introduction en 2000 des instructions SSE2 dans les processeurs Intel, l'argument de performances plus élevées en utilisant la simple précision plutôt que la double précision du fait de calculs moins coûteux car moins précis était spécieux car toutes les opérations sur les nombres flottants étaient faites, quel que soit leur format, sur un co-processeur arithmétique dans un format étendu sur 80 bits. À partir de 2000, les nouveaux processeurs ont offert la possibilité de faire les calculs flottants sur le co-processeur arithmétique 80 bits ou dans une mémoire propre aux instructions SSE. Il est alors possible de réaliser plusieurs opérations identiques simultanément sur des données différentes (modèle SIMD) et l'utilisation de la simple précision peut alors offrir de meilleures performances au détriment de la précision des calculs.

En règle générale, on considère cependant aujourd'hui que, à l'exception des cas où la précision requise ou l'espace de stockage des données sont faibles, il y a peu d'intérêt à utiliser la simple précision et l'on se bornera par défaut à utiliser la double précision pour tous les calculs. C'est d'ailleurs le choix fait par le langage Python et par la calculatrice Numworks pour les calculs approchés.

Des formats plus petits que la simple précision sont cependant de plus en plus utilisés dans des applications comme les *réseaux de neurones profonds* pour l'apprentissage dans le cadre de l'Intelligence Artificielle. Dans ces applications, la précision des calculs n'est pas cruciale. Par contre, il faut pouvoir représenter une quantité colossale de valeurs et faire des opérations simples dessus avec de très bonnes performances. Des formats de nombres flottants sur 16 bits — demie-précision — ont été employés par Suyog Gupta *et al.*[46] avant d'être remplacés par des formats encore plus petits sur 8 bits. Le sacrifice de précision limite cependant ces nouveaux

formats à ces applications de niches où ils excellent.

Lecture et affichage des nombres flottants : une question de précision. Le standard IEEE 754 considère qu'un format binaire de nombres flottants offre d chiffres décimaux de précision si tout nombre décimal à d chiffres peut être représenté de manière univoque par un nombre flottant, dans la limite des capacités du format. On doit ainsi pouvoir faire un aller-retour : un nombre décimal v doit pouvoir être stocké comme un nombre flottant qui, affiché avec d chiffres redonne exactement v.

Pour les entiers, on peut trouver la précision décimale d'un format sur b bits en cherchant le plus petit d tel que :

$$10^d > 2^b.$$

Le nombre de valeurs représentables en décimal doit être au moins égal au nombre de valeurs représentables en binaire. On obtient ainsi la formule :

$$d = \left\lceil \frac{b \ln 2}{\ln 10} \right\rceil \tag{5}$$

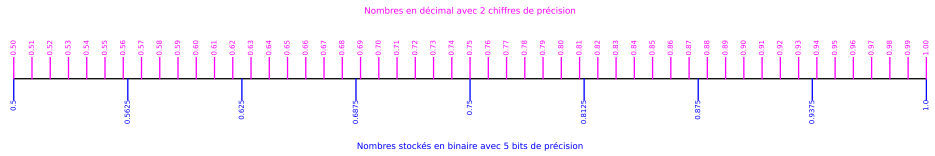


FIGURE 29 – Passage de base 10 à base 2 dans le domaine $[0.5, 1]$ (5 chiffres binaires, 2 chiffres décimaux).

Considérons le format $\mathbb{F}_5^1$, dont le signifiant est codé sur 5 bits. On peut donc s'attendre, selon la formule (5) à avoir $\mathrm{d}5 \ln 2 / \ln 10\mathrm{e} = 2$ chiffres décimaux de précision. Pourtant, si l'on considère les valeurs dans le domaine $[0.5, 1]$ (figure 29), on constate qu'il n'est pas possible de faire un aller-retour pour toutes les valeurs décimales avec deux chiffres de précision. C'est par exemple le cas pour la valeur 0.61, qui sera stockée sous la valeur arrondie 0.625 ; la valeur décimale sur deux chiffres la plus proche de 0.625 est 0.62 (en tenant compte d'un arrondi au plus proche-pair) et pas 0.61.

En 1967, Bennett GOLDBERG[47] a montré que l'équation (5) n'était pas utilisable pour les nombres flottants car elle suppose un écart constant entre les différentes valeurs. Suivant le domaine que l'on considère, on peut avoir plus ou moins de valeurs représentables en décimal ou en binaire avec une précision fixée. On voit bien sur la figure 29 qu'il y a plus de valeurs décimales sur deux chiffres que de valeurs binaires sur 5 bits entre 0.5 et 1 ; il est donc illusoire de vouloir effectuer un passage du décimal au binaire et retour de manière non ambiguë. Par contre, une telle opération ne poserait pas de problème dans le domaine $[1, 2]$ (figure 30).

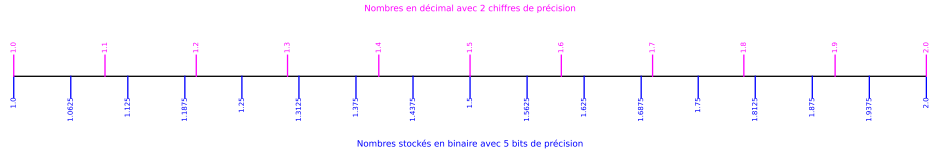


FIGURE 30 – Passage de base 10 à base 2 dans le domaine $[1, 2]$ (5 chiffres binaires, 2 chiffres décimaux).

[46] Suyog GUPTA et al. « Deep learning with limited numerical precision ». In : *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France : JMLR.org, juill. 2015, p. 1737-1746

[47] I. Bennett GOLDBERG. « 27 bits are not enough for 8-digit accuracy ». In : *Communications of the ACM* 10.2 (fév. 1967), p. 105-106

[48] David W. Matula. « In-and-out conversions ». In : *Communications of the ACM* 11.1 (jan. 1968), p. 47-50

[49] David W. Matula. « The base conversion theorem ». In : *Proceedings of the American Mathematical Society* 19.3 (1968), p. 716-723

David Matula[48,49] a montré qu'il est possible de faire un aller-retour du décimal au binaire et vice-versa avec des nombres flottants si l'on a la relation :

$$10^d < 2^{b-1} \tag{6}$$

où b et d sont, respectivement, le nombre de bits et de chiffres de précision en base 2 et 10. Les références citées présentent un résultat plus général pour n'importe quelle base de départ et d'arrivée mais la relation (6) suffit à nos besoins.

Si l'on applique la relation (6) à notre exemple, on voit que l'on n'a qu'un seul chiffre de précision pour un signifiant de 5 bits (figure 31).

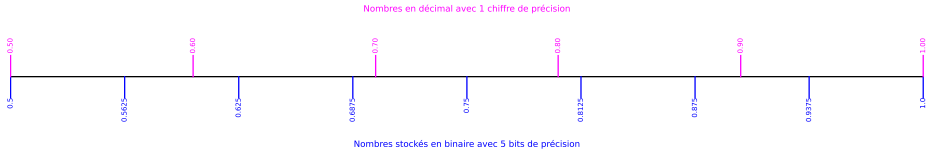


Figure 31 – Passage de base 10 à base 2 dans le domaine $[0.5, 1]$ (5 chiffres binaires, 1 chiffre décimal).

On peut aussi se poser le problème inverse et se demander combien de chiffres décimaux il faut calculer pour que la valeur décimale puisse être relue comme le nombre flottant de départ. Matula a montré que la formule (6) était valide pour différentes bases et l'on a la relation à vérifier :

$$2^b < 10^{d-1}$$

Pour les nombres flottants double précision (signifiant de 53 bits), cela veut dire que le nombre de chiffres décimaux d à calculer doit vérifier :

$$d > \frac{b \ln 2}{\ln 10} + 1.$$

La plus petit valeur entière vérifiant cette relation est $d = 17$.

On doit donc afficher un nombre flottant en double précision sur 17 chiffres pour être sûr qu'il sera relu comme le nombre flottant de départ. À l'inverse, en utilisant la relation (6), on voit que l'on ne peut pas avoir plus de $52 \ln 2 / \ln 10$ chiffres décimaux de précision, soit 15 chiffres, si l'on veut pouvoir stocker un nombre décimal de manière non ambiguë en double précision.

Lorsque l'on affiche en décimal un nombre flottant v, on peut avoir plusieurs valeurs susceptibles d'être relues sous la forme v. Le standard IEEE impose que la chaîne affichée soit la plus courte. Exemple : la valeur flottante double précision :

$$1.1001100110011001100110011001100110011001100110011010_2 \times 2^{-4}$$

peut être affichée sous différentes formes qui seront toutes relues comme cette valeur, par exemple :

```
>>> 0.10000000000000001
0.1
>>> 0.1
0.1
```

La convention choisie par le standard limite les surprises pour l'utilisateur, même si elle n'empêche pas certaines difficultés. Par exemple, le programme python :

```
v = 0.0
for i in range(10):
    v += 0.1
    print(v)
```

affiche :

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

car, par exemple, il n'est pas possible d'afficher la valeur obtenue en faisant l'opération $\mathrm{fl}(0.1 + 0.1 + 0.1)$ avec moins de 17 chiffres sans ambiguïté.

Représentation des réels dans le langage Python

Par défaut, les nombres à virgule manipulés en Python sont des nombres flottants au format IEEE 754 en double précision[50]. Le nom du type rapporté par la commande `type()` est « `float` » :

```
>>> type(4.5)
<class 'float'>
```

Ce nom peut prêter à confusion car dans d'autres langages de programmation (C, C++, Java, par exemple), il est utilisé pour parler de nombres flottants en simple précision.

Les infinis et les NaNs peuvent être créés grâce au constructeur du type `float` :

```
>>> float("inf")
inf
>>> float("nan")
nan
```

La division par un infini donne bien 0 ou -0 en fonction du signe des opérandes :

```
>>> 1/float("inf")
0.0
>>> (-1)/float("inf")
-0.0
```

Python ne respecte cependant pas complètement le standard IEEE 754 car il interdit la division par 0, pourtant parfaitement définie sur les nombres flottants :

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

De même, les calculs trop grands lèvent en général une exception plutôt que de retourner un infini :

```
>>> from math import exp
>>> exp(710.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: math range error
```

Il est cependant possible de retrouver un comportement un peu plus standard en utilisant le type `float64` de la bibliothèque *Numpy* :

```
>>> from numpy import float64, exp
>>> float64(1)/float64(0)
<stdin>:1: RuntimeWarning: divide by zero encountered in double_scalars
inf
>>> exp(float64(710))
inf
```

Un avertissement est affiché mais le résultat retourné correspond à ce que l'on attend dans le cadre du standard IEEE 754. Grâce à *Numpy*, il est même possible de faire des calculs en simple précision avec le type `float32`.

Le support des NaNs par Python est aussi partiel. Certaines opérations retournent bien un NaN, comme attendu par le standard IEEE 754 :

```
>>> float("inf")/float("inf")
nan
```

La plupart des cas de création de NaNs sont cependant souvent remplacés par la levée d'une exception :

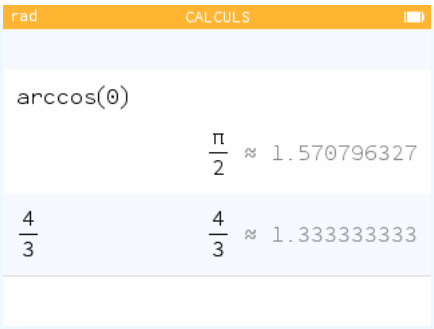
```
>>> from math import sqrt
>>> sqrt(-1.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

[50] En toute rigueur, la documentation du langage ne garantit pas l'utilisation du format IEEE 754 double précision, même si cela semble être universellement le cas en pratique.

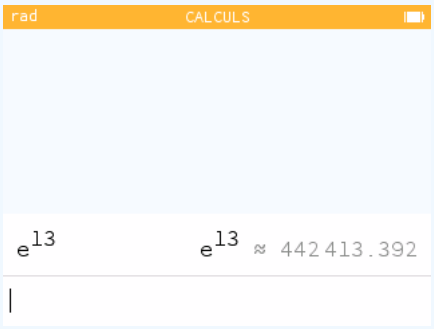
Représentation des « réels » dans la calculatrice Numworks

La calculatrice Numworks utilise plusieurs représentations des réels en fonction des besoins. Un nombre peut être représenté comme un rationnel (une paire d'entier et un bit de signe), un nombre binaire à virgule flottante au format IEEE 754 double précision, ou un nombre décimal à virgule flottante stocké dans un type *ad hoc*.

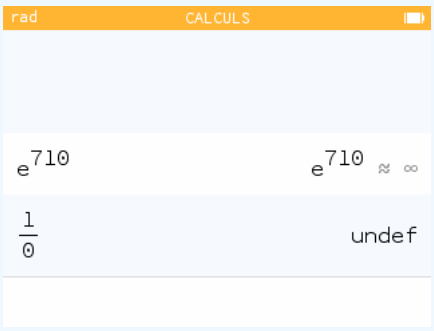
Dans le mode « *Calculs* », la calculatrice essaye de simplifier chaque expression avant d'en fournir une valeur numérique. Cela lui permet d'obtenir ainsi un résultat plus précis. Si le calcul a une valeur rationnelle ou une valeur transcendante bien identifiée, elle est retournée sous cette forme :



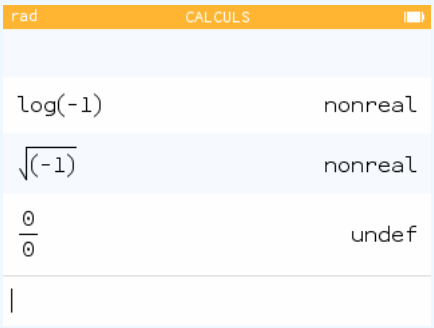
Sinon, une approximation en double précision est calculée :



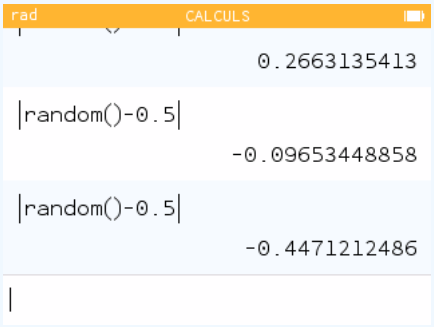
La gestion de l'infini n'est pas complète car la division par 0 n'est pas supportée :



Les NaNs n'existent pas en tant que tel sur la calculatrice. À la place, on trouve le résultat « `nonreal` » pour une opération définie sur les complexes mais pas sur les réels, et le résultat « `undef` » quand le calcul n'a pas de valeur définie :



La simplification des expressions avant leur évaluation peut augmenter la précision des calculs mais a des effets parfois indésirables lorsqu'elle requiert de précalculer certaines parties. Par exemple, l'expression $|\text{random}() - 0.5|$ retourne une fois sur deux une valeur négative car l'expression à l'intérieur de la valeur absolue est pré-évaluée pour en connaître le signe. S'il est positif, toute l'expression est remplacée par random() − 0.5, sinon par −(random() − 0.5). Lors de l'évaluation finale de l'expression pour afficher une valeur à l'utilisateur, la valeur retournée par la fonction random() change, ce qui peut invalider la simplification faite auparavant (issue #2040) :



3.2.2 Calculs d'erreurs

Étant donnée une constante c positive — le cas négatif est analogue — exprimée en décimal, le standard IEEE 754 impose que le nombre flottant utilisé pour la représenter soit celui qui lui est le plus proche, avec la règle de parité pour résoudre le cas d'équidistance, comme on l'a fait dans les exemples de la section précédente.

Soient ϕ_g et ϕ_d des éléments de $\mathbb{F}_p^{\text{emax}}$, avec ϕ_g (resp. ϕ_d) le plus grand flottant inférieur (resp., le plus petit flottant supérieur) ou égal à c. Si c est représentable dans $\mathbb{F}_p^{\text{emax}}$, on a :

$$\text{fl}(c) = c = \phi_g = \phi_d$$

Sinon, l'écart entre c et sa représentation arrondie au plus proche est inférieur ou égal à la moitié de la distance entre les flottants qui l'encadrent (figure 32) :

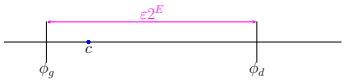


FIGURE 32 : Arrondi de la valeur c, avec $\phi_g = m_{\phi_g} 2^E$.

$$|\text{fl}(c) - c| \leqslant \frac{\phi_d - \phi_g}{2} \tag{7}$$

On doit considérer deux cas :

— *Si ϕ_g est normalisé.* Dans ce cas, c s'exprime sous la forme[51] $m_c 2^E$, avec $1 \leqslant m_c < 2$ et $\phi_g = m_{\phi_g} 2^E$ avec $1 \leqslant m_{\phi_g} < 2$. Si on note ε l'*epsilon de la machine* égal à 2^{1-p} (c'est aussi l'écart entre la valeur 1 et next (1)), on a : $\phi_d = \phi_g + \varepsilon \times 2^E$. De l'équation (7), on obtient :

$$|\text{fl}(c) - c| \leqslant \frac{\varepsilon}{2} \times 2^E.$$

On sait que c est différent de zéro car ce n'est pas un nombre flottant. On peut donc écrire :

$$\frac{|\text{fl}(c) - c|}{c} \leqslant \frac{\varepsilon/2 \times 2^E}{m_c \times 2^E}$$

Comme $1 \leqslant m_c < 2$, il vient :

$$|\text{fl}(c) - c| \leqslant \frac{\varepsilon}{2} c$$

Ou, alternativement :

$$\text{fl}(c) = c(1 + \delta), \quad \text{avec } |\delta| \leqslant \frac{\varepsilon}{2} \tag{8}$$

[51] Notez que, comme $c \in \mathbb{R}$, m_c possède potentiellement un nombre infini de bits après la virgule.

— *Si ϕ_g est dénormalisé ou nul.* La distance entre deux nombres dénormalisés adjacents est constante pour tous les nombres dénormalisés et égale à $\mu = \varepsilon \times 2^{1-emax}$. De l'équation (7), on a :

$$\left|\mathrm{fl}(c) - c\right| \leqslant \frac{\mu}{2}$$

que l'on peut exprimer comme :

$$\mathrm{fl}(c) = c + \eta, \text{ avec } \left|\eta\right| \leqslant \frac{\mu}{2} \tag{9}$$

On peut unifier les équations (8) et (9) :

$$\mathrm{fl}(c) = c(1+\delta) + \eta, \text{ avec } \begin{cases} \delta\eta = 0, \\ \left|\delta\right| \leqslant \frac{\varepsilon}{2}, \\ \left|\eta\right| \leqslant \frac{\mu}{2} \end{cases} \tag{10}$$

où δ est nul en cas d'*underflow* et η est nul sinon.

Le standard IEEE 754 impose que les opérations arithmétiques entre flottants (addition, soustraction, multiplication, division) ainsi que l'extraction de la racine carrée soient *correctement arrondies*, ce qui veut dire que leur résultat doit être la valeur flottante la plus proche du résultat réel. Le modèle (10) s'applique donc à ces opérations et l'on a pour les opérations binaires :

$\forall (\phi_1, \phi_2) \in (\mathbb{F}_p^{\mathrm{emax}})^2, \forall \mathrm{op} \in \mathrm{f}+, -, \times, \div \mathrm{g}:$

$$\mathrm{fl}(\phi_1 \,\mathrm{op}\, \phi_2) = (\phi_1 \,\mathrm{op}\, \phi_2)(1+\delta) + \eta, \text{ avec } \begin{cases} \delta\eta = 0, \\ \left|\delta\right| \leqslant \frac{\varepsilon}{2}, \\ \left|\eta\right| \leqslant \frac{\mu}{2} \end{cases} \tag{11}$$

Erreur absolue et erreur relative

Connaissant une valeur réelle exacte y et une valeur approchée calculée $\mathrm{fl}(y)$ de y, on définit l'*erreur absolue* par l'écart (en valeur absolue car, en général, seule nous intéresse la magnitude) entre ces deux valeurs :

$$E_a(\mathrm{fl}(y), y) = \left|\mathrm{fl}(y) - y\right|$$

Lorsque $y \neq 0$, on peut aussi définir l'*erreur relative* par :

$$E_r(\mathrm{fl}(y), y) = \frac{\left|\mathrm{fl}(y) - y\right|}{\left|y\right|}$$

L'erreur absolue est souvent moins intéressante lorsque l'on considère les calculs sur les nombres flottants car elle va forcément croître lorsque l'on s'éloigne de 0, comme l'écart entre deux flottants consécutifs augmente. Par exemple, lorsqu'en double précision le nombre $y = 2^{53} + 1$ est arrondi à $\mathrm{fl}(y) = 2^{53}$, l'erreur absolue est égale à 1, ce qui semble important. Pourtant la valeur calculée est la meilleure possible. Si l'on calcule l'erreur relative, on trouve $E_r = 1/(2^{53}+1)$, ce qui nous indique bien que l'erreur commise est inférieure à l'écart entre deux flottants consécutifs.

Avec ce modèle de calcul, on peut borner les erreurs maximales induites par un calcul sur les nombres flottants. Considérons, par exemple, deux algorithmes pour faire la somme de trois nombres flottants $(x, y, z) \in (\mathbb{F}_p^{\mathrm{emax}})^3$:

$$A_1 : (x+y)+z \qquad A_2 : x+(y+z)$$

Les deux algorithmes sont équivalents sur les réels du fait de l'associativité de l'addition. Qu'en est-il avec des nombres flottants ? Pour simplifier, on va supposer qu'aucun *underflow* n'a lieu lors des calculs.

On a :

$$\begin{aligned}\mathrm{fl}((a+b)+c) &= \mathrm{fl}(\mathrm{fl}(a+b)+c)\\ &= ((x+y)(1+\delta_1)+z)(1+\delta_2) \text{ avec } |\delta_1| \leqslant \frac{\varepsilon}{2}, |\delta_2| \leqslant \frac{\varepsilon}{2}\\ &= (x+y+z)+(x+y+z)\delta_2+(x+y)(\delta_1+\delta_1\delta_2)\end{aligned}$$

Si l'on néglige les très petits termes d'erreurs multipliés entre eux, il vient :

$$\mathrm{fl}((x+y)+z) \overset{1}{=} (x+y+z)\left[1+\delta_2+\delta_1\frac{x+y}{x+y+z}\right]$$

En calculant l'erreur relative, on trouve :

$$E_r(\mathrm{fl}((x+y)+z), x+y+z) = \frac{x+y}{x+y+z}\delta_1+\delta_2 \tag{12}$$

En faisant la même analyse, on va trouver :

$$E_r(\mathrm{fl}(x+(y+z)), x+y+z) = \delta_3+\frac{y+z}{x+y+z}\delta_4 \tag{13}$$

Des équations (12) et (13), on déduit que $(x+y)+z$ est différent de $x+(y+z)$ dans le cas général, car les erreurs d'arrondi commises sont différentes.

On voit aussi que l'on doit choisir l'algorithme à utiliser en fonction des valeurs $|x+y|$ et $|y+z|$: si $|x+y| \gg |y+z|$, l'erreur δ_1 dans l'équation (12) sera beaucoup plus amplifiée que l'erreur δ_4 dans l'équation (13), et inversement.

L'absorption

La somme de deux nombres flottants a et b est faite suivant un algorithme en trois étapes :

1. On égalise les exposants : le signifiant du nombre dont l'exposant est le plus petit est décalé vers la droite et son exposant augmenté de 1 pour chaque décalage d'un bit jusqu'à avoir des exposants identiques ;
2. On effectue la somme des signifiants ;
3. On renormalise le signifiant, si nécessaire, en ajustant l'exposant.

Par exemple, effectuons l'addition $5.0+0.1$ en simple précision. On va avoir une étape supplémentaire car avant de faire l'addition, il faut arrondir la valeur 0.1 :

$$\begin{aligned}0.1_{10} &= 1.10011001100110011001100\mathbf{11001}\cdots_2 \times 2^{-4}\\ \mathrm{fl}(0.1_{10}) &= 1.10011001100110011001101_2 \times 2^{-4}\end{aligned}$$

On a aussi : $5.0_{10} = 1.01000000000000000000000_2 \times 2^2$.

Il vient :

$$\begin{array}{rll} & 1.01000000000000000000000_2 & \times 2^2\\ + & 1.10011001100110011001101_2 & \times 2^{-4}\\ \hline \end{array}$$

On décale le signifiant du deuxième opérande de six positions vers la droite de façon à égaliser les exposants

$$\begin{array}{rll} & 1.01000000000000000000000_2 & \times 2^2\\ + & 0.00000110011001100110011\mathbf{001101}_2 & \times 2^2\\ \hline \end{array}$$

On doit arrondir de nouveau le deuxième opérande car les bits rouges ne sont pas représentables, puis on ajoute les significants :

$$\begin{array}{rll} & 1.01000000000000000000000_2 & \times 2^2 \\ + & 0.00000110011001100110011_2 & \times 2^2 \\ \hline & 1.01000110011001100110011_2 & \times 2^2 \end{array}$$

On a donc perdu de la précision à deux endroits : lors de la représentation de 0.1_{10} et lors de l'alignement des exposants. La perte de précision lors de l'alignement est ce que l'on appelle l'*erreur d'absorption*. Dans le pire des cas, si l'écart entre les deux exposants est supérieur ou égal à p (24, en simple précision et 53 en double précision), l'opérande le plus petit va perdre tous les bits de son significant et cela reviendra à sommer le plus grand opérande avec 0.

On peut voir le phénomène d'absorption à l'œuvre lorsque l'on fait une somme naïve des valeurs d'un tableau avec des magnitudes très différentes. C'est malheureusement la méthode utilisée par la fonction `sum()` de Python (les éléments sont sommés de la gauche vers la droite) :

```
>>> sum([2.0**52,0.5,-2.0**52])
0.0
```

La somme est faite en double précision. On commence par faire $2^{52} + 0.5$; l'alignement des exposants annule le deuxième opérande et le résultat est donc 2^{52}. En ajoutant ensuite -2^{52}, on obtient 0 au lieu de 0.5. Notez qu'ici, tous les éléments du tableau sont représentables en double précision sans erreur.

La fonction `SUM()` de LibreOffice Calc souffre du même problème et doit donc être utilisée avec prudence (figure 33).

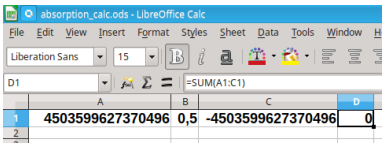


FIGURE 33 : Absorption avec la fonction `SUM()` de LibreOffice Calc.

On a vu précédemment que l'addition n'est pas transitive sur les nombres flottants. Plus largement, le choix de l'algorithme d'évaluation d'une expression a un impact non négligeable sur la correction du résultat final et des expressions équivalentes sur les nombres réels peuvent avoir des propriétés très différentes sur les nombres flottants. De nombreux travaux sont d'ailleurs dévolus à la recherche d'algorithmes corrects pour évaluer certaines expressions récurrentes.

Les langages de programmation peuvent rendre la situation encore plus compliquée car certains ne spécifient pas entièrement la façon d'évaluer une expression. La simple addition $a+b+c$ sans parenthèses peut être évaluée indifféremment comme $(a+b)+c$ ou $a+(b+c)$ en fonction des environnements dans des langages comme C ou C++[52]. L'usage des parenthèses est alors crucial pour garantir l'ordre d'évaluation souhaité par le programmeur, à condition que le compilateur n'essaye pas par la suite d'optimiser le code produit sans tenir compte des propriétés réduites de l'arithmétique sur les nombres flottants[53].

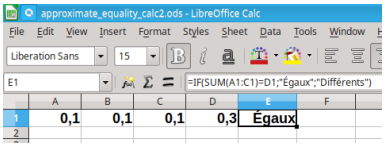


FIGURE 34 : Bénéfices de l'égalité approximative dans LibreOffice Calc.

[52] Source : cppreference.

[53] En Python, le langage garantit, lui, l'évaluation de gauche à droite.

L'égalité approximative. Les erreurs d'arrondi sont toujours source de surprise pour les utilisateurs non avertis et l'on trouve quantité de discussions dans les forums en ligne sur les dangers de l'arithmétique flottante quand des relations aussi simples que celle ci-dessous ne sont pas vérifiées :

```
>>> 0.1+0.1+0.1==0.3
False
```

Face à cela, certaines applications choisissent de mettre en œuvre une *égalité approximative* plutôt qu'une égalité stricte entre deux résultats flottants. Dans LibreOffice Calc, par exemple, l'exemple ci-dessus donne le résultat attendu (figure 34). Pour cela, LibreOffice Calc ne vérifie jamais l'égalité stricte de deux valeurs a et b ; il s'assure seulement que

l'écart relatif entre a et b est « suffisamment petit » :

$$|a-b| \leqslant \max(|a|,|b|)\varepsilon \implies a \doteq b, \tag{14}$$

avec $\doteq$ le symbole d'égalité approximative.

Cette « solution » pose autant de problème qu'elle en résout car l'égalité n'est plus alors une relation d'équivalence (on peut avoir $a = b$ et $b = c$ mais $a \neq c$). De plus, si a ou b est nul (mais pas les deux), l'égalité approximative suivant la formule (14) n'est jamais vérifiée. La figure 35 montre ce dernier problème dans LibreOffice Calc : la difference entre $0.222507386 \times 10^{-307}$ et 0 est très inférieure à celle entre $\mathrm{fl}(0.1 + 0.1 + 0.1)$ et $\mathrm{fl}(0.3)$ et pourtant, l'égalité avec 0 n'est pas vérifiée. De même des valeurs qui sont visiblement différentes (e^1 et $e^{1+2^{-52}}$) sont considérées comme égales.

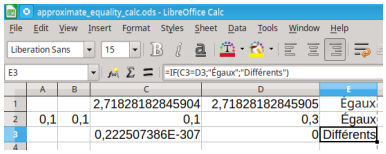


FIGURE 35 : Inconvénients de l'égalité approximative dans LibreOffice Calc.

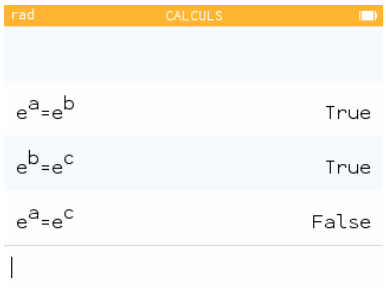


FIGURE 36 : Conséquence de l'égalité approchée sur la calculatrice Numworks.

La calculatrice Numworks utilise la même notion d'égalité que LibreOffice Calc dès lors que des résultats sont représentés par des nombres flottants en double précision. La figure 36 montre la perte de la relation d'équivalence : on crée une variable a égale à $1 - 2^{-52}$, une variable b égale à 1 et une variable c égale à $1 + 2^{-52}$. En prenant les exponentielles, e^a est suffisamment proche de e^b pour être déclaré égal ; il en est de même pour e^b et e^c. Par contre, la différence entre e^a et e^c est légèrement trop grande et l'on considère ces deux quantités comme différentes.

La cancellation

Lorsque deux nombres flottants sont très proches, leur soustraction peut être très précise car l'alignement des exposants est faible ou inexistant ; On a même des théorèmes par STERBENZ et HAUSER qui garantissent que la soustraction de deux nombres flottants x et y est sans erreur ($\mathrm{fl}(x-y) = x-y$) si $y/2 \leqslant x \leqslant 2y$.

Lorsque les deux opérandes de la soustraction sont issus de calculs, ils peuvent être entachés d'erreurs ; si les approximations sont proches, seuls les bits entachés d'erreur resteront après la soustraction. Considérons, par exemple les valeurs approchées en simple précision :

$$\begin{cases} \widehat{x} &= 1.11001101010010011101101 \times 2^{-5} \\ \widehat{y} &= 1.11001101010010011011010 \times 2^{-5}, \end{cases}$$

où les bits en rouge sont entachés d'erreur. La soustraction $\mathrm{fl}(\widehat{x} - \widehat{y})$ donne :

$$\begin{array}{rr} 1.11001101010010011101101 & \times 2^{-5} \\ - \quad 1.11001101010010011011010 & \times 2^{-5} \\ \hline 0.00000000000000000010011 & \times 2^{-5} \end{array}$$

Après normalisation, on obtient le résultat $\mathrm{fl}(\widehat{x} - \widehat{y}) = 1.0011 \times 2^{-25}$, où les seuls bits significatifs du résultat qui restent sont ceux entachés d'erreur. C'est ce que l'on appelle une *cancellation catastrophique*. Ce phénomène est susceptible d'apparaitre dans l'évaluation de nombreuses expressions importantes, comme le discriminant $\Delta = b^2 - 4ac$, où b^2 et ac peuvent être proches et entachés d'une erreur d'arrondi.

Analyse de la résolution par radicaux d'une équation quadratique. Les livres de mathématiques nous apprennent que l'on peut trouver les racines d'un polynôme quadratique de la forme $ax^2 + bx + c$ en calculant d'abord le discriminant :

$$\Delta = b^2 - 4ac. \tag{15}$$

— Si Δ est négatif, le polynôme n'admet pas de racine réelle ;
— si Δ est nul, le polynôme admet une racine double :

$$x_1 = x_2 = -\frac{b}{2a}$$

[54] Pat H. STERBENZ. *Floating-point computation*. Prentice-Hall series in automatic computation. [Englewood Cliffs, N.J : Prentice-Hall, 1973. ISBN : 978-0-13-322495-5

[55] John R. HAUSER. « Handling floating-point exceptions in numeric programs ». In : *ACM Transactions on Programming Languages and Systems* 18.2 (mars 1996), p. 139-174. (Visité le 20/05/2020)

— Si Δ est strictement positif, le polynôme admet deux racines :

$$x_1 = \frac{-b+\sqrt{\Delta}}{2a} \qquad x_2 = \frac{-b-\sqrt{\Delta}}{2a} \tag{16}$$

La mise en œuvre directe de cet algorithme dans un langage de programmation se heurte à la réalité du calcul sur les nombres flottants. D'un point de vue purement qualitatif, on peut s'attendre à avoir des problèmes lorsque le calcul de Δ requiert la soustraction de deux valeurs positives, b^2 et $4ac$, qui sont des résultats de calculs (cancellation potentielle). Or, la précision est cruciale pour la prise de décision qui suit (une légère variation de Δ peut avoir un impact important sur le nombre de solutions attendues). Analysons l'erreur commise lors du calcul du discriminant :

$$\mathrm{fl}\big(b^2 - 4ac\big) = \big[b^2(1+\delta_1) - 4ac(1+\delta_2)\big](1+\delta_3), \quad |\delta_i| \leqslant \frac{\varepsilon}{2}$$

Notez que δ_2 ne quantifie que l'erreur de la multiplication ac. En l'absence d'*overflow*, la multiplication par 4 qui suit ne génère pas d'erreur et correspond à une simple modification de l'exposant du résultat de ac. On obtient :

$$\mathrm{fl}\big(b^2 - 4ac\big) = b^2(1+\delta_1+\delta_3+\delta_1\delta_3) - 4ac(1+\delta_2+\delta_3+\delta_2\delta_3)$$

et donc

$$\mathrm{fl}\big(b^2 - 4ac\big) - (b^2 - 4ac) = b^2(\delta_1+\delta_3+\delta_1\delta_3) - 4ac(\delta_2+\delta_3+\delta_2\delta_3)$$

On peut ignorer les termes d'erreurs multipliés entre eux car ils sont très petits par rapport aux autres. De même, on peut borner $\delta_1+\delta_3$ et $\delta_2+\delta_3$ par δ tel que $|\delta| \leqslant \varepsilon$. Il vient alors :

$$\big|\mathrm{fl}\big(b^2 - 4ac\big) - (b^2 - 4ac)\big| \leqslant (b^2 + 4\,|ac|)\delta$$

L'erreur relative dans le calcul du discriminant est donc :

$$\frac{\big|\mathrm{fl}\big(b^2 - 4ac\big) - (b^2 - 4ac)\big|}{\big|b^2 - 4ac\big|} \leqslant \frac{(b^2 + 4\,|ac|)}{\big|b^2 - 4ac\big|}\delta$$

On en déduit que si b^2 est très proche de $4ac$, l'erreur relative peut devenir très grande.

Exemple 10. *Considérons l'équation quadratique* $x^2 + (1+2^{-52})x + (\frac{1}{4} + 2^{-53}) = 0$. *On peut calculer le discriminant :*

$$\Delta = (1+2^{-52})^2 - 4\times 1\times(\frac{1}{4} + 2^{-53}) = 2^{-104}$$

Comme Δ *est strictement positif, il y a deux solutions :*

$$\begin{cases} x_1 &= \frac{-(1+2^{-52})+\sqrt{2^{-104}}}{2} &= \frac{-1-2^{-52}+2^{-52}}{2} &= \frac{-1}{2} \\ x_2 &= \frac{-(1+2^{-52})-\sqrt{2^{-104}}}{2} &= \frac{-1-2^{-52}-2^{-52}}{2} &= \frac{-1-2^{-51}}{2} \end{cases}$$

La valeur de b^2 *ne peut pas être représentée en double précision et l'on obtient* $\widehat{b^2} = 1+2^{-51}$. *La calculatrice Numworks évalue le discriminant de manière naïve. Lorsque l'on utilise son mode « Equations » pour trouver les solutions de notre équation, on en obtient une seule (figure 37) et sa valeur est erronée :*

$$x_1 = -\frac{4503599627370497}{9007199254740992} = \frac{-1-2^{-52}}{2}$$

Le code Python fourni dans le paquetage polynomial.py *pré-installé sur la calculatrice utilise le même algorithme et a donc le même problème (figure 38).*

Il n'existe pas de solution magique pour résoudre le problème lié au calcul du discriminant, pas de réécriture de l'expression $b^2 - 4ac$ *qui garantirait la précision nécessaire pour déterminer le bon nombre de solutions. Par contre, il est toujours possible d'évaluer le discriminant avec une précision plus grande en espérant résoudre la majorité des cas.*

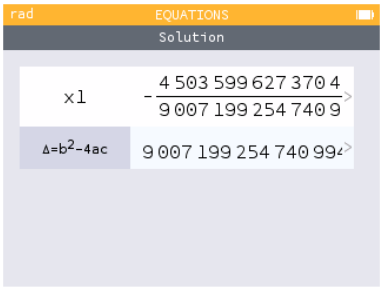


FIGURE 37 : Résolution de l'équation $x^2 + (1+2^{-52})x + (\frac{1}{4} + 2^{-53}) = 0$ à deux solutions avec le mode « Equations » de la calculatrice Numworks.

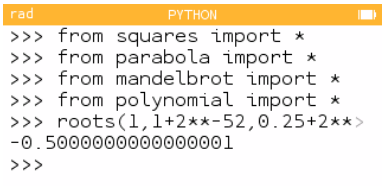


FIGURE 38 : Résolution de l'équation $x^2 + (1+2^{-52})x + (\frac{1}{4} + 2^{-53}) = 0$ à deux solutions avec le paquetage *polynomial.py* de la calculatrice Numworks.

Le calcul du discriminant n'est cependant pas le seul problème qui se pose à nous : lorsque b est proche de $\sqrt{\Delta}$, le calcul de x_1 (si $b > 0$) ou de x_2 (si $b < 0$) peut se trouver entaché d'une erreur importante par cancellation. Cela arrivera par absorption dès lors que $b^2 \gg 4ac$.

Exemple 11. *Considérons l'équation quadratique*

$$x^2 + 2^{27}x + \frac{3}{4} = 0.$$

On a :

$$\Delta = 2^{27^2} - 4 \times 1 \times \frac{3}{4} = 2^{54} - 3$$

Il y a donc deux solutions :

$$x_1 = \frac{-2^{27} + \sqrt{2^{54} - 3}}{2}$$
$$x_2 = \frac{-2^{27} - \sqrt{2^{54} - 3}}{2}$$

Si l'on effectue le calcul en double précision, on peut se rappeler que dans le domaine $[2^{53}, 2^{54}]$, on n'est capable de représenter que les entiers pairs. Donc $2^{54} - 3$ va être arrondi. Par ailleurs, $\sqrt{2^{54} - 3}$ est très proche de 2^{27}. Aussi, on va avoir une cancellation lors du calcul de x_1 en soustrayant de $-b$ une valeur approchée de $\sqrt{\Delta}$.

Les solutions de cette équation sont :

$$x_1 \approx -5.587935447692871 \times 10^{-9}$$
$$x_2 \approx -1.342177280000000 \times 10^{8}$$

Si l'on utilise les formules (16) pour x_1 et x_2, on obtient en double précision :

$$x_1 = -7.450580596923828 \times 10^{-9}$$
$$x_2 = -1.34217728 \times 10^{8}$$

La solution x_2 est calculée avec précision ; par contre la solution x_1, victime de cancellation, n'a aucun chiffre significatif correct.

Pour éviter la cancellation, on peut réécrire différemment la formule donnant x_1 en multipliant le numérateur et le dénominateur par $-b - \sqrt{\Delta}$:

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} = \frac{-b + \sqrt{\Delta}}{2a} \times \frac{-b - \sqrt{\Delta}}{-b - \sqrt{\Delta}}$$

Notons que $-b - \sqrt{\Delta}$ ne peut être nul que si $ac = 0$. Dans ce cas, il est possible de résoudre l'équation dégénérée différemment.

En simplifiant, il vient :

$$x_1 = \frac{2c}{-b - \sqrt{\Delta}}$$

Le point important est que cette réécriture a inversé le signe de l'opération entre $-b$ et $\sqrt{\Delta}$. On n'aura donc pas de cancellation. Si on évalue x_1 avec cette nouvelle formule, on obtient :

$$x_1 = -5.587935447692871 \times 10^{-9}$$

comme attendu.

Lorsque b est négatif, il suffit de réécrire la formule pour x_2 de la même manière.

Il semble que la calculatrice Numworks évalue les solutions en utilisant les formules (16). Elle obtient donc une solution erronée, comme dans notre exemple (figure 39).

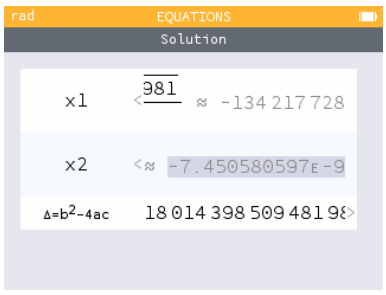


FIGURE 39 : Cancellation sur une solution d'équation quadratique par la calculatrice Numworks.

La résolution d'une équation quadratique est l'un de ces problèmes faussement simples pour lesquels il existe des méthodes mathématiques qui sont bien connues de tous mais qui ne peuvent pas être directement mises en œuvre sur un ordinateur utilisant l'arithmétique flottante sous peine de sévères déceptions. Comme pour un certain nombres de problèmes classiques, il existe de nombreux travaux sur des méthodes visant à calculer précisément les solutions en arithmétique flottante[56,57], travaux trop souvent ignorés par les développeurs et développeuses de code numérique.

[56] George E. Forsythe. *How do you solve a quadratic equation ?* Technical Report CS-TR-66-40. Computer Science Department, Stanford University, juin 1966

[57] William Kahan. « On the cost of floating-point computation without extra precise arithmetic ». Nov. 2004. URL : http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf

Les fonctions transcendantes. Depuis sa première version en 1985, le standard impose que les opérateurs arithmétiques ainsi que la racine carrée soient *correctement arrondis*. La version de 1985 ne considérait pas du tout le cas des fonctions transcendantes (log, sin, ...) et la dernière version de 2019 ne fait que suggérer un arrondi correct pour ces fonctions, sans l'imposer. Contrairement aux opérateurs arithmétiques, où la borne maximum du nombre de bits à calculer pour savoir comment arrondir le résultat est connue à l'avance et de taille raisonnable, ce n'est pas le cas pour les fonctions transcendantes. Il a fallu de nombreuses années de recherche et parfois des calculs exhaustifs pour trouver des bornes pour la plupart des fonctions transcendantes, mais ces bornes sont parfois tellement élevées que le coût d'un arrondi correct devient très important. Afin de ne pas faire payer ce prix à des applications qui n'en auraient pas l'usage — on peut cependant se demander qui n'a pas besoin d'un résultat correct ? —, il a été décidé de ne pas imposer cette contrainte pour les fonctions transcendantes. Un système peut donc respecter le standard IEEE 754 et retourner des valeurs fantaisistes pour toutes ces fonctions.

Exemple 12 (Calcul de $\ln(x)$). *Pour la double précision, Vincent Lefèvre et Jean-Michel Muller ont montré*[58] *que le pire cas du point de vue de la détermination du sens de l'arrondi lors du calcul du logarithme népérien était atteint avec la valeur*

$$x = 1.737429606443346566788426 \times 10^{204}.$$

On a alors :

$$\ln(x) = 1.\overbrace{10101100100011110011110101110100111110010010111000 1}^{52\ bits}\underbrace{0000\cdots000}_{65\ zéros}1110\cdots \times 2^8$$

Pour cette fonction, il faut donc prévoir de calculer jusqu'à 119 bits (53 bits de signifiant, 65 zéros et un bit supplémentaires à 1) pour savoir comment arrondir le résultat.

[58] Vincent Lefèvre et Jean-Michel Muller. « Worst Cases for Correct Rounding of the Elementary Functions in Double Precision ». report. INRIA,LIP, 2000. URL : https://hal.inria.fr/inria-00072594

Exemple — Comment tirer un 7 avec un dé à six faces.

La calculatrice Numworks offre une fonction `randint(a,b)` pour tirer aléatoirement un entier dans le domaine $[a, b]$ suivant une loi uniforme. On l'a utilisée pour simuler cent millions de tirages d'un dé à six faces et l'on a noté les valeurs obtenues. La figure 40 présente la distribution des résultats. Pas de surprise, ou presque : les entiers de 1 à 6 sont tirés à peu près aussi souvent les uns que les autres. Par contre, on voit que la valeur 7 a été tirée cinq fois... Comment est-ce possible ?

L'algorithme utilisé pour tirer un entier dans un domaine $[a, b]$ est le suivant :

1. On calcule un nombre entier v_e pseudo-aléatoire dans le domaine $[0, 2^{32}-1]$;
2. On divise v_e par 2^{32} pour obtenir une valeur en simple précision $v_f \in [0, 1)$;
3. On retourne
$$\left\lfloor v_f(b+1-a)+a \right\rfloor. \tag{17}$$

L'erreur faite n'est pas unique à la calculatrice Numworks et on la retrouve dans

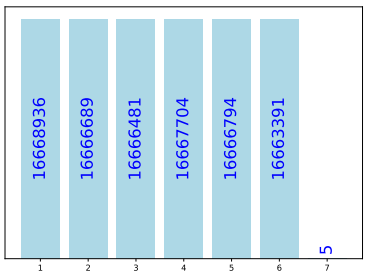


Figure 40 : Simulation de 100 000 000 tirages d'un dé à 6 faces avec la fonction `randint()` de la calculatrice Numworks.

d'autres langages et applications[59], comme Scratch et Snap!: en divisant un entier dans $[0, 2^{32} - 1]$ par 2^{32}, on obtient tous les multiples de 2^{-32} entre 0 et $1 - 2^{-32}$. Cependant, tous ces multiples ne sont pas représentables en simple précision. Le flottant immédiatement avant 1, par exemple, est $1-2^{-24}$. Tous les multiples de 2^{-32} supérieurs ou égaux à $1-2^{-25}$ seront donc arrondis à 1. Dans ce cas, la formule (17) retournera la valeur 7 pour $a = 1$ et $b = 6$. On a donc une probabilité de $2^7/2^{32}$ — soit approximativement une chance sur trente-trois millions — de retourner une valeur en dehors du domaine demandé. Incidemment, le fait que certains résultats de la division par 2^{32} sont arrondis pour produire v_f implique qu'il existe un biais dans la probabilité de chaque valeur flottante d'être produite (il existe une seule façon d'obtenir 2^{-32} mais 128 façons d'obtenir 1, par exemple); le tirage n'est donc pas complètement uniforme.

[59] Frédéric Goualard. « Generating Random Floating-Point Numbers by Dividing Integers : a Case Study ». In : *Proceedings of the International Conference on Computational Science*. Sous la dir. de V. Krzhizhanovskaya. T. 12138. Lecture Notes in Computer Science. Amsterdam, The Netherlands : Springer, juin 2020, p. 15-28

3.2.3 Calcul exact

Du fait de la représentation interne des nombres flottants, certains calculs sont toujours exacts. C'est le cas de la multiplication par une puissance de 2 — si elle ne conduit pas à un *overflow* — car elle correspond à une simple incrémentation de l'exposant. De même, la division par une puissance de 2 se fait par simple décrémentation de l'exposant; elle est donc exacte si elle n'induit pas d'*underflow* (lorsque les calculs nous amènent dans le domaine des nombres dénormalisés, l'exposant ne varie plus mais le signifiant est décalé vers la droite et peut donc perdre des bits significatifs).

En observant que l'erreur d'arrondi d'une addition ou d'une soustraction de deux nombres flottants est un nombre flottant représentable, Ole Møller[60] mit au point dès 1965 une méthode pour calculer exactement la somme de deux nombres flottants a et b sous la forme de deux autres nombres flottants : la somme arrondie $s = \mathrm{fl}(a + b)$ et l'erreur $e = (a + b) - \mathrm{fl}(a + b)$. On a ainsi : $a + b = s + e$. Par la suite, des méthodes du même type ont été développées pour la multiplication. On a aujourd'hui de nombreux algorithmes qui utilisent cette arithmétique à deux résultats pour garder trace des différentes erreurs et les compenser, obtenant ainsi un résultat de meilleure qualité. Ces méthodes sont devenues tellement importantes que Jason Riedy et James Demmel proposèrent[61] en 2018 de les intégrer au standard IEEE 754, ce qui fut fait en dans la révision de 2019 avec les *opérations augmentées* recommandées d'addition, de soustraction et de multiplication[62].

[60] Ole Møller. « Quasi double-precision in floating point addition ». In : *BIT* 5.1 (mars 1965), p. 37-50

[61] Jason Riedy et James Demmel. « Augmented Arithmetic Operations Proposed for IEEE-754 2018 ». In : *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. Juin 2018, p. 45-52

[62] *IEEE Standard for Floating-Point Arithmetic*. Rapp. tech. IEEE Std 754-2019 (Revision of IEEE 754-2008). Institute of Electrical et Electronics Engineers (IEEE), juill. 2019

Exemple — Somme exacte

Étant donnés deux nombres flottants a et b, avec a de magnitude supérieure ou égale à celle de b, l'algorithme *Fast2Sum* de Theodorus Dekker[63] retourne dans s la somme arrondie au plus proche-pair $\mathrm{fl}(a + b)$ et l'erreur $e = (a + b) - \mathrm{fl}(a + b)$, telles que $a + b = s + e$:

```
def Fast2Sum(a,b):
    s = a + b
    z = s - a
    e = b - z
    return (s,e)
```

Exemple d'utilisation :

```
>>> Fast2Sum(0.25,0.23)
(0.48, 2.7755575615628914e-17)
>>> Fast2Sum(2.0**53,1)
```

```
(9007199254740992.0, 1.0)
```

4 Alternatives aux nombres flottants

Certaines applications requièrent des garanties sur les calculs effectués. L'expertise pour définir et mettre en œuvre un code de calcul apportant ces garanties n'est pas toujours disponible. Il existe cependant heureusement des alternatives aux nombres flottants dès lors que les performances ne sont pas le goulot d'étranglement de l'application.

En Python, on peut utiliser la bibliothèque decimal pour faire du calcul en nombre flottant décimal plutôt que binaire, en choisissant la précision voulue :

```
>>> from decimal import *
>>> Decimal("0.1")+Decimal("0.1")+Decimal("0.1")==Decimal("0.3")
True
```

Il existe aussi des bibliothèques de calcul en nombres flottants binaires avec une précision définissable par l'utilisateur. Cela ne résout évidemment pas tous les problèmes de représentation des valeurs à développement infini mais cela permet d'étudier le comportement des calculs avec différentes précisions. Les bibliothèques gmp et mpfr sont des piliers dans ce domaine. Elles sont accessibles en Python avec la bibliothèque gmpy2 :

```
>>> import gmpy2
>>> from gmpy2 import mpfr
>>> gmpy2.get_context().precision=100
>>> mpfr("0.1")
mpfr('0.1000000000000000000000000000002',100)
```

Exemple — La précision variable : une panacée ?

Considérons la fonction de Rump suivante :

$$f(a,b) = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

Quelle est la valeur de $f(77617, 33096)$? Le programme Python ci-dessous évalue f en simple et en double précision :

```
from numpy import single, double

def rump(a,b):
    T=type(a)
    return (T(333.75) - a**2)*b**6 + a**2*(T(11)*a**2*b**2 -
                                           T(121)*b**4 -
                                           T(2))+T(5.5)*b**8+a/(T(2)*b)

print(f"{rump(single(77617.0),single(33096.0)):.6f}")
print(f"{rump(double(77617.0),double(33096.0)):.15f}")
```

On obtient les résultats :

```
1.172604
1.172603940053179
```

[63] Theodorus J. Dekker. « A floating-point technique for extending the available precision ». In : *Numerische Mathematik* 18.3 (juin 1971), p. 224-242

Les deux résultats concordent dans les limites des précisions respectives des formats utilisés. On a donc une certaine confiance dans la valeur obtenue. Pourtant, on peut montrer que :

$$f(77617, 33096) \approx -0.827396059946821$$

Même le signe est faux ! Utilisons la bibliothèque gmpy2 pour évaluer f avec des précisions croissances :

```
import gmpy2
from gmpy2 import mpfr

def rump(a,b):
    T=type(a)
    return (T(333.75) - a**2)*b**6 + a**2*(T(11)*a**2*b**2 -
                                           T(121)*b**4 -
                                           T(2))+T(5.5)*b**8+a/(T(2)*b)

for prec in [24,53,113,120,122,130,200,300]:
    gmpy2.get_context().precision=prec
    print(prec,"\t",rump(mpfr('77617'),mpfr('33096')))
```

On obtient le résultat :

```
24      1.17260396
53      1.1726039400531787
113     1.172603940053178631858834904520 1838
120     1.172603940053178631858834904520 1837076
122     -0.827396059946821368141165095479816292 01
130     -0.827396059946821368141165095479816291 99961
200     -0.827396059946821368141165095479816291 99903311578438
           481991781452
300     -0.827396059946821368141165095479816291 99903311578438
           481991781484167270969301426154218032390621
```

Par une analyse minutieuse, on peut montrer[64] qu'il faut calculer avec une précision d'au minimum 122 bits pour obtenir la valeur de la fonction avec une précision satisfaisante, ce que confirme notre programme ci-dessus. On voit donc que, sans analyse poussée, on ne peut pas simplement ré-évaluer une fonction avec une précision croissante en considérant que des résultats concordants sont un brevet de correction.

Il est aussi possible d'utiliser l'arithmétique flottante classique à l'intérieur de méthodes de calcul destinées à déterminer la précision du résultat : c'est l'*arithmétique stochastique*, implémentée par exemple dans l'outil CADNA[65]. L'expression à évaluer est recalculée plusieurs fois en faisant varier aléatoirement le sens de l'arrondi de chaque opération qui la compose. L'ensemble des résultats est ensuite analysé pour déterminer le nombre de chiffres correct à prendre en compte.

Enfin, l'*arithmétique d'intervalles* remplace chaque opération sur les nombres flottants par une opération sur une paire de nombres flottants correspondant aux bornes de l'intervalle contenant le résultat réel avec certitude. La bibliothèque PyInterval offre ce genre d'arithmétique en Python :

```
>>> from interval import interval
>>> interval(1)/interval(10)
interval([0.09999999999999999, 0.1])
```

[64] Eugene Loh et G. William Walster. « Rump's Example Revisited ». In : *Reliable Computing* 8.3 (juin 2002), p. 245-248

[65] À notre connaissance, il n'existe pas encore de bibliothèque Python pour utiliser ce type de méthode.

5 Pour aller plus loin

Le livre « *Handbook of floating-point arithmetic* » par Jean-Michel Muller *et al.*[66] est une très bonne introduction à l'arithmétique en virgule flottante et va loin dans la présentation de ses propriétés et même de sa mise en œuvre dans les processeurs. Le chapitre « *Analyser et encadrer les erreurs dues à l'arithmétique flottante* » par Claude-Pierre Jeannerod et Nathalie Revol[67] du livre « *Informatique Mathématique : une photographie en 2017* » présente plus succinctement les éléments permettant d'analyser le calcul en nombres flottants. Il est disponible gratuitement en ligne et en français. Le livre « *Accuracy and stability of numerical algorithms* » par Nicholas Higham[68] est une référence très claire dans l'analyse des algorithmes utilisant l'arithmétique flottante, en particulier de toutes les méthodes linéaires. L'article « *What every computer scientist should know about floating-point arithmetic* » de David Goldberg[69] est un classique depuis sa parution, même si son style très sec et académique le réserve à des lecteurs déjà aguerris. Le livre « *The end of error : unum computing* » par John Gustafson[70], présente, malgré un style enflammé non absent de mauvaise foi, une description intéressante des inconvénients de l'arithmétique flottante et propose une nouvelle manière de faire des calculs sur ordinateur. Le livre « *Elementary functions : algorithms and implementation* », par Jean-Michel Muller[71], est une présentation très précise de l'implémentation des fonctions classiques sur les nombres flottants. À réserver à des lecteurs ou lectrices passionné-e-s par le sujet, cependant. Enfin, l'article « *Pitfalls in computation, or why a math book isn't enough* » de George Forsythe,[72] quoiqu'un peu ancien, est une lecture intéressante car il met l'accent sur ces formules mathématiques classiques qui ne peuvent pas être réutilisées impunément sur un ordinateur, et il se fait l'avocat d'une meilleure compréhension des propriétés de l'arithmétique flottante.

[66] Jean-Michel Muller et al., éd. *Handbook of floating-point arithmetic*. Boston, Mass. Basel Berlin : Birkhäuser, 2010. ISBN : 978-0-8176-4704-9

[67] Claude-Pierre Jeannerod et Nathalie Revol. *Analyser et encadrer les erreurs dues à l'arithmétique flottante*. Pages : 115-144. CNRS Editions, jan. 2017. URL : https://hal.inria.fr/hal-01658296

[68] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. 2nd ed. Philadelphia : Society for Industrial et Applied Mathematics, 2002

[69] David Goldberg. « What every computer scientist should know about floating-point arithmetic ». In : *ACM Computing Surveys (CSUR)* 23.1 (mars 1991), p. 5-48. (Visité le 20/05/2020)

[70] John L. Gustafson. *The end of error : unum computing*. Chapman & Hall/CRC computational science series. CRC Press, an imprint of the Taylor & Francis Group, 2015

[71] J. M. Muller. *Elementary functions : algorithms and implementation*. 2nd ed. Boston : Birkhäuser, 2006

[72] George E. Forsythe. *Pitfalls in computation, or why a math book isn't enough*. Technical Report. Stanford, CA, USA : Stanford University, 1970

Références

[1] David P. Bellamy, Jeffrey C. Lagarias et Felix Lazebnik. *Proposed Problem ; Large Values of Tan n*. Mars 2022. URL : https://cpb-us-w2.wpmucdn.com/sites.udel.edu/dist/3/7714/files/2022/03/tan_n.pdf.

[2] David P. Bellamy et Felix Lazebnik. « 10656 ». In : *The American Mathematical Monthly* 105.4 (1998). Publisher : Mathematical Association of America, p. 366-366. ISSN : 0002-9890. (Visité le 03/04/2023).

[3] David P. Bellamy et al. « Large Values of Tangent : 10656 ». In : *The American Mathematical Monthly* 106.8 (1999), p. 782-784.

[4] Sylvie Boldo. « Formal Verification of Programs Computing the Floating-Point Average ». In : *Formal Methods and Software Engineering*. Sous la dir. de Michael Butler, Sylvain Conchon et Fatiha Zaïdi. Lecture Notes in Computer Science. Springer International Publishing, 2015, p. 17-32.

[5] Sylvie Boldo. « Kahan's Algorithm for a Correct Discriminant Computation at Last Formally Proven ». In : *IEEE Transactions on Computers* 58.2 (fév. 2009), p. 220-225.

[6] Carlos F. Borges. « Algorithm 1014 : An Improved Algorithm for hypot(x,y) ». In : *ACM Transactions on Mathematical Software* 47.1 (déc. 2020), 9 :1-9 :12.

[7] W. Buchholz. « Fingers or fists ? (the choice of decimal or binary representation) ». In : *Communications of the ACM* 2.12 (déc. 1959), p. 3-11.

[8] Arthur W. Burks, Herman H. Goldstine et John von Neumann. *Preliminary Discussion of the logical design of an electronic computing instrument*. Technical Report. The Institute for Advanced Study, juin 1946.

[9] Lord Byron. *'Alas ! the love of Women !' Byron's Letters and Journals*. Sous la dir. de Leslie A. Marchand. T. 3. The Belknap Press of Harvard University Press, 1974.

[10] David H.C. Chen. *The removal/demotion of MinNum and MaxNum operations from IEEE 754-2008*. Rapp. tech. Fév. 2017.

[11] Stephen Chrisomalis. *Numerical notation : a comparative history*. Cambridge, New York : Cambridge University Press, 2010.

[12] J. T. Coonen. « An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic ». In : *Computer* 13.1 (jan. 1980), p. 68-79.

[13] Theodorus J. Dekker. « A floating-point technique for extending the available precision ». In : *Numerische Mathematik* 18.3 (juin 1971), p. 224-242.

[14] Jan Dürre, Guillermo Payá-Vayá et Holger Blume. « Teaching Digital Logic Circuit Design via Experiment-Based Learning - Print your own Logic Circuit ». In : *Proceedings of the World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2016)*. Juill. 2016.

[15] George E. Forsythe. *How do you solve a quadratic equation ?* Technical Report CS-TR-66-40. Computer Science Department, Stanford University, juin 1966.

[16] George E. Forsythe. *Pitfalls in computation, or why a math book isn't enough*. Technical Report. Stanford, CA, USA : Stanford University, 1970.

[17] Bob Fraley et Steve Walther. « Proposal to eliminate denormalized numbers ». In : *ACM SIGNUM Newsletter* 14.si-2 (oct. 1979), p. 22-23.

[18] David Goldberg. « What every computer scientist should know about floating-point arithmetic ». In : *ACM Computing Surveys (CSUR)* 23.1 (mars 1991), p. 5-48. (Visité le 20/05/2020).

[19] I. Bennett Goldberg. « 27 bits are not enough for 8-digit accuracy ». In : *Communications of the ACM* 10.2 (fév. 1967), p. 105-106.

[20] Frédéric Goualard. « Generating Random Floating-Point Numbers by Dividing Integers : a Case Study ». In : *Proceedings of the International Conference on Computational Science*. Sous la dir. de V. Krzhizhanovskaya. T. 12138. Lecture Notes in Computer Science. Amsterdam, The Netherlands : Springer, juin 2020, p. 15-28.

[21] Suyog Gupta et al. « Deep learning with limited numerical precision ». In : *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France : JMLR.org, juill. 2015, p. 1737-1746.

[22] John L. Gustafson. *The end of error : unum computing*. Chapman & Hall/CRC computational science series. CRC Press, an imprint of the Taylor & Francis Group, 2015.

[23] John R. Hauser. « Handling floating-point exceptions in numeric programs ». In : *ACM Transactions on Programming Languages and Systems* 18.2 (mars 1996), p. 139-174. (Visité le 20/05/2020).

[24] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. 2nd ed. Philadelphia : Society for Industrial et Applied Mathematics, 2002.

[25] Gyula Horváth et Tom Verhoeff. « Numerical difficulties in pre-university informatics education and competitions ». In : *Informatics in education* 2.1 (jan. 2003), p. 21-38.

[26] David G. Hough. « The IEEE Standard 754 : One for the History Books ». In : *Computer* 52.12 (déc. 2019), p. 109-112.

[27] *IEEE Standard for Binary Floating-Point Arithmetic*. American National Standard (ANSI) IEEE Std 754-1985. The Institute of Electrical et Electronics Engineers, Inc, 1985.

[28] *IEEE Standard for Floating-Point Arithmetic*. Rapp. tech. IEEE Std 754-2019 (Revision of IEEE 754-2008). Institute of Electrical et Electronics Engineers (IEEE), juill. 2019.

[29] Claude-Pierre Jeannerod et Nathalie Revol. *Analyser et encadrer les erreurs dues à l'arithmétique flottante*. Pages : 115-144. CNRS Editions, jan. 2017. URL : https://hal.inria.fr/hal-01658296.

[30] William Kahan. « Miscalculating Area and Angles of a Needle-like Triangle ». Manuscript. Sept. 2014.

[31] William Kahan. « On the cost of floating-point computation without extra precise arithmetic ». Nov. 2004. URL : http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf.

[32] William M. Kahan. « Interval Arithmetic Options in the Proposed IEEE Floating Point Arithmetic Standard ». In : *Interval Mathematics 1980*. Sous la dir. de KARL L. E. Nickel. Academic Press, jan. 1980, p. 99-128.

[33] Felix Lazebnik. « Surprises ». In : *Mathematics Magazine* 87.3 (2014). Publisher : [Mathematical Association of America, Taylor & Francis, Ltd.], p. 212-221.

[34] Vincent Lefèvre et Jean-Michel Muller. « Worst Cases for Correct Rounding of the Elementary Functions in Double Precision ». report. INRIA,LIP, 2000. URL : https://hal.inria.fr/inria-00072594.

[35] Jennifer S. Light. « When Computers Were Women ». In : *Technology and Culture* 40.3 (1999), p. 455-483.

[36] Eugene Loh et G. William Walster. « Rump's Example Revisited ». In : *Reliable Computing* 8.3 (juin 2002), p. 245-248.

[37] Boris Nikolaevitch Malinovsky. *Pioneers of Soviet Computing*. Sous la dir. d'Anne Fitzpatrick. Seconde éd. SIGCIS : The Special Interest Group for Computing, Information, et Society, 2010.

[38] David W. Matula. « In-and-out conversions ». In : *Communications of the ACM* 11.1 (jan. 1968), p. 47-50.

[39] David W. Matula. « The base conversion theorem ». In : *Proceedings of the American Mathematical Society* 19.3 (1968), p. 716-723.

[40] Robert K. Merton. *The Sociology of Science : Theoretical and Empirical Investigations*. Sous la dir. de Norman W. Storer. Chicago, IL : University of Chicago Press, sept. 1979.

[41] Ole Møller. « Quasi double-precision in floating point addition ». In : *BIT* 5.1 (mars 1965), p. 37-50.

[42] J. M. Muller. *Elementary functions : algorithms and implementation*. 2nd ed. Boston : Birkhäuser, 2006.

[43] Jean-Michel Muller. « On the Error of Computing $ab + cd$ using Cornea, Harrison and Tang's Method ». In : *ACM Transactions on Mathematical Software* 41.2 (fév. 2015), p. 1-8.

[44] Jean-Michel Muller et al., éd. *Handbook of floating-point arithmetic*. Boston, Mass. Basel Berlin : Birkhäuser, 2010. ISBN : 978-0-8176-4704-9.

[45] John von Neumann. *First draft of a report on the EDVAC*. Technical Report W-670-ORD-4926. Moore School of Electrical Engineering, University of Pennsylvania, juin 1945.

[46] Mary Payne et William Strecker. « Draft proposal for a binary normalized floating point standard ». In : *ACM SIGNUM Newsletter* 14.si-2 (oct. 1979), p. 24-28.

[47] Peter Prinz et Tony Crawford. *C in a nutshell : the definitive reference*. Second edition. O'Reilly Media, 2015.

[48] Jason Riedy et James Demmel. « Augmented Arithmetic Operations Proposed for IEEE-754 2018 ». In : *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. Juin 2018, p. 45-52.

[49] R. Rojas. « Konrad Zuse's legacy : the architecture of the Z1 and Z3 ». In : *IEEE Annals of the History of Computing* 19.2 (avr. 1997), p. 5-16.

[50] Raul Rojas. « The Z1 : Architecture and Algorithms of Konrad Zuse's First Computer ». Preprint arXiv :1406.1886 [cs]. Juin 2014.

[51] Siegfried M. Rump. « Algorithms for Verified Inclusions : Theory and Practice ». In : *Reliability in Computing*. Sous la dir. de Ramon E. Moore. Academic Press, jan. 1988, p. 109-126.

[52] Claude Elwood Shannon. « A symbolic analysis of relay and switching circuits ». Mém. de mast. Massachusetts Institute of Technology, 1937.

[53] Pat H. Sterbenz. *Floating-point computation*. Prentice-Hall series in automatic computation. [Englewood Cliffs, N.J : Prentice-Hall, 1973. isbn : 978-0-13-322495-5.

[54] Michael R. Williams. *A history of computing technology*. 2nd ed. Los Alamitos, Calif : IEEE Computer Society Press, 1997.

Annexes

A Représentation des entiers dans Python

Jusqu'en 2001, la version 2 de Python offrait deux types d'entiers :

— Les « `int` », typiquement mis en œuvre par des entiers de 32 ou 64 bits, suivant l'architecture de la machine ;

— Les « `long` »[73], qui étaient des entiers avec une précision variable.

Les deux types coexistaient de manière bien séparée ; une expression dont les opérandes étaient de type « `int` » avait un résultat de type « `int` », quitte à lever une exception si le résultat était trop grand pour y être codé sans erreur.

Avec le *Python Enhancement Proposal 237*, il fut décidé que le passage du type `int` au type `long` se ferait de façon implicite dès lors que le type `int` n'était plus assez grand pour représenter un résultat. D'un point de vue mise en œuvre, le type `int` de Python 2 est représenté par le type `long` de C, d'une taille minimale de 32 bits ; sur une machine relativement récente, la taille est plutôt de 64 bits. Dans ce cas, on peut représenter tous les entiers entre -2^{63} et $2^{63}-1$ avec le type `int`. En dehors de ce domaine, les valeurs passent automatiquement en précision variable :

```
Python 2.7.17 (default, Nov 28 2022, 18:51:39)
[GCC 7.5.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 9223372036854775807 # 2**63-1
9223372036854775807
>>> 9223372036854775808 # 2**63
9223372036854775808L
>>> (-2)**63
-9223372036854775808
>>> (-2)**63-1
-9223372036854775809L
```

Le suffixe « `L` » identifie les nombres au format « `long` ».

Depuis la version 3 de Python, il n'existe plus qu'un seul type d'entier : le type « `int`, » correspondant désormais à un entier en précision variable. Chaque entier apparaissant dans un programme Python 3 est stocké à la demande dans une partie de la mémoire de l'ordinateur appelée « tas » sous la forme d'un tableau d'entiers. Cela requiert une *allocation dynamique* de mémoire, qui est un processus coûteux en temps. De plus, la structure contenant un entier occupe *au minimum* 28 octets en mémoire, soit trois fois plus de place qu'un entier classique sur 64 bits :

```
>>> from sys import getsizeof
>>> a=4
>>> getsizeof(a)
28
```

Un entier n en précision variable est stocké dans une structure contenant deux champs :

— Un tableau de k entiers sur 32 bits correspondant aux « chiffres » de n exprimés en base 2^{30} ;

— Un entier signé sur 32 ou 64 bits contenant la valeur k si n est positif et $-k$ sinon.

Pour coder l'entier 370 087 804 339 228 847 192, on aurait la représentation interne de la figure 41 car :

$$370\,087\,804\,339\,228\,847\,192 = 321 \times 2^{30^2} + 1267 \times 2^{30^1} + 76888 \times 2^{30^0}$$

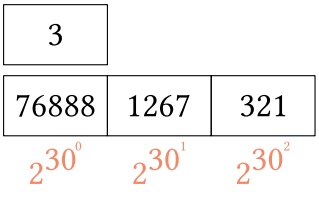


FIGURE 41 : Représentation interne du nombre 370 087 804 339 228 847 192 dans Python 3.

[73] Après avoir appelé « `float` » les nombres flottants en double précision alors que plusieurs langages majeurs utilisaient ce terme pour la simple précision, les développeurs de Python ont récidivé en utilisant « `int` » et « `long` » pour faire référence à des choses différentes des habitudes admises — en C, C++ et Java, un « `int` » fait généralement 32 bits et un « `long` », 64 bits — et différentes de la version 2 à la version 3 du langage.

Afin d'augmenter les performances du langage, les petits entiers compris dans l'intervalle $[-5, 256]$ voient leur représentation interne pré-construite en mémoire. Ainsi, si un entier dans ce domaine apparait dans du code Python, on évite le coût de la construction de la structure pour le représenter.

En plus de cela, les entiers dans le domaine $[-2^{30}+1, 2^{30}-1]$ bénéficient d'un traitement spécial : comme ils ne nécessitent qu'une seule case pour les stocker, Python essaye de les manipuler avec la même efficacité que des entiers de taille fixe.

```
>>> a=2**30-1
>>> getsizeof(a)
28
>>> a=2**30
>>> getsizeof(a)
32
```

Le programme ci-dessous montre l'impact de cette optimisation : une première boucle effectue mille fois la somme des entiers de 0 à 46340, qui est inférieure à 2^{30} ; une deuxième boucle fait la même chose mais en décalant tous les entiers manipulés de 2^{30}. À l'exécution, la première boucle s'exécute en trois fois moins de temps que la deuxième.

```
import time

# Somme des entiers de 0 à 46340
debut = time.perf_counter()
for i in range(1000):
    L = list(range(46341))
    somme = sum(L)
fin = time.perf_counter()
print(fin-debut)

# Somme des entiers de 2**30+0 à 2**30+46340
debut = time.perf_counter()
for i in range(1000):
    L = [i+2**30 for i in range(46341)]
    somme = sum(L)
fin = time.perf_counter()
print(fin-debut)
```

Le coût de manipulation des entiers reste cependant élevé dans le langage Python 3 du fait de l'indirection effectuée à chaque accès à un entier, qui n'existe pas dans des langages de programmation utilisant une représentation de taille fixe standard.

B Représentation finie et infinie

On a vu que certains nombres dont la représentation est finie en base 10 ont une représentation infinie en base 2. Dans quelles conditions cela peut-il arriver ? À l'inverse, existe t-il des nombres avec une représentation finie en base 2 qui nécessiteraient un nombre infini de chiffres en base 10 ?

Pour répondre à ces questions, généralisons le problème et demandons-nous quelles sont les conditions pour qu'un nombre x ait une représentation finie dans une base b. Afin de ne pas devoir intégrer le signe dans les notations, on ne considérera que des nombres positifs, le cas des nombres négatifs étant tout à fait analogue.

Représentation finie en base b

Soit $(c_{k-1}c_{k-2}\cdots c_0.c_{-1}c_{-2}\cdots)_b$ la représentation en base b d'un nombre x. Si x admet une représentation finie sur $n+k$ chiffres en base b, on peut décaler la virgule vers la droite jusqu'à la positionner après le dernier chiffre c_{-n} en partie fractionnaire, ce qui revient à multiplier le nombre par b^n :

$$\begin{aligned} x &= (c_{k-1}c_{k-2}\cdots c_0.c_{-1}c_{-2}\cdots c_{-n})_b \\ &= \frac{(c_{k-1}c_{k-2}\cdots c_0c_{-1}c_{-2}\cdots c_{-n})_b}{b^n} \end{aligned}$$

On aura donc :

Proposition 1. *Un nombre x admet une représentation finie en base b s'il existe un entier positif ou nul n tel que xb^n est un entier.*

Quelle que soit la base considérée, x ne peut admettre une représentation finie s'il est irrationnel. On peut donc se concentrer sur les nombres rationnels seulement. Dans ce cas, on peut exprimer x comme la *fraction irréductible* de deux entiers :

$$x = \frac{p}{q},\ (p,q) \in \mathbb{N}^2$$

D'après la proposition 1, pour que x ait une représentation finie en base b, on doit pouvoir trouver n tel que :

$$\frac{p}{q}b^n \in \mathbb{N}$$

Comme p/q est irréductible, p et q ne partagent aucun facteur premier. La seule possibilité pour que xb^n soit entier est donc que b^n contiennent tous les facteurs premiers de la décomposition en facteurs de q. Avec $b^n = qr$ (pour $r \in \mathbb{N}$), on obtient :

$$\frac{p}{q}b^n = pr \in \mathbb{N}$$

On en déduit :

Proposition 2. *Un nombre positif rationnel $x = p/q$ (avec $pgcd(p,q) = 1$) admet une représentation finie en base b si et seulement si*[74] *:*

$$\mathcal{D}_q \subseteq \mathcal{D}_b$$

[74] **Notation.** Étant donné un entier positif a, on note $\mathcal{D}_a$ l'ensemble des diviseurs premiers de a. Exemple : $\mathcal{D}_{15435} = \{1,3,5,7\}$ car $15435 = 1 \times 3^2 \times 5 \times 7^3$.

Exemple 13. *Le nombre $x = 0.43512$ a une représentation finie en base 10 car on a :*

$$x = \frac{43512}{100000} = \frac{5439}{12500} = \frac{5439}{2^2 \times 5^5}$$

Il suffit de prendre $n = 5$ pour avoir :

$$x \times 10^5 = \frac{5439}{2^2 \times 5^5} \times (2 \times 5)^5 = 5439 \times 2^3 \in \mathbb{N}$$

À l'inverse, le nombre $x = \frac{1}{3}$ *n'admet pas de représentation finie en base* 10 *car* 3 *et* 10 *n'ont aucun facteur premier en commun. Par contre, il a une représentation finie en base* 3 *puisque l'on peut prendre* $n = 1$ *pour avoir :*

$$x \times 3^1 = \frac{1}{3} \times 3^1 = 1 \in \mathbb{N}$$

Représentation finie en bases b_1 et b_2

Sous quelles conditions sur b_1 et b_2, un nombre positif x ayant une représentation finie en base b_1 a t-il aussi une représentation finie dans une base b_2 ?

Considérons l'ensemble des nombres ayant une représentation finie en base b_1 :

$$\mathcal{U} = \left\{ p/q \mid \text{pgcd}(p,q) = 1 \text{ et } \mathcal{D}_q \subseteq \mathcal{D}_{b_1} \right\}$$

Une condition suffisante simple pour que tous les éléments de $\mathcal{U}$ aient une représentation finie en base b_2 est que tous les diviseurs de b_1 soient aussi des diviseurs de b_2 : $\mathcal{D}_{b_1} \subseteq \mathcal{D}_{b_2}$.

On peut facilement vérifier que c'est aussi une condition nécessaire. En effet, considérons le nombre $1/r$ avec r un diviseur de b_1 mais pas de b_2. Dans ce cas, d'après la proposition 2, $1/r$ est représentable en base b_1 mais pas en base b_2.

On a donc :

Proposition 3. *Tous les nombres entiers ayant une représentation finie en base* b_1 *ont une représentation finie en base* b_2 *si et seulement si tous les diviseurs de* b_1 *sont des diviseurs de* b_2 *:*

$$\mathcal{D}_{b_1} \subseteq \mathcal{D}_{b_2}$$

Application aux base 2 et 10

On peut appliquer directement la proposition 3 aux base 2 et 10 :

- Comme tous les diviseurs de 2 sont aussi des diviseurs de 10, tous les nombres ayant une représentation finie en base 2 ont aussi une représentation finie en base 10 ;
- À l'inverse, tous les diviseurs de 10 ne sont pas des diviseurs de 2. En particulier, comme l'on a :

$$\mathcal{D}_2 = \text{f1, 2g}$$
$$\mathcal{D}_{10} = \text{f1, 2, 5g}$$

tout nombre x s'exprimant sous la forme de la fraction irréductible p/q, avec $5 \in \mathcal{D}_q$ n'admet pas de représentation finie en base 2.

C Galerie d'exemples

C.1 Étude d'une fonction quadratique dans GeoGebra

On veut étudier avec GeoGebra la fonction quadratique déjà vue page 42 :

$$f(x) = x^2 + (1 + 2^{-52})x + (\frac{1}{4} + 2^{-53})$$

La figure 42 montre que GeoGebra représente incorrectement la fonction f en considérant qu'elle ne s'annule qu'une seule fois, pour $x = -0.5$.

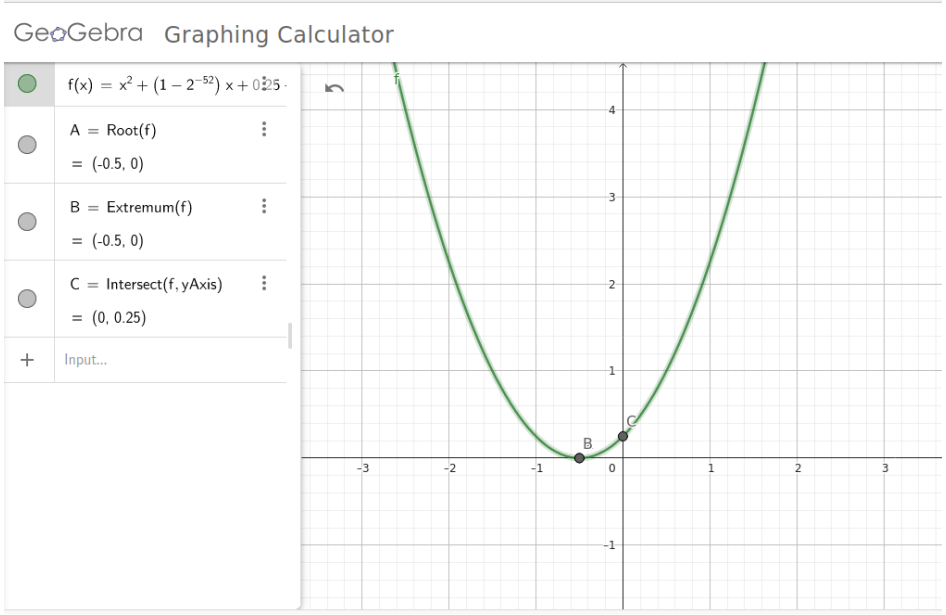


FIGURE 42 – Étude de la fonction $f(x) = x^2 + (1 + 2^{-52})x + (\frac{1}{4} + 2^{-53})$ dans GeoGebra.

On a vu précédemment que cette fonction admet deux racines très proches et qu'il est important de calculer le discriminant du polynôme quadratique associé avec une précision étendue pour en obtenir la bonne valeur.

À l'inverse, GeoGebra semble implémenter correctement le calcul des racines de f lorsque le discriminant est calculé précisément. Par exemple, on voit sur la figure 43 que l'outil détermine correctement les deux racines du polynôme vu page 44 :

$$f(x) = x^2 + 2^{27}x + \frac{3}{4}.$$

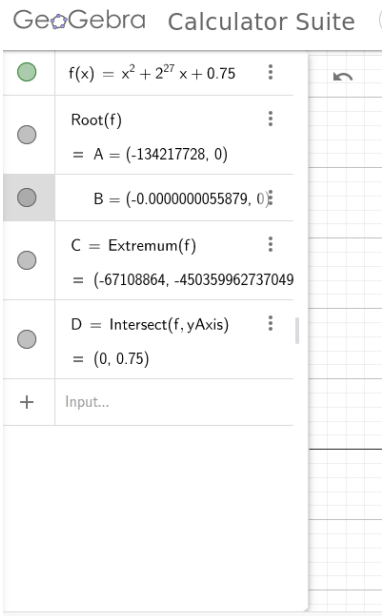


FIGURE 43 : Étude de la fonction $f(x) = x^2 + 2^{27}x + \frac{3}{4}$ dans GeoGebra.

C.2 Détermination de la limite d'une suite

La suite de KAHAN et MULLER :

$$\begin{cases} U_0 & = 2 \\ U_1 & = -4 \\ U_{n+2} & = 111 - \frac{1130}{U_{i+1}} + \frac{3000}{U_i U_{i+1}} \end{cases} \tag{18}$$

converge vers la valeur 6. Vérifions cela avec un programme Python qui calcule les 50 premier termes :

```
n = 50
Ui  =  2
Ui1 = -4
for i in range(2,n+1):
    Ui2 = 111 - 1130/Ui1 + 3000/(Ui*Ui1)
    Ui  = Ui1
    Ui1 = Ui2
print(Ui2)
```

On obtient :

```
100.0
```

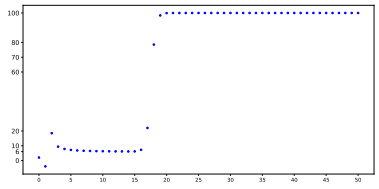


FIGURE 44 : Les 50 premiers termes de la suite de KAHAN-MULLER calculés en Python.

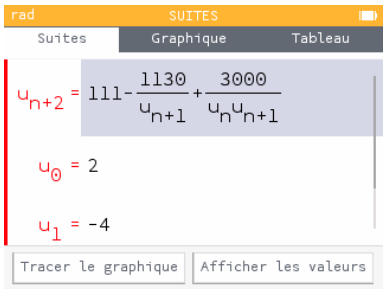


FIGURE 45 : Calcul des termes de la suite de KAHAN-MULLER.

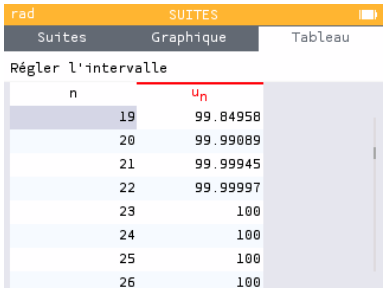


FIGURE 46 : Calcul des 50 premiers termes de la suite de KAHAN-MULLER avec la calculatrice Numworks.

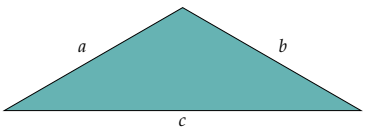


FIGURE 47 : Calcul de l'aire d'un triangle.

La figure 44 présente les différents termes calculés.

Testons alors en utilisant l'application « Suites » de la calculatrice Numworks (figure 45). À partir de U_{23}, on atteint aussi le point-fixe 100 (figure 46).

La récurrence (18) admet trois points-fixes : 5, 6 et 100. Les deux premiers points-fixes sont répulsifs alors que le troisième est attractif. Le point-fixe atteint à partir de $U_0 = 2$ et $U_1 = -4$ est 6 si l'on travaille avec des nombres réels. Il suffit d'une toute petite erreur de calcul (ici, les arrondis successifs lors des calculs sur les nombres flottants) pour que l'on sorte de la trajectoire menant à 6. Comme le point-fixe 100 est attractif, la nouvelle trajectoire nous y conduit invariablement.

C.3 Aire d'un triangle avec la formule de Héron

L'aire d'un triangle de longueurs des côtés a, b et c (figure 47) peut se calculer avec la formule de HÉRON :

$$S = \sqrt{p(p-a)(p-b)(p-c)}, \text{ avec } p = \frac{a+b+c}{2}.$$

Une implémentation directe en Python serait :

```
def heron(a,b,c):
    """
        Calcul de l'aire d'un triangle de longueurs de côtés `a`, `b` et `c`.
        Précondition: les valeurs `a`, `b` et `c` doivent vérifier
        l'inégalité triangulaire.
    """
    s = 0.5*(a+b+c)
    return sqrt(s*(s - a)*(s - b)*(s - c))
```

Comme rappelé par le commentaire dans la fonction, on doit s'assurer avant l'appel que les longueurs a, b et c définissent bien un triangle.

On voit aisément que si p est proche de a, b ou c, on va avoir une cancellation lors de la soustraction $p-a$, $p-b$, ou $p-c$. Pour le triangle $(10000, 10000, 19999.999999999996)$, on obtient, par exemple, en utilisant des nombres flottants en double précision :

```
>>> heron(10000,10000,19999.999999999996)
2.697398304697218
```

Cependant, l'aire de ce triangle vaut approximativement 1.9073486328124996. On a :

$$c \approx 2a$$
$$\mathrm{fl}(p) = 2a$$

Lors de la soustraction $p - c$, on a une cancellation qui va se répercuter sur le résultat.

On a le même problème avec des triangles où l'un des côtés est très petit par rapport aux deux autres. Le triangle $(10^{10}, 10^{10}, 10^{-4})$ a une aire proche de 500000. Avec la formule de Héron, on obtient :

```
>>> heron(1e10,1e10,1e-4)
495910.64453125
```

Une formule alternative proposée par Kahan[75] permet d'éviter le phénomène de cancellation. On trie a, b, et c de façon à avoir $a \geqslant b \geqslant c$ et l'on calcule :

$$\frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}$$

Lors du calcul, il est important que les parenthèses soient respectées ; il est crucial de ne pas chercher à optimiser l'expression, un travail qu'un compilateur optimisant risque malheureusement de faire dans le dos du programmeur, ruinant ainsi le travail effectué.

Une mise en œuvre de cette formule en Python serait :

```
def kahan(a,b,c):
    """
       Calcul de l'aire d'un triangle de longueurs `a`, `b` et `c`
       avec la formule de Kahan.
    """
    (a,b,c)=sorted((a,b,c),reverse=True)
    return 0.25*sqrt((a + (b + c))*(c - (a - b))*(c + (a - b))*(a + (b - c)))
```

Pour les deux triangles vus précédemment, on obtient les résultats attendus :

```
>>> kahan(10000,10000,19999.999999999996)
1.9073486328124996
>>> kahan(1e10,1e10,1e-4)
500000.00000000006
```

C.4 Étude d'une propriété de la fonction tan

David Bellamy, Jeffrey Lagarias et Felix Lazebnik ont montré[76,77] en 1999 qu'il existe un nombre infini d'entiers positifs n vérifiant $|\tan(n)| > n$ et $\tan(n) > n/4$. Le problème reste cependant ouvert pour $\tan(n) > n$. On connaît un certain nombre de valeurs n vérifiant cette dernière relation ; la suite de tels entiers est répertoriée dans l'*on-line encyclopedia of integer sequences* sous l'identifiant A249836.

On se propose d'écrire un programme Python pour vérifier la relation pour les seize premières valeur de la suite A249836 :

```
from math import tan
L= [1, 260515, 37362253, 122925461, 534483448, 3083975227, 902209779836,
    74357078147863, 214112296674652, 642336890023956, 18190586279576483,
    248319196091979065, 1108341089274117551, 11855429981233835451 6058,
    14285991290206085825486 71, 42857973870618257476460 13 ]
for n in L:
    if tan(n) > n:
        print(n)
```

On obtient à l'exécution :

[75] William Kahan. « Miscalculating Area and Angles of a Needle-like Triangle ». Manuscript. Sept. 2014

[76] David P. Bellamy et al. « Large Values of Tangent : 10656 ». In : *The American Mathematical Monthly* 106.8 (1999), p. 782-784

[77] David P. Bellamy, Jeffrey C. Lagarias et Felix Lazebnik. *Proposed Problem ; Large Values of Tan n*. Mars 2022. URL : https://cpb-us-w2.wpmucdn.com/sites.udel.edu/dist/3/7714/files/2022/03/tan_n.pdf

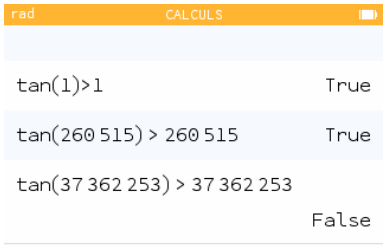


FIGURE 48 : Tests de la relation $\tan(n) > n$ pour les premiers termes de la suite A249836.

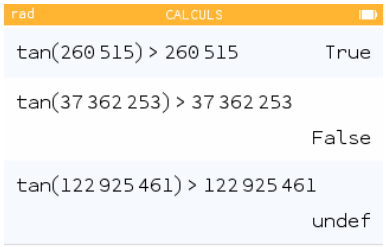


FIGURE 49 : Tests plus poussés pour la relation $\tan(n) > n$.

```
1
260515
37362253
122925461
534483448
3083975227
902209779836
74357078147863
214112296674652
642336890023956
```

La relation semble n'être vérifiée que pour les dix premières valeurs. Que s'est-il passé ? Les entiers manquants sont tout à fait représentables dans le type `int` de Python. Mais, pour calculer la tangente, Python doit transformer ces entiers en des nombres flottants en double précision, car c'est le type que « `tan()` » prend en entrée. Affichons la liste des entiers qui ne sont pas représentables dans ce format :

```
for n in L:
    if n != float(n):
       print(n)
```

On obtient :

```
18190586279576483
248319196091979065
1108341089274117551
118554299812338354516058
14285991290206085825486 71
42857973870618257476460 13
```

soit, l'ensemble des entiers non affichés par notre premier programme. Python offre un type `int` avec une précision variable mais dès qu'un calcul doit être effectué avec une fonction transcendante, les opérandes sont automatiquement transformés en nombres flottants en double précision, ce qui limite les calculs possibles.

Lorsque l'on décide de faire le même test sur la calculatrice Numworks, on obtient des résultats très différents (figure 48) : la relation n'est plus vérifiée dès le troisième terme. Plus étonnant encore, si l'on poursuit les tests (figure 49), on obtient `undef` en résultat.

Calculons la tangente de 37362253. Avec Python, on obtient :

```
>>> tan(37362253)
37754853.36177291
```

ce qui est très proche de la valeur réelle :

$$\tan(37362253) = 37754853.361772908592175824730311870706726\cdots$$

Avec la calculatrice Numworks, on obtient 36703174.26, ce qui justifie la réponse négative au test.

Le calcul de toutes les fonctions périodiques est fait sur un sous-domaine $\mathcal{S}$ très restreint de leur domaine de définition ; tout argument en dehors de $\mathcal{S}$ doit y être ramené par un processus appelé *réduction d'argument*. Ce calcul est difficile à faire de façon rigoureuse pour des arguments très éloignés de $\mathcal{S}$. Pour la fonction tan, un argument proche d'un multiple de $\pi/2$ — comme 37362253, qui est très proche de $23785549\pi/2$ — requiert un grand soin dans la réduction d'argument car une petite erreur se traduit par un énorme écart dans l'évaluation. La bibliothèque mathématique utilisée par Python

fait ce calcul rigoureusement ; il semble que ce ne soit pas le cas pour celle utilisée par la calculatrice Numworks. Cela explique aussi le « undef » observé dans la figure 49 : la valeur 122925461 est tellement proche d'un multiple de $\pi/2$ que la calculatrice assimile le résultat à un infini (remplacé par undef, ici).

Index

Licence de ce document