Arithmétique et Précision des Calculs sur Ordinateur

Jean-Michel Muller

Arithmétique et Précision des Calculs sur Ordinateur

Jean-Michel Muller, CNRS, Laboratoire de l'Informatique du Parallélisme (UMR 5668)

Introduction

Définir avec soin les unités du système international permet d'effectuer des mesures plus précises, et plus reproductibles. Cependant, c'est rarement le produit « brut » d'une mesure qui nous intéresse : à partir de valeurs mesurées, nous allons effectuer des calculs, parfois longs et complexes, afin de trouver le résultat recherché. Si ces calculs sont peu précis, ou s'ils sont difficilement reproductibles parce que leur résultat est très dépendant de l'environnement de calcul utilisé (processeur, langage, compilateur, etc.), une partie de l'avantage résultant de la normalisation de la mesure risque d'être perdue.

De fait, jusqu'au milieu des années 1980, les arithmétiques des ordinateurs était très diverses, et on pouvait observer des comportements particulièrement étranges. Un article de W. Kahan datant de 1981 [4] en donne une liste impressionnante : multiplications par 1 qui entraînent un dépassement de capacité, tests nous disant que $y = z$ mais que $y - t \neq z - t$… En ajoutant à ceci le fait que les formats de représentation des nombres étaient très variables, et en particulier que les seuils de dépassement différaient fortement d'un système à l'autre, on comprend aisément que les programmes numériques étaient difficilement portables d'un ordinateur à un autre, et il fallait souvent reprendre complètement les programmes lorsqu'on changeait de machine.

Il était donc important de normaliser l'arithmétique des ordinateurs. Ceci est fait par le biais d'une norme la norme IEEE-754 [2], dont la première version date de 1985, et la plus récente de 2019. Cette norme spécifie les formats de l'arithmétique virgule flottante (qui est de loin le système arithmétique le plus utilisé pour représenter des nombres réels, voir plus loin), ainsi que le comportement des principales opérations arithmétiques. Depuis 2008 elle émet aussi des recommandations (qui ne sont toutefois pas contraignantes) concernant le comportement de quelques fonctions mathématiques (exponentielles, logarithmes, fonctions trigonométriques, etc.)

Construire un système arithmétique, c'est souvent devoir élaborer un compromis entre divers souhaits, qui ne sont pas vraiment compatibles : on aimerait que nos calculs soient rapides, précis, qu'ils nécessitent peu de mémoire ou de surface de silicium et consomment peu d'énergie, et que leur programmation soit simple, de sorte que la probabilité d'introduire un « bug » soit la plus petite possible. Les compromis dépendront souvent des domaines abordés : un supercalculateur utilisé pour faire des calculs de dynamique des fluides, un calculateur embarqué dans un avion, un téléphone portable et un ordinateur individuel ne répondent pas du tout aux mêmes besoins. De ce point de vue, de gros changements sont intervenus dans les deux dernières décennies, demandant à revoir et à diversifier notre manière de faire du calcul : alors que dans les premières années de l'informatique, on utilisait des ordinateurs essentiellement pour faire de la simulation numérique et des calculs financiers, de nombreuses nouvelles applications sont apparues : systèmes embarqués, informatique nomade, jeux, intelligence artificielle, etc.

La représentation «virgule flottante» des nombres

La représentation « virgule flottante » binaire, de loin la plus utilisée pour représenter les nombres réels en informatique, correspond à l'adaptation à la base 2 de ce que les fabricants de calculettes appellent la « notation scientifique ». Dans une telle notation, lorsque l'on voit affiché 2.786E04, cela signifie 2.786×10^4, c'est-à-dire le nombre que l'on obtient en multipliant 2,786 par 10000, soit 27860. Dans cette représentation, 2.786 est appelé la *mantisse* du nombre, et 4 son *exposant.* Cette « notation scientifique » utilise la base 10 : l'exposant représente une multiplication par une puissance de 10, et la mantisse est *écrite en base dix* (c'est-à-dire dans notre système usuel de représentation des nombres).
Cette écriture en base 10 n'est pas la mieux adaptée au calcul sur ordinateur. C'est pourquoi la plupart des machines utilisent une représentation interne des nombres en base 2 (même si les résultats finaux d'un calcul sont ensuite convertis en base 10 pour affichage).

La norme IEEE 754, dans sa version de 2019, spécifie des formats binaires (les plus connus, binary32 et binary64, encore improprement appelés *simple précision* et *double précision* pour des raisons historiques, correspondent à des représentations sur respectivement 32 et 64 bits) ainsi que des formats décimaux. Mais la norme spécifie bien plus que les formats de représentation : son intérêt principal réside dans sa spécification des *arrondis* et des *exceptions*.

Arrondi correct : algorithmes et preuves

En général, la somme, le produit, le quotient, … de deux nombres virgule flottante n'est pas exactement représentable dans le même format virgule flottante : il faut donc l'arrondir. Pendant longtemps, la seule chose que l'on savait était que le résultat d'une opération n'était « pas trop loin » du résultat exact. Considérons le petit programme suivant :

```
s = a+b ; z = s-a ; r=b-z      (1)
```

Si les opérations arithmétiques étaient exactes, à la fin du calcul, *z* vaudrait *b* et *r* vaudrait zéro. Si maintenant la seule information dont on dispose est que les opérations sont « presque exactes » on va en déduire que *z* n'est pas très éloigné de *b*, et que *r* n'est que du « bruit numérique ». Et pourtant, comme nous allons le voir, lorsqu'on l'exécute sur une machine respectant la norme IEEE 754, ce programme nous fournit un résultat très intéressant.

Le principal apport de la norme IEEE 754 réside dans la notion d'*Arrondi correct*. L'utilisateur choisit une *fonction d'arrondi* parmi plusieurs définies par le standard : arrondi au plus près (c'est la fonction par défaut, bien entendu la valeur à retourner lorsque l'entrée est exactement entre deux nombres virgule flottante est clairement spécifiée par la norme), arrondi vers le haut, vers le bas, vers zéro. Lorsqu'on effectue une opération arithmétique, le résultat doit être la fonction d'arrondi appliquée au résultat exact : on dit que l'opération est *correctement arrondie*. Par exemple, si on note RN la fonction « arrondi au plus près », lorsqu'on calcule en machine $a + b$, où a et b sont des nombres virgule flottante, le résultat effectivement obtenu (sauf si l'utilisateur choisit de changer la fonction d'arrondi) est RN(a +

b). Cette simple spécification, ajoutée à la connaissance de la structure des nombres virgule flottante, permet de montrer des résultats intéressants et utiles. Par exemple, si *a* et *b* sont deux nombres virgule flottante tels que $|a| \geq |b|$, l'exécution du programme (1) va nous donner $a + b = s + r$. Le nombre *r* est donc exactement l'erreur de l'addition en machine de *a* et *b*. On sait également, par un algorithme tout aussi simple, calculer l'erreur d'une multiplication. Ceci est très important car on pourra ensuite ré-injecter ces termes d'erreur dans un calcul, pour « compenser », au moins partiellement, les erreurs commises. Toute une famille d'algorithmes *compensés* [6] utilise de tels résultats pour calculer avec précision des sommes des produits scalaires ou pour simuler une arithmétique plus précise que celle fournie par le processeur utilisé [3], etc.

La spécification d'arrondi correct permet également de montrer des propriétés portant sur les domaines dans lesquels peuvent se trouver des résultats d'un calcul. Considérons le petit exemple suivant, dû à Kahan [5]. On cherche à calculer

$$z = \frac{x}{\sqrt{x^2 + y^2}}$$

dans le but de calculer ensuite une fonction *g* de *z* qui n'est définie qu'entre -1 et 1 (par exemple un arcsinus). Un programmeur naïf de 1975 se serait probablement dit que lorsque *x* et *y* sont des nombres réels, $x/\sqrt{x^2 + y^2}$ est toujours compris entre -1 et 1, et il aurait donc simplement écrit la ligne de programme :

```
z = x/sqrt(x*x + y*y)      (2)
```

Mais sur une machine de 1975, si le résultat exact est très proche de 1, il est possible que, à cause des erreurs d'arrondi, le résultat calculé soit très légèrement supérieur à 1. À ce moment-là, le calcul de *g*(*z*) peut entraîner une erreur grave. Un programmeur expérimenté de 1975 devait donc effectuer des tests pour détecter de tels cas. Il est cependant possible de montrer que si l'addition, la multiplication et la racine carrée sont correctement arrondies (ce qui est le cas sur toutes les machines respectant la norme IEEE 754, c'est-à-dire tous les ordinateurs usuels) le résultat calculé en utilisant le programme (2) sera toujours compris entre -1 et 1. Le programmeur expérimenté d'aujourd'hui écrira donc le même programme que le programmeur naïf de 1975 !

Exceptions

Un autre domaine dans lequel la norme IEEE 754 a apporté beaucoup est celui de la gestion des cas exceptionnels. Que faut-il faire lorsque le résultat exact est trop grand pour être représentable, ou lorsque l'opération effectuée n'est pas définie mathématiquement (comme 0/0, $\sqrt{-1}$ en arithmétique réelle, etc.) ? Avant la norme, chaque système avait sa façon propre de gérer ce genre de situation, ce qui rendait très difficile le portage des programmes. La norme a imposé des représentations particulières pour les résultats de telles opérations et a complètement spécifié quel doit être le comportement des opérations arithmétiques lorsque de telles représentations apparaissent en entrée. La conséquence est une bien meilleure portabilité des programmes numériques (en changeant de machine, le programme se comportera de la même manière dans tous ces cas bizarres), et la possibilité de prouver à

l'avance quel va être le comportement d'un programme. Venons-en justement à cette idée de preuve…

Preuve formelle, automatisation de la conception de bibliothèques

Pour des applications critiques (par exemple la conduite automatique ou l'assistance à la conduite de véhicules), on a besoin d'être *certains* du bon comportement d'un programme numérique. Pour cela, on ne peut en général plus se contenter de l'essayer avec un grand nombre d'entrées. Il devient donc nécessaire de prouver son comportement, ce qui se fait en utilisant les spécifications de la norme IEEE 754. On tombe alors sur un autre écueil : une telle preuve est en général très longue et complexe, car il faut tenir compte d'une foule de cas particuliers. Il est alors difficile d'avoir une confiance totale en la preuve : est-on certain de ne pas avoir oublié un cas, de ne pas avoir fait une faute quelque part ? Pour résoudre ce problème, tout un travail a été effectué depuis quelques années pour rendre possible l'élaboration de preuves vérifiées par ordinateur, à l'aide d'environnements comme l'assistant de preuve Coq. Le lecteur intéressé par ce sujet pourra consulter avec intérêt l'ouvrage récent de S. Boldo et G. Melquiond [1].

La situation n'est pas figée

La norme IEEE 754 évolue. La version de 2019 intègre de nouvelles fonctionnalités permettant de faciliter la reproductibilité des calculs. De manière plus générale, de nouveaux systèmes de représentation des nombres sont proposés, qui essaient de prendre en compte l'évolution technologique. En particulier, si la vitesse des unités de calculs et celle des unités mémoire de nos ordinateurs ont considérablement augmenté depuis 30 ans, elles ne l'ont pas fait avec la même cadence : le rapport entre le temps de lecture/écriture en mémoire et le temps des opérations arithmétiques a été multiplié par environ 100 depuis 1990. Bien prendre ceci en compte demande à repenser à tous les niveaux notre façon de calculer.

Références

[1] Sylvie Boldo et Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. ISTE Press - Elsevier, Décembre 2017.

[2] IEEE Standard for Floating-Point Arithmetic, IEEE 754-2019
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8766229&isnumber=8766228

[3] M. Joldes, J.-M. Muller et V. Popescu, *Tight and rigourous error bounds for basic buiding blocks of double-word arithmetic*, ACM Transactions on Mathematical Software, Vol. 44 No 2, Octobre 2017.

[4] W. Kahan, Why we need a floating-point standard?, Dept. Of Computer Science, Univ. Berkeley, 1981.

[5] W. Kahan, *What can you learn about Floating-Point arithmetic in one hour*, disponible à l'adresse https://people.eecs.berkeley.edu/~wkahan/ieee754status/cs267fp.pdf

[6] J.-M. Muller (coordinateur), N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol et S. Torres, *Handbook of Floating-Point Arithmetic* (2ème edition), Birkhauser, 2018.