Arithmétique des ordinateurs et preuves formelles

Sylvie Boldo, Guillaume Melquiond

Arithmétique des ordinateurs et preuves formelles

Sylvie Boldo Guillaume Melquiond

École des Jeunes Chercheurs en Informatique Mathématique
22 mars 2012

Table des matières

1 Introduction

Initialement l'informatique, *computer science*, n'était que la science des ordinateurs et des calculateurs. Le but premier était de pouvoir calculer rapidement et sans risque d'erreur humaine. Les processeurs actuels ont parfaitement rempli l'objectif de rapidité : effectuer 3 milliards d'opérations par secondes est devenu banal. Une question raisonnable est alors celle du risque d'erreur et donc de la validité de ces calculs. Au bout d'une heure, on a fait plus de $10\,000\,000\,000\,000$ calculs, mais la réponse donnée est-elle juste ? À peu près juste ?

Cette question se pose particulièrement dans le cas des calculs sur les nombres réels. En effet, l'ordinateur calcule avec une précision finie et peut donc commettre à chaque opération une petite erreur d'arrondi. L'accumulation de ces erreurs d'arrondi peut rendre ce résultat faux ou faire échouer le programme à cause d'un comportement inattendu (division par zéro par exemple).

En pratique, les nombres sont représentés dans l'ordinateur en notation scientifique, avec un exposant pour l'ordre de grandeur et une mantisse avec un nombre fixé de chiffres. Par exemple avec 3 chiffres décimaux, on peut parler de π comme 3,14, de la vitesse de la lumière comme 3,00 $\times 10^8$ m/s ou de la dette de la France (fin du troisième trimestre 2011) comme 1,69 $\times 10^{12}$ €. Ces valeurs sont arrondies (ici au plus proche) mais restent très proches de la valeur exacte. Après plusieurs calculs, la distance à la valeur exacte peut augmenter, par exemple $\pi^2 \hookrightarrow 3{,}14^2 \hookrightarrow 9{,}86$ mais $\pi^2 = 9{,}869\,604\ldots$ donc le nombre flottant le plus proche est $9{,}87$. Avec seulement deux arrondis, on n'a plus exactement le flottant le plus proche du résultat exact. Alors avec 3 milliards de calculs…

Un autre point à aborder est celui de la confiance. Peut-on avoir confiance en un ordinateur ? un programme ? un algorithme ? une démonstration mathématique ? En effet, si quelqu'un prouve un algorithme, mais que cette preuve ou sa réalisation en programme contient une erreur, le programme se trompera ou échouera. Cela peut avoir des conséquences importantes en termes commerciaux ou de vies humaines. Comment donc peut-on augmenter la confiance en une preuve ?

Nous allons donc présenter la façon dont les ordinateurs calculent en section 2 avant de présenter comment prouver avec une grande sûreté des propriétés sur les flottants en section 3. Nous présenterons de nombreux exemples en section 4 avant de conclure.

2 Arithmétique à virgule flottante

2.1 Nombres entiers

La représentation machine des entiers est en binaire. Pour $a_i \in \{0, 1\}$, le nombre $a_n a_{n-1} \cdots a_1 a_0$ représente la valeur $\sum_{i=0}^{n} a_i 2^i$. Pour représenter les négatifs, la solution habituellement choisie est le complément à 2. Il est nécessaire de fixer la valeur de n : la représentation de -1 dépend du nombre de bits choisi. Pour un nombre binaire en complément à 2 sur n bits, on sépare le premier bit s et les $n-1$ bits suivants. Ces $n-1$ bits représentent (en binaire usuel) une valeur positive v. La valeur de ce nombre est alors v si $s = 0$ et la valeur est $-2^{n-1} + v$ si $s = 1$.

Les valeurs possibles stockées vont de -2^{n-1} à $2^{n-1} - 1$: il n'y a pas de valeur perdue, ni de représentations multiples de 0. On garde également un discriminant facile du signe : les valeurs strictement négatives ont le premier bit à 1 et les valeurs positives ou nulles leur premier bit à 0.

Quand les valeurs maximales sont dépassées, par exemple -2^{31} et $2^{31} - 1$ pour des nombres sur 32 bits, on a le phénomène de dépassement de capacité (*overflow*). Dans ce cas, selon le langage et l'environnement, on peut avoir de l'arithmétique modulo, une saturation ou un échec.

2.2 Nombres à virgule flottante

Dans le processeur, on ne représente qu'une partie des réels en « notation scientifique », ce qui signifie que l'on garde un exposant et un nombre fixé de chiffres. On appelle cela un nombre à virgule

flottante. Ces nombres, ainsi que les calculs sont régis par un standard, l'IEEE-754 qui définit la répartition et l'usage des bits d'un nombre [33]. Les formats usuels sont la simple précision (`float` en C) sur 32 bits, appelée *binary32* et la double précision (`double` en C) sur 64 bits, appelée *binary64*. Des formats étendus (avec plus de bits pour l'exposant et/ou la fraction) existent souvent mais ne sont pas toujours accessibles au programmeur.

La suite de bits est divisée en 3 parties : signe, exposant et fraction de tailles respectives 1, w_e et w_f :

s	e'	f
signe	exposant biaisé sur w_e bits	fraction sur w_f bits

On notera $B = 2^{w_e-1} - 1$ la valeur du biais. Le type et la valeur du flottant dépendent de la valeur de l'exposant :

– si $0 < e' < 2^{w_e} - 1$, le flottant est appelé normal et sa valeur est

$$(-1)^s \cdot 1.f \cdot 2^{e'-B}.$$

– si $e' = 0$, le flottant est dit dénormalisé [1] et sa valeur est

$$(-1)^s \cdot 0.f \cdot 2^{1-B}.$$

Dans le cas où $f = 0$, on obtient deux valeurs qui sont $+0$ et -0.

– si $e' = 2^{w_e} - 1$ et $f = 0$, on a deux valeurs qui sont $+\infty$ et $-\infty$.

– si $e' = 2^{w_e} - 1$ et $f \neq 0$, on a une valeur spéciale qui est le NaN (*Not-a-Number*).

On notera la précision $p = w_f + 1$. En pratique, la précision est de 24 en simple précision et 53 en double précision.

Voici l'exemple d'un nombre à virgule flottante simple précision normal:

$$11100011010010011110000111000000$$

$\boxed{1}$		$\boxed{11000110}$		$\boxed{10010011110000111000000}$
s		e'		f
signe		exposant		fraction
$\downarrow$		$\downarrow$		$\downarrow$
$(-1)^s$	$\times$	$2^{e'-B}$	$\times$	$1 \bullet f$
$(-1)^1$	$\times$	$2^{198-127}$	$\times$	$1.10010011110000111000000_2$

$$-2^{54} \times 206727 \approx -3{,}7 \times 10^{21}$$

La répartition des flottants ressemble donc à celle de la figure 2.2. L'espace entre les flottants double à chaque changement d'exposant.

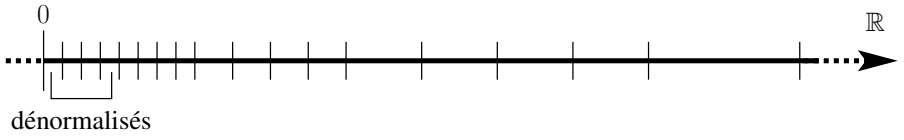

FIGURE 1 – Exemple de répartition des nombres flottants

Dans le cas où l'on obtient un dénormalisé, on parle d'*underflow*. Cela correspond au fait où l'une des valeurs à arrondir est inférieure au plus petit flottant normal 2^{1-B}.

Pour un flottant x, on appelle ulp (*unit in the last place*) et on notera $\mathrm{ulp}(x)$ la valeur de son dernier bit, c'est-à-dire $2^{e'-B-w_e}$ pour un flottant normal. Pour un dénormalisé, son ulp vaut 2^{1-B-w_e}, qui est

1. Depuis la révision de la norme en 2008, le terme anglais choisi est *subnormal*.

également le plus petit flottant strictement positif. On notera $e_{\min} = 1 - B - w_e$. Ainsi en double précision, on a $\mathrm{ulp}(1) = 2^{-52}$ et $e_{\min} = -1074$.

Le résultat d'un calcul n'est généralement pas un nombre flottant. Il est donc nécessaire de l'arrondir, c'est-à-dire de fournir un nombre flottant proche de ce résultat exact. La norme IEEE-754 [33] définit 5 modes d'arrondis :

- arrondi au plus proche pair (*rounding to nearest, ties to even*, arrondi par défaut), noté $\circ$. Il renvoie le flottant le plus proche du réel à arrondir. Lorsqu'un réel est au milieu de deux flottants, il choisit celui qui a la mantisse paire.
- arrondi vers $-\infty$, noté $\bigtriangledown$. Il renvoie le plus grand flottant inférieur au réel.
- arrondi vers $+\infty$, noté $\bigtriangleup$. Il renvoie le plus petit flottant supérieur au réel.
- arrondi vers zéro. Si le réel est positif, il renvoie l'arrondi vers $-\infty$, sinon il renvoie l'arrondi vers $+\infty$.
- arrondi au plus proche avec choix loin de zéro (*rounding to nearest, ties away from zero*). Il renvoie le flottant le plus proche. Lorsqu'un réel est au milieu de deux flottants, il choisit celui qui a la plus grande valeur absolue. Cet arrondi est utilisé notamment par les banques.

Étant donnés ces arrondis, chaque calcul élémentaire (addition, soustraction, multiplication, division, racine carrée) est parfait : on obtient le même résultat que si l'on avait calculé avec une précision infinie et arrondi ensuite au format choisi. C'est une propriété très puissante qui permet de raisonner sur chaque étape de calcul et garantit une bonne précision sur un calcul. Sur certains processeurs récents, une autre opération est disponible, le FMA (*Fused Multiply-and-Add*) qui permet de calculer $a \times x + y$ avec un seul arrondi final.

Revenons sur les valeurs spéciales. Il y a tout d'abord des infinis qui ont la sémantique mathématique attendue : $+\infty + 1$ donne $+\infty$ et $+\infty \times (-3)$ donne $-\infty$ par exemple. Il y a par ailleurs deux zéros signés qui sont considérés comme égaux par le test d'égalité, mais qui ont deux représentations binaires distinctes. Ils se comportent comme attendu vis-à-vis des infinis : $1/+0$ donne $+\infty$ et $1/-\infty$ donne -0 par exemple. Finalement, le NaN est la réponse donnée lorsqu'il n'y a pas de valeur numérique raisonnable à répondre : par exemple $0/0$ ou $\infty - \infty$ ou $\sqrt{-1}$.

2.3 Propriétés simples

Nous allons maintenant voir quelques propriétés simples de l'arithmétique flottante telles que la répartition des nombres flottants et les erreurs relatives de l'opération d'arrondi. Nous allons nous placer dans un monde idéal où les nombres flottants n'ont pas d'exposant maximal, c'est-à-dire que l'on peut trouver des flottants arbitrairement grands et que l'arrondi d'un réel ne donnera jamais un infini. Nous noterons $\mathbb{F} \subseteq \mathbb{R}$ un tel format défini de manière unique par une précision p et un exposant minimal $e_{\min}$. Ainsi, on peut voir $\mathbb{F}$ comme l'ensemble des $m \times 2^e$ avec m et e entiers tels que $|m| < 2^p$ et $e \geq e_{\min}$. Nous noterons $\square$ un arrondi quelconque parmi les arrondis usuels.

2.3.1 Monotonie

Lemme 2.1 (Idempotence)

$$\forall x \in \mathbb{F},\ \square(x) = x.$$

Dans la suite, nous supposerons que nous avons affaire à un mode d'arrondi qui respecte localement la propriété de monotonie, ce qui est le cas de tous les arrondis standards. Cette propriété stipule que, pour tout réel y situé entre un réel x et son arrondi $\square(x)$, y s'arrondit dans la même direction que x, c'est-à-dire que l'on a $\square(y) = \square(x)$. On déduit de cette propriété le lemme suivant :

Lemme 2.2 (Monotonie)

$$\forall x, y \in \mathbb{R},\ x \leq y \Rightarrow \square(x) \leq \square(y).$$

Preuve. Supposons tout d'abord qu'il existe un nombre $z \in \mathbb{F}$ tel que $x \leq z \leq y$. Par définition, $\triangle(x) \leq z$ et $z \leq \triangledown(y)$. Par conséquent, $\square(x) \leq \triangle(x) \leq \triangledown(y) \leq \square(y)$, ce qui conclut la preuve.

Supposons maintenant qu'il n'existe pas un tel z. Autrement dit, $\triangledown(x) = \triangledown(y)$ et $\triangle(x) = \triangle(y)$. Si $\square(y) = \triangledown(x)$, alors par monotonie locale, $\square(x) = \square(y)$. Sinon, $\square(y) = \triangle(x) \geq \square(x)$. Dans les deux cas, la propriété $\square(x) \leq \square(y)$ est donc satisfaite. ■

Corollaire 2.3 (Comparaison avec un nombre représentable)

$$\forall x \in \mathbb{F},\ \forall y \in \mathbb{R},\ x \leq y \Rightarrow x \leq \square(y).$$

Lemme 2.4 (Successeur) *Étant donné un nombre représentable $x = m_x \cdot 2^{e_x} \geq 0$, le nombre $y = (m_x + 1) \cdot 2^{e_x}$ est dans $\mathbb{F}$. Qui plus est, si $m_x \cdot 2^{e_x}$ est sa représentation canonique, alors il n'existe aucun nombre $z \in \mathbb{F}$ tel que $x < z < y$.*

Preuve. Puisque le nombre x est représenté par $m_x \cdot 2^{e_x}$, les propriétés $|m_x| < 2^p$ et $e_x \geq e_{\min}$ sont satisfaites. Deux cas se présentent, soit $|m_x + 1| < 2^p$, soit $m_x + 1 = 2^p$. Dans le premier cas, y est dans $\mathbb{F}$ de représentation $(m_x + 1) \cdot 2^{e_x}$. Dans le second cas, y vaut $2^p \cdot 2^{e_x}$ qui est dans $\mathbb{F}$ puisque représentable par $1 \cdot 2^{p+e_x}$.

Supposons maintenant que la représentation est canonique, c'est-à-dire que $e_x = e_{\min}$ ou que $2^{p-1} \leq |m_x| < 2^p$. Supposons par l'absurde qu'il existe $z = m_z \cdot 2^{e_z}$ représentable entre x et y. La position de z entre x et y force $e_x > e_z$ et $m_z > 2^{e_x - e_z} m_x \geq 2m_x$. Cela implique $m_z > 2^p$ ou $e_z < e_{\min}$, ce qui contredit le fait que $m_z \cdot 2^{e_z}$ est une représentation valide de z. ■

2.3.2 Bornes d'erreur

L'interrogation principale concernant un algorithme numérique est généralement l'erreur de calcul qui est commise. L'approche usuelle consiste à borner les erreurs commises à chaque opération puis à les combiner en une erreur totale. Voyons ici comment borner ces erreurs locales.

Lemme 2.5 (Erreur d'arrondi en arrondi au plus près) *Pour tout $x \in \mathbb{R}$, il existe ε et δ tel que*

$$\circ(x) = x \cdot (1 + \varepsilon) + \delta \quad \textit{et} \quad |\varepsilon| \leq 2^{-p} \quad \textit{et} \quad |\delta| \leq 2^{e_{\min}-1}.$$

Qui plus est, $\delta = 0$ ou $\varepsilon = 0$.

Preuve. Sans perte de généralité, nous pouvons supposer $0 < x$. Si $x \in \mathbb{F}$, alors $\delta = \varepsilon = 0$ satisfont la formule. Supposons donc que x n'est pas représentable, c'est-à-dire que $\triangledown(x)$ et $\triangle(x)$ sont différents. Soit $m \cdot 2^e$ la représentation canonique de $\triangledown(x)$. Le lemme 2.4 assure que $\triangle(x) = (m + 1) \cdot 2^e$. Par conséquent, $|\circ(x) - x| \leq (\triangle(x) - \triangledown(x))/2 = 2^{e-1}$. Séparons deux cas suivant que $\triangledown(x)$ est strictement plus petit ou plus grand ou égal au plus petit nombre normal $2^{e_{\min}+p-1}$.

Si $\triangledown(x)$ est dans le domaine des nombres dénormalisés, alors nous choisissons $\varepsilon = 0$ et $\delta = \circ(x) - x$. Comme sa représentation est canonique, $e = e_{\min}$, ce qui assure $|\delta| \leq 2^{e_{\min}-1}$.

Voyons maintenant le cas où $\triangledown(x)$ est un nombre normal, c'est-à-dire que $2^{p-1} \leq |m| < 2^p$. Cette fois, nous choisissons $\delta = 0$ et $\varepsilon = (\circ(x) - x)/x$. Nous avons donc $|\varepsilon| \leq 2^{e-1}/(2^{p-1} \cdot 2^e) = 2^{-p}$. ■

Il existe une variante du théorème valable pour tous les arrondis :

Lemme 2.6 (Erreur d'arrondi en arrondi dirigé) *Pour tout $x \in \mathbb{R}$, il existe ε et δ tel que*

$$\square(x) = x \cdot (1 + \varepsilon) + \delta \quad \textit{et} \quad |\varepsilon| < 2^{-p+1} \quad \textit{et} \quad |\delta| < 2^{e_{\min}}.$$

Qui plus est, $\delta = 0$ ou $\varepsilon = 0$.

2.3.3 Addition dénormalisée

Les bornes d'erreur précédentes ne sont pas tout à fait satisfaisantes, puisque les nombres calculés sont très souvent dans le domaine des nombres normaux et que δ vaut alors zéro après chaque opération. Dans ces conditions, la présence de δ dans les bornes d'erreur n'est qu'une source de bruit dont le seul but est de compliquer les calculs d'erreur. Il ne peut cependant pas être ignoré à moins de prouver qu'il n'y a pas de nombres dénormalisés produits par les calculs, par exemple qu'il n'y a pas de multiplication entre des nombres très petits.

L'opération d'addition occupe cependant une place particulière. En effet, il est possible de prouver les deux lemmes suivants et donc d'ignorer δ pour toutes les additions.

Lemme 2.7 (Exactitude de l'addition dénormalisée)

$$\forall x, y \in \mathbb{F},\ |x+y| \leq 2^{e_{\min}+p} \Rightarrow x+y \in \mathbb{F}.$$

Preuve. Si $|x+y| = 2^{e_{\min}+p}$, alors $x+y$ est représentable. Nous pouvons donc supposer l'inégalité stricte. Soient $m_x \cdot 2^{e_x}$ et $m_y \cdot 2^{e_y}$ les représentations de x et y. La somme $x+y$ vaut $m \cdot 2^{e_{\min}}$ avec $m = m_x \cdot 2^{e_x - e_{\min}} + m_y \cdot 2^{e_y - e_{\min}}$. Le nombre m est un entier et, au vu de l'inégalité précédente, $|m| < 2^p$. Par conséquent, $m \cdot 2^{e_{\min}}$ est une représentation valide de $x+y$ qui est donc dans $\mathbb{F}$. ■

Corollaire 2.8 (Erreur d'arrondi pour l'addition)

$$\forall x, y \in \mathbb{F},\ \exists \varepsilon,\ \circ(x+y) = (x+y)\cdot(1+\varepsilon) \quad \textit{et} \quad |\varepsilon| \leq 2^{-p}.$$

Preuve. Le lemme 2.5 traite le cas où $x+y$ est dans le domaine des nombres normaux. Il reste donc le cas où $|x+y| < 2^{e_{\min}+p-1}$. Par application du lemme précédent, nous savons que $x+y \in \mathbb{F}$ et donc que $\circ(x+y) = x+y$, ce qui conclut la preuve. ■

2.4 Opérations de base

Dans cette section, nous allons décrire comment les opérateurs flottants sont implantés en pratique dans les circuits, ce qui soulignera quelques unes des particularités de l'arithmétique à virgule flottante.

2.4.1 Alignement de mantisse et arrondi

Considérons un nombre strictement positif $x = m \cdot 2^e$ pas forcément représentable que l'on chercherait à arrondir et à mettre sous forme canonique en précision p. Nous allons ici ignorer les problèmes liés aux underflows. Nous noterons q la longueur effective de m : $2^{q-1} \leq m < 2^q$. Deux cas se présentent : soit $q \leq p$ soit $q > p$.

Dans le premier cas, x est représentable mais n'est pas sous forme canonique. Cette dernière est $(m \cdot 2^{p-q}) \cdot 2^{e+q-p}$. Il suffit donc de faire un décalage vers la gauche de la mantisse m et de décrémenter l'exposant en conséquence.

Dans le second cas, x n'est pas représentable sauf cas dégénéré. La représentation canonique de son arrondi vers zéro vaut $m' \cdot 2^{e'}$ avec $m' = \lfloor m \cdot 2^{p-q} \rfloor$ et $e' = 2^{e+q-p}$. Les autres arrondis doivent choisir entre les deux flottants représentables $m' \cdot 2^{e'}$ et $(m'+1) \cdot 2^{e'}$, comme l'indique le lemme 2.4. Le choix entre ces deux flottants consécutifs se fait en regardant la valeur de $\delta = m' - m \cdot 2^{p-q}$. Si δ vaut zéro, le résultat est nécessairement $m' \cdot 2^{e'}$. Si δ est non nul et l'arrondi vers l'infini, le résultat est $(m'+1) \cdot 2^{e'}$. Sinon, il faut comparer δ avec $1/2$ et choisir en conséquence. Dans le cas particulier de l'arrondi au plus près pair, si δ vaut exactement $1/2$, le résultat est celui dont la mantisse m' ou $m'+1$ est paire.

En pratique, il n'est pas question de calculer explicitement δ. Il suffit de regarder le bit de m en position $q-p-1$ pour savoir si δ est plus petit que $1/2$ ou non. Ce bit est usuellement noté r comme

rounding. Il suffit ensuite de regarder s'il existe un bit de m entre les positions 0 et $q-p-2$ qui vaut 1 pour savoir si δ est strictement plus grand que 0 ou $1/2$ (en fonction de r) ou non. Cette condition se représente elle aussi par un seul bit, noté cette fois s comme *sticky*.

Le choix entre les deux cas présentés ici, $q \leq p$ et $q > p$, se décide généralement de façon statique. Le décalage ne sera donc effectué que dans un sens donné. Notez qu'aussi bien le calcul de q que le décalage sont des opérations d'autant plus coûteuses qu'il y a de possibilité. Ainsi, un décaleur de type *barrel shifter* devant potentiellement décaler de 0 à 53 bits nécessite 6 étages de multiplexeurs. Il sera donc important de trouver les plages minimales nécessaires pour chaque opération afin de limiter leur coût.

2.4.2 Multiplication

Commençons par l'opération la plus simple : la multiplication. Étant donnés deux nombres flottants canoniques $x = m_x \cdot 2^{e_x}$ et $y = m_y \cdot 2^{e_y}$, le résultat de l'opération est $\square(m_x m_y \cdot 2^{e_x+e_y})$. Sans perte de généralité, nous pouvons considérer que les deux nombres sont positifs, la règle des signes donnant facilement les autres cas. Supposons tout d'abord que les deux entrées sont des nombres normaux et que $e_x + e_y + p \geq e_{\min}$.

L'entier $z = m_x m_y$ vérifie $2^{2p-2} \leq z < 2^{2p}$. L'opérateur de multiplication effectue cette multiplication entière sur $2p$ bits. Ici, q vérifie $2p > q \geq 2p-2 > p$. La décision entre les deux valeurs possibles de q se fait en regardant le bit de poids fort de z. Le décalage vers la droite ne peut lui aussi prendre que deux valeurs, p ou $p-1$, et s'implante donc à l'aide d'une simple rangée de multiplexeurs.

En présence d'entrées dénormalisées, la situation se complique. En effet, le produit d'un nombre dénormalisé par un nombre normal peut être dans le domaine des nombres normaux, mais l'entier z pourra avoir entre $p+1$ et $2p-2$ bits et il faudra donc un décaleur conséquent. Autre cas, le produit de deux nombres normaux peut être un nombre dénormalisé et il faudra là encore effectuer un décalage arbitraire de z vers la droite. Le matériel ignore généralement ces problèmes et se concentre sur le cas où entrées et sorties sont normales. Les cas dénormalisés sont alors pris en charge par des circuits dédiés, par du microcode dans les processeurs, ou plus prohibitif encore par le système d'exploitation.

2.4.3 Addition

Considérons maintenant le cas de l'addition flottante de deux nombres normaux canoniques $x = m_x \cdot 2^{e_x}$ et $y = m_y \cdot 2^{e_y}$. Sans perte de généralité, nous pouvons supposer que les nombres sont ordonnés $e_x \geq e_y$ et que x est positif. Le signe de m_y par contre n'est connu que dynamiquement. Le résultat vaut $\square((m_x \cdot 2^{e_x-e_y} + m_y) \cdot 2^{e_y})$. Il est cependant hors de question de calculer cette valeur ainsi en pratique, puisque $e_y - e_x$ peut être arbitrairement grand et le décaleur et l'additionneur aussi par conséquent.

Plutôt que de décaler m_x vers la gauche, la solution est de décaler m_y vers la droite, ce qui fait que le décalage est d'au plus p bits. Le résultat est alors $\square(m_x + \lfloor m_y \cdot 2^{e_y-e_x} \rceil) \cdot 2^{e_x})$. Le problème est que de l'information nécessaire pour arrondir est perdue lors du décalage vers la droite. Cela empêche non seulement d'arrondir correctement le résultat, mais en cas d'annulation catastrophique entre m_x et $\lfloor m_y \cdot 2^{e_y-e_x} \rceil$, des bits de la mantisse résultat pourraient être perdus.

Pour éviter cela, il est important de séparer deux cas. Il n'y a potentiellement annulation catastrophique que si m_y est négatif et $e_y \geq e_x - 1$. Dans tous les autres cas, la somme $m_x + \lfloor m_y \cdot 2^{e_y-e_x} \rceil$ aura entre $p-1$ et $p+1$ bits.

Considérons d'abord le cas de l'annulation catastrophique. m_y est décalé d'un bit au plus vers la droite, ou de façon plus simple, m_x est décalé d'un bit vers la gauche. Comme il y a annulation catastrophique, il faudra potentiellement un décaleur entre 1 et p bits vers la gauche pour renormaliser le résultat. Il n'y aura par contre pas besoin de circuit pour effectuer l'arrondi, le résultat étant exact.

L'autre cas nécessite un décaleur vers la droite pour m_y. Comme l'information nécessaire pour l'arrondi est perdue lors de ce décalage, l'idée est de calculer à l'avance les bits r et s en regardant

seulement m_y. La difficulté est que leur position n'est pas connue exactement tant que le décalage final de 1, 0 ou -1 bits n'a pas été décidé, ce qui nécessite le résultat de l'addition. Pour contourner ce problème, des bits g de garde sont calculés en plus de r et s afin de pouvoir corriger r et s en cas de mauvaise estimation de leur position [20].

2.4.4 Division et racine carrée

Contrairement à l'addition et la multiplication, il est difficile d'implanter division et racine carrée dans des circuits purement combinatoires. La plupart des circuits pour ces opérateurs sont en fait itératifs : ils sont appelés en boucle jusqu'à avoir calculé le résultat final. Les deux familles d'itération les plus utilisées sont le calcul chiffre à chiffre et les convergences quadratiques.

Le calcul chiffre à chiffre correspond à l'approche manuelle : à chaque itération, un nouveau chiffre du résultat est calculé. Une fois que suffisamment de chiffres ont été calculés, l'arrondi s'effectue en regardant la valeur du reste [20]. Il y a cependant deux différences. Tout d'abord, un chiffre n'est pas un bit, c'est un groupe de bits, ce qui permet de diminuer le nombre d'itérations nécessaires sans trop augmenter leur coût.

Ensuite, les algorithmes s'autorisent des chiffres négatifs. Cela permet de n'effectuer qu'un calcul approché mais rapide à chaque itération, ce qui peut provoquer des erreurs en excès du résultat qui seront rattrapées lors des itérations suivantes. Par exemple, en décimal, le calcul du quotient de 350 par 57 peut se faire en ne considérant que la division des premiers chiffres : $35/5 = 7$. Le reste vaut alors -49. Le chiffre suivant du résultat sera alors $-49/5 \simeq -9$ noté $\overline{9}$ et le reste $2{,}3$. À ce stade, le quotient vaut ainsi $7{,}\overline{9} = 6{,}1$ et $350 = 57 \times 6{,}1 + 2{,}3$.

L'autre approche consiste à utiliser une suite ayant une convergence quadratique de type Newton. Au lieu de produire à chaque itération un nombre fixé de nouveaux chiffres, le nombre de chiffres corrects du résultat va doubler à chaque itération. La section 4.2.4 donnera un exemple d'un tel algorithme.

La principale différence entre ces deux approches est que la première effectue un calcul exact alors que la seconde raffine un résultat approché. Ainsi, dans le cas d'une récurrence chiffre à chiffre effectuant le quotient de x par y, à l'étape i, le quotient partiel q_i de i chiffres et le reste calculé de la division r_i vérifient $x = yq_i + r_i$.

Notez que, contrairement à l'addition et à la multiplication, ces deux approches ne se prêtent pas à des calculs pipelinés : les mêmes unités de calcul sont utilisées à chaque itération.

En fait, une troisième approche est parfois employée. Il s'agit juste d'effectuer une approximation polynomiale de $1/y$, sur le modèle de ce qui est fait pour les fonctions élémentaires en section 4.4.

2.5 Compilation et sémantique

Passons maintenant au cas du logiciel. Comme mentionné en section 2.3.2, un développeur est généralement intéressé par la qualité numérique de son code et une façon de l'obtenir est d'effectuer une analyse d'erreur en s'appuyant sur les bornes d'erreurs de chaque opération flottante. Encore faut-il les connaître ! On pourrait croire que l'existence d'une norme sur l'arithmétique flottante fixe de façon claire le comportement d'un programme numérique, mais c'est sans compter sur tous les autres acteurs impliqués dans son exécution : langage, compilateur, système d'exploitation, processeur. Le même programme peut ainsi produire des résultats numériques bien différents en fonction des circonstances [21, 34].

2.5.1 Parenthésage

Commençons par le langage et intéressons nous aux deux langages les plus utilisés en matière de calculs numériques : Fortran et C. Le langage Fortran autorise de reparenthéser toutes les expressions arithmétiques qui ne sont pas explicitement parenthésées dans le programme, à condition de respecter

les règles d'associativité et de commutativité des réels. Ainsi, la somme flottante de quatre nombres `a+b+c+d` pourrait aussi bien être interprétée comme `((a+b)+c)+d` que comme `(a+b)+(c+d)`. La deuxième forme présente ainsi l'avantage, sur une architecture parallèle ou super-scalaire, d'effectuer deux additions en parallèle et donc de terminer plus rapidement que la première forme.

Malheureusement, les opérateurs flottants ne sont pas associatifs. Considérons la somme `delta + 1. + (-1.)` avec `delta` suffisamment petit, par exemple $\delta = 2^{-p}$. Si la somme est exécutée comme `(delta + 1.) +(-1.)`, le résultat sera alors zéro. En effet, δ est suffisamment petit pour que $\circ(\delta + 1) = 1$. Si par contre la somme est exécutée comme `delta + (1. - (1.))`, alors le résultat sera la valeur de `delta`. Cet exemple peut sembler artificiel, mais l'ordre des opérations est parfois fondamental pour certains algorithmes, comme décrit en section 4.1.2.

Le langage C n'autorise pas de telles réécritures des expressions : en l'absence de parenthésage explicite, les opérateurs sont considérés comme étant associatifs à gauche. Par contre, rien n'empêche un compilateur peu scrupuleux de s'affranchir de ces contraintes dans le but d'optimiser le programme au maximum. Il devient ainsi important de choisir avec précaution les options de compilation pour éviter ce genre de désagrément.

Dans ces conditions, il peut être intéressant de faire une analyse de l'erreur qui soit indépendante de l'ordre des opérations. Les bornes de la section 2.3.2 ne sont alors plus utilisables, mais il est possible d'en trouver de nouvelles, dans le cas de l'addition tout du moins [12]. L'idée est de chercher des bornes telles que, si l'on effectue une analyse d'erreur pour un certain ordre des opérations (par exemple en choisissant systématiquement l'associativité à gauche), l'erreur totale obtenue sera valable quelque soit l'ordre de n additions. Cela signifie que le corollaire 2.8 doit alors être oublié et la borne d'erreur prend la forme

$$|\circ(x + y) - (x + y)| \leq \varepsilon(|x| + |y|) + \delta$$

avec ε un peu plus grand que $n2^{-p}$ et $\delta = n2^{e_{\min}}$. L'analyse d'erreur devient alors plus grossière, mais elle reste suffisamment précise pour traiter des algorithmes qui ne se préoccupent pas de l'ordre des opérations.

2.5.2 Précision intermédiaire

Supposons maintenant que le compilateur respecte scrupuleusement la norme C, nous ne sommes pas sortis de l'auberge pour autant. Elle autorise en effet, dans certaines conditions, d'utiliser pour les calculs une précision intermédiaire plus grande que celle que l'on pourrait déduire du type des opérandes. Ainsi, ce n'est pas parce que les deux entrées sont des flottants *binary64* ($p = 53$) que leur somme ne sera pas faite en *binary80* ($p = 64$). Les processeurs de type x86 ont ainsi une unité flottante qui fournit des registres et opérateurs en précision étendue, et les compilateurs ne se gênent pas pour les utiliser s'ils en ont la possibilité.

On pourrait croire que l'usage d'une précision étendue ne peut qu'améliorer la qualité d'un calcul, mais ce n'est pas forcément le cas. Là encore, les algorithmes de la section 4.1.2 ne renverraient plus que des résultats absurdes. Mais même sur des algorithmes moins subtils, la situation n'est pas si simple. Le coupable est le phénomène de *double arrondi*. En arrondi au plus près, si jamais le résultat en précision étendue $p + p'$ est arrondi vers le milieu de deux flottants en précision p, alors au moment de stocker le résultat du calcul dans une variable en précision p, la valeur sera choisie de façon arbitraire (parité de la mantisse) entre les deux flottants à égale distance du résultat en précision étendue. Autrement dit, si le mauvais flottant est choisi, l'erreur totale sera plus grande que si le calcul avait été directement effectué en précision p.

Comme pour l'ordre des calculs, il est à nouveau possible de prendre en compte ces mauvaises surprises en jouant sur la valeur de ε dans la borne d'erreur [32, 12].

Jusqu'à présent, nous n'avons considéré que l'influence du langage et du compilateur, avec une petite contribution du processeur en ce qui concerne la présence de formats exotiques, mais le système

d'exploitation a lui aussi son grain de sable à caser. Par exemple, le système Microsoft Windows force les processeurs de type x86 à effectuer les opérations en précision $p = 53$ même quand les opérateurs 80 bits sont utilisés. Cela a le bon goût d'éviter les problèmes de double arrondi introduits par les compilateurs [2], mais cela signifie aussi qu'un programmeur utilisant explicitement des opérations *binary80* obtiendrait des résultats calculés avec une précision bien plus faible que prévue.

2.5.3 Le cas du FMA

Pour finir, voyons ce qu'il en est du FMA, car cet opérateur ne fait pas partie des opérateurs arithmétiques de base que l'on trouve dans les langages. Étant légèrement plus rapide qu'une multiplication suivie d'une addition, les compilateurs aimeraient bien introduire cet opérateur partout où l'utilisateur a écrit `a*b+c` ou `c+a*b`. La norme C autorisant l'usage d'une précision supérieure pour les calculs intermédiaires, il suffit de dire que la multiplication est effectuée en précision infinie (ou simplement $2p$) pour pouvoir employer un FMA à cet endroit. La norme C99 introduit ainsi la notion d'opération *contractante* pour préciser le cadre de cette optimisation.

Notez que la notion de parenthésage revient alors de façon insidieuse. En effet, l'expression `a*b + c*d` peut se compiler aussi bien vers `fma(a,b,c*d)` que `fma(c,d,a*b)` et le résultat dépendra donc du compilateur. Là encore, quelques surestimations dans le choix des bornes d'erreur permettent de faire une analyse d'erreur qui soit correcte même si le compilateur a fusionné des additions avec des multiplications [12].

Comme avec le double arrondi, le FMA peut lui aussi introduire des résultats inattendus. Considérez l'exemple suivant [37].

```
double f(double a, double b) {
  return a >= b ? sqrt(a * a - b * b) : 0;
}
```

Cette fonction ne calcule `sqrt(a*a-b*b)` que si $a \geq b$. Elle renvoie zéro sinon. A priori, par monotonie de l'opérateur d'arrondi, si $a \geq b$, alors `a*a-b*b` devrait être positif ou nul et la racine carrée est alors bien définie. C'est sans compter sur la présence potentielle d'un FMA. Si c'est le cas, pour $a = b$, la valeur de `a*a-b*b` peut très bien être négative. En effet, si l'expression a été compilée en `fma(a,a,-(b*b))`, il suffit que $a^2 < \circ(a^2) = \circ(b^2)$ pour que le résultat renvoyé par la fonction `f` soit un NaN, ce qui arrivera donc souvent quand $a = b$.

3 Preuves

Très vite, les théorèmes traitant d'arithmétique à flottante deviennent trop longs et compliqués pour que leurs preuves inspirent vraiment confiance. En effet, les vérifier représente alors une trop grande quantité de travail pour un humain. Mais plutôt que de se débarrasser de ces preuves qui ont perdu une part de leur utilité, l'idée est de les confier à un système formel implanté sur un ordinateur. C'est alors la machine qui va se charger automatiquement de vérifier la correction des preuves et leur adéquation avec les théorèmes.

3.1 Coq

Plusieurs systèmes formels sont adaptés à la vérification de preuves de théorèmes mathématiques. Nous allons ici nous intéresser au système Coq [3].

2. Il reste cependant des histoires louches pour les exposants très grands et très petits, donc la situation n'est pas tout à fait parfaite.

3. `http://coq.inria.fr/`

Ce système formel propose une logique intuitionniste d'ordre supérieure. Il s'appuie sur l'isomorphisme de Curry-Howard : un raisonnement prouve un théorème donné si et seulement si le programme (dans le λ-calcul) associé à ce raisonnement a le type associé au théorème. Par exemple, un théorème de la forme $A \Rightarrow B \Rightarrow C$ est correct dans ce formalisme si et seulement s'il existe une fonction qui, étant données des preuves de A et B, est capable de construire une preuve de C.

C'est à la charge de l'utilisateur d'exhiber une telle fonction, le système se contentant de vérifier que la fonction a le bon type. Cela en fait un *prouveur interactif*, aussi appelé *assistant de preuves*. Le système fournit cependant un ensemble de *tactiques* qui permettent de construire cette fonction implicitement en appliquant des étapes s'apparentant à du raisonnement mathématique [1]. La particularité par rapport à une preuve papier usuelle est qu'une démonstration Coq est une *preuve en arrière* : plutôt que de déduire des conséquences des hypothèses, l'utilisateur va chercher des propriétés qui impliquent la conclusion du théorème. L'approche consiste donc, étant donné un but à démontrer, à appliquer un théorème puis à prouver que ses hypothèses peuvent être satisfaites.

L'avantage de cette approche est qu'il suffit de faire confiance à une petite portion d'un assistant de preuve, dans le cas de Coq son algorithme de typage, pour être sûr qu'un théorème est correct. L'inconvénient est qu'une étape de raisonnement qui peut nous sembler évidente ne l'est pas toujours pour l'ordinateur et une preuve formelle peut donc nécessiter un bien plus gros travail qu'une preuve papier.

3.2 Formalismes pour l'arithmétique flottante

La première formalisation Coq de l'arithmétique flottante réellement utilisable pour prouver des algorithmes a été Pff [4] [16, 2]. Les nombres flottants y sont décrits par des paires d'entiers mantisse-exposant. Les formats sont décrits par leur exposant minimal et la précision des mantisses. Il s'agit donc d'une formalisation très proche du modèle décrit dans les sections précédentes.

Voici un exemple d'énoncé de théorème : entre plusieurs représentations valides de nombres flottants, celle canonique est d'exposant minimal.

```
Theorem FcanonicLeastExp :
  forall (b : Fbound) (x y : float),
  x = y :>R -> Fbounded b x -> Fcanonic b y -> (Fexp y <= Fexp x)%Z.
```

Ce théorème est quantifié universellement par un format `b` et deux paires d'entiers `x` et `y` (représentant implicitement les nombres réels $m_x \cdot 2^{e_x}$ et $m_y \cdot 2^{e_y}$). La première hypothèse énonce que les réels représentés sont les mêmes. La seconde (`Fbounded`) dit que `x` est une représentation valide, c'est-à-dire que $|m_x| < 2^p$ et $e_x \geq e_{\min}$. La troisième (`Fcanonic`) dit que `y` est non seulement valide mais canonique. La conclusion du théorème affirme alors que $e_y \leq e_x$.

Après Pff, d'autres formalisations pour Coq de l'arithmétique flottante ont été développées. Elles avaient d'autres objectifs que la preuve d'algorithmes flottants ; d'autres choix de conception que ceux de Pff ont donc été faits. Ainsi, Pff représente l'opération d'arrondi comme une relation entre un réel et une paire d'entiers, ne propose pas de calculs effectifs sur les nombres flottants, ne permet pas de traiter d'autres formats de nombres que celui décrit ci-dessus. Les formalismes suivants ont donc contourné ces limitations, chacun de leur côté.

La bibliothèque Flocq [5] a été conçue pour réunir au sein d'un même cadre toutes ces évolutions [9]. Cette fois-ci, les nombres flottants sont un sous-ensemble des nombres réels et les fonctions d'arrondi sont des fonctions des réels dans les réels. L'ensemble des nombres flottants pour un format donné est alors défini comme étant l'image d'une telle fonction. Plus précisément, une fonction d'arrondi $\circ$ est la

4. `http://lipforge.ens-lyon.fr/www/pff/`
5. `http://flocq.gforge.inria.fr/`

donnée de deux fonctions $\mathrm{rnd} : \mathbb{R} \to \mathbb{Z}$ et $\phi : \mathbb{Z} \to \mathbb{Z}$.

$$\circ(x) = \mathrm{rnd}\left(x \times 2^{-\phi(e)}\right) \times 2^{\phi(e)} \quad \text{avec } e = \lfloor \log_2(x) + 1 \rfloor.$$

Afin de vérifier les propriétés usuelles d'un arrondi, ces deux fonctions doivent être contraintes. Par exemple, rnd est choisie de telle sorte qu'elle soit l'identité sur les entiers et monotone sur $\mathbb{R}$. Le choix de la fonction ϕ permet de représenter divers formats de nombre :

- virgule fixe : $\phi(e) = e_{\min}$,
- virgule flottante sans exposant minimal : $\phi(e) = e - p$,
- virgule flottante : $\phi(e) = \max(e - p, e_{\min})$,
- virgule flottante avec dénormalisation abrupte : $\phi(e) = \begin{cases} e_{\min} + p - 1 & \text{si } e - p < e_{\min}, \\ e - p & \text{sinon.} \end{cases}$

Même si ce cours ne s'intéresse qu'à la base 2, aussi bien Pff que Flocq sont capables de gérer les autres bases de numérations, les définitions précédentes utilisent en fait un paramètre $\beta \geq 2$ à la place de 2. Les propriétés de l'arithmétique flottante décimale s'obtiennent ainsi en instanciant β par 10.

3.3 Automatisation

Les systèmes formels obligent l'utilisateur à décrire ses preuves dans ses moindres détails, ce qui est rarement intéressant. Les sections suivantes vont donc s'intéresser à quelques méthodes permettant d'automatiser la preuve formelle en rapport avec l'arithmétique des ordinateurs. L'objectif est ici de borner des expressions, par exemple pour prouver que des calculs ne vont pas provoquer de dépassement de capacité, et de borner des erreurs pour s'assurer qu'une valeur calculée ne sera pas trop éloignée de la valeur attendue.

3.3.1 Arithmétique d'intervalle

L'arithmétique d'intervalle et ses variations fournissent un moyen simple de vérifier des propriétés sur des expressions de façon mécanique. Dans cette section, le but n'est plus d'utiliser des preuves formelles pour garantir qu'un théorème à propos d'arithmétique flottante est vrai, mais au contraire d'utiliser des techniques courantes en arithmétique des ordinateurs pour faciliter la preuve de théorèmes mathématiques.

Considérons l'exemple suivant. Étant donnés un domaine D, une fonction f et une propriété P, l'objectif est de vérifier

$$\forall x \in D,\ P(f(x)).$$

Une formule équivalente s'obtient en considérant l'image $f(D)$ du domaine D par f :

$$\forall y \in f(D),\ P(y).$$

En fait, une façon suffisante de prouver cette formule serait d'exhiber un ensemble I tel que

$$f(D) \subseteq I \wedge \forall y \in I,\ P(y).$$

Voyons comment trouver un tel I de telle sorte que la partie gauche de la formule soit vraie par construction. Il ne restera alors plus qu'à prouver la partie droite. Dans cet optique, il est préférable que I soit aussi petit que possible sous peine d'obtenir une formule improuvable.

Soient x et y deux expressions réelles comprises dans des intervalles à bornes flottantes $[\underline{x}, \overline{x}]$ et $[\underline{y}, \overline{y}]$. Des intervalles encadrant les expressions composées suivantes peuvent être calculés en s'appuyant sur les propriétés de monotonie des différents opérateurs réels [35].

$$\begin{aligned} x + y &\in [\bigtriangledown(\underline{x} + \underline{y}), \bigtriangleup(\overline{x} + \overline{y})] \\ x - y &\in [\bigtriangledown(\underline{x} - \overline{y}), \bigtriangleup(\overline{x} - \underline{y})] \\ x \times y &\in [\min_{\gamma \in \Gamma}(\bigtriangledown(\gamma)), \max_{\gamma \in \Gamma}(\bigtriangleup(\gamma))] \quad \text{avec } \Gamma = \{\underline{x}\underline{y}, \overline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\overline{y}\} \end{aligned}$$

Nous pouvons ainsi définir des opérations sur des intervalles qui sont compatibles avec les opérations sur les nombres réels. Qui plus est, l'arithmétique d'intervalles à bornes flottantes s'accommode bien des dépassements de capacité, la borne de gauche devenant alors $-\infty$ tandis que la borne de droite devient $+\infty$.

Cette approche permet d'obtenir par le calcul un encadrement prouvé d'une expression $f(x)$ connaissant un intervalle contenant x. Partant de là, c'est donc un moyen de prouver automatiquement et formellement des bornes sur une expression, à supposer que le système formel sache faire de l'arithmétique flottante [30].

L'arithmétique d'intervalle est cependant caractérisée par une perte de *corrélation* dans les calculs qu'elle effectue. Supposons par exemple que nous cherchions à borner $(x+1)-x$ sachant $x \in [0, 10]$. En effectuant le calcul par intervalle, nous trouvons que $x+1 \in [1, 11]$ puis que $(x+1)-x \in [-9, 11]$, ce qui est un encadrement assez mauvais de 1. Il existe de nombreuses façons d'améliorer l'arithmétique d'intervalles pour limiter ce genre de problème.

Une façon simple consiste à découper l'intervalle d'entrée en sous-intervalles, à effectuer le calcul par intervalles sur chacun d'eux, puis à faire l'union de tous les intervalles résultants. Cette méthode est effective à condition de découper des sous-intervalles suffisamment petits, ce qui peut être coûteux en temps.

Considérons maintenant une autre approche. Elle permet de prouver des bornes sur des expressions dérivables. Le théorème de Taylor nous apprend pour une fonction f dérivable sur un intervalle D contenant x et x_0 que

$$f(x) = f(x_0) + (x - x_0)f'(\xi) \quad \text{avec } \xi \in D.$$

En passant aux intervalles, la formule devient alors

$$f(x) \in f([x_0, x_0]) + (D - [x_0, x_0])f'(D).$$

Autrement dit, au lieu d'évaluer l'expression $f(D)$ par intervalle, c'est maintenant l'expression $f'(D)$ qui est évaluée. Cette méthode peut être combinée avec la précédente, par exemple pour éviter les points où la fonction n'est pas dérivable, ou tout simplement pour limiter les pertes de corrélation qui apparaissent dans le calcul de $f(D)$ [31].

3.3.2 Analyse de l'erreur directe

À cause de la corrélation qui existe entre $\circ(x)$ et x, une utilisation directe de l'arithmétique d'intervalle pour borner une erreur du type $\circ(x) - x$ conduirait à des valeurs correctes mais sans intérêt. La solution est d'utiliser le lemme 2.5 afin d'obtenir directement un encadrement de cette erreur. Par exemple, si $|x| \leq m$ avec m dans le domaine des nombres normaux, alors $|\circ(x) - x| \leq 2^{-p}m$.

Qu'en est-il dans le cas général de l'erreur $x - y$ entre deux termes x et y proches ? Si x est de la forme $x = \circ(x')$, une première étape consiste à découper l'erreur en deux morceaux par l'inégalité triangulaire :

$$|x - y| \leq |\circ(x') - x'| + |x' - y|.$$

Nous nous ramenons ainsi à borner une expression $a - y$ un peu plus simple qui reste néanmoins l'erreur entre deux termes proches.

Supposons maintenant que x et y ont la même structure, c'est-à-dire qu'il existe un opérateur $\diamond$ tel que $x = x_1 \diamond x_2$ et $y = y_1 \diamond y_2$. Qui plus est, supposons que x_1 et x_2 sont proches et qu'il en va de même pour y_1 et y_2. Dans le cas de l'addition ou de la soustraction, nous pouvons alors appliquer l'inégalité suivante pour se ramener à des erreurs plus simples :

$$|x - y| \leq |x_1 - x_2| + |y_1 - y_2|.$$

Pour la multiplication, l'idée est similaire mais en passant par l'erreur relative plutôt que l'erreur absolue :

$$|x/y - 1| \leq \varepsilon_x + \varepsilon_y + \varepsilon_x\varepsilon_y \quad \text{avec } |x_1/x_2 - 1| \leq \varepsilon_x \text{ et } |y_1/y_2 - 1| \leq \varepsilon_y.$$

À condition que les termes x et y aient la même structure et en répétant ce processus itérativement, nous pouvons ainsi nous ramener à des erreurs de la forme $\circ(u) - u$ que nous savons borner. De façon plus générale, nous pourrions nous ramener à des erreurs de la forme $u - v$ dont les bornes pourraient être connues par ailleurs.

Ces idées se retrouvent dans l'outil Gappa [6] [15]. Il sert à prouver des formules de la forme

$$e_1 \in I_1 \wedge \ldots e_n \in I_n \Rightarrow e \in I$$

avec $e_1, \ldots, e_n, e$ des expressions arithmétiques et $I_1, \ldots, I_n, I$ des intervalles à bornes numériques. Il a été conçu pour aider à vérifier la correction de fonctions en virgule flottante approchant des fonctions élémentaires [17]. Les expressions peuvent donc contenir des opérateurs d'arrondi. L'outil utilise de l'arithmétique d'intervalle naïve pour borner les expressions mais il cherche aussi à les décomposer de façon à se ramener à des termes corrélés qu'il sait traiter par des méthodes d'analyse d'erreur. Une fois qu'il a réussi à prouver une formule, l'outil génère une preuve formelle de cette formule que Coq peut alors vérifier automatiquement [7].

3.4 Exemple : le cas du cosinus

Afin d'illustrer les méthodes précédentes, nous allons considérer l'exemple suivant. L'objectif est de calculer en virgule flottante simple précision une approximation de la fonction cosinus sur un petit intervalle autour de zéro : $[-2^{-4}, 2^{-4}]$. Au vu du domaine d'entrée et de la précision du résultat, il s'agit d'un exemple jouet. La section 4.4 détaillera comment une telle fonction est implantée en pratique. Voici le code C de notre fonction jouet :

```
float my_cos(float x) {
  return 0xf.fffffp-4f - x * x * 0x7.ff5c7p-4f;
}
```

3.4.1 Analyse de l'algorithme

Ce code effectue l'évaluation d'un polynôme de degré 2 en virgule flottante simple précision ($p = 24$). Les nombres `0xf.fffffp-4f` et `0x7.ff5c7p-4f` utilisent la représentation hexadécimale du C99 : la mantisse est représentée en base 16, l'exposant en base 10, et la puissance est 2. Chacun de ces deux nombres est représentable en simple précision puisque leur mantisse fait 6 chiffres, soit 24 bits. Le premier vaut environ $16 \cdot 2^{-4} = 1$ tandis que le second vaut environ $8 \cdot 2^{-4} = 1/2$. Autrement dit, le polynôme vaut environ $1 - x^2/2$, ce qui n'est guère étonnant puisque l'objectif est d'approcher la fonction cosinus autour de zéro.

Notre but est de calculer la valeur mathématique du cosinus, mais nous implantons une évaluation de polynôme en virgule flottante. Le cosinus aura sa notation usuelle $\cos(x)$, la fonction $f(x)$ désignera la valeur effectivement calculée par `my_cos`, tandis que $P(x)$ représentera l'évaluation du polynôme en précision infinie, autrement dit sans aucune erreur d'arrondi. La distance entre $f(x)$ et $P(x)$ s'appelle *l'erreur d'arrondi*. Celle entre entre $\cos(x)$ et $P(x)$ s'appelle *l'erreur de méthode*. L'erreur totale entre la valeur souhaitée $\cos(x)$ et la valeur obtenue $f(x)$ est donc bornée par la somme de ces deux erreurs.

Nous allons ici nous intéresser à l'erreur de méthode. Remarquez tout d'abord que ce n'est pas le polynôme $1 - x^2/2$ qui a été choisi pour P mais un polynôme proche. En effet, le polynôme $1 - x^2/2$, bien que très proche de $\cos(x)$ autour de zéro, devient une mauvaise approximation quand on s'éloigne,

6. `http://gappa.gforge.inria.fr/`

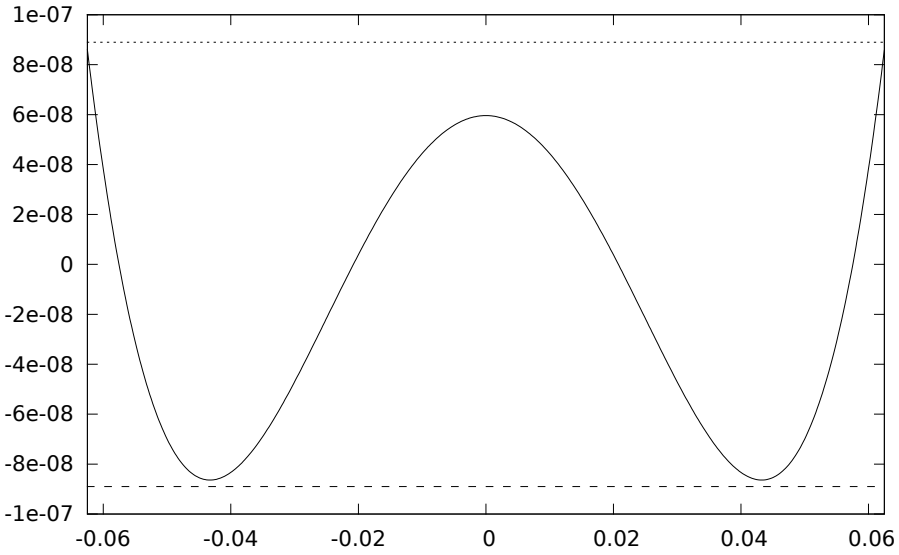

FIGURE 2 – L'erreur de méthode entre cos et le polynôme est bornée par $8{,}9 \cdot 10^{-8}$ pour $|x| \leq 2^{-4}$.

l'erreur allant jusqu'à atteindre $6{,}4 \cdot 10^{-7}$ en $x = 1/16$. Le polynôme choisi est l'un des meilleurs polynômes de degré 2 approchant le cosinus sur l'intervalle $[-1/16, 1/16]$ dont les coefficients soient représentables par des nombres simple précision. Cette fois, l'erreur maximale commise est $8{,}6 \cdot 10^{-8}$, soit 8 fois meilleure que ce que nous aurions obtenu avec le polynôme de Taylor tronqué. La fonction d'erreur $\cos(x) - P(x)$ est représentée sur la figure 2. Remarquez que l'erreur atteint ses extrema locaux en cinq point alors que le polynôme est de degré 2 et que ses extrema sont (presque) tous égaux en valeur absolue. Le théorème de Chebyshev s'applique donc, ce qui justifie que le polynôme P est (presque) optimal.

3.4.2 Spécification et preuve

En règle générale et dans cet exemple en particulier, l'erreur d'arrondi ne compense pas l'erreur de méthode, elle vient s'y ajouter. L'erreur totale sera donc supérieure à $8{,}6 \cdot 10^{-8}$. En prenant en compte seulement la dernière opération (la soustraction) qui renvoie une valeur inférieure à 1, nous pouvons estimer que l'erreur d'arrondi sera au moins égale à $1/2 \times 2^{-24} \simeq 3 \cdot 10^{-8}$. L'erreur totale sera donc d'au moins $1{,}2 \cdot 10^{-7} \simeq 2^{-23}$. Soyons optimiste et supposons que l'erreur totale ne sera en fait pas plus grande. Quelques entrées choisies au hasard confirment que c'est effectivement le cas.

Nous avons donc une fonction censée calculer une approximation de cosinus. Sa spécification est que, pour toute entrée x telle que $|x| \leq 1/16$, le résultat $f(x)$ doit vérifier $|f(x) - \cos(x)| \leq 2^{-24}$. La borne d'erreur étant optimiste, il est fort possible que la fonction ne satisfasse pas sa spécification. Il reste donc à prouver formellement que la fonction et la spécification sont bien en adéquation.

Nous allons utiliser l'outil Frama-C [7] à cet effet, en conjonction avec le *plugin* Jessie [8] et la plateforme Why3 [9]. Étant donnés un programme écrit en C et sa spécification, ces logiciels produisent un ensemble d'obligations de preuve par calcul de plus faible précondition [19]. Une fois que toutes ces obligations sont prouvées, soit par l'utilisateur soit par des outils automatiques, le programme est vérifié : il satisfait sa spécification. Cette spécification est fournie sous forme de commentaires dans le code. Pour l'exemple, cela donne le programme suivant :

7. http://frama-c.com/
8. http://krakatoa.lri.fr/
9. http://why3.lri.fr/

```
/*@ requires \abs(x) <= 0x1p-4;
  @ ensures \abs(\result - \cos(x)) <= 0x1p-23;
  @*/
float my_cos(float x) {
  return 0xf.fffffp-4f - x * x * 0x7.ff5c7p-4f;
}
```

Le commentaire `requires` indique l'hypothèse qui est faite sur les entrées de la fonction ; il s'agit d'une *précondition*. Les obligations de preuves générées pour les fonctions appelant `my_cos` demanderont de prouver que cette hypothèse est toujours satisfaite. Le commentaire `ensures` indique la propriété vérifiée par le résultat de la fonction ; il s'agit d'une *postcondition*. Les obligations de preuves générées pour les fonctions appelant `my_cos` pourront la supposer. Quant à l'obligation de preuve générée pour la fonction `my_cos`, elle demande de prouver que la postcondition est vraie sachant que la précondition l'est et connaissant le code de la fonction. C'est sur cette obligation de preuve que nous allons nous concentrer.

À partir de là, on pourrait espérer que tout soit prouvé automatiquement, à supposer que la spécification est effectivement correcte. Cependant, les approches automatiques de la section 3.3 n'arrivent pas toujours à correctement séparer l'erreur de méthode de l'erreur d'arrondi. Pour les aider un peu, nous allons ajouter une assertion dans le code qui prétend que l'erreur de méthode $P(x) - \cos(x)$ est bornée :

```
float my_cos(float x) {
  //@assert \abs(0xf.fffffp-4f - x * x * 0x7.ff5c7p-4f -\cos(x)) <= 8.9e-8;
  return 0xf.fffffp-4f - x * x * 0x7.ff5c7p-4f;
}
```

L'obligation de preuve va maintenant se décomposer en deux morceaux : le premier demande de prouver que l'assertion est vraie connaissant la précondition, le second demande de prouver que la postcondition est vraie connaissant l'assertion. Le premier peut alors être transformé en un but Coq que l'utilisateur doit prouver :

```
Theorem WP_parameter_my_cos_ensures_default :
  forall (x_0:floating_point.Single.single),
  ((Rabs (floating_point.Single.value x_0)) <= (1 / 16)%R)%R ->
  ((Rabs (((16777215 / 16777216)%R - (((floating_point.Single.value x_0) *
     (floating_point.Single.value x_0))%R * (8385991 / 16777216)%R)%R)%R -
     (Rtrigo_def.cos (floating_point.Single.value x_0)))%R) <= (89 /
     1000000000)%R)%R.
```

Remarquez que la borne de l'erreur de méthode a été changée de $8{,}6 \cdot 10^{-8}$ en $8{,}9 \cdot 10^{-8}$ afin de faciliter sa preuve en Coq. Cela pourrait bien sûr rendre improuvable la borne sur l'erreur totale (le deuxième morceau de l'obligation de preuve) ; mais dans le cas présent, il s'avère que Gappa y arrive automatiquement. La combinaison des preuves Coq et Gappa conclut la vérification de la fonction `my_cos` : étant donnée une entrée x bornée par $1/16$, elle calcule effectivement une valeur à une distance au plus 2^{-23} de $\cos(x)$.

Nous n'avons pas abordé ici la question des dépassements de capacité. En fait, les outils génèrent une deuxième obligation de preuve qui demande de prouver qu'aucune des quatre opérations flottantes de la fonction ne déborde. Gappa n'a aucune difficulté à la prouver.

4 Applications

4.1 Calculs exacts

En tant qu'approximation des nombres réels, les calculs à virgule flottante semblent imparfaits et imprécis. Il est néanmoins possible de calculer juste avec des flottants. Plus précisément, pour certaines

opérations, on peut calculer exactement l'erreur commise et cela en utilisant uniquement des calculs flottants. Nous allons ici parler de calcul exact et de calcul d'erreur.

4.1.1 Sterbenz

Le calcul exact le plus connu est la soustraction exacte, connue également sous le nom de théorème de Sterbenz [41].

Théorème 4.1 (Sterbenz) *Soient a et b dans $\mathbb{F}$ tels que*

$$\frac{b}{2} \leq a \leq 2b.$$

Alors le réel $a-b$ est représentable. En particulier, il est calculé sans erreur par la soustraction flottante.

Preuve. La preuve est assez simple. Considérons a et b tels que $b \leq a \leq 2b$, l'autre cas se traitant par symétrie. Nous en déduisons facilement que a et b sont positifs ou nuls. Alors, si nous considérons les représentations canoniques, nous avons $e_b \leq e_a$. Nous pouvons prendre pour $a - b$ le flottant qui lui est égal $(m_a - m_b \times 2^{e_b - e_a}) \cdot 2^{e_a}$. Son exposant est convenable et sa mantisse est telle que

$$\left|m_a - m_b \times 2^{e_b - e_a}\right| = |a - b|\, 2^{-e_a} = (a - b)\, 2^{-e_a} \leq a\, 2^{-e_a} = m_a < 2^p.$$

L'exposant et la mantisse de ce flottant sont acceptables, donc $a - b$ est représentable. ■

Une soustraction entre deux nombres proches est donc exacte. Elle est pourtant souvent appelée « annulation catastrophique » (*cancellation*) car elle fait ressortir les erreurs précédentes. Si le résultat est un dénormalisé, le résultat reste valide, c'est alors équivalent au lemme 2.7.

4.1.2 Erreur de l'addition

Si l'on ne considère que l'arrondi au plus proche, l'erreur commise lors d'une addition flottante est représentable par un flottant. Si l'on reprend la façon dont on calcule le résultat d'une addition de la section 2.4, on voit que si les nombres sont proches, alors l'erreur est à la suite du résultat arrondi de l'addition et tient sur un flottant. Si les nombres sont éloignés, le plus grand est le résultat tandis que le petit est l'erreur. Ce n'est pas le cas en arrondi dirigé. De plus, cette erreur exacte peut être calculée en utilisant uniquement des opérations flottantes [18, 28] :

Théorème 4.2 (FastTwoSum) *Soient a et b deux flottants. On calcule*

$$\begin{aligned} r_1 &= \circ(a + b) \\ b' &= \circ(r_1 - a) \\ r_2 &= \circ(b - b') \end{aligned}$$

Alors, si $|a| \geq |b|$, on a l'égalité mathématique $a + b = r_1 + r_2$.

Ainsi, si on connaît le plus grand des deux flottants a et b, on peut très facilement (2 additions supplémentaires) calculer l'erreur exacte de leur addition. Le théorème suivant permet de traiter le cas où l'ordre de a et b n'est pas connu à l'avance, au prix de 5 additions supplémentaires.

Théorème 4.3 (TwoSum) *Soient a et b deux flottants. On calcule*

$$\begin{aligned} r_1 &= \circ(a+b) \\ b' &= \circ(r_1 - a) \\ a' &= \circ(r_1 - b') \\ \epsilon_b &= \circ(b - b') \\ \epsilon_a &= \circ(a - a') \\ r_2 &= \circ(\epsilon_a + \epsilon_b) \end{aligned}$$

Alors on a l'égalité mathématique $a + b = r_1 + r_2$.

Dans les deux algorithmes précédents, il n'y a aucun souci en cas d'underflow : si a, b, r_1 ou r_2 est un dénormalisé, on aura toujours l'égalité mathématique $a + b = r_1 + r_2$.

4.1.3 Erreur de la multiplication

Le cas de la multiplication est en fait plus simple que l'addition : l'erreur de la multiplication est représentable quel que soit le mode d'arrondi si elle n'est pas trop petite. En effet, comme vu en section 2.4.2, multiplier deux flottants sur p bits donne un flottant sur $2p$ bits que l'on peut scinder en deux morceaux : le résultat arrondi et l'erreur. Parfois, l'erreur est trop petite pour être représentable, par exemple, si l'on multiplie $2^{e_{\min}}$ par 2^{-1}, l'arrondi est 0 et l'erreur est $2^{e_{\min}-1}$ qui n'est pas représentable par un flottant.

Reste ensuite à calculer cette erreur en n'utilisant que des calculs flottants. Pour cela, nous nous servons de l'algorithme suivant qui coupe en deux (partie haute et partie basse) un flottant. Ici, il s'agit d'un flottant double précision dont la mantisse fait 53 bits. La partie haute sera sur 26 bits. La partie basse devrait être sur $53 - 26 = 27$ bits, mais grâce au signe, on gagne un bit et la partie basse est également représentable sur 26 bits.

Théorème 4.4 (Veltkamp) *On calcule avec $p = 53$. Soit $C = 2^{27} + 1$ (flottant représentable) et soit a un flottant. On calcule*

$$\begin{aligned} p &= \circ(a \times C) \\ q &= \circ(a - p) \\ a_1 &= \circ(p + q) \\ a_2 &= \circ(a - a_1) \end{aligned}$$

Alors $a = a_1 + a_2$ et a_1 représente les 26 bits de poids fort et a_2 est également représentable sur 26 bits.

Pour calculer l'erreur de la multiplication, l'idée est alors assez simple : nous coupons en 2 les entrées a et b. Nous multiplions entre eux chacun des morceaux, car les multiplications sont alors exactes. Nous ajoutons ensuite dans le bon ordre les produits partiels [18] comme indiqué dans le théorème suivant.

Théorème 4.5 (Dekker) *Soient a et b deux flottants. On calcule*

$$\begin{aligned} (a_1, a_2) &= \mathrm{Veltkamp}(a) \\ (b_1, b_2) &= \mathrm{Veltkamp}(b) \\ r_1 &= \circ(a \times b) \\ t_1 &= \circ(-r + \circ(a_1 \times b_1)) \\ t_2 &= \circ(t_1 + \circ(a_1 \times b_2)) \\ t_3 &= \circ(t_2 + \circ(a_2 \times b_1)) \\ r_2 &= \circ(t_3 + \circ(a_2 \times b_2)) \end{aligned}$$

Alors, si l'erreur de la multiplication est représentable, $a \times b = r_1 + r_2$.

Il faut noter que cet algorithme est très coûteux (16 opérations supplémentaires). Si le processeur possède un FMA, calculer l'erreur de la multiplication devient trivial [27] :

Théorème 4.6 (FastTwoMult) *Soient a et b deux flottants. On calcule:*

$$\begin{aligned} r_1 &= \circ(a \times b) \\ r_2 &= \circ(a \times b - r_1) \end{aligned}$$

Alors $a \times b = r_1 + r_2$.

L'erreur de la multiplication étant un flottant, il est calculé exactement par le FMA en une seule opération s'il n'y a pas d'underflow.

4.1.4 Erreur du FMA

De la même façon que l'addition ou la multiplication, on peut vouloir calculer l'erreur exacte d'un FMA. Dans ce cas, l'erreur ne peut généralement pas être représentée sur un seul flottant, mais elle peut l'être comme somme de deux flottants, si le résultat de la multiplication n'est pas trop petit (voir ci-dessus). Nous avons un algorithme pour calculer ces erreurs [10]:

Théorème 4.7 (FmaErr) *Soient a, x et y des flottants. On calcule*

$$\begin{aligned} r_1 &= \circ(ax + y) \\ (u_1, u_2) &= \mathrm{FastTwoMult}(a, x) \\ (\alpha_1, \alpha_2) &= \mathrm{TwoSum}(y, u_2) \\ (\beta_1, \beta_2) &= \mathrm{TwoSum}(u_1, \alpha_1) \\ \gamma &= \circ(\circ(\beta_1 - r_1) + \beta_2) \\ (r_2, r_3) &= \mathrm{FastTwoSum}(\gamma, \alpha_2) \end{aligned}$$

Alors, si l'erreur de la multiplication du FMA est représentable, on a $ax + y = r_1 + r_2 + r_3$. De plus, r_1, r_2 et r_3 vérifient $|r_2 + r_3| \leq \frac{1}{2}\mathrm{ulp}(r_1)$ et $|r_3| \leq \frac{1}{2}\mathrm{ulp}(r_2)$.

Au final, il est possible d'identifier des calculs exacts ou de calculer exactement des erreurs avec un surcoût parfois important. Nous allons maintenant regarder des exemples où nous nous accommodons des calculs inexacts, mais où nous souhaitons au moins évaluer l'incertitude.

4.2 Exemples

4.2.1 Discriminant

Un premier exemple est celui du calcul précis du déterminant 2×2. Cela sert pour le test d'orientation (un point est-il à droite ou à gauche d'une droite) mais aussi pour le calcul du discriminant $b^2 - a \times c$. Nous allons nous intéresser à ce dernier cas. L'algorithme naïf est assez peu précis et peut facilement donner une réponse incorrecte. La figure 3 décrit un algorithme plus complexe dû à Kahan [26] qui donne le bon résultat à 2 ulps près.

La fonction `fabs` est la valeur absolue. La fonction `exactmult` correspond à la valeur r_2 de l'algorithme de Dekker présenté en section 4.1.3 : c'est l'erreur de la multiplication dont l'arrondi est donné en dernier argument. Ainsi, la somme $p + dp$ est mathématiquement égal à b^2.

L'idée de cet algorithme est la suivante : le test `p+q <= 3*fabs(p-q)` décide si $b^2 \approx a \times c$ et donc si les erreurs de calculs dans les multiplications risquent de fausser le résultat. Si ce test est vrai, il n'y aura pas d'annulation catastrophique et on peut choisir l'algorithme naïf qui donnera une bonne

```
double Kahan_discr (double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=exactmult(b,b,p);
    dq=exactmult(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

FIGURE 3 – Algorithme de Kahan pour le calcul précis du discriminant.

précision. Si ce test est faux, l'algorithme naïf est insuffisant et on calcule les erreurs des multiplications `dp` et `dq` et on tient compte de ces termes d'erreurs.

La preuve initiale est assez complexe et prouve que le résultat final est à moins de 2 ulps du résultat exact. La preuve complète est bien plus compliquée puisque le test flottant peut se tromper. En effet, le résultat exact du test $p + q \leq 3\,|p - q|$ peut différer de celui de `p+q <= 3*fabs(p-q)` aux limites. Ce fait n'était pas prévu par l'auteur de l'algorithme, mais ne remet pas en cause sa validité. Les cas d'erreur du test correspondent aux cas où $b^2 \approx \frac{a \times c}{2}$ et où $a \times c \approx \frac{b^2}{2}$ et dans ces cas, tous les chemins d'exécution donnent une erreur inférieure à 2 ulps [4].

Ici, il n'y a pas de problème supplémentaire dû à un underflow. Si les termes d'erreurs de la multiplication sont calculés exactement, alors l'algorithme fonctionne, même si ces derniers sont dénormalisés.

4.2.2 Évaluation polynomiale

Pour l'évaluation de fonctions élémentaires ($\cos$, $\exp \ldots$), on évalue souvent un polynôme avec un argument petit (comme un polynôme de Taylor). En pratique, cela fait des évaluations particulièrement précises, surtout lorsqu'on utilise la méthode de Horner. Ici, nous n'allons en fait nous intéresser qu'aux dernières opérations. En effet, les résultats précédents étant multipliés par l'argument réputé petit, leur influence sera minime. Mais nous voulons également pouvoir comparer le résultat au résultat mathématique exact, par exemple comparer $\exp(x)$ et `1+x+x*x/2`, ce qui nécessite de tenir compte d'une erreur de méthode [6].

Nous allons ici nous intéresser à la notion d'arrondi fidèle : f est un flottant arrondi fidèle d'un réel x si f est l'arrondi vers le haut ou vers le bas de x. Autrement dit, f est l'arrondi de x pour un certain mode d'arrondi et nous le noterons donc $f = \square(x)$. Nous ne garantissons pas l'arrondi exact, mais une très grande proximité et une erreur strictement inférieure à un ulp.

Théorème 4.8 (Axpy) *Soient les réels a_0, x_0 et y_0. Soient les nombres flottants représentables a, x et y avec une précision $p \geq 6$. Les nombres flottants représentables t et u sont définis par $t = \circ(a \times x)$ et $u = \circ(t + y)$. Si*

$$\left(3 + 2^{4-p}\right) \times |a \times x| \leq |y|, \text{ et}$$

$$|y_0 - y| + |a_0 \times x_0 - a \times x| < \frac{2^{1-p}}{12} \times |y|,$$

alors $u = \square(a_0 \times x_0 + y_0)$.

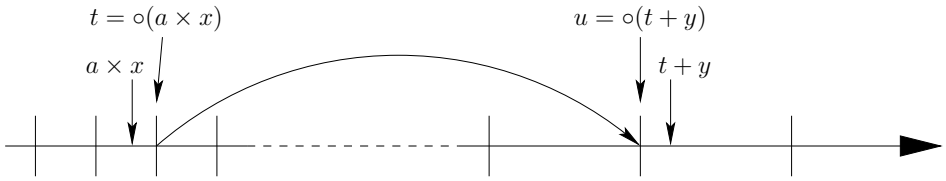

FIGURE 4 – Dernier pas de l'évaluation de Horner

Autrement dit on calcule le flottant renvoyé u comme montré en figure 4.2.2. En négligeant les dépassements de capacité, si 3 $|a \times x| \lesssim |y|$ et $err(y) + err(a \times x) < \frac{2^{1-p}}{12} \times |y|$, alors u est l'un des flottants encadrant le réel exact [11]. Bien que le théorème ne fasse pas apparaître l'exposant minimal, il tient précisément compte des dénormalisés et de tous les cas possibles d'underflow.

4.2.3 Émulation du FMA

Le FMA est un opérateur récent et beaucoup de processeurs ne le possèdent pas. Se pose alors la question d'émuler un FMA à l'aide des opérateurs flottants existants. Mais il est impossible d'émuler un FMA en utilisant seulement l'addition et la multiplication. Nous allons également utiliser un arrondi spécial appeler arrondi impair noté $\square_{\text{odd}}$ défini par

$$\square_{\text{odd}}(x) = \begin{cases} x & \text{si } x \in \mathbb{F}, \\ \triangle(x) & \text{si sa mantisse est impaire,} \\ \triangledown(x) & \text{sinon.} \end{cases}$$

Cet arrondi est un arrondi fidèle. De plus, il renvoie toujours un flottant impair, sauf lorsqu'il est exact. Cet arrondi peut être calculé comme un *ou logique* entre l'arrondi vers zéro et l'un des drapeaux que lève le processeur pour signaler un calcul inexact. Cela peut également être fait en logiciel en utilisant des tests [8].

Avec cet arrondi, on peut alors émuler un FMA de la façon suivante.

Théorème 4.9 (FmaEmul) *Soient a, x et y des flottants. On calcule*

$$\begin{aligned} (u_h, u_l) &= \text{ExactMult}(a, x) \\ (t_h, t_l) &= \text{TwoSum}(y, u_h) \\ v &= \square_{\text{odd}}(t_l + u_l) \\ z &= \circ(t_h + v) \end{aligned}$$

Alors, si l'erreur de la multiplication du FMA est représentable et que la précision $p \geq 5$, on a $z = \circ(a \times x + y)$.

L'arrondi impair permet également de calculer $\circ(a + b + c)$ [8].

4.2.4 Division entière sur Itanium

Une application contraire à l'intuition est la suivante : utiliser les opérateurs flottants pour effectuer des opérations entières. C'est le cas pour l'architecture IA-64 [14] qui ne contient pas d'unité calculant la division mais propose des FMAs. Nous ne présenterons ici que le cas de la division 16 bits.

Pour calculer a/b, on utilise une table qui donne des valeurs approchées sur 11 bits de $1/b$ pour les valeurs de b avec une erreur inférieure à 2^{-8}. L'idée est alors de partir de cette approximation et d'en doubler la précision. Il y a ici une difficulté supplémentaire : si nous utilisons une itération usuelle,

le résultat risque d'être incorrect car juste inférieur à la valeur voulue. Pour que les erreurs rendent la valeur toujours supérieure à la valeur voulue et donc tronquée au résultat correct, on ajoute une constante magique 2^{-17} qui est suffisante pour empêcher un mauvais arrondi trop petit mais pas assez grande pour créer un mauvais arrondi trop grand.

Théorème 4.10 *Soient a et b des entiers 16 bits. On calcule en précision $p = 64$:*

$$\begin{aligned} y_0 &= \mathrm{ApproxTable}(1/b) \\ q_0 &= \circ(a \times y_0) \\ e_0 &= \circ(1 + 2^{-17} - b \times y_0) \\ q_1 &= \circ(q_0 + e_0 \times q_0) \\ q &= \lfloor q_1 \rfloor \end{aligned}$$

Alors, $q = \lfloor a/b \rfloor$.

Les calculs se font en précision étendue avec $p = 64$. Regardons cet algorithme. Tout d'abord, on a $y_0 = 1/b(1+\varepsilon_0)$ avec $|\varepsilon_0| \leq 2^{-8,886}$. Puis $a \times y_0$ est calculé exactement car y_0 est sur 11 bits et a sur 16, donc $q_0 = a \times y_0 = a/b(1+\varepsilon_0)$ et est sur 27 bits. Puis, e_0 est calculé par un FMA. En fait, cette opération est exacte : c'est quasiment le théorème de Sterbenz car $b \times y_0 \approx 1$. Donc $e_0 = 1 + 2^{-17} - b \times y_0 = 2^{-17} - \varepsilon_0$. Le calcul de q_1 est également exact car $q_0 + e_0 \times q_0 = q_0(1 + e_0)$ et q_0 est sur 27 bits et $1 + e_0$ loge sur 28 bits, ce qui fait 55 bits et nous en avons 64. Donc $q_1 = q_0 + e_0 \times q_0 = a/b(1 + \varepsilon_0)(1 + e_0)$. Au final, q_1 est calculé avec une erreur faible et positive.

Un certain nombre de généralisations, ainsi que les preuves formelles, ont été faites par Harrison [23].

4.3 Exemple en analyse numérique : équation des ondes

Une partie de l'analyse numérique est consacrée à la recherche de schémas numériques, c'est-à-dire des algorithmes pour calculer une approximation de la solution exacte d'une équation aux dérivées partielles (EDP) ou d'une équation différentielle ordinaire. Nous présentons ici une EDP assez simple, l'équation des ondes en dimension 1 avec un schéma numérique très simple dont nous allons étudier l'erreur.

L'équation des ondes en dimension 1 est la solution de l'EDP

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = s(x,t)$$

avec des valeurs fournies pour la position $u_0(x)$ et la vitesse initiales $u_1(x)$, ainsi que pour la vitesse de propagation c. Elle correspond à l'oscillation d'une corde, au son, aux ondes radar...

Nous considérons une grille finie de taille $(\Delta x, \Delta t)$. Nous voulons calculer u_j^k qui approche $u(j\Delta x, k\Delta t)$, la solution exacte au point de la grille. Pour cela, nous utilisons le schéma à 3 points aux différences finies centré :

$$\frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} - c^2 \frac{u_{j+1}^{k-1} - 2u_j^{k-1} + u_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

comme expliqué en figure 5, ainsi que d'autres formules similaires pour les initialisations.

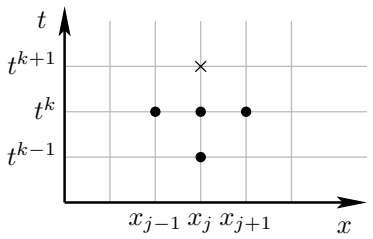

FIGURE 5 – Schéma à 3 points : u_j^k dépend de u_{j-1}^{k-1}, u_j^{k-1}, u_{j+1}^{k-1} et u_j^{k-2}.

Ce schéma est traduit en un programme C dont voici un extrait après les initialisations de `u[.][0]` et `u[.][1]` :

```
for (k=1; k<nk; k++) {
  u[0][k+1] = 0.;
  for (i=1; i<ni; i++) {
    du = u[i+1][k] - 2.*u[i][k] + u[i-1][k];
    u[i][k+1] = 2.*u[i][k] - u[i][k-1] + a*du;
   }
  u[ni][k+1] = 0.;
}
```

Pour prouver le bon comportement de ce programme, il faut donc à la fois borner son erreur d'arrondi (car les calculs ne sont pas exacts) et son erreur de méthode (car le schéma n'est pas la solution exacte).

4.3.1 Erreur de méthode

L'idée ici est de prouver que le schéma numérique sans erreurs d'arrondi est proche de la solution exacte de l'EDP. C'est un fait bien connu en mathématiques. La traduction vers la preuve formelle s'est cependant révélée plus ardue que prévu [5]. Nous ne donnerons ici que l'idée générale.

Nous utilisons un grand O uniforme où la première fonction a 2 variables et la seconde une seule. La relation $f = O(g)$ est définie par

$$\exists \alpha, C > 0, \quad \forall \mathbf{x}, \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|.$$

En fait, f dépend du point (espace, temps) et de la taille de la grille tandis que g ne dépend que de la taille de la grille.

La preuve mathématique se fait alors en trois temps [5] :

- La consistance mesure à quel point la solution exacte résout le schéma numérique. La preuve se base sur le théorème de Taylor et beaucoup de manipulations symboliques.
- La stabilité garantit que les valeurs du schéma numérique restent bornées. La preuve se base sur la définition d'une énergie discrète dont on donne les propriétés. La preuve consiste en des manipulations de produits scalaires et de normes.
- La convergence est la preuve finale à partir des résultats précédents et des initialisations.

Au final, nous prouvons que le schéma numérique est borné. Plus précisément, notons e_j^k l'erreur de convergence définie par $u(j\Delta x, k\Delta t) - u_j^k$ et définissons la norme utilisée par $\|q_\mathrm{h}\|_{\Delta x}^2 = \sum_{i=0}^{i_\mathrm{max}} q_i^2 \Delta x$. Nous prouvons alors la formule attendue par les numériciens :

$$\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O\left| \begin{array}{l} t \in [0, t_\mathrm{max}] \\ (\Delta x, \Delta t) \to 0 \\ 0 < \Delta x \wedge 0 < \Delta t \wedge \\ c\frac{\Delta t}{\Delta x} \leq 1 - \xi \end{array} \right. (\Delta x^2 + \Delta t^2).$$

Quand la taille de la grille décroît, Δx et Δt décroissent et l'erreur de convergence décroît de façon quadratique.

4.3.2 Erreur d'arrondi

Avec une technique naïve, on obtient une erreur d'arrondi de l'ordre de $2^k\,2^{-53}$ au bout de k pas de temps, ce qui est à la fois énorme et bien supérieur à ce que l'on constate en pratique. En effet, les erreurs d'arrondi se compensent lors des calculs. Pour exprimer (et prouver) ce fait, nous allons donner une expression précise de l'erreur d'arrondi signée en fonction d'erreurs élémentaires [3]. Nous mettrons ainsi en évidence les compensations.

Nous rappelons l'itération principale :

```
dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
```

En supposant que les `p[.][.]` sont exacts, nous notons ε_i^{k+1} l'erreur d'arrondi commise lors de ces deux lignes de calcul. Comme le schéma numérique est borné et que nous connaissons les initialisations, nous pouvons garantir que les flottants sont inférieurs à 2. Nous en déduisons donc par arithmétique d'intervalles que pour tout n, m, nous avons $|\varepsilon_n^m| \leq 78 \times 2^{-52}$.

Nous définissons ensuite une suite mathématique (qui est une solution du schéma avec d'autres initialisations) par

$$\begin{array}{lll} \alpha_0^0 = 1 & \forall i \neq 0,\ \alpha_i^0 = 0 & \\ \alpha_{-1}^1 = \alpha_1^1 = (1-a) & \alpha_0^1 = 2a & \forall i \notin \{-1,0,1\},\ \alpha_i^1 = 0 \\ \alpha_i^k = a \times (\alpha_{i-1}^{k-1} + \alpha_{i+1}^{k-1}) + 2(1-a) \times \alpha_i^{k-1} - \alpha_i^{k-2} & & \end{array}$$

Nous prouvons alors que l'erreur d'arrondi des p_i^k est

$$p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l\ \varepsilon_{i+j}^{k-l}.$$

Cette double sommation permet d'exhiber les compensations d'erreurs et donne au final une erreur d'arrondi tout à fait acceptable, de l'ordre de $k^2\ 2^{-53}$:

$$\left| p_i^k - exact\left(p_i^k\right) \right| \leq 78 \times 2^{-53} \times (k+1) \times (k+2).$$

Au final, nous avons complètement annoté (154 lignes d'annotations pour 32 lignes de C) et prouvé le programme (149 obligations de preuves) en utilisant des outils automatiques (117 obligations de preuves automatiquement prouvées) et Coq (15000 lignes et 30 minutes de compilation).

4.4 Fonctions élémentaires

L'évaluation des fonctions élémentaires constitue un domaine où des techniques très pointues d'arithmétique des ordinateurs sont mises en œuvre. Les fonctions élémentaires désignent généralement l'ensemble des fonctions obtenues par composition de polynômes, de l'exponentielle complexe, et de l'inverse de fonction. On y trouve ainsi le logarithme, le sinus, ou encore l'arctangente hyperbolique.

Une particularité des fonctions élémentaires usuelles est que leur évaluation peut se faire en trois étapes [36]. Prenons l'exemple de l'exponentielle que l'on cherche à évaluer en x :

1. Réduction d'argument : trouver x' proche de zéro et $k \in \mathbb{Z}$ tels que $x \simeq x' + k \log 2$.
2. Évaluation sur un petit intervalle : calculer $y' \simeq \exp(x')$.
3. Reconstruction du résultat : renvoyer $y = y' \cdot 2^k$.

Remarquez que l'étape de reconstruction est ici triviale : y s'obtient en incrémentant de k l'exposant du nombre flottant y'. Pour d'autres fonctions, ce pourra être la réduction d'argument qui est triviale tandis que la reconstruction est compliquée ; c'est par exemple le cas du logarithme en base 2. Enfin, il y

a des fonctions pour lesquelles aussi bien la réduction d'argument que la reconstruction du résultat sont compliquées.

L'intérêt d'effectuer ces étapes 1 et 3 est de simplifier et d'accélérer autant que possible l'étape 2. En effet, l'évaluation se fait généralement à l'aide d'un polynôme. Or plus l'intervalle est petit, plus l'approximation polynomiale sera précise, ou, à précision égale, plus le degré du polynôme sera petit et donc son évaluation rapide.

4.4.1 Réduction d'argument

La première étape consiste à ramener l'entrée de la fonction dans un petit intervalle. Par exemple, dans le cas d'une fonction circulaire, on pourrait vouloir obtenir x' et k tels que $x \simeq x' + k\pi/4$ et $0 \leq x' \lesssim \pi/4$. En fait, les fonctions circulaires étant périodiques, la valeur exacte de k est sans intérêt, seule sa valeur modulo 8 importe. Autrement dit, nous allons chercher comment calculer x' et k tels que $x' + k\pi/4 - x \simeq 0\ [2\pi]$. Nous supposerons que $x \geq 0$.

Considérons d'abord la formule naïve suivante :

$$k = \lfloor \circ(x \times \circ(4/\pi)) \rfloor \quad \text{et} \quad x' = \circ(x - \circ(\circ(\pi/4) \times k)).$$

Si k est non nul, la valeur de $\circ(\pi/4) \times k$ est suffisamment proche de x pour que le théorème 4.1 s'applique. La soustraction de x' est donc exacte. Par conséquent, l'erreur absolue totale commise lors du calcul de x' est celle de $\circ(\circ(\pi/4) \times k)$, c'est-à-dire quelques $\mathrm{ulp}(x)$ puisqu'il n'y a que des multiplications et pas de risque de valeurs dénormalisées. Autrement dit, l'erreur absolue sur x' augmente avec x alors que la valeur de x' reste plus petite que $\pi/4$. Pour éviter cela, il faudrait augmenter la précision à laquelle est connue la constante $\pi/4$ et sont effectués les calculs intermédiaires. Ce dernier point rend cet algorithme inutilisable en pratique pour les grandes valeurs de x.

L'objectif est donc de trouver un moyen de garantir une erreur absolue faible tout en ayant une précision intermédiaire constante et donc un temps de calcul constant. Tout d'abord, modifions légèrement l'algorithme de la façon suivante :

$$z = \circ(x \times \circ(4/\pi)) \quad \text{et} \quad k = \lfloor z \rfloor \quad \text{et} \quad x' = \circ(\circ(\pi/4) \times \circ(z - k)).$$

La situation n'est pas plus satisfaisante qu'avant, l'erreur absolue finale étant de l'ordre de celle commise lors du calcul de z. Remarquez que cette borne sur l'erreur absolue est directement proportionnelle à la constante $4/\pi$ qui apparaît dans z. S'il était possible de la remplacer par une constante plus petite, l'erreur absolue pourrait être ramenée dans des bornes raisonnables. C'est l'idée de l'algorithme suivant [38] :

$$z = \circ(x \times C_{e_x}) \quad \text{et} \quad k = \lfloor z \rfloor \quad \text{et} \quad x' = \circ(\circ(\pi/4) \times \circ(z - k)).$$

L'objectif est donc de trouver toute une famille de constante C_i. La constante C_{e_x} utilisée lors de la réduction d'argument dépend de l'exposant de x puisque plus x est grand plus il faut que la constante soit petite pour garder une erreur absolue bornée. Évidemment, puisque cette constante n'est plus une approximation de $4/\pi$, la propriété $x \simeq x' + k\pi/4$ n'est plus vérifiée. La propriété $0 \leq x \lesssim \pi/4$ est par contre vérifiée. Il reste donc à choisir C_{e_x} de telle sorte que $x' + k\pi/4 - x \simeq 0\ [2\pi]$.

Les calculs intermédiaires seront effectués dans une précision p tandis que $x = m_x \cdot 2^{e_x}$ est représenté avec une précision p_x moindre. Soit $w = e_x - 3$. Représentons la valeur $4/\pi$ comme la somme $\alpha_h \cdot 2^{-w} + \alpha_l$ avec α_h un entier et α_l un réel entre 0 et 2^{-w}. Fixons $C_{e_x} = \bigtriangledown(\alpha_l)$.

Le produit $x \times C_{e_x}$ vaut donc

$$\begin{aligned} x \times C_{e_x} &= x \times 4/\pi - m_x \cdot 2^{e_x}(\alpha_h \cdot 2^{-w} + \alpha_l - \bigtriangledown(\alpha_l)) \\ &= x \times (4/\pi + \alpha_l - \bigtriangledown(\alpha_l)) - 8 m_x \alpha_h. \end{aligned}$$

Le produit $8m_x\alpha_h$ est un entier multiple de 8. Il n'a donc aucune influence sur la valeur de k modulo 8 ; il n'en a pas non plus sur la valeur de x'. L'erreur absolue commise est maintenant de l'ordre de $x \cdot (\alpha_l - \bigtriangledown(\alpha_l))$ qui est plus petit que $2^{e_x+p_x-w-p+1} = 2^{p_x-p+4}$. Notez que la précision p des calculs intermédiaires doit bien être supérieure à celle p_x de l'entrée mais qu'elle ne dépend pas de l'amplitude, ce qui était l'objectif recherché.

On pourrait penser *a priori* que l'ensemble des constantes C_i nécessite un espace de stockage de l'ordre de $p \times e_{\max}$ bits. Ce n'est pas le cas. En effet, les constantes C_i et C_{i+1} diffèrent peu : l'exposant est décrémenté de 1 tandis que les mantisses sont identiques à un décalage de 1 près. Par conséquent, environ $p + e_{\max}$ bits du développement binaire de $4/\pi$ suffisent pour effectuer la réduction d'argument.

Ceci n'est qu'un aperçu des algorithmes de réduction d'argument ; il en existe de nombreux autres. Qui plus est, il est parfois nécessaire d'avoir une borne sur l'erreur relative et non pas l'erreur absolue et cet algorithme ne serait donc pas directement utilisable.

4.4.2 Calcul en haute précision

Tout d'abord, à moins d'avoir une reconstruction triviale, des erreurs d'arrondi supplémentaires viendront s'ajouter après l'évaluation polynomiale. Qui plus est, certaines reconstructions peuvent amplifier les erreurs commises lors de la réduction d'argument et l'évaluation polynomiale. Par conséquent, il n'est généralement pas possible de faire les calculs intermédiaires en précision p si l'objectif est d'obtenir une erreur relative finale de l'ordre de 2^{-p}. La précision des calculs intermédiaires doit donc être plus élevée, parfois le double ou le triple de la précision attendue comme c'est le cas pour les fonctions correctement arrondies que nous aborderons plus tard.

Si l'architecture cible propose plusieurs précisions de calcul dont une suffisamment grande pour effectuer les calculs intermédiaires, il n'y a pas de difficulté. Si ce n'est pas le cas, il faut émuler cette précision étendue. Une approche consiste à employer des *expansions* : les nombres en précision étendue sont stockés sous la forme d'une somme non évaluée de plusieurs nombres flottants [39]. En étageant les exposants de ces différents nombres, il est ainsi possible d'augmenter artificiellement la précision de l'expansion. Par exemple, un nombre x pourrait être représenté par une expansion de taille 2, $x_h + x_l$ avec $|x_l| \leq \mathrm{ulp}(x_h)/2$, offrant ainsi au moins $2p$ bits de mantisse virtuelle. Les nombres représentés par une expansion auront par contre la même plage d'exposant que les nombres flottants usuels. Les expansions peuvent avoir une taille aussi longue que l'on souhaite, la seule limite étant qu'à partir d'un certain nombre de termes, toute la plage d'exposants est nécessairement couverte ; ajouter de nouveaux termes ne permet alors plus de représenter de nouveaux nombres.

Il reste à définir des opérations sur ces nombres. Nous allons voir ici le cas de l'addition [18]. Supposons que nous ayons deux expansions de taille 2, $x_h + x_l$ et $y_h + y_l$, et que nous cherchions une expansion de taille 2 dont la valeur est proche de la somme des deux entrées. Sans perte de généralité, nous pouvons supposer $|x_h| \geq |y_h|$. L'algorithme suivant permet de calculer l'expansion somme :

$$\begin{aligned} (r_h, r_l) &= \mathrm{FastTwoSum}(x_h, y_h) \\ s &= \circ(\circ(r_l + y_l) + x_l) \\ (t_h, t_l) &= \mathrm{FastTwoSum}(r_h, s) \end{aligned}$$

Comparer les magnitudes des différents nombres montre que si r_h n'est pas nul, alors il est plus grand que s et l'hypothèse du deuxième FastTwoSum est donc satisfaite. En conséquence de quoi, les exposants de t_h et t_l sont correctement étagés dans l'expansion. Appliquons le lemme 2.8 à chacune des deux additions arrondies de s, ce qui donne des ε_1 et ε_2 tels que

$$\begin{aligned} t_h + t_l &= r_h + s \\ &= x_h + y_h - r_l + ((r_l + y_l)(1+\varepsilon_1) + x_l)(1+\varepsilon_2) \\ &= x_h + x_l + y_h + y_l + (r_l + y_l)\varepsilon_1(1+\varepsilon_2) + (x_l + y_l + r_l)\varepsilon_2 \end{aligned}$$

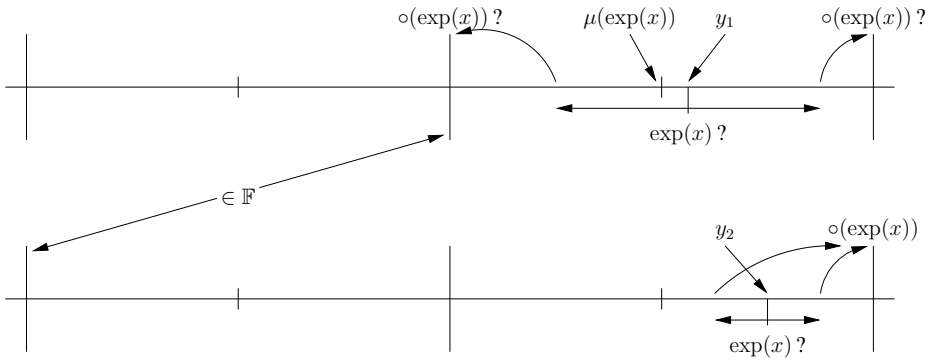

FIGURE 6 – Deux approximations successives y_1 et y_2 de $\exp(x)$ dans le but de calculer $\circ(\exp(x))$.

Les erreurs relatives locales étant inférieures à 2^{-p} et les termes x_l, y_l et r_l étant négligeables devant x_h, l'erreur relative totale est donc de l'ordre de quelques ulps en l'absence d'une annulation catastrophique de $x_h + y_h$. Il existe d'autres versions de l'addition qui, au prix d'opérations supplémentaires, permettent d'obtenir une erreur moindre pour les expansions de taille 2 [24].

4.4.3 Dilemme du fabriquant de table

Les paragraphes précédents ont expliqué comment calculer une approximation flottante d'une fonction élémentaire. Une fois la reconstruction terminée, ce qui est calculé est une valeur y représentée par la somme de plusieurs flottants ou par un nombre flottant en précision étendue. Supposons maintenant que l'objectif est en fait de renvoyer $\circ(\exp(x))$, c'est-à-dire l'unique flottant le plus proche de la valeur mathématique de $\exp(x)$. Les conséquences de cette propriété sont nombreuses : précision, portabilité, reproductibilité, préservation des propriétés de monotonie, ... Elles sont toutes souhaitables, ce qui justifie donc de vouloir calculer des fonctions élémentaires *correctement arrondies*.

Comme ce qui a été calculé est y, l'approche naturelle est de calculer $\circ(y)$, à supposer que ce soit égal à $\circ(\exp(x))$. À partir du moment où y est suffisamment proche de $\exp(x)$, c'est-à-dire quand l'erreur relative est inférieure à 2^{-p}, nous avons une propriété d'arrondi fidèle : $\circ(y)$ est l'un des deux flottants encadrant $\exp(x)$. Les flottants $\circ(y)$ et $\circ(\exp(x))$ seront donc différents si et seulement si y et $\exp(x)$ sont de part et d'autre du milieu $\mu(\exp(x))$ entre ces deux flottants encadrant $\exp(x)$.

Une condition suffisante pour avoir l'arrondi correct est donc que l'erreur $|y - \exp(x)|$ soit inférieure à la distance au milieu $|y - \mu(\exp(x))|$. Remarquez que, si une borne sur l'erreur est connue, ce test peut se faire dynamiquement lors de l'évaluation de la fonction. Si la condition n'est pas satisfaite, une première solution est donc de recommencer tout le calcul à zéro en utilisant une plus grande précision intermédiaire et un plus grand degré de polynôme pour diminuer l'erreur. Un exemple est donné sur la figure 6. Le processus est répété jusqu'à ce que l'erreur soit plus petite que la distance au milieu [42]. Cette méthode termine puisque y converge vers $\exp(x)$ et que $|\exp(x) - \mu(\exp(x))|$ ne vaut zéro pour aucun x flottant.

L'inconvénient est que l'on ne sait pas *a priori* jusqu'à quelle précision il va falloir monter et donc quelles sont les complexités temporelle et spatiale de l'évaluation d'une fonction élémentaire. C'est le *dilemme du fabriquant de table* : jusqu'à quel ordre doit calculer un éditeur de table de logarithmes pour garantir que tous les chiffres imprimés sont corrects ?

Dans le cas de l'arithmétique à virgule flottante, connaître $\delta = \min\{|\exp(x) - \mu(\exp(x))| \ ; \ x \in \mathbb{F} \cap [-M; M]\}$ résoudrait donc le problème. Comme le nombre de flottants est fini, c'est effectivement un minimum et il ne vaut pas zéro. Cependant, même si le nombre de flottants est fini, obtenir cette valeur par calcul exhaustif n'est pas raisonnable et il est donc important de mettre en œuvre des méthodes plus subtiles [29].

5 Conclusion

Ce document ne constitue qu'un petit aperçu des questions liées à l'arithmétique des ordinateur et à l'utilisation de la preuve formelle dans ce domaine. Voici une sélection de quelques ouvrages pour creuser certains des sujets abordés.

Concernant l'arithmétique à virgule flottante au sens de la norme IEEE-754 et ses utilisations avancées, l'ouvrage de référence est *Handbook of Floating-Point Arithmetic* [37]. Pour avoir plus de détails sur la façon dont elle est implantée efficacement en matériel, consultez *Digital Arithmetic* [20].

De plus amples informations sur le calcul en précision plus haute que celle fournie par le matériel se trouvent dans cet article dédié à la géométrie algorithmique : *Adaptative precision floating-point arithmetic and fast robust geometric predicates* [40]. Quant aux algorithmes pour faire du calcul efficace multi-précision en général, l'ouvrage de référence est *Modern Computer Arithmetic* [13].

Nous avons en général peu abordé les questions d'analyse numérique et en particulier pas du tout celles liées à l'algèbre linéaire et au calcul de l'erreur inverse. Un bon ouvrage sur ces deux sujets est *Accuracy and Stability of Numerical Algorithms* [25].

Enfin, dans le domaine de la preuve formelle pour l'arithmétique à virgule flottante, on pourra lire cette version longue d'un article dédié à la vérification de bout en bout d'une fonction élémentaire en HOL Light : *Floating-point verification in HOL Light: the exponential function* [22]. Pour ce qui est de l'utilisation de Coq pour la preuve de propriété an arithmétique flottante, se reporter à la thèse *Preuves formelles en arithmétiques à virgule flottante* [2].

Références

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[2] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, 2004.

[3] Sylvie Boldo. Floats & Ropes: a case study for formal numerical program verification. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *LNCS*, pages 91–102, Rhodos, Greece, 2009.

[4] Sylvie Boldo. Kahan's algorithm for a correct discriminant computation at last formally proven. *IEEE Transactions on Computers*, 58(2):220–225, 2009.

[5] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal proof of a wave equation resolution scheme: the method error. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the first Interactive Theorem Proving Conference*, volume 6172 of *LNCS*, pages 147–162, Edinburgh, Scotland, 2010.

[6] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 37(1-4):45–60, 2004.

[7] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sarcedoti Coen, and Stephen M. Watt, editors, *Proceedings of the 16th Calculemus Symposium*, volume 5625 of *LNAI*, pages 59–74, Grand Bend, ON, Canada, 2009.

[8] Sylvie Boldo and Guillaume Melquiond. Emulation of FMA and correctly-rounded sums: Proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4):462–471, 2008.

[9] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, 2011.

[10] Sylvie Boldo and Jean-Michel Muller. Exact and approximated error of the FMA. *IEEE Transactions on Computers*, 60(2):157–164, 2011.

[11] Sylvie Boldo and César Muñoz. Provably faithful evaluation of polynomials. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, volume 2, pages 1328–1332, Dijon, France, April 2006.

[12] Sylvie Boldo and Thi Minh Tuyen Nguyen. Proofs of numerical programs when the compiler optimizes. *Innovations in Systems and Software Engineering*, 7:1–10, 2011.

[13] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[14] Marius Cornea, Cristina Iordache, John Harrison, and Peter Markstein. Integer divide and remainder operations in the IA-64 architecture. In Jean-Claude Bajard, Christiane Frougny, Peter Kornerup, and Jean-Michel Muller, editors, *Proceedings of the 4th Conference on Real Numbers and Computers*, pages 161–184, Schloss Dagstuhl, Germany, 2000.

[15] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1), 2010.

[16] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[17] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[18] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.

[19] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[20] Miloš D. Ercegovac and Tomas Láng. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[21] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[22] John Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[23] John Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[24] Yodo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, CO, USA, 2001.

[25] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 2nd edition, 2002.

[26] William Kahan. On the cost of floating-point computation without extra-precise arithmetic. World-Wide Web document, 2004.

[27] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software*, 23(4):561–589, 1997.

[28] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, second edition, 1981.

[29] Vincent Lefèvre, Jean-Michel Muller, and Arnaud Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, 1998.

[30] Guillaume Melquiond. Floating-point arithmetic in the Coq system. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 93–102, Santiago de Compostela, Spain, 2008.

[31] Guillaume Melquiond. Proving bounds on real-valued functions with computations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *LNAI*, pages 2–17, Sydney, Australia, 2008.

[32] Guillaume Melquiond and Sylvain Pion. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications*, 41(1):57–70, 2007.

[33] Microprocessor Standards Committee. IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2008*, pages 1–58, August 2008.

[34] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on programming languages and systems*, 30(3):12, 2008.

[35] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, Philadelphia, PA, USA, 2009.

[36] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 1997.

[37] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[38] Mary H. Payne and Robert N. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[39] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, 1991.

[40] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[41] Pat H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.

[42] Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, 1991.