Numéro d'ordre : 482
Numéro attribué par la bibliothèque : 07ENSL0 482

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

spécialité : Informatique

Laboratoire de l'Informatique du Parallélisme

École Doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 17 octobre 2008 par

Christoph Quirin LAUTER

Arrondi correct de fonctions mathématiques

Fonctions univariées et bivariées, certification et automatisation

Directeur de thèse :	Florent DE DINECHIN
Co-encadrant de thèse :	Jean-Michel MULLER
Après avis de :	Marius CORNEA
	Philippe LANGLOIS
	Wolfgang WALTER

Devant la Commission d'examen formée de :

Jean-Marie CHESNEAUX	Membre
Marius CORNEA	Membre/Rapporteur
Florent DE DINECHIN	Membre
Philippe LANGLOIS	Membre/Rapporteur
Jean-Michel MULLER	Membre
Wolfgang WALTER	Membre/Rapporteur

À Olya, ma femme géniale,
sans qui les choses seraient différentes et tristes...

À mes parents artistes,
qui ont essayé de me léguer de leur créativité...

À ma sœur,
la meilleur sœur qui soit...

Remerciements

Je tiens tout d'abord à remercier mes deux directeurs de thèse, Florent de Dinechin et Jean-Michel Muller, pour leurs excellentes qualités d'encadrants et pour m'avoir donné la liberté de découvrir beaucoup d'intéressantes choses seul et surtout avec eux. Merci à eux d'avoir toujours été disponibles pour répondre à mes question, de leurs conseils avisés et d'avoir une oreille ouverte dans les moments de doute et de difficulté. Un grand merci particulièrement à Florent pour m'avoir indirectement admis dans sa famille et pour être un ami non seulement professionnel mais surtout aussi personnel. Merci encore, Florent, pour ton aide lors de mes périples en Mairies et Préfectures. Merci aussi à Nicolas Brisebarre pour avoir été une vraie aide dans ces moments.

Je tiens évidemment à remercier également Jean-Marie Chesneaux, professeur à l'université Pierre et Marie Curie, pour avoir accepté de présider le jury lors de ma soutenance. Je remercie aussi Philippe Langlois, Wolfgang Walter et Marius Cornea pour avoir relu ce document avec soin malgré les brefs délais. Merci, Danke et Mulţumesc pour leurs commentaires qui m'ont permis, je l'espère, d'améliorer la qualité et la clarté de ce document. Je remercie de plus Marius Cornea pour m'avoir proposé un poste chez Intel et de m'y avoir défendu après ma candidature.

Merci beaucoup aussi à mes coauteurs Florent de Dinechin, Jean-Michel Muller, Sylvain Chevillard, Guillaume Melquiond, Peter Kornerup, Vincent Lefèvre, Nicolas Louvet et David Defour pour le temps formidable que j'ai pu passer avec eux lors de la publication d'articles et pour avoir supporté et corrigé sans cesse mes phrases beaucoup trop longues...

Je tiens à remercier profondément mes cobureaux, Guillaume Melquiond, Sylvain Chevillard, Romain Michard, Francisco Chávez et Jérémie Detrey. Merci de m'avoir supporté malgré mon penchant pour les entêtements, pour le chaos sur mon bureau et pour d'autres tics qu'ils ne m'ont jamais reprochés. Un grand merci à Guillaume pour être mon extraterrestre préféré pour des questions de casts en C/configuration Debian/Shell/HTML/Gappa/etc. Je tiens à remercier particulièrement Sylvain pour sa coopération amicale dans nos recherches et développements logiciels, les échanges fructueux de problèmes et solutions que j'ai pu avoir avec lui et grâce à lui. Merci également, Sylvain, d'être aussi têtu que moi, de ne pas avoir manqué d'occasion de me convaincre du danger de manger à 11h30, d'avoir subi mes taquineries et bien sûr, pour fpminimax (pas) dans Sollya et la condition de Haar.

Mes remerciements vont aussi à tous les autre membres de l'équipe Arénaire et au LIP entier. Merci pour m'avoir si chaleureusement accueilli, pour vos remarques et conseils, pour l'administration de ce cadre excellent de recherche qu'est le LIP. Merci donc à Gilles Villard, Claude-Pierre Jeannerod, et à Nathalie Revol ainsi qu'à tous les autres doctorants de l'équipe, en particulier à Guillaume Revy. Merci bien sûr aussi à Sylvie Boyer, Corinne Iafrate et Isabelle Pera ainsi qu'à Dominique Ponsard et Serge Torres. Merci, Serge, aussi pour

ton aide au développement logiciel numérique.

Je tiens à remercier aussi tous ceux et celles que j'aurais oublié de citer par leur nom. Je vous présente mes plus plates et confuses excuses pour cet oubli dans cette liste, qui est loin d'être exhaustive. Merci également à toute la France pour avoir financé cette thèse.

Je tiens particulièrement à remercier ma famille de m'avoir supporté durant cette thèse, surtout dans les moments de plus graves doutes. Danke ! Un grand merci va aussi du fond de mon cœur à Olya, ma femme, pour être venue en France, avoir appris le français à cause de cette thèse. Merci, Lelya, pour avoir épousé le Чувырло que je suis, pour me supporter au jour le jour, en particulier dans les moments de doute, et pour ton aide et tes conseils de sens pratique. Спасибо, Лёля!

Table des matières

Table des figures

Introduction

Un problème historique

Vers 1620, les anglais Napier et Briggs ainsi que le suisse Bürgi publient les premiers tables de logarithmes [103, 20, 15]. Par ces tables, ils ne simplifient pas seulement les calculs que leur contemporains devaient faire de tête. Ils fondent aussi une nouvelle branche dans les mathématiques de l'époque.

Pourtant, avec les méthodes de l'époque et sans aucune assistance mécanique, calculer ces tables relevait du défi. Briggs, par exemple, mettra sept ans à compléter en 1624 sa deuxième table de logarithmes [16]. Elle est précise à 14 décimales !

Mais ces 14 décimales sont-elles vraiment justes ? Il semble injuste de reprocher à ces fiers pionniers une éventuelle naïveté dans l'estimation des erreurs qu'il commettaient forcement en approchant leurs logarithmes par quelque procédé arithmétique. Mais comment Briggs déterminait-il la quatorzième décimale ? Regardons par exemple l'entrée pour 12530 dans la table des logarithmes en base 10. Avec les moyens d'aujourd'hui il est facile de voir que

$$\log_{10}(12530) = 4.0979510709941\ 5000028259\ldots$$

Si Briggs avait par exemple décidé de mener ses calculs intermédiaires sur 17 décimales en tronquant les décimales en trop, il aurait pu obtenir le résultat suivant :

$$\log_{10}(12530) \approx 4.0979510709941\ 499$$

Ensuite, il aurait pu se dire : les deux nombres sur 14 décimales autour de mon résultat intermédiaire sont 4.0979510709941 et 4.0979510709942. Leur milieu est $4.0979510709941\ 500$. Alors le premier nombre est plus proche de mon résultat intermédiaire que le deuxième nombre. Donc je vais reporter 4.0979510709941 dans ma table. À ce moment-là, il se serait trompé sur la dernière décimale ! Avec le résultat obtenu très précisément avec les moyens modernes, on voit que le logarithme de 12530 est en fait plus grand que le milieu des deux nombres. Il aurait fallu marquer 4.0979510709942. Ce n'est qu'en calculant sur 20 décimales d'abord, Briggs aurait pu obtenir une entrée correcte :

$$\log_{10}(12530) \approx 4.0979510709941\ 500001.$$

Briggs et ses homologues, tous fabricants de tables, se trouvaient alors devant le dilemme suivant : devraient-ils calculer vraiment sur 20 décimales pour n'en donner que 14 ? Quel gâchis de temps et de peine ! En plus, qui pouvait leur assurer que 20 décimales suffisaient ? Peut-être en aurait-il fallu encore plus. Par ailleurs, le fait d'avoir toutes les décimales justes ne pouvait leur servir qu'à une chose : comparer leurs tables à celles des autres fabricants pour déceler des fautes de calcul.

Le dilemme des fabricants de tables dans l'informatique

On pourrait croire que ce dilemme des fabricants de tables est un problème anecdotique des mathématiques de la Renaissance. Mais il arrive systématiquement quand on essaie d'arrondir correctement, c'est-à-dire de choisir, dans un ensemble discret, l'élément le plus proche d'un résultat mathématique que l'on ne sait qu'approcher.

L'importance du dilemme des fabricants de tables pour l'arithmétique des ordinateurs est observée pour la première fois en 1974 par Kahan [71]. Reinsch le redécouvre en 1979 pour les fonctions de conversion de base [109]. Bien que les machines soient souvent binaires, l'entrée et la sortie de nombres se fait souvent en base 10 pour accommoder les humains. Lors des conversions, un arrondi est à effectuer. Tout en remarquant les avantages d'une arithmétique bien spécifiée, Reinsch reste très pessimiste sur l'arrondi correct. Il traite même de coupeurs de cheveux en quatre les gens qui veulent obtenir un dernier bit correct à tout prix [109].

En 1985, l'arithmétique virgule flottante est normalisée par la norme IEEE 754 [5]. Toujours par crainte du dilemme énoncé, cette norme demande l'arrondi correct seulement pour des opérations qui sont calculables exactement ou qui ont une inverse qui l'est.

En 1990, Dunham doute encore de la faisabilité de l'arrondi correct de fonctions mathématiques [46]. Pourtant, en 1991, Ziv propose un procédé qui en prouve la faisabilité pratique [130]. L'approche consiste à prendre du recul avant d'arrondir : quand l'approximation est très proche d'un endroit où l'arrondi change – le milieu des deux nombres dans les exemples précédents, on n'arrondit pas tout de suite. On augmente itérativement le nombre de décimales du résultat intermédiaire jusqu'à ce que l'arrondi correct devienne possible. Si l'erreur de l'approximation est connue et bornée, ce test est simple.

Comme on peut l'observer, les cas où beaucoup de décimales supplémentaires sont nécessaires sont rares [52]. Une implantation basée sur le procédé de Ziv délivrera donc l'arrondi correct avec une performance moyenne très bonne [52, 130, 40]. Seuls les cas difficiles à arrondir, mais rares, connaîtront des performances moindres.

En 2001 et 2002, les compagnies IBM et Sun publient chacune une bibliothèque de fonctions mathématiques correctement arrondies nommées IBM libultim et Sun libmcr basées sur le procédé de Ziv. La bibliothèque proposé par IBM sera intégré plus tard dans la bibliothèque standard de la plupart des systèmes Linux. Ainsi, l'arrondi correct commence à se démocratiser.

Bien que le procédé de Ziv trouve un moyen pratique pour le dilemme des fabricants de tables, il ne le résout pas encore complètement. Une question se pose toujours : combien d'itérations du procédé de Ziv, c'est-à-dire combien de décimales, faudra-t-il au maximum pour arrondir correctement et, d'ailleurs, termine-t-elle ? En d'autres mots, les implantations basées sur ce procédé ne peuvent pas garantir leurs performances en temps et en utilisation mémoire. Cette situation peut être fâcheuse par exemple dans des systèmes temps réel ou embarqués.

Ce n'est en 1996 que Muller et Lefèvre commencent à résoudre le dilemme des fabricants de tables avec une approche aussi pragmatique qu'ambitieuse [88, 84]. Un peu comme Briggs, il calcule juste tous les résultats possibles d'une fonction donnée sur un ensemble discret de points. Cette approche par recherche exhaustive est difficile : si Briggs a calculé la fonction $\log$ sur trente mille points [16], Muller et Lefèvre devraient l'évaluer sur au moins 2^{53}, c'est-à-dire neuf milles de milliards de points [84]. Pour chaque point, ils devraient tester comme Ziv si l'arrondi est difficile ou non. Ceci est encore trop lent. Ils se servent donc

d'une méthode particulière qui leur permet d'éliminer des plages entières de flottants, sur lesquelles ils montrent qu'aucun cas difficile à arrondi ne peut exister. Puis, ils calculent la fonction sur les arguments restant, dont le nombre reste tout de même grand et exponentiel en la précision considéré. Bref, le procédé leur permet de trouver finalement le pire des cas dans lequel se peut trouver une implantation utilisant le procédé de Ziv [84].

Entre 2000 et 2003, Defour développe la première implantation d'une fonction correctement arrondie qui tire profit de ces pires cas calculés par Lefèvre [31]. Cette fonction, l'exponentielle, est intégrée dans la bibliothèque CRLibm qui vient de naître. Elle est donc la première implantation qui puisse garantir non seulement l'arrondi correct mais aussi un temps d'évaluation maximal dans le pire cas. En principe, la voie maintenant libre pour répandre les implantations de fonctions mathématiques correctement arrondies. Leur normalisation également devient possible et en pratique, normalisation veut souvent dire démocratisation.

Les enjeux de cette thèse

Cette thèse, commencée en 2005, est également intitulée « l'arrondi correct des fonctions mathématiques ». En pensant aux travaux de précédents, on pourrait croire à un non-sens. Mais c'est justement en regardant de près ces travaux que l'on se rend compte que le problème de l'arrondi n'a été traité que dans un cadre bien précis. Defour en a démontré la faisabilité à l'exemple d'une seule fonction univariée. Son implantation est correcte mais peut être largement optimisée.

La préoccupation principale dans cette thèse est alors d'élargir l'espace de recherche pour les implantations correctement arrondies de fonctions mathématiques. Pour ce faire, on considérera d'abord les questions suivantes :

- Diverses fonctions mathématiques sont habituellement utilisées et implantées. Dans le cadre de la virgule flottante, toutes les techniques d'implantations se comporteront-elles identiquement en termes de performance ? À titre d'exemple, on considérera la fonction $\log$ pour laquelle plusieurs techniques de réduction d'argument ont dû être envisagées en CRLibm avant qu'un code optimisé soit trouvé.
- Que faire de familles de fonctions, paramétrées par un entier n par exemple ? On s'intéressera alors à titre d'exemple aux fonctions x^n. On y atteindra une amélioration nette en temps de calcul par rapport à des approches précédentes.
- Quand on s'intéresse alors aux fonctions bivariées, comme par exemple x^y, une explosion combinatoire rend les recherches exhaustives de pires cas intraitables. Devant ce constat d'échec, quelles sont les propriétés exploitables de la fonction x^y particulière mais importante ? On se rendra compte que c'est en fait toujours les recherches de pires cas, exécutées sur un domaine très restreint, qui en améliorent les performances.
- Dans le pire cas, arrondir correctement demande d'évaluer une fonction avec une précision conséquente. La question qui se pose alors est : Quelle est l'arithmétique la mieux appropriée pour fournir cette précision rapidement sur les machines actuelles ? L'étude montrera en particulier qu'une représentation des valeurs intermédiaires manipulées dans une implantation de fonction correctement arrondie sous la forme d'une somme non évaluée de trois flottants (un triple-double) est bien adapté aux besoins de précision et aux caractéristiques des machines actuelles.

Ces questions se posent naturellement quand la maturité d'un domaine permet de dépasser la spécialisation initiale. Les objectifs de cette thèse vont plus loin : C'est avec une lenteur incomparable que des spécialistes ont développé les premières implantations de fonc-

tions mathématiques dans la bibliothèque CRLibm. On compte quelques mois par fonction. Cela n'empêche non seulement la généralisation de l'arrondi correct auprès de compagnies industrielles qui doivent minimiser le temps d'ingénieur passé par fonction. Elle restreint également le domaine de recherche qui peut être parcouru lors de l'optimisation d'une implantation. Cette thèse essayera d'analyser cette problématique et d'y remédier :

- Dans un premier temps, on observera que la lenteur du processus de développement est due à l'effort nécessaire pour certifier une implantation. L'arrondi correct ne vaut rien sans certification formelle. En effet, les cas difficiles à arrondir dont pourrait résulter un mauvais arrondi suite à une erreur d'implantation sont si rares que des tests non-exhaustifs ont peu de chance de les détecter. Les premières preuves de certification étaient faites à la main et ainsi fastidieuses. Elles ne résistaient à aucun changement dans l'implantation et inspiraient peu de confiance [41]. Dans cette thèse, on s'intéressera alors à la question de savoir comment ce processus de preuve peut être mécanisé. L'étude passera d'abord par une compréhension de l'outil de preuve Gappa [41, 96]. L'expérience acquise avec quelques preuves Gappa faites à la main permettra ensuite d'en automatiser la rédaction.
  Cette étude montera également que les approches de preuve suivies dans les travaux précédents ont une faille. Elles utilisaient dans la démonstration une majoration de la norme infini (norme sup) d'une fonction. Le calcul de cette norme était pourtant effectué avec un algorithme non certifié qui, justement, n'en donne pas de majoration mais une minoration. On présentera un nouvel algorithme pour le calcul certifié de normes infinis [24]. Contrairement à d'autres possibilités de majoration de normes infini, l'utilisation de cet algorithme ne demande pas de travail manuel.
- La tâche d'implanter une fonction mathématique consiste en plusieurs étapes. Les plus importantes entre elles sont l'approximation de la fonction par un polynôme et la détermination d'une séquence d'opérations machines qui évaluent ce polynôme. En toute brièveté, comme la précision a un coût, la clé pour la performance est dans l'ajustement de précisions. Dans le cadre donné, des dizaines de précisions attachées à des paramètres d'implantation doivent être minorées et adaptées. On s'intéressera alors à l'automatisation de ce travail extrêmement fastidieux. On proposera des techniques automatisées pour chacun des problèmes principaux, la génération d'une séquence d'évaluation et l'approximation polynomiale [82].

L'algorithme complet d'implantation automatique de fonctions permet d'évaluer le temps effectif d'évaluation d'une fonction dans un vaste domaine de recherche. Par exemple, il sera possible de générer et de certifier formellement une implantation d'une fonction comme e^x pour en mesurer la performance sur quelques milliers de combinaisons largeur de domaine d'implantation et précision d'évaluation. Avec un espace de recherche ainsi considérablement élargi, l'optimisation d'implantation devient aisée. La plupart du travail étant automatisé, il est également facile d'implanter une fonction mathématique avec arrondi correct.

Organisation du document

Ce document suit l'ordre des questions évoquées ci-dessus. Cet ordre naturel est presque chronologique. Le chapitre 1 fixe d'abord le cadre formel des travaux précédents et présentés. Il se constitue ensuite de la suite des trois études pour $\log$, x^n et x^y. Le chapitre 3 s'intéresse à l'arithmétique multi-double utilisée dans des implantation rapides des fonctions mathématiques correctement arrondies. Les études sur l'outil de certification Gappa et sur

le calcul de normes infini forment le chapitre 4. Le chapitre 5 présente l'outil logiciel Sollya, développé au cours de cette thèse. Cet outil sert à automatiser l'implantation de fonctions mathématiques. Les résultats de cette étude d'automatisation et quelques perspectives sont présentés dans le chapitre 6. Le lecteur trouvera les conclusions de cette thèse ainsi que des perspectives dans le chapitre 7.

CHAPITRE 1

Arrondi correct de fonctions mathématiques

J'aime les calculs faux car ils donnent des résultats plus justes.

Jean Arp, *peintre franco-allemand*

Nous allons voir les bases de l'arrondi correct de fonctions mathématiques. Mais bizarrement, ce qui est important dans l'arrondi correct, ce ne sont pas les propriétés mathématiques des fonctions. C'est l'arithmétique sur machine, avec ses propriétés et arcanes, dans laquelle une implantation d'une fonction mathématique peut être correctement arrondie ou non. Et c'est surtout l'interaction entre des approximations aux fonctions mathématiques et une arithmétique qui régit sur l'arrondi correct.

La première section 1.1 de ce chapitre fixe alors d'abord l'arithmétique flottante sur laquelle seront basé quasiment tous les travaux de cette thèse. La notion de l'arrondi correct est ensuite formellement définie. Puis, l'état de l'art pour le fournir pour des fonction mathématiques transcendantes est résumé. La deuxième section 1.2 présente nos travaux [43] concernant la mise en œuvre de cet état de l'art pour la fonction log.

1.1 Cadre général de l'arrondi correct

La virgule flottante est la représentation en machine des nombres réels la plus répandue. Elle est utilisée dans une multitude d'applications allant du calcul scientifique à la finance ou encore aux jeux vidéo. C'est dans ce cadre que nous allons considérer l'arrondi correct, même si cette notion se définit pour tout représentation discrète.

1.1.1 La virgule flottante normalisée IEEE 754

Historique de la virgule flottante normalisée Le concept de virgule flottante est ancien [109, 101]. On parle d'abord de notation scientifique. Déjà certains des premiers ordinateurs implantent la virgule flottante avec ses quelques opérations de base, comme l'addition et soustraction [4]. Elle est clairement omniprésente dans les machines des années 70 et 80 [109, 101].

En revanche, jusqu'à la publication de la norme internationale IEEE 754 pour l'arithmétique en virgule flottante (binaire) en 1985, il n'y pas *une* virgule flottante : les différentes implantations sur les divers systèmes se comportent différemment [101]. Ces variations sont dues à trois facteurs :

- les représentations en virgule flottante sont paramétrées, comme par exemple par la précision de calcul, et le choix de ces paramètres se faisait différemment sur les systèmes,
- la présence ou l'absence de l'arrondi correct, qui comme on verra, rendait les résultats quasiment indéterministes et ainsi différents d'un système à l'autre, et
- la représentation différente de valeurs et conditions spéciales, pour une division par 0 par exemple.

La norme IEEE 754 de 1985 a éliminé ces variations en standardisant chacun de ces trois points [5] : des formats uniques fixent les paramètres de précision et autres, une compatibilité binaire est spécifiée par l'arrondi correct pour les 5 opérations $+$, $-$, $\times$, $/$ et $\sqrt{}$ définies par le standard, et les valeurs spéciales sont gérées uniformément. La norme publiée en 1985 définit une arithmétique flottante binaire, c'est-à-dire les nombres sont écrits en base 2.

L'implantation de la norme IEEE 754 a d'abord été promue par la compagnie Intel. Elle a publié en 1985 un premier coprocesseur x87 de calcul flottant IEEE 754 [101]. Depuis, tous les autres constructeurs ont également fourni et fournissent des systèmes implantant cette norme. Un support est maintenant disponible quasiment partout.

La norme IEEE 754 publié en 1985 a résolu beaucoup de problèmes du calcul flottant. Tout de même, elle présente certaines lacunes :

- Le support pour une arithmétique en base 2 seule a freiné et quasiment empêché la diffusion de la norme dans le domaine du calcul financier.
- L'avancement de la technique a fait naître de nouveaux opérateurs qui sont déjà implantés sur certains systèmes sans être sanctionnés par la norme ancienne.
- Certaines comportements pour la gestion de cas spéciaux prévus par la norme n'ont quasiment jamais été implantés en réalité. En revanche, certains calculs mélangeant des formats et précisions, réalisés en pratique, ne sont pas prévus.
- Dans l'uniformisation et la portabilité, la norme se restreint à 5 opérations de base et laisse de côté par exemple les fonctions mathématiques.

Pour remédier à ces problématiques, la norme a été révisée à partir de l'année 2000. Il dépasserait le cadre de ce document de décrire tous les changements. Principalement, les révisions ont permis la standardisation d'un support pour

- l'arithmétique décimale (en base 10),
- une nouvelle opération `fma` $x \cdot y + z$, réalisant à la fois une multiplication et une addition avec un seul arrondi,
- une meilleure gestion de cas particuliers et de la précision et
- des fonctions mathématiques correctement arrondies, dont la réalisation pratique a été le but de cette thèse.

La norme IEEE 754-2008 a été votée en juin 2008. Elle est en vigueur comme norme IEEE depuis le 29 août 2008 [67].

Spécifications techniques de la norme IEEE 754-2008 Considérons alors les spécifications techniques de la norme IEEE 754-2008 qui sont importantes pour les travaux présentés ici [67].

Selon la norme, un nombre virgule flottante x fini s'écrit

$$x = s \cdot \beta^E \cdot m.$$

Ici, les composants s, β, E et m portent les noms suivants :
- s est appelé signe du nombre x,
- β est appelée base de la représentation,
- E est nommé exposant du nombre x et
- m est la mantisse ou le signifiant du nombre x.

Pour former un flottant de précision k chiffres avec un exposant[1] de taille w bits, ces valeurs doivent satisfaire les bornes suivantes :

$$\begin{aligned} s &\in \{-1; 1\} \\ \beta &\in \{2; 10\} \\ E &\in \mathbb{Z} \\ -2^{w-1} \leq E &\leq 2^{w-1} - 1 \\ m &\in \frac{\mathbb{Z}}{\beta^{k-1}} \\ 1 \leq m &< \beta \end{aligned}$$

Des représentations spéciales sont prévues pour ± 0, $\pm\infty$ et des données Not-A-Number (NaN) qui ne représentent pas un nombre réel.

La norme révisée IEEE 754-2008 n'ayant été votée qu'en 2008, les travaux présentés ici ne portent que sur $\beta = 2$ prévue par la norme IEEE 754-1985. Dans la suite, nous allons donc supposer de travailler en base binaire avec $\beta = 2$.

Une combinaison d'une précision k et d'une taille w d'exposant est appelée un format de nombres flottants. La norme IEEE 754 en définit plusieurs. Le nom des formats les plus répandus et importants, en nomenclature IEEE 754-1985 et IEEE 754-2008, sont les suivants [5, 67] :

Précision k	Taille de l'exposant w	Nom IEEE 754-1985	Nom IEEE 754-2008
24	8	simple précision	binary32
53	11	double précision	binary64
64	15	précision double étendue	extended-binary64

Dans ce document, les noms de la nomenclature de la norme IEEE754-1985 seront utilisés. Nous noterons $\mathbb{F}_k$ l'ensemble des flottants de précision k sans spécifier la taille w de l'exposant. Les nombres flottants double précision jouent le rôle le plus important de nos travaux ; on notera $\mathbb{D}$ l'ensemble qu'ils forment.

Le résultat mathématique $z = x \bullet y$ d'une opération $\bullet$ sur deux nombres flottants x et y n'est un nombre flottant que dans des cas particuliers. En général, le résultat est un réel. L'arithmétique IEEE 754 prévoit alors des arrondis $\star_k : \mathbb{R} \to \mathbb{F}_k$ projetant les réels sur les flottants. Quatre modes d'arrondi sont standardisés [5].

[1] Pour des raisons de place réduite et de clarté, nous faisons abstraction des détails techniques dans le cas d'un nombre subnormal. Voir [31, 8, 101] pour une présentation plus exhaustive.

En voici la liste :

- L'arrondi au plus proche pair si milieu (*round-to-nearest-ties-to-even*) $\circ_k$ associe au réel $x \in \mathbb{R}$ le flottant $y \in \mathbb{F}_k$ le plus proche de x. Dans le cas d'égalité de distance entre deux flottants, le flottant ayant une mantisse $m \cdot \beta^{k-1}$ paire est choisi.
- L'arrondi vers le haut $\triangle_k$ associe au réel x le flottant y le plus petit supérieur ou égal à x.
- L'arrondi vers le bas $\triangledown_k$ arrondit au réel x le flottant y le plus grand inférieur ou égal à x.
- L'arrondi vers zéro $\bowtie_k$ associe au réel x positif le flottant $\triangledown_k(x)$ et au réel x négatif le flottant $\triangle_k\,(x)$.

Remarquons que la norme IEEE754-1985 révisée prévoit un cinquième mode d'arrondi qui est l'opposé de l'arrondi vers zéro.

Avec les arrondis $\star_k : \mathbb{R} \to \mathbb{F}_k$, il devient alors possible de définir les opérations correctement arrondies spécifiées par le standard IEEE 754 :

Définition 1 (Arrondi correct) *Soit $f : \mathbb{R}^n \to \mathbb{R}$ une fonction n-aire sur les réels. Soit $\star_k : \mathbb{R} \to \mathbb{F}_k$, $\star_k \in \{\circ_k, \triangledown_k, \triangle_k, \bowtie_k\}$ un arrondi. Soit $F : \mathbb{F}_k^n \to \mathbb{F}_k$ une opération arithmétique flottante implantant f.*

Alors l'opération F se dira correctement arrondie si et seulement si

$$\forall \overrightarrow{x} \in \mathbb{F}_k^n,\ F\left(\overrightarrow{x}\right) = \star_k\left(f\left(\overrightarrow{x}\right)\right),$$

c'est-à-dire pour tous ses arguments $\overrightarrow{x}$, F renvoie un résultat flottant comme si la fonction f avait été évalué avec une précision infinie et que ce résultat avait été arrondi selon $\star_k$.

Sur cette base, la norme IEEE 754-2008 spécifie six opérations arithmétiques correctement arrondies [67]. Elles doivent être obligatoirement implantées par un système compatible avec la norme. Le tableau suivant les résume :

Opération	Notation	Fonction implantée
Addition	$\oplus : \mathbb{F}_k^2 \to \mathbb{F}_k$	$(x, y) \mapsto x + y$
Soustraction	$\ominus : \mathbb{F}_k^2 \to \mathbb{F}_k$	$(x, y) \mapsto x - y$
Multiplication	$\otimes : \mathbb{F}_k^2 \to \mathbb{F}_k$	$(x, y) \mapsto x \cdot y$
Division	$\oslash : \mathbb{F}_k^2 \to \mathbb{F}_k$	$(x, y) \mapsto x/y$
Racine carrée	`sqrt` $: \mathbb{F}_k \to \mathbb{F}_k$	$x \mapsto \sqrt{x}$
FMA	`fma` $: \mathbb{F}_k^3 \to \mathbb{F}_k$	$(x, y, z) \mapsto x \cdot y + z$

L'arrondi correct des opérations de base La spécification de l'arrondi correct pour les opérations de base dans la norme IEEE 754 et l'adoption de cette norme ont eu plusieurs avantages :

- Par l'arrondi correct, les opérations deviennent déterministes. En effet, la fonction f implantée et l'arrondi $\star_k$ ont des comportement complètement déterministes. L'opération F, qu'elle soit implantée en matériel ou en logiciel, par le constructeur A ou B, produira toujours le même résultat, si elle est correctement arrondie. Des programmes numériques deviennent ainsi portables [31, 101].

– Par la définition de l'arrondi correct comme la composée de deux fonctions mathématiquement faciles à décrire, des preuves du comportement numérique deviennent possible. En particulier, elles peuvent être précises au niveau du bit au lieu d'être de grossières majorations d'erreur d'arrondi [8, 96, 101]. La manipulation formelle de techniques de représentations de nombres à précision doublée par des sommes non-évaluées de flottants se simplifie également [48, 8, 106, 96] (cf. aussi section 3).
– Le support de plusieurs arrondis au lieu du seul arrondi au plus proche est la clef pour une arithmétique d'intervalle efficace. Elle devient précise, donnant des encadrements optimaux, si l'arrondi correct est disponible [58].

L'arrondi correct des cinq opérations de base a été spécifié dans la norme IEEE 754-1985 parce qu'il est relativement simple. En effet, pour ces six opérations simples, le résultat intermédiaire infiniment exact avant arrondi nécessaire pour l'arrondi correct peut être facilement calculé et écrit sur un nombre fini et petit de chiffres. Pour l'addition, la soustraction, la multiplication et l'opération `fma`, cela se voit trivialement. Pour la division et la racine carrée, il suffit de considérer un reste euclidien. En résumé, l'arrondi correct est simple pour des fonctions algébriques de degré relativement petit, typiquement 2 ou 3 [18].

1.1.2 L'arrondi correct de fonctions mathématiques

Cette simplicité[2] de l'arrondi correct des opérations de base n'est plus observée quand il faut le fournir pour des fonctions $f : \mathbb{R}^n \to \mathbb{R}$ transcendantes ou algébriques de haut degré. Nous allons voir pourquoi.

Fonctions algébriques et transcendantes Une fonction $f : \mathbb{R}^n \to \mathbb{R}$ s'appelle algébrique si et seulement s'il existe un polynôme

$$p \in \mathbb{Z}_d[y, x_1, \ldots, x_n]$$

tel que

$$p(f(x_1, \ldots, x_n), x_1, \ldots, x_n) = 0$$

pour tous les arguments $x_1, \ldots, x_n$ où f est définie. Le degré d du polynôme p, défini comme la somme des degrés des monômes, est le degré de la fonction algébrique. Une fonction qui n'est pas algébrique s'appelle transcendante.

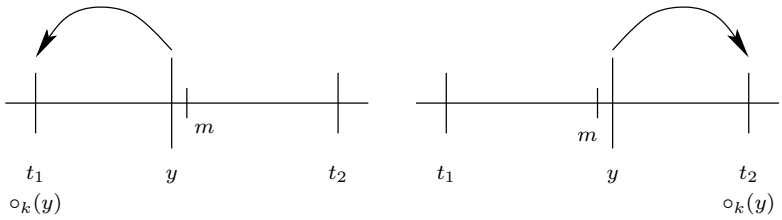
Supposons alors que l'on ait des arguments $x_1, \ldots, x_n$ d'une fonction $f : \mathbb{R}^n \to \mathbb{R}$ et que l'on veuille arrondir $y = f(x_1, \ldots, x_n)$ par exemple au flottant le plus proche[3] dans $\mathbb{F}_k$. Dès lors que l'on a une approximation $\widehat{y} = y \cdot (1 + \varepsilon)$ suffisamment bonne de la fonction, il nous est possible de trouver les deux flottants t_1 et t_2 de $\mathbb{F}_k$ qui encadrent[4] la valeur de la fonction y. Pour arrondir y au plus proche, nous devons alors décider lequel de t_1 et t_2 est plus proche de y. Cela veut dire que l'on doit tester si y est inférieur ou supérieur au milieu $m = \frac{t_1+t_2}{2}$ entre t_1 et t_2. La figure 1.1 illustre cette situation.

Ce test si y est inférieur ou supérieur à m ne peut évidemment pas remplacé directement par un test sur l'approximation $\widehat{y} = y \cdot (1 + \varepsilon)$ de la fonction. Si l'approximation $\widehat{y}$ est plus

[2] Simplicité relative, la certification d'un opérateur de division flottant peut déjà être très technique [93, 26]

[3] Pour les autres modes d'arrondi, la discussion est analogue.

[4] Évidemment, décider cet encadrement est aussi dur que l'arrondi correct lui-même. Il est équivalent de l'arrondi correct dirigé. Un léger sur-encadrement ne nuit pourtant pas à l'argument qui suit.



(a) Arrondi vers le bas (b) Arrondi vers le haut

FIG. 1.1 – Arrondi au plus proche

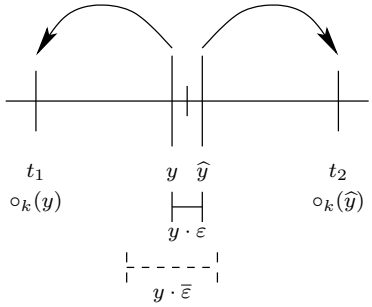


FIG. 1.2 – Arrondi d'une approximation

proche de m que de la valeur y infiniment exacte, c'est-à-dire si $|\widehat{y} - m| \leq |y \cdot \varepsilon|$, l'approximation $\widehat{y}$ et la valeur exacte y peuvent être d'un côté et de l'autre du milieu m. L'approximation $\widehat{y}$ et y ne s'arrondissent pas au même flottant. La figure 1.2 illustre cette situation.

Si la fonction f est algébrique de degré d petit, le problème de test de l'arrondi $y \overset{?}{<} m$ peut être résolu par une évaluation exacte du polynôme $p \in \mathbb{R}_d[y, x_1, \ldots, x_n]$ qui doit exister pour f. En effet, l'évaluation de p sur l'approximation $\widehat{y}$ et les arguments $x_1, \ldots, x_n$ n'implique que des multiplications et additions dont les résultats peuvent être exactement représentées sur machine. Pour d le degré de la fonction et k le nombre des chiffres des flottants $x_1, \ldots, x_n$, cette représentation nécessite typiquement $\mathcal{O}(k \cdot d)$ chiffres. Le signe de la valeur

$$p(\widehat{y}, x_1, \ldots, x_n)$$

décide finalement l'arrondi – les zones de monotonie de p sont connues en général.

De tels tests sont appliqués en pratique. Par exemple Jeannerod et al. [69] décident l'arrondi correct d'une racine carrée $y = \sqrt{x}$ par un test a posteriori sur une approximation $\widehat{y}$. Leur test se lit simplement $\widehat{y} \cdot \widehat{y} \overset{?}{<} x$.

En revanche, si la fonction f à arrondir correctement est transcendante, un test a posteriori par évaluation exacte d'un polynôme n'est pas possible : aucun tel polynôme n'existe. De la même façon, la technique ne s'applique pas en pratique quand le degré d de la fonction est grand. Ceci empêche l'évaluation exacte en pratique car des nombres de taille $\mathcal{O}(k \cdot d)$ devraient être représentés en machine.

Ce dernier point sur le calcul de l'arrondi correct de fonctions algébriques est important : dans cette thèse, nous nous intéressons à l'arrondi correct de la fonction x^y. Comme nous allons voir, les méthodes habituelles ne suffisent pas. Cette fonction n'est certainement pas algébrique en toute généralité, mais pour un format flottant avec une précision et un do-

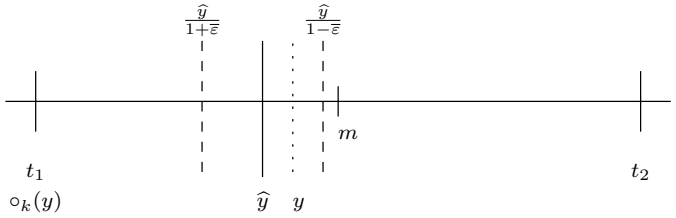


FIG. 1.3 – Arrondi d'une approximation $\widehat{y}$ pour une fonction transcendante $y = f(x)$

maine d'exposant bornés, la fonction x^y définie sur ce format satisfait $p(x^y, x, y) = 0$ pour un polynôme $p \in \mathbb{Z}_d[z, x, y]$. En effet, il y a un exposant $\underline{E}$ minimal du flottant y qu'il suffit de sortir

$$x^y = \sqrt[2^{-\underline{E}}]{x^{y \cdot 2^{-\underline{E}}}}.$$

On obtient une racine d'un rationnel, ce qui est clairement une fonction algébrique. Il s'avère pourtant que le degré d de ce polynôme p est si grand (à peu près $2^{64+53+1} = 2^{118}$ pour la précision double) que l'arrondi correct de la fonction ne peut pas se faire en pratique par l'évaluation de ce polynôme p.

Dans la suite, nous commettrons un léger abus de langage en subsumant, quand le contexte le permet, les fonctions algébriques de grand degré sous les fonctions transcendantes.

Dilemme du fabricant de tables Alors, pour les fonctions f transcendantes, l'arrondi correct de $y = f(x_1, \ldots, x_n)$ doit passer par une approximation $\widehat{y} = y \cdot (1 + \varepsilon)$ si bonne qu'il puisse être assuré que si $\widehat{y}$ est d'un côté de la frontière d'arrondi m, la valeur infiniment exacte y est aussi de ce côté, donc que y et $\widehat{y}$ s'arrondissent de la même façon.

En d'autres mots, il faut exhiber une erreur maximale $\overline{\varepsilon} \in \mathbb{R}^+$ telle que l'erreur ε de l'approximation $\widehat{y}$ par rapport à la valeur exacte y reste bornée par $\overline{\varepsilon}$ et que $\frac{\widehat{y}}{1-\overline{\varepsilon}}$ et $\frac{\widehat{y}}{1+\overline{\varepsilon}}$ s'arrondissent à la même valeur (cf. figure 1.3),

$$\star_k\left(\frac{\widehat{y}}{1-\overline{\varepsilon}}\right) = \star_k\left(\frac{\widehat{y}}{1+\overline{\varepsilon}}\right).$$

En effet, et l'approximation et la valeur exacte de la fonction $y = \frac{\widehat{y}}{1+\varepsilon}$ seront alors encadrées par un même intervalle

$$\widehat{y}, y \in \left[\frac{\widehat{y}}{1+\overline{\varepsilon}}; \frac{\widehat{y}}{1-\overline{\varepsilon}}\right]$$

dont les deux bornes et ainsi toutes les valeurs s'arrondissent pareil. Donc l'arrondi correct de la valeur exacte $y = f(x_1, \ldots, x_n)$ inconnue est égal à $\star_k(\widehat{y})$. La figure 1.3 illustre la situation.

Il est normal de se demander comment il serait possible de se donner une erreur maximale $\overline{\varepsilon}$ pour qu'une approximation $\widehat{y}$ calculée à cette précision, perturbée par $\frac{1}{1+\overline{\varepsilon}}$ et $\frac{1}{1-\overline{\varepsilon}}$, donne deux valeurs s'arrondissant pareil. Après s'être donné une erreur maximale à une certaine valeur, comment peut-on être sûr qu'il ne faudrait pas une erreur maximale plus petite ? Ce phénomène, que l'erreur maximale autorisée sur une approximation permettant de garantir l'arrondi correct est inconnue, s'appelle *le dilemme du fabricant des tables* [71,

84, 101]. Il est la principale difficulté à surmonter pour fournir l'arrondi correct des fonctions mathématiques transcendantes.

Déjà, on peut se demander si, pour une fonction transcendante f donnée, il existe pour tous arguments $\overrightarrow{x} = (x_1, \ldots, x_n)$ une telle erreur maximale $\overline{\varepsilon}$, qui doit être forcément non-nulle car seule une approximation est possible et pas un calcul exact. Posée comme cela, la question a une réponse négative. En effet, il existe des arguments dits *exacts* à des fonctions comme $f = \exp$, où la valeur de la fonction est exactement égale à une frontière d'arrondi m. Prenons par exemple $x = 0$ et $e^0 = 1$ avec un arrondi dirigé, pour lequel $m = 1$ est une frontière.

Pour les fonctions élémentaires simples, c'est-à-dire l'exponentielle, le logarithme népérien et les fonctions trigonométriques et leurs inverses, un théorème par Lindemann montre qu'un seul argument exact est observé par fonction [59]. Dans une implantation, un simple test suffit pour la gestion de cet unique cas particulier. Pour d'autres fonctions comme les logarithmes en base 2 ou 10, quelques arguments exacts sont observés, qui par leur nature particulière sont faciles à filtrer. Pour la fonction x^y, le test est légèrement plus compliqué ; on y consacre le chapitre 2.2.

Après avoir filtré les cas où la valeur de la fonction tombe exactement sur une frontière d'arrondi, on est sûr qu'il existe une erreur maximale $\overline{\varepsilon} \in \mathbb{R}^+$ telle que pour toute approximation $\widehat{y}$ à une erreur ε plus petite que $\overline{\varepsilon}$ s'arrondit correctement à la même valeur que la valeur infiniment exacte y. La raison est simple : l'ensemble des nombres flottants d'un format, comme par exemple $\mathbb{D}$ de la double précision, est fini et la distance entre $y = f(x_1, \ldots, x_n)$ et une frontière m n'est jamais nulle ; il existe donc évidemment une distance minimale donnant $\overline{\varepsilon}$. L'argument $\overrightarrow{x} = (x_1, \ldots, x_n)$ de la fonction f pour lequel cette distance entre y et la frontière d'arrondi m est minimale et $\overline{\varepsilon}$ est atteint, est communément appelé le *pire cas* de la fonction f, le format de flottants et l'arrondi considérés.

Pour autant, étant donné une fonction f et un format de flottants, on ne connaît pas la valeur de $\overline{\varepsilon}$. Il faut la calculer. Actuellement, ce calcul est très cher : il n'existe pas d'autre technique[5] que d'évaluer la fonction sur un grand nombre de ses arguments flottants $\overrightarrow{x}$ possibles et de regarder la distance à la prochaine frontière d'arrondi. Le nombre de flottants dans un format croît exponentiellement avec la taille en bit du format. Typiquement, le format double précision consiste en à peu près 2^{64} valeurs. S'il s'agit pour f d'une fonction k-variée, ce nombre est bien sûr encore mis à la puissance k des combinaisons. Même si certaines plages de flottants peuvent souvent être éliminées par un calcul simple parce que la fonction s'arrondirait toujours pareil ou serait indéfinie, calculer le pire cas $\overline{\varepsilon}$ relève du défi. Allons voir comment l'arrondi correct des fonctions transcendantes peut quand même être réalisé en pratique.

1.1.3 État de l'art pour l'arrondi correct de fonctions mathématiques

Malgré le dilemme du fabricant des tables, qui est que l'approximation d'une fonction avant arrondi correct doit être beaucoup plus précise que le résultat final et que cette précision intermédiaire est inconnue, des techniques ont pu être développées qui permettent de fournir l'arrondi correct avec une bonne performance.

[5] Ce n'est pas tout à fait vrai : la méthode SLZ exclut des domaines de flottants qui ne peuvent pas être le pire cas sans évaluation de la fonction [116]. Elle utilise pourtant des polynômes de Taylor à la place de la fonction sur de très petits domaines ; ceci équivaut à une évaluation.

La première approche, proposée par Ziv [130], se base sur deux observations. Premièrement, après filtrage des cas de frontière d'arrondi, une précision maximale nécessaire, donc une valeur $\overline{\varepsilon} > 0$, existe, même si elle est inconnue. Deuxièmement, étant donné une approximation $\widehat{y} = y \cdot (1 + \varepsilon)$ avec une précision connue $\widetilde{\varepsilon} \in \mathbb{R}^+$, c'est-à-dire telle que $|\varepsilon| \leq \widetilde{\varepsilon}$, il est possible de tester si l'arrondi de $\widehat{y}$ est égal à l'arrondi de y. Il suffit de vérifier si $\frac{\widehat{y}}{1+\widetilde{\varepsilon}}$ et $\frac{\widehat{y}}{1-\widetilde{\varepsilon}}$ s'arrondissent à la même valeur. Si c'est le cas, y, qui est entre ces valeurs, s'y arrondit aussi. Ce test s'appelle un *test d'arrondi* [101, 33]. L'approche de Ziv consiste donc simplement à calculer itérativement des approximations $\widehat{y}_1, \widehat{y}_2, \ldots$ de plus en plus précises à y jusqu'à ce que le test d'arrondi assure l'arrondi correct. Comme la borne d'erreur de pire cas $\overline{\varepsilon}$ existe, ce processus de Ziv s'arrête sous la condition que les cas de frontière aient été filtrés.

Avec cette approche très simple qui est le processus de Ziv, une très bonne performance en moyenne peut être obtenue. En effet, le temps moyen T_{moyen} jusqu'à ce que l'implantation rende un résultat correctement arrondi peut s'estimer par

$$T_{\text{moyen}} = T_1 + \sum_{i=2} p_i \cdot T_i.$$

Ici, T_i est le temps nécessaire pour calculer et tester la i-ième approximation et p_i est la probabilité qu'il faille recourir à au moins i étapes d'approximation [130]. Les cas difficiles à arrondir, pour lesquels la précision doit être augmentée une ou plusieurs fois, sont évidemment très proches de frontières d'arrondi, qui sont soit des nombres flottants, soit des milieux de flottants consécutifs. Ces nombres ont une mantisse de longueur finie ; seuls des zéros peuvent suivre. Les cas difficiles ont donc une mantisse particulière contenant une longue suite de t zéros (ou uns) consécutifs après le bit d'arrondi [101] : par exemple pour un arrondi au plus proche à k bits, on aurait en binaire

$$f(x) = 2^E \cdot \underbrace{1.01101010\ldots01}_{k \text{ bits}} \quad \underbrace{1}_{\text{bit d'arrondi}} \quad \underbrace{000000\ldots000}_{\text{suite de } t \text{ bits}} \quad 10_2.$$

En partant de l'hypothèse[6] que les bits après les k bits du format vers lequel on arrondit dans la mantisse d'une fonction transcendante forment une suite aléatoire [47, 130], il est évident de voir que la probabilité d'une suite de zéros ou uns de longueur t est de 2^{-t}. En conséquence, la probabilité p_i de devoir lancer la i-ième étape du processus de Ziv est également $\mathcal{O}\left(2^{-r \cdot i}\right)$, où r est le nombre de bits significatifs gagnés à chaque étape. Le temps de calcul T_i des différentes étapes croît beaucoup plus lentement que les probabilités p_i ne décroissent [47, 14, 130]. En pratique, on observe que le temps moyen d'une implantation correctement arrondie basée sur l'approche de Ziv est $T_{\text{moyen}} = \tau \cdot T_1$ avec $\tau \leq 1.2$. Sachant que T_1 est typiquement le temps de calcul d'une implantation de fonction non correctement arrondie, l'arrondi correct n'est pas coûteux en moyenne [40].

Le processus de Ziv permet de fournir l'arrondi correct avec une bonne performance en moyenne. Pourtant l'approche n'est pas sans inconvénients non plus. Tant que personne n'est capable de donner une valeur à la précision du pire cas $\overline{\varepsilon}$, que l'itération de Ziv ne dépassera évidemment pas[7], deux problèmes se posent :

– Il est impossible de donner une borne pour le temps de calcul au pire cas. De la même façon, les bits des approximations successives devant s'écrire quelque part, aucune borne pour l'utilisation mémoire ne peut être indiquée. Cela empêche l'utilisation

[6] hypothèse – on l'avoue – légèrement pseudo-mathématique [32, 40], mais en pratique vérifiée

[7] ou juste de quelques bits, parce que l'on ne fait pas de pas d'itération d'un seul bit

d'une telle implantation correctement arrondie dans des environnements de temps réel ou encore embarquées, ayant une mémoire petite et très limitée.
– L'algorithme numérique et son implantation de calcul d'approximations de la fonction de plus en plus précises doivent être conçus pour une précision arbitraire. L'arrondi correct n'a pas de sens sans preuve de correction : il est tout à fait improbable de devoir calculer sur une précision très grande mais tout à fait probable d'avoir un code mal conçu ou mal implanté. Les preuves de correction de code à une précision fixée sont déjà difficiles (cf. chapitre 4). Elles le sont encore plus si elles doivent être génériques pour toute précision. D'ailleurs, pour la même raison, les codes de précision arbitraire sont bien un ordre de grandeur plus lents que des implantations à précision fixée [43].

Tous ces inconvénients liés au processus de Ziv disparaissent si on peut calculer le pire cas des fonctions usuelles.

Le calcul des pire cas pour des formats flottants jusqu'à la précision IEEE 754 simple et des fonctions univariées peut se faire par la méthode naïve consistant à évaluer la fonction donnée sur les au plus 2^{32} arguments possibles et à regarder la distance de sa valeur aux frontières d'arrondi. Pour les formats plus précis, l'utilisation de la méthode naïve implique des temps de calcul beaucoup trop grands à cause de l'explosion combinatoire observée. Ne restant pas sur ce constat d'échec, Muller et Lefèvre puis Stehlé et Zimmermann ont cherché à développer des méthodes particulièrement ciblées pour réaliser une recherche de pire cas au moins pour un arrondi en précision double et ils en ont proposé tout un ensemble [88, 84, 86, 116, 85]. Avec ces techniques, il est possible de calculer le pire cas des fonctions mathématiques univariées usuelles. On peut même déterminer toute une courte liste de cas particulièrement difficiles à arrondir, que l'on appelle, par abus de langage, les *pires cas* de la fonction donnée. La double précision peut être traitée relativement sans problèmes, la précision double-étendue, fournissant non 53 mais 64 bits de mantisse, est atteignable [116, 85]. Au-delà, les méthodes étant toutefois exponentielles, les temps de calcul redeviennent démesurément grands. Remarquons toujours que la fonction doit satisfaire certaines propriétés, en particulier, elle doit bien s'approcher par des polynômes de petit degré sur des plages assez larges de flottants consécutifs. Ceci n'est par exemple pas le cas pour les fonctions $\sin$ et $\cos$ dans les grands arguments pour lesquelles d'autres méthodes sont nécessaires [57].

Effectuées pour la précision double et les fonctions mathématiques les plus usuelles comme $\exp$, 2^x, $\log$, $\log_2$, $\log_{10}$, $\sin$, $\cos$, asin, etc., ces recherches de pires cas montrent qu'une précision intermédiaire correspondant à approximativement 110 à 160 bits est nécessaire pour cet arrondi double précision. Le tableau 1.4 donne un résumé plus précis (bien que non-exhaustif) des résultats publiés dans [86, 32, 33]. Le tableau fait abstraction du domaine de définition des fonctions ou des domaines parcourus par la recherche. Sauf pour les fonction trigonométriques périodiques, il s'agit des domaines intéressants en pratique.

Une précision intermédiaire de 110 à 160 bits garantissant l'arrondi correct de la fonction peut assez facilement être atteinte : il s'agit de 2.1 à 3.0 fois la précision double de base, trouvée sur matériel. Il devient alors possible de borner statiquement le nombre d'itérations nécessaires pour le processus de Ziv. Des garanties de temps maximal d'évaluation et d'utilisation maximale de mémoire peuvent maintenant être données. Comme la précision intermédiaire maximale est connue statiquement, beaucoup d'optimisations peuvent se faire, par exemple sur les polynômes d'approximation de la fonction ou sur l'arithmétique utilisée qui ne doit plus être capable de supporter de précision arbitraire. La situation est donc bien simplifiée.

f	exp	cos	sin	asin	tan	atan	sinh
$\overline{\varepsilon}$	2^{-159}	2^{-143}	2^{-127}	2^{-127}	2^{-133}	2^{-127}	2^{-127}

f	asinh	log	2^x	$\log_2$	acos	cosh	acosh
$\overline{\varepsilon}$	2^{-127}	2^{-119}	2^{-114}	2^{-110}	2^{-117}	2^{-112}	2^{-116}

FIG. 1.4 – Précisions nécessaires au pire cas $\overline{\varepsilon}$ en précision double

En principe, quand les résultats de recherche de pire cas effacent le dilemme du fabricant des tables, il serait possible de ne calculer qu'une approximation – à la bonne précision bien sûr – et d'arrondir toute de suite correctement. En pratique, une telle approche serait pourtant sous-optimale en termes de performance. Les implantations de fonctions mathématiques dans les bibliothèques libm classiques fournissent l'arrondi correct dans bien 99% des cas. Elles réussissent à faire cela en approchant la fonction avec juste 8 bits de garde : comme on a vu, la probabilité que l'on ait alors une mantisse correspondant à un cas difficile – avec 7 zéros ou uns consécutifs – tombe à $2^{-7} \leq 1 - 99\%$. Alors, avec une approche d'arrondi correct en une étape d'approximation, on calculerait toujours avec $110 - 53 = 57$ à $160 - 53 = 107$ bits de garde au lieu de 8 bits pour ne produire un résultat meilleur que dans 1% des cas. Une telle implantation correctement arrondie serait donc bien sous-optimale.

Alors, il s'avère qu'une approche en deux étapes est optimale en pratique sur les systèmes actuels [31, 40]. La fonction est d'abord approchée avec à peu près 8 bits de garde, donc $53 + 8 = 61$ bits pour la double précision. Cette étape d'approximation correspond à ce qui est fait dans une bibliothèque libm classique non correctement arrondie ; elle sert à obtenir une performance en moyenne élevé. On parle donc de *phase rapide*. Un test d'arrondi classique à la Ziv [33] vérifie ensuite si l'arrondi correct peut déjà se faire et lance, si nécessaire, une deuxième étape d'approximation. Le coût du test d'arrondi est négligeable [40]. La deuxième étape fournit une approximation à la précision demandée par le pire cas. Il est alors possible de juste l'arrondir pour obtenir l'arrondi correct de la fonction mathématique à implanter. Comme cette deuxième étape est optimisée moins pour la performance mais plus pour la précision, on parle de *phase précise* [33].

La bibliothèque CRLibm[8] implante des fonctions mathématiques correctement arrondies selon ce principe de deux étapes basées sur les résultats de recherche de pire cas par Muller et Lefèvre [33, 43]. Elle a été fondée par David Defour durant sa thèse [31]. Nous l'avons étendue par 10 implantations de fonctions durant cette thèse, tout en réécrivant certaines des implantations existantes sur des nouvelles bases mieux optimisées.

La prochaine section 1.2 illustre ce travail à l'exemple de la fonction logarithme. Elle montrera également les techniques générales employées pour approcher une fonction mathématique transcendante sur ordinateur.

1.2 Étude d'une fonction univariée : $\log x$

La fonction logarithme a été un candidat excellent pour des essais de différentes implantations dans CRLibm. Avant de donner les différentes techniques essayées à la section 1.2.2

[8] `http://lipforge.ens-lyon.fr/www/crlibm/`

et avant de présenter l'implantation finalement retenue à la section 1.2.3, considérons les techniques générales employées pour les fonction mathématiques usuelles. La section 1.2.4 donnera ensuite des mesures de performance.

1.2.1 Techniques générales pour l'approximation de fonctions

Les processeurs actuels ont une performance accrue sur les opération flottantes d'addition et de multiplication – loin devant celle pour la division par exemple [93, 26]. Les fonction mathématiques $\widetilde{f} : \mathbb{R} \to \mathbb{R}$ sont alors implantées en logiciel (ou – de la même façon – en microcode) suivant l'approche suivante, qui combine une tabulation de valeurs avec une approximation polynomiale [121, 93, 26, 101]. L'approche consiste en quatre étapes élémentaires :

1. L'étape appelé *réduction d'argument* utilise des propriétés algébriques de la fonction $\widetilde{f}$ pour la faire correspondre à une fonction f. L'argument x de cette fonction f appartient – justement par vertu de la réduction d'argument – à un petit domaine dom, le plus souvent centré en 0. Cette étape de réduction peut utiliser des tables avec des valeurs précalculées. Elle peut provoquer une erreur, appelée *erreur de réduction*, quand la relation entre $\widetilde{f}$ et f est basée sur une approximation.

2. Dans le petit domaine dom, la fonction est ensuite approchée par un polynôme $p = c_0 + x \cdot (c_1 + x \dots)$ d'un certain degré d et ayant des coefficients flottants $c_i \in \mathbb{F}$. Cette étape se nomme *approximation polynomiale* et provoque *l'erreur d'approximation* $\varepsilon = \frac{p-f}{f}$. Le calcul d'un polynôme d'approximation peut se faire par plusieurs moyens, comme par exemple des polynômes de Taylor, de Tchebychev ou d'erreur minimale. C'est un point si important que l'on y consacrera toute la section 6.3.

3. Le polynôme p est ensuite implanté en arithmétique flottante, par exemple comme $P = c_0 \oplus x \otimes (c_1 \oplus x \dots)$. La séquence d'instruction pour P est appelée *code d'évaluation polynomiale*. Comme une arithmétique flottante est utilisé, les arrondis dans évaluation provoquent *l'erreur d'arrondi*, notée $E = \frac{p-P}{p}$ [41].

4. Finalement, l'étape dite de *reconstruction* combine les valeurs tabulées lues lors de la réduction d'argument avec la valeur du polynôme évalué afin d'inverser la réduction d'argument et récupérer la valeur de la fonction $\widetilde{f}$. Comme des opérations flottantes sont utilisées, cette étape peut engendrer une erreur, appelée *erreur de reconstruction* [121, 93, 26, 101].

Les erreur des différentes étapes se combinent en une erreur totale, qui correspond à la différence (relative) entre la valeur flottante (intermédiaire) retournée par le code et la valeur exact de $\widetilde{f}$. C'est cette erreur qui doit être inférieure à la précision du pire cas $\overline{\varepsilon}$ pour assurer l'arrondi correct.

Dans le cas d'une implantation correctement arrondie en deux étapes à la CRLibm, les quatre étapes ci-dessus sont principalement mises en œuvre deux fois pour la phase rapide et pour la phase précise. Il est pourtant souvent possible de partager au moins l'étape de la réduction d'argument ainsi que d'utiliser le même schéma d'algorithme global. Le chapitre 3 analysera ce point plus en détail.

1.2.2 La recherche de réductions d'argument pour $\log x$

La conception d'une réduction d'argument performante est souvent la partie la plus difficile dans une implantation de fonction mathématique. Dans cette section, nous illustrons ce processus de recherche à l'exemple de la fonction $\log x$ en base népérienne.

Toute réduction d'argument pour la fonction logarithme utilise le fait que la virgule flottante est un système de numération semi-logarithmique [101]. L'argument x de la fonction peut trivialement[9] être décomposé en un exposant E et un signifiant m, $1 \leq m < 2$:

$$x = 2^E \cdot m.$$

On obtient donc

$$\log x = \log\left(2^E \cdot m\right) = E \cdot \log 2 + \log m.$$

A ce stade, une certaine réduction est déjà obtenue : m ne varie plus qu'entre 1 et 2 ou bien, comme on préfère souvent pour des raisons de cancellation, qu'entre à peu près 0.70 et 1.42 [33, 44, 43].

Quoi qu'il en soit, le domaine $[0.70; 1.42]$ est encore beaucoup trop large pour approcher $\log$ avec un polynôme de degré raisonnable et une précision de 60 voire 120 bits (phase rapide respectivement précise). Une réduction supplémentaire s'impose.

Il est possible de choisir juste un découpage linéaire du domaine $[0.70; 1.42]$. En revanche, la fonction $\log$ définie par exemple sur $[0.70; 0.79]$ (correspondant à un découpage en 8 morceaux par exemple) ne ressemble à aucune des fonctions $\log$ définies sur les autres domaines $[0.70 + i \cdot 0.09; 0.79 + i \cdot 0.09]$. Il est alors nécessaire d'utiliser non *un* polynôme d'approximation mais autant de polynômes qu'il y a de sous-domaines. Le chargement des coefficients des polynômes dans l'évaluation doit se faire alors par un accès mémoire indirect, ce qui nuit à la performance. Comme beaucoup de valeurs pour les coefficients doivent être stockées, seul un découpage en un nombre assez réduit de sous-domaines est possible. Le degré des polynômes reste donc assez grand. L'approche est pourtant celle qui a été choisie dans les premières implantations de CRLibm [44].

La recherche d'une réduction d'argument peut alors mener à l'observation suivante : la fonction $\log$ a un zéro en 1 et elle s'approche le mieux autour de 1. En partant d'une mantisse m vérifiant $1 \leq m < 2$, on observe qu'il faut juste diviser[10] par une approximation c_k de m pour toujours obtenir une valeur autour de 1. Il est possible d'obtenir une telle approximation c_k de m en effectuant une simple troncature de ses bits de mantisse. Formellement, on obtient alors par exemple

$$m = c_k + t = 1 + \frac{k}{64} + t$$

où $k = 1, \ldots, 64$. Alors on se sert de la propriété de base du logarithme que

$$\log(a \cdot b) = \log a + \log b$$

[9] La décomposition marche même dans le cas d'un argument x subnormal. Juste un peu plus de soin est nécessaire dans le code [33, 44, 43].

[10] Il s'agit d'une division mathématique que l'on implantera par une multiplication par une inverse lue dans une table.

et obtient :

$$\begin{aligned}
\log(m) &= \log\left(c_k \cdot \frac{m}{c_k}\right) \\
&= \log c_k + \log\left(\frac{m}{c_k}\right) \\
&= \log c_k + \log\left(\frac{2m}{2c_k}\right) \\
&= \log c_k + \log\left(\frac{m+m}{c_k+c_k}\right) \\
&= \log c_k + \log\left(\frac{m+c_k+m-c_k}{m+c_k-(m-c_k)}\right) \\
&= \log c_k + \log\left(\frac{1+\frac{m-c_k}{m+c_k}}{1-\frac{m-c_k}{m+c_k}}\right) \\
&= \log c_k + \log\left(\frac{1+r}{1-r}\right)
\end{aligned}$$

où $r = \frac{m-c_k}{m+c_k}$ est l'argument réduit de la fonction $f(r) = \log\left(\frac{1+r}{1-r}\right)$. On vérifie facilement qu'il est borné par $|r| \leq 1/64$, ce qui correspond à l'effet de réduction cherché.

Cette réduction d'argument pour le logarithme semble être due à Markstein [93] et reportée par Muller [101]. Elle a plusieurs inconvénients :

- La réduction d'argument relate $\widetilde{f} = \log$ à la fonction différente $f(x) = \log\left(\frac{1-r}{1+r}\right)$. Cela empêche la réutilisation du polynôme d'approximation pour une optimisation qui saute la réduction d'argument pour les arguments x proches de 1, pour lesquels aucune réduction n'est nécessaire.
- Une division flottante est utilisée pour calculer $r = \frac{m-c_k}{m+c_k}$. Sur la plupart des systèmes actuels, la division est une opération très coûteuse, d'autant qu'elle peut provoquer des blocages de pipeline.
- La réduction n'est pas exacte, c'est-à-dire des erreurs d'arrondi entacheront la valeur de r calculée en arithmétique flottante. Dans une implantation à deux phases à la CR-Libm, cela veut dire que la phase précise ne peut pas réutiliser l'argument réduit de la phase rapide mais doit le recalculer plus précisément.

Pour ces raisons, cette réduction d'argument, qui a pourtant l'avantage de n'utiliser aucune mémoire pour tabulation, n'a jamais été implantée en CRLibm.

Nous avons plutôt choisi la réduction suivante – due à Wong et Goto[129] – pour notre réimplantation de la fonction logarithme dans CRLibm [43], que nous allons décrire dans la section 1.2.3 : on observe toujours que la multiplication de la mantisse m par une approximation r de son inverse, $r = \frac{1}{m} \cdot (1+\varepsilon_r)$ donne toujours une valeur autour de 1. Formellement, on a

$$m \cdot r = m \cdot \frac{1}{m} \cdot (1+\varepsilon_r) = 1 + \varepsilon_r.$$

La valeur z, définie par

$$z = m \cdot r - 1$$

vaut donc l'erreur relative de l'inverse r,

$$z = m \cdot r - 1 = 1 + \varepsilon_r - 1 = \varepsilon_r$$

et reste très petite, comme l'erreur relative. En plus, cette réduction d'argument est compatible avec le logarithme :

$$\begin{aligned}\log m &= \log\left(\frac{1}{r}\cdot m\cdot r\right)\\ &= \log\left(m\cdot r\right)-\log r\\ &= \log\left(1+m\cdot r-1\right)-\log r\\ &= \log\left(1+z\right)-\log r.\end{aligned}$$

L'inverse approximative r de x peut être lue dans une table indexée par les premiers bits de poids fort de la mantisse m. Typiquement, avec une table indexée sur k bits, une précision d'à peu près k bit peut être obtenue sur r, ce qui correspond à $|z| = |\varepsilon_r| \leq 2^{-k}$. L'approximation polynomiale de $\log(1+x)$ se fait sans problèmes.

La table pour r contient aussi les valeurs approchées des $-\log(r)$, lues à la bonne précision et par la phase rapide et par la phase précise. Pour que la reconstruction de $\log x$ par

$$\log x = \log\left(2^E\cdot m\right) = E\cdot\log 2-\log r+\log\left(1+z\right)$$

ne provoque pas de problèmes de cancellation catastrophique, un peu de soin est nécessaire pour la décomposition de x en son exposant E et une mantisse m [43].

1.2.3 Les diverses implantations de $\log$ dans CRLibm

Nous avons implanté la fonction logarithme en CRLibm sous deux variantes :

- La première implantation est portable sur tous les systèmes compatible IEEE 754. Elle utilise les arithmétiques en précision double, double-double et triple-double (cf. chapitre 3) pour les calculs intermédiaires.
- La deuxième implantation ne peut être exécutée que sur des systèmes offrant la précision double-étendue, comme les systèmes x86/x87 ou Itanium. Elle utilise les arithmétiques double-étendue et double-double-étendue.

Toutes les variantes implantent l'algorithme basé sur la dernière réduction d'argument mentionnée, due à Wong et Goto.

Les phases rapides donnent typiquement une précision de 60 bits avant d'effectuer le test d'arrondi. Les phase précises approchent la fonction $\log$ à une précision de 120 bits valides. Selon Muller et Lefèvre [87], le pire cas pour la fonction $\log$ contient une suite de 64 zéros ou uns consécutifs, ce qui correspond à $\overline{\varepsilon} = 2^{-119}$.

Les implantations commencent toutes par gérer les cas spéciaux comme $x = \mathrm{NaN}$ ou $x \leq 0$, auquel cas NaN est retourné ou comme $x = +\infty$, qui donne $+\infty$. Comme la fonction logarithme a un unique zéro en 1 et que la grille flottante est large autour de 1, une valeur subnormale ne peut être produite par cette fonction [33, 43]. Les codes peuvent donc faire abstraction de cette problématique. Les nombres flottants x subnormaux en entrée sont traités par un cas à part dans la décomposition $x = 2^E\cdot m$. Ce branchement est même nécessaire dans l'implantation basée sur le format double-étendue, offrant en principe un domaine d'exposant plus large, parce que la décomposition est réalisée par une manipulation des bits de la représentation en mémoire des nombres double précision.

La réduction d'argument $z = m\cdot r-1$ est implantée dans tous les cas de sorte qu'elle soit exacte, c'est-à-dire qu'elle ne provoque pas d'erreur d'arrondi. Ainsi, la réutilisation de l'argument réduit dans la phase précise est possible.

- En arithmétique double-étendue, la valeur pour r tabulée est stocké avec une mantisse sur au plus 11 bits. Ainsi, la multiplication de flottant double m (53 bits) par r (11 bits) devient exacte sur le format double-étendue (64 bits). La soustraction de 1 est exacte par le lemme de Sterbenz [118].
- En arithmétique double, la valeur de l'inverse r est tabulée sur un nombre simple précision complet. Une séquence de multiplication exacte due à Dekker (cf. chapitre 3) est utilisée pour produire une somme non-évaluée de deux doubles $t_{hi} + t_{lo} = m \cdot r$. La soustraction de 1 étant toujours exacte, il est possible d'obtenir un argument réduit exact $z_{hi} + z_{lo} = m \cdot r - 1$ sur une somme non-évaluée de deux doubles. Remarquons qu'il est impossible de provoquer une cancellation (bénigne) suffisamment grande dans la soustraction de 1 pour que l'argument réduit tienne sur un flottant double [33, 43].

Dans tous les cas, l'argument réduit z est borné en valeur absolue par 2^{-8}.

Sur ce domaine, la fonction $\log(1 + x)$ peut être approchée par des polynômes de petit degré.

- Dans l'implantation portable, un polynôme de degré 7 est suffisant pour la phase rapide. La phase précise utilise un polynôme de degré 14. Certains des coefficients des polynômes sont stockées sous la forme d'une somme non-évaluée de flottants double précision, c'est-à-dire sur un double-double.
- L'implantation en double-étendue utilise aussi un degré 7 pour la phase rapide. Tous ses coefficients tiennent sur des flottants double-étendues. Un polynôme également de degré 14 est utilisé pour la phase précise. Pour ce polynôme, certains coefficients doivent être stockés sous la forme de sommes non-évaluées de nombres double- étendue, c'est-à-dire des double-double-étendues.

Remarquons que pour ces implantations de la fonction logarithme dans CRLibm, nous ne disposions pas encore des techniques d'optimisations d'approximations polynomiales à base de l'algorithme LLL [17], que nous décrirons à la section 6.3.3 de ce document.

L'évaluation des polynômes dans les phases rapides ne se fait jamais sous schéma de Horner pur afin de tirer plus de profit du parallélisme d'instruction supporté par les machines super-scalaires actuelles :

- Le code portable commence par une évaluation de Horner en trois étapes, qui ne prennent pas en compte la partie basse z_{lo} de l'argument réduit. Une séquence ad hoc fournit ensuite un résultat sous forme de somme non-évaluée de deux flottants doubles ayant au moins 60 bits significatifs.
- Le code en double-étendue utilise le schéma d'Estrin [101].

Dans les phases précises, où la performance compte moins, l'évaluation se fait selon le schéma de Horner.

La reconstruction de la valeur de $\log$ à partir des valeurs $-\log r$ lues dans les tables, de la valeur du polynôme et de $E \cdot \log 2$ est implantée avec des séquences ad hoc qui essaient de minimiser le nombre d'opérations tout en gardant la précision au niveau qu'il faut pour l'arrondi correct.

Le test d'arrondi utilise la séquence de Ziv dans l'implantation portable et un test ad hoc par manipulation de bits dans l'implantation en double-étendue [33, 43]

Plus de détails sur nos implantations de la fonction logarithme, qui dépasserait le cadre de cette thèse sans apporter d'information pertinente, peuvent être trouvés dans [33] et surtout dans [43].

1.2.4 Résultats expérimentaux

Donnons maintenant les mesures de performances que nous avons pu obtenir avec nos implantations correctement arrondies de la fonction logarithme dans CRLibm.

Performances en temps d'exécution

Nous considérons des nombres flottants double précision positifs distribués de façon aléatoire. Plus précisément, nous prenons des nombres entier aléatoires sur 63 bits que nous coërçons en des flottants double précision.

En moyenne, la phase précise est lancée dans moins de 1% des cas dans toutes les implantations. Ceci assure que le surcoût de la phase précise reste négligeable, surtout quand on considère les coût relatifs des phases rapide et précise (cf. tables ci-dessous).

La table 1.5 donne les mesures de temps des implantations du logarithme en arithmétique double-étendue et triple-double. Elles sont comparées à l'implantation CRLibm présentée dans [44] basée sur la bibliothèque `scslib` ainsi qu'aux implantations correctement arrondies concurrentes dans la bibliothèque `libmcr` de Sun et `libultim` d'IBM. Les mesures pour la `libm` par défaut, non correctement arrondie, sont également indiquées. Les mesure ont été effectuées sur un Pentium 4 Xeon avec une Linux Debian 3.1 et gcc 3.3 comme compilateur[11].

Implantation	moyenne	pire cas
`libmcr` de Sun	1055	831476
`libultim` d'IBM	677	463488
CRLibm, en `scslib`	706	55804
CRLibm, en triple-double	634	5140
CRLibm, en double-étendue	339	4824
`libm` par défaut (sans arrondi correct)	*323*	*8424*

FIG. 1.5 – Comparaison des performance de $\log$ sur Pentium 4

La table 1.6 donne des mesures de temps pour une variété de processeurs, toutes obtenues sur un Linux récent et le compilateur gcc 3.3. La table montre que nos implantation correctement arrondies, de la fonction $\log$ en l'occurrence, ont une performance en moyenne proche des bibliothèques `libm` installées par défaut sur les systèmes. Remarquons toujours que la bibliothèque par défaut sur l'architecture Linux/PowerPC est dérivée de la bibliothèque `libultim` d'IBM, qui fournit l'arrondi correct sans le prouver.

Ces tables montrent aussi le bénéfice tiré d'une phase précise écrite de façon à n'excéder la précision du pire cas que de peu : nous obtenons un temps d'exécution qui est au moins deux ordres de grandeur meilleur que celui des implantations correctement arrondies dans les bibliothèques `libmcr` et `libultim`, qui ne disposent pas d'information sur le pire cas.

[11] La bibliothèque `libmcr` ne compile pas avec les nouvelles versions de gcc.

Opteron (cycles)	moyenne	pire cas
CRLibm en double-étendue	118	862
`libm` par défaut (sans arrondi correct)	*189*	*8050*
Pentium 4 (cycles)	moyenne	pire cas
CRLibm en double-étendue	339	4824
`libm` par défaut (sans arrondi correct)	*323*	*8424*
Pentium 3 (cycles)	moyenne	pire cas
CRLibm en double-étendue	150	891
`libm` par défaut (sans arrondi correct)	*172*	*1286*
Power5 (unités de mesure)	moyenne	pire cas
CRLibm (sans FMA)	50	259
CRLibm (avec FMA)	42	204
`libm` par défaut	52	28881
Itanium 1 (cycles)	moyenne	pire cas
CRLibm en double-étendue avec FMA	73	2150
`libm` par défaut (sans arrondi correct)	*54*	*8077*

FIG. 1.6 – CRLibm comparé aux bibliothèques `libm` par défaut sur divers processeurs

Empreintes de mémoire

Dans toutes nos implantations de la fonction $\log$ en CRLibm, nous utilisons une table à 128 entrées, dont chacune consiste en un nombre flottant simple précision sur 4 octets et soit un nombre triple-double sur 24 octets (cf. chapitre 3) ou un nombre double-double-étendue sur 20 octets. Les tailles des tables sont donc :

– $128 \times (4 + 3 \times 8) = 3584$ octets pour la version portable en précision triple-double et
– $128 \times (4 + 2 \times 10) = 3072$ octets pour la version utilisant la précision double-étendue.

Les valeurs des coefficients des polynômes sont directement compilées dans le code. Remarquons que l'utilisation de mémoire sur les processeurs 64 bit peut effectivement être plus grande à cause de contraintes d'alignement mémoire. Itanium y est un candidat particulièrement dépensier, ce qu'il compense par sa taille importante de caches. Notons que l'utilisation mémoire de ces implantations de logarithme est plus grande que celle de la première en CRLibm présentée dans [44].

Afin d'être honnête, mentionnons que l'implantation de la fonction $\log$ dans les bibliothèques par défaut sur des architectures x86/x87 utilise très peu de mémoire car elle se sert des instructions spéciales d'évaluation de logarithme `fyl2xp1` et `fyl2x`. Seuls 40 octets sont alors nécessaire pour l'implantation de la fonction. En revanche, tout utilisateur de processeur x86 doit payer pour le bout de silicium qu'il faut pour implanter ces instructions. Nos implantations utilisent de la mémoire régulière pas très chère.

CHAPITRE 2

Vers l'arrondi correct des fonctions puissance

Non quia difficilia sunt non audemus, sed quia non audemus difficilia sunt.

Lucius Annæus Seneca, *stoïcien romain*

Au précédent chapitre 1, nous avons vu quelles difficultés présente l'arrondi correct et comment elles peuvent être résolues pour des fonction univariées. Nous avons vu que la situation change pour des fonction bivariées car les recherches de pire cas deviennent très difficiles voire impossibles.

Les fonctions puissance, notamment x^n mais aussi x^y, sont très importantes en calcul financier : en effet, l'intérêt global d'un placement, avec tous les intérêts sur les intérêts, se calcule par une fonction puissance. Elles sont donc si importantes qu'elles ont été intégrées aux opérations recommandées par la norme IEEE754-2008 [67].

Dans le cadre de cette thèse, nous avons essayé de trouver des solutions pour fournir l'arrondi correct de x^n et dans un deuxième temps de x^y. La section 2.1, qui résume nos travaux de [74], s'intéresse à une implantation particulière et rapide de x^n. Ensuite, la section 2.2 présente une technique qui passe à l'étape de la fonction bivariée x^y. Notre contribution principale y est un nouvel algorithme de détection de bornes d'arrondi pour cette fonction [83].

2.1 Une famille de fonctions univariées : x^n

2.1.1 Un contexte particulier pour une implantation

Dans [75], Kornerup, Lefèvre et Muller survolent une technique pour calculer les pires cas en double précision de la famille des fonctions x^n pour des valeurs de n entières et petites, typiquement $3 \leq n \leq 1000$. Ces pire cas permettent donc d'assurer l'arrondi correct pour la double précision pour x un flottant double précision. Ils proposent également deux algorithmes itératifs pour le calcul approximatif de x^n. Avec ces algorithmes, l'arrondi correct de la fonction x^n, $n \in \mathbb{N}$, $3 \leq n \leq 1000$, $x \in \mathbb{D}$, peut donc pratiquement être réalisé.

Dans ce domaine de n, les recherches de pire cas montrent qu'une précision de 116 bits significatifs est nécessaire pour qu'une approximation de x^y garantisse l'arrondi correct. Pour

obtenir une telle précision, Kornerup et al. [75] choisissent alors d'utiliser l'arithmétique double-double-étendue. Cette arithmétique est basée sur le format double-étendue qui est défini, autorisé mais non mandaté par la norme IEEE 754. Ce format fournit des nombres flottant avec un signifiant sur 64 bits et un domaine d'exposant étendu par rapport à la double précision IEEE 754. Actuellement, la précision double-étendue n'est supportée en matériel que sur des processeurs x86 (x87) et Itanium. Avec une somme non-évaluée $x_{hi}+x_{lo}$ de deux nombres double-étendue $x_{hi}, x_{lo} \in \mathbb{F}_{64}$, il devient possible de représenter jusqu'à 129 bits significatifs ; il s'y agit de la double-double-étendue bien connue [100, 36, 9, 81, 106, 92, 75, 74].

Les algorithmes itératifs proposés pour x^n doivent effectuer des multiplications sur des nombres double-double-étendue : ils sont tous basés sur un remplacement de la puissance par une suite de multiplications et/ou mises au carrée. On les détaillera dans la suite. Une brique de base pour la multiplication double-double-étendue consiste en une séquence calculant pour deux nombres flottants double-étendue $a, b \in \mathbb{F}_{64}$ deux flottants $x_{hi}, x_{lo} \in \mathbb{F}_{64}$ tels que $x_{hi} + x_{lo} = a \cdot b$ sans erreur d'arrondi. Deux séquences sont connues qui peuvent servir à cet effet. La première séquence, due à Dekker [36], est portable car elle n'utilise que des opérations d'addition et de multiplication flottantes. Elle nécessite pourtant 17 opérations flottantes. La deuxième séquence utilise l'instruction FMA (ou plutôt FMS) qui calcule $\circ(a \cdot b + c)$ (respectivement $\circ(a \cdot b - c)$) correctement arrondi. La séquence ne consiste qu'en deux opérations flottantes [75] :

$$\begin{aligned} x_{hi} &= \circ_{64}(a \cdot b) \\ x_{lo} &= \circ_{64}(a \cdot b - x_{hi}) \end{aligned}$$

Comme les algorithmes itératifs pour x^n présentés dans [75] sont très gourmands en telles multiplications, Kornerup et al. supposent que l'opération FMA est disponible afin d'éviter de ralentir l'algorithme d'un facteur proche de $17/2$. Actuellement, l'opération FMA n'est supporté en matériel que par les systèmes Itanium d'Intel, les PowerPC d'IBM, les processeurs Cell et éventuellement quelques processeurs graphiques.

Par la nécessité d'avoir un support et de la précision double-étendue et de l'opération FMA, les algorithmes itératifs précédemment proposés pour le calcul de x^n ne sont actuellement exploitables que sur des systèmes Itanium. Or cette architecture Itanium est particulièrement super-scalaire. Elle n'est pas bien adaptée à des algorithmes impliquant des boucles d'un corps très court et un nombre d'itérations trop réduit pour qu'un pipeline logiciel soit efficace [93, 26]. Les algorithmes itératifs proposés pour x^n sont typiquement de tels codes. Une approximation de x^n est calculée soit par des multiplications répétées $x^n = x \cdot x^{n-1}$ soit par la méthode des paysans russes épelant les bits de n, mettant au carré et multipliant pour des bits à un, $x^n = \left(x^2\right)^{\frac{n-1}{2}} \cdot x$. Pour les valeurs de n considérées, les boucles sont donc très courtes, typiquement le longueur $\log_2 733 < 10$. Elles ne contiennent dans leur corps qu'une séquence très courte pour la mise au carré et la multiplication. Dans l'étude que nous avons faite, nous avons donc essayé d'attaquer ce problème particulier, c'est-à-dire de trouver une méthode plus linéaire, impliquant beaucoup moins de branchements et de sauts. Notre méthode n'utilisera aucune boucle.

Un problème de la méthode pour x^n proposée dans [75] est aussi le fait que la précision intermédiaire ne peut pas être adaptée aux besoins actuels durant le calcul. Toutes les opérations se font en double-double-étendue. Dans notre approche, seules quelques opérations sont faites en double-double-étendue en moyenne. Cette meilleure adaptation provoque un

gain en vitesse.

Les méthodes dans [75] ont été révisées pour répondre aux problématiques que l'on vient d'aborder. La publication [74] intègre ces nouvelles adaptations ainsi que notre algorithme tel qu'il sera décrit par la suite.

2.1.2 Un algorithme pour x^n basé sur l'exponentielle et le logarithme

Quand on pense à la complexité binaire asymptotique [14], il peut sembler ridicule de vouloir calculer x^n par la formule

$$x^n = 2^{n \cdot \log_2 x}.$$

Pourtant nous allons montrer dans cette section que sur un processeur super-scalaire actuel avec des pipelines profonds et avec des n suffisamment grands mais pas énormes non plus ($5 \leq n \leq 1000$), la situation est différente. En plus, c'est justement cette configuration particulière à laquelle on s'intéresse en pratique [75]. Sur l'architecture Itanium fournissant non seulement à la fois la précision double-étendue et l'instruction FMA mais aussi une multitude d'unités virgule flottante exploitables en parallèle, nous obtenons une performance très haute : dans les mesures de temps reportées à la section 2.1.3, on observe que le temps d'évaluation en moyenne d'une puissance x^n arrondie correctement est équivalent à celui de 21 multiplications séquentielles sur Itanium 2.

Grandes lignes de l'algorithme pour x^n

Comme dans les autres implantations de fonction mathématiques correctement arrondies, nous choisissons une approche en deux étapes d'approximation pour x^n. Nous combinons ces étapes avec l'approche basée sur un calcul de logarithme et d'exponentielle d'une façon qui permet de profiter à la fois de la rareté des cas difficiles à arrondir, de la précision double-étendue et d'une réutilisation des valeurs déjà calculés à la phase rapide pour la phase précise. Comme nous ne disposons[1] d'une information de pire case que pour les valeurs $3 \leq n \leq 733$, nous ciblons notre implantation à ce domaine. Pour ce domaine, nous garantissons donc un arrondi correct de la fonction x^n.

Le schéma d'évaluation proposé est illustré à la figure 2.1. La fonction $2^{n \cdot \log_2(x)}$ est d'abord approché à une précision relative de $2^{-59.17}$. Ces 6.17 bits supplémentaires par rapport à la précision double, vers laquelle l'arrondi final se fait, impliquent que la probabilité de passer à la phase précise est d'à peu près $2 \cdot 2^{-6.17} \approx 2.8\%$. Cette phase précise assure l'arrondi en fournissant une précision relative de 2^{-116}. Le pire cas pour x^n dans le domaine considéré est x^{458} avec x s'écrivant en binaire

$$x = 1.00001111001110001100111110101010110010110111000110 10_2$$

En effet, on a pour cette valeur

$$x^{458} = \underbrace{1.00011111000010110000100001110110101110100000000100101}_{53 \text{ bits}} 1$$
$$\underbrace{00000000 \cdots 00000000}_{61 \text{ zéros}} 1110 \cdots_2 \times 2^{38}$$

[1] À l'heure où nous écrivons ces lignes, le domaine a grandi jusqu'à $n = 1039$ et ne cesse de croître. Une valeur consécutive de n est accomplie à peu près toutes les 5 heures maintenant.

Avec ses 116 bits significatifs dans l'approximation de la phase précise, l'arrondi se fera alors en effet correctement : $116 = 53 + 1 + 61 + 1$.

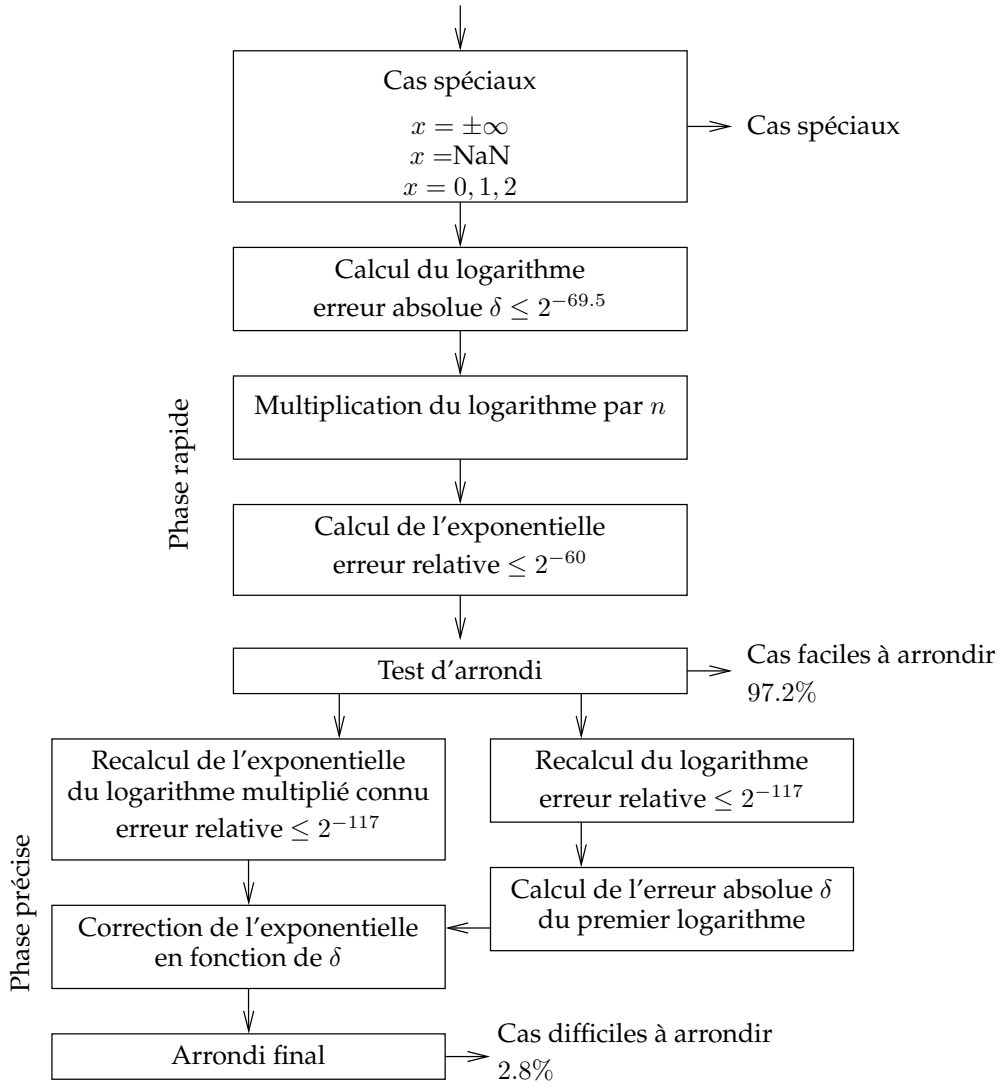


FIG. 2.1 – Approche en deux étapes basée sur des logarithmes et des exponentielles

Nous avons cherché à adapter la deuxième étape au matériel super-scalaire donné, qui offre également un grand nombre de registres. Comme l'approximation du logarithme $\ell = \log_2 x + \delta$, calculé à la phase rapide, est déjà disponible, il est possible de réaliser le calcul des logarithme et exponentielle précises en parallèle. En effet, en considérant

$$x^n = 2^{n \cdot \log_2 x} = 2^{n \cdot \ell} \cdot 2^{n \cdot (\log_2 x - \ell)},$$

on voit que l'on peut paralléliser le calcul approximatif de $2^{n \cdot \ell}$ et de $\ell' = \log_2 x$. La correction nécessaire ensuite, c'est-à-dire la multiplication de l'exponentielle par $2^{n \cdot (\log_2 x - \ell)} = 2^{n \cdot (\ell' - \ell)} = 2^{n \cdot \delta}$ peut être remplacé par son développement au premier ordre : $2^{n \cdot \delta} \approx 1 + c \cdot n \cdot \delta$.

Détails de l'implantation et bornes d'erreur

Les deux sous-algorithmes de calcul de logarithme et d'exponentielle suivent les principes basiques d'approximation par tables et polynômes. Ils sont des variantes des techniques présentées dans [129, 26, 80, 43]. Nous utilisons 8 kilo-octets de tables dans notre mise en œuvre. Les polynômes d'approximations ont des coefficients flottants optimisés [17] (cf. aussi section 6.3.3).

Logarithme Dans les deux phases rapide et précise, le logarithme $\log_2 x$ est encore une fois basé sur la réduction d'argument finalement retenue à la section 1.2.2 :

$$\begin{aligned}\log_2 x &= \log_2\left(2^E \cdot m\right)\\ &= E + \log_2\left(m \cdot r\right) - \log_2 r\\ &= E + \log_2\left(1 + (m \cdot r - 1)\right) - \log_2 r\\ &= E + \log_2\left(1 + z\right) + \log_2 r\\ &= E + p(z) + logtblr[m] + \delta\end{aligned}$$

Ici, comme l'implantation se fait pour l'architecture Itanium, la décomposition de x en E et m peut se faire à l'aide des instructions Itanium `getf` et `fmerge`. Cela économise un grand nombre de cycles.

La valeur r est produite par l'instruction Itanium `frcpa`. Elle donne une approximation de l'inverse de m avec au moins 8.886 bits significatifs [26]. Comme l'instruction est basée sur une petite table indexée par les 8 premiers bits du signifiant de x (sans le 1 de poids fort), il est possible de tabuler les valeurs de $\log_2 r$ dans une petite table indexée par ces 8 premiers bits du signifiant de x.

L'argument réduit z peut ensuite être calculé exactement à l'aide d'une instruction Itanium FMS :

$$z = \circ_{64}\left(m \cdot r - 1\right).$$

En effet, comme on peut facilement vérifier sur ces 256 cas possibles, l'instruction `frcpa` [26] retourne son résultat r sur des nombres flottants ayant 11 bits de mantisse non-nuls au plus. Comme x est un nombre double précision, $x \cdot r$ tient sur $53 + 11 = 64$ bits, donc sur un flottant double-étendue. Il n'y a pas d'arrondi lors de la soustraction $x \cdot r - 1$ grâce au lemme de Sterbenz [118].

Le fait que l'argument z soit exact permet sa réutilisation dans la phase précise de l'algorithme. Il convient de remarquer que la latence jusqu'à obtenir cet argument réduit est petite. Il est produit par seulement 3 opérations dépendantes : un `fmerge`, un `frcpa` et un `fms`.

Les valeurs tabulées $logtbl[m]$ pour $\log_2 r$ sont stockées comme des nombres double-double-étendue $logtblr_{hi}[m] + logtblr_{lo}[m]$. L'erreur absolue des entrées par rapport à la valeur exacte de $\log_2 r$ est bornée par 2^{-130}. Les deux nombres double-étendue d'une entrée sont lus lors de la phase rapide. La phase précise peut donc les réutiliser directement.

La valeur absolue de l'argument réduit z se majore comme suit : on a $r = \frac{1}{m} \cdot (1 + \varepsilon_r)$ avec $|\varepsilon_r| \leq 2^{-8.886}$. Donc on a

$$z = m \cdot r - 1 = \frac{1}{m} \cdot (1 + \varepsilon_r) \cdot m - 1 = \varepsilon_r$$

d'où on déduit que $|z| \leq 2^{-8.886}$.

La fonction $\log_2(1+z)$ est approchée par un polynôme de degré 6 pour la phase rapide et de degré 12 pour la phase précise. Les erreurs d'approximation correspondantes sont bornées par $2^{-69.49}$ respectivement par $2^{-129.5}$. Les polynômes ont des coefficients flottants optimisés. Nous minimisons le nombre de flottants double-étendue et double parce que l'architecture Itanium peut charger des flottants double et simple précision plus rapidement [26, 93].

Ces polynômes d'approximation sont évalués par un code mélangeant le schéma d'Estrin avec le schéma de Horner [26, 101, 43, 111]. Dans la phase rapide, la précision double-étendue est suffisante pour garantir que l'erreur d'arrondi reste inférieure à 2^{-70}. Dans la phase précise, nous utilisons l'arithmétique double-double-étendue bien connue [100, 36, 9, 81, 106, 92, 74]. Cela nous permet de garder l'erreur d'arrondi absolue en-dessous de 2^{-130}.

La reconstruction de la valeur de $\log_2 x$ à partir de la valeur du polynôme approximateur et des valeurs tabulées est réalisée dans les deux étapes en arithmétique double-double-étendue. La valeur de $\log_2 x$ est retourné dans trois registres E, ℓ_{hi} et ℓ_{lo}. Dans la phase rapide nous utilisons une version modifiée de l'algorithme d'addition exacte bien connu **Fast2Sum** [100, 36, 9, 81, 106, 92] (cf. aussi chapitre 3). Cette version assure que ℓ_{hi} s'écrit sur 53 bits au plus, c'est-à-dire sur un flottant double.

Exponentielle La réduction d'argument suivante [26, 80] est utilisée pour le calcul de l'exponentielle 2^t :

$$\begin{aligned} 2^t &= 2^M \cdot 2^{t-M} \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{t-\left(M+i_1 \cdot 2^{-7}+i_2 \cdot 2^{-14}\right)} \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{u-\Delta} \\ &= 2^M \cdot exptbl_1[i_1] \cdot (1 + exptbl_2[i_2]) \cdot q(u) \cdot (1+\varepsilon) \end{aligned}$$

Ici, les valeurs M, i_1 et i_2 sont des entiers. Elles sont calculées à partir de

$$t = n \cdot E + n \cdot \ell_{hi} + n \cdot \ell_{lo}$$

à l'aide de l'instruction FMA, des décalages et l'instruction Itanium `getf` qui retourne le signifiant d'un nombre flottant :

$$\begin{aligned} s &= \circ_{64}\left(n \cdot \ell_{hi} + \left(2^{49}+2^{48}\right)\right) \\ a &= \circ_{64}\left(s - \left(2^{49}+2^{48}\right)\right) = \left\lfloor n \cdot \ell_{hi} \cdot 2^{14} \right\rceil \\ b &= \circ_{64}\left(n \cdot \ell_{hi} - a\right) \\ u &= \circ_{64}\left(n \cdot \ell_{lo} + b\right) \\ k &= \texttt{getf}(s) \\ M &= k \div 2^{14} \\ i_1 &= \left(k \div 2^7\right) \bmod 2^7 \\ i_2 &= k \bmod 2^7 \end{aligned}$$

Dans cette séquence de calculs, toutes les opérations flottantes sauf celles produisant s et u sont exactes grâce au lemme de Sterbenz [118]. L'erreur dont est entaché s est compensée dans la suite des opérations ; en fait, elle est égale à b. L'erreur absolue Δ dont est entaché u

est bornée par 2^{-78} parce que u est majoré en valeur absolue par 2^{-15} et les calculs se font sur des mantisses de 64 bits.

Pour approcher la fonction 2^u avec $u \in \left[-2^{-15}; 2^{-15}\right]$, nous utilisons un polynôme de degré 3 dans la phase rapide et un polynôme de degré 6 dans la phase précise. Ces polynômes garantissent une erreur d'approximation relative inférieure à $2^{-62.08}$, respectivement $2^{-118.5}$.

Les valeurs des tables $exptbl_1[i_1]$ et $exptbl_2[i_2]$ sont toutes stockées sous la forme de nombres double-double-étendue. Seules les parties de poids fort sont lues dans la phase rapide. La phase précise les réutilise et lit en plus les parties de poids faible. De la même façon, la reconstruction se fait en précision double-étendue dans la phase rapide et en précision double-double-étendue dans la phase précise.

La phase rapide retourne son résultat final $2^{n \cdot \log_2 x} \cdot (1 + \varepsilon_1)$ sous la forme de deux nombres flottants r_{hi} et r_{lo}. La valeur r_{hi} est un nombre double précision ; r_{lo} représente donc une estimation de l'erreur d'arrondi lors de l'arrondi de x^n envers la précision double.

Dans la phase précise, l'exponentielle $2^{n \cdot \ell}$ est finalement corrigée par

$$2^{\delta''} = 2^{n \cdot \left(E + i_1 \cdot 2^7 + i_2 \cdot 2^{14} + u\right) - n \cdot (E + \ell')} = 2^{\delta - \Delta},$$

où ℓ' est l'approximation du logarithme calculée par la phase précise. L'étape de correction approche d'abord $\delta'' = n \cdot \left(E + i_1 \cdot 2^7 + i_2 \cdot 2^{14} + u\right) - n \cdot (E + \ell')$ à une précision de 58 bits et utilise ensuite un polynôme linéaire pour approcher la fonction de correction $2^{\delta''}$. Le résultat final est stocké dans un nombre double-double-étendue.

La fonction x^n a quelques arguments pour lesquels sa valeur est égale à un nombre flottant ou le milieu de deux nombres flottants consécutifs. Un exemple est 9^{17}. Pour permettre l'arrondi correct de ces cas, des règles particulières doivent être appliquées, comme par exemple la règle de l'arrondi au flottant de mantisse paire pour l'arrondi au plus proche. La séquence d'arrondi final après la phase précise doit donc séparer deux cas : le cas de l'arrondi inexact d'après les règles de base et le cas de l'arrondi de ces cas exacts et mi-ulps. La séparation est pourtant simple parce que les pires cas de x^n sont connus : si l'approximation de x^n est plus proche à un flottant ou un milieu de flottants consécutifs que la fonction x^n peut jamais être (pire cas), x^n est exactement sur le flottant ou le milieu [83]. La section 2.2 éclaircira ce point technique.

Bornes d'erreurs complètes Une preuve complète et potentiellement formelle des bornes d'erreur pour les deux étapes de notre algorithme pour x^n dépasserait le cadre de ce travail de thèse. Remarquons donc juste que les calculs d'erreurs suivants peuvent être passés assez facilement en l'outil de preuve arithmétiques Gappa [41], que nous présenterons plus en détail à la section 4.3. Les bornes d'erreur d'approximation données ici ont été certifiées par des techniques de calcul certifié de normes infini [24], qui seront décrites à la section 4.2.

Dans la suite, nous donnons donc une analyse d'erreur classique pour ce code numérique de calcul de fonction mathématique qu'est notre implantation de x^n. Nous nous concentrons sur la phase rapide de l'algorithme. L'analyse de la phase précise suit le même schéma et elle est plutôt plus facile à analyser car moins optimisé pour une latence minimisée sur machine.

Nous allons utiliser les notations suivantes :

- $E + \ell = E + \ell_{hi} + \ell_{lo}$ représente l'approximation du logarithme $\log_2 x$,
- δ est l'erreur totale associée,
- δ_{table}, δ_{approx}, $\delta_{\text{éval}}$ et δ_{reconstr} sont les erreurs absolues engendrées par la tabulation, l'approximation et la reconstruction du logarithme.

- $r_{hi} + r_{lo}$ dénotent l'approximation de $x^n = 2^{n \cdot \log_2 x}$,
- $\varepsilon_{\text{phaserapide}}$ est l'erreur relative associée,
- ε_1 est l'erreur relative totale due à l'approximation seule de l'exponentielle sans prendre en compte l'erreur du logarithme,
- $\varepsilon_{\text{table}}$, $\varepsilon_{\text{approx}}$, $\varepsilon_{\text{éval}}$ et $\varepsilon_{\text{reconstr}}$ sont les erreurs relatives dont sont entachées les valeurs de table, l'approximation, l'évaluation et la reconstruction de l'exponentielle et, finalement,
- Δ représente l'erreur absolue dont est entaché l'argument réduit u de l'exponentielle.

On peut donc faire l'analyse d'erreur suivante :

$$\begin{aligned} r_{hi} + r_{lo} &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot p(u) \cdot \\ &\quad \cdot (1 + \varepsilon_{\text{reconstr}}) \cdot (1 + \varepsilon_{\text{table}}) \cdot (1 + \varepsilon_{\text{éval}}) \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{u - \Delta} \\ &\quad \cdot 2^{\Delta} \cdot (1 + \varepsilon_{\text{reconstr}}) \cdot (1 + \varepsilon_{\text{table}}) \cdot (1 + \varepsilon_{\text{éval}}) \cdot \left(1 + \varepsilon_{\text{approx}}\right) \\ &= 2^{n \cdot (E + \ell_{hi} + \ell_{lo})} \cdot (1 + \varepsilon_1) \end{aligned}$$

Ici ε_1 est borné par

$$|\varepsilon_1| \leq \varepsilon_{\text{reconstr}} + \varepsilon_{\text{table}} + \varepsilon_{\text{éval}} + \varepsilon_{\text{approx}} + 2 \cdot \Delta + \mathcal{O}\left(\varepsilon^2\right)$$

Avec $|\varepsilon_{\text{reconstr}}| \leq 3 \cdot 2^{-64}$, $|\varepsilon_{\text{table}}| \leq 3 \cdot 2^{-64}$, $|\varepsilon_{\text{éval}}| \leq 4 \cdot 2^{-64}$, $\left|\varepsilon_{\text{approx}}\right| \leq 2^{-62.08}$ et $|\Delta| \leq 2^{-78}$, on en déduit que

$$|\varepsilon_1| \leq 2^{-60.5}.$$

Ensuite, on obtient pour $E + \ell_{hi} + \ell_{lo}$:

$$\begin{aligned} E + \ell_{hi} + \ell_{lo} &= E + logtblr_{hi}[m] + logtblr_{lo}[m] + p(z) + \delta_{\text{éval}} + \delta_{\text{reconstr}} \\ &= E + \log_2(r) + \log_2(1 + z) + \delta_{\text{table}} + \delta_{\text{approx}} + \delta_{\text{éval}} + \delta_{\text{reconstr}} \\ &= \log_2(x) + \delta_{\text{table}} + \delta_{\text{approx}} + \delta_{\text{éval}} + \delta_{\text{reconstr}} \\ &= \log_2(x) + \delta \end{aligned}$$

Comme $\left|\delta_{\text{approx}}\right| \leq 2^{-69.49}$, $|\delta_{\text{éval}}| \leq -\log_2\left(1 - 2^{-8.886}\right) \cdot 3 \cdot 2^{-64} \leq 2^{-70.7}$, $|\delta_{\text{table}}| \leq 2^{-128}$ et $|\delta_{\text{reconstr}}| \leq 2^{-117}$, on a

$$|\delta| \leq 2^{-68.9}.$$

Ces bornes donnent finalement :

$$\begin{aligned} r_{hi} + r_{lo} &= 2^{n \cdot (\ell_{hi} + \ell_{lo})} \cdot (1 + \varepsilon_1) \\ &= 2^{n \cdot \log_2(x)} \cdot 2^{n \cdot \delta} \cdot (1 + \varepsilon_1) \\ &= x^n \cdot \left(1 + \varepsilon_{\text{phaserapide}}\right) \end{aligned}$$

Avec $n \leq 733$, cela implique que

$$\left|\varepsilon_{\text{phaserapide}}\right| \leq 2^{733 \cdot 2^{-68.9}} \cdot \left(1 + 2^{-60.5}\right) - 1 \leq 2^{-59.17}.$$

Pour la phase précise, l'analyse des bornes d'erreur est similaire. On obtient la borne d'erreur relative totale $\left|\varepsilon_{\text{phaseprécise}}\right| \leq 2^{-116}$. Ceci montre en combinaison avec l'information de pires cas que notre implantation donne un arrondi correct en double précision pour n inférieur ou égal à 733.

2.1.3 Mesures de temps expérimentales pour x^n

Comparons maintenant notre implantation de la fonction x^n correctement arrondie aux différentes autres approches, présentées dans [75] et [74]. Remarquons ici que la séquence de multiplication double-double-étendue dans [75] est légèrement différente de celle utilisée dans [74], ce qui justifie de comparer aux deux implantations.

Les mesures de temps ont été effectuées sur un ordinateur a base d'Intel/HP Itanium 2. Les codes ont été compilés avec icc 10.1 et exécutés sur un GNU/Linux 2.6.18-6-mckinley #1 SMP ia64. Les temps sont indiqués en cycles machine ; une multiplication a une latence de 4 cycles. Nous donnons un temps en moyenne, reflétant surtout la phase rapide, et un temps au pire cas. Comme la performance des approches itératives dépend de n, nous indiquons les temps pour des valeurs petites et grandes de n. Nous excluons des entrées provoquant des arrondis subnormaux en résultat car ceux-ci sont gérés sur Itanium par une procédure assistance logicielle ce qui fausserait nos mesures. Remarquons que les implantations basées sur les approches itératives ne garantissent pas une gestion correcte des valeurs spéciales comme les infinités et les NaNs. Elles sont donc légèrement avantagées par rapport à notre implantation à base de logarithmes et d'exponentielles qui la garantit. Remarquons également que notre implantation de la phase précise est préliminaire et demanderait encore de l'optimisation. Cette optimisation a été faite pour notre phase rapide. Le tableau suivant liste les temps d'exécution mesurés :

Cas	Notre approche		1ère approche itérative [75]		2ème approche itérative [74]	
n	moyenne	pire cas	moyenne	pire cas	moyenne	pire cas
$n = 3$	84.2	327	77	77	65	65
$n = 4$	84.2	327	112	112	86	86
$n = 611$	84.2	327	369	369	261	261
$3 \leq n \leq 611$ (moyenne)	84.2	327	309.6	369	221.2	261

Les observations suivantes peuvent être faites sur ces mesures de temps :

- Notre approche nécessite en moyenne 84.2 cycles, ce qui correspond à la latence de 21 multiplications sur Itanium 2. Dans ce délai, 77 cycles (19.25 latences multiplications) sont nécessaires pour la phase rapide.
- Dans le cadre donné pour une implantation, calculer x^n par des logarithmes et des exponentielles est donc plus rapide en moyenne qu'une approche itérative dès que $n \geq 4$.
- Le temps nécessaire au pire cas dans notre approche reste inférieur au temps au pire cas pour l'algorithme présenté dans [75]. Il est légèrement plus grand que le temps de l'algorithme présenté dans [74]. La différence est pourtant plus petite que le temps pour notre phase rapide : $327-261 = 66 \leq 77$. On ne gagnerait donc rien en remplaçant notre phase précise par cet algorithme. Au vu du code, il semble que l'on puisse gagner cette différence par optimisation de notre phase précise.
- Avec n croissant davantage au delà de 611, la performance des algorithmes itératifs décroîtra encore. À part pour des n très grands, celle de notre algorithme à base de logarithmes et exponentielle ne croîtra que très légèrement. Pour une implantation

correctement arrondie de x^n moins expérimentale, on préconise donc notre approche, avec une optimisation très poussée en fonction des pire cas qui restent à être cherchés.

2.1.4 Conclusions sur x^n

Les fonctions x^n, $x^{\frac{1}{n}}$ et $(1+x)^n$ ainsi que bien sûr x^y sont devenues des opérations recommandées dans la norme révisée IEEE 754-2008 [67]. Les recherches de pire cas pour x^n en double précision avancent rapidement ; une fois $n = 1024$ atteint, elles s'accélèreront probablement encore car le domaine de recherche pour x sera alors réduit à moins d'un exposant : pour $n \geq 1025$, on a $(2-\delta)^n \approx 2^n - n \cdot \delta^{n-1} \geq 2^{1024}$ et 2^{1024} est un dépassement de capacité. Elles donnent non seulement les pires cas pour x^n mais aussi ceux de $x^{\frac{1}{n}}$. Il conviendra de s'intéresser sérieusement à une implantation portable (et non seulement Itanium) de x^n et $x^{\frac{1}{n}}$, une fois que l'on jugera que le domaine pour n connu est suffisamment large.

Pour x^y, l'espoir de pouvoir effectuer des recherches de pire cas pour l'arrondi correct est moins grand. Elles sont pourtant toujours utiles, comme le montre la section suivante.

La fonction $(1+x)^n$ est également définie par la norme IEEE754-2008 et par des bibliothèques mathématiques [67]. Nous ne l'avons pas encore étudiée. Son utilisation semble pourtant être très spécifique à quelques domaines.

2.2 Une fonction bivariée : x^y

2.2.1 Introduction à la problématique avec x^y

À la section 1.1.3, nous avons vu l'état de l'art actuel pour l'arrondi des fonctions mathématiques. Le procédé principal a été le processus de Ziv, où la fonction est approchée de plus en plus précisément jusqu'à ce que l'on soit sûr que l'arrondi de cette approximation soit égal à l'arrondi correct de la fonction infiniment exacte. Nous avons vu aussi qu'une recherche de pire cas était possible pour les fonctions mathématiques univariées usuelles [116, 85]. L'information de pire cas étant disponible, le nombre d'itérations dans le processus de Ziv peut être réduit à deux ; on parle des phases rapide et précise.

Afin de pouvoir garantir la terminaison du processus de Ziv, il est nécessaire de connaître à l'avance les cas où l'image $f(x)$ d'une fonction tombe exactement sur ce que l'on appelle une frontière d'arrondi [130, 101, 124]. Une frontière d'arrondi est un point où l'arrondi change de valeur : les valeurs inférieures à la frontière sont arrondies vers le bas et les valeurs supérieures à la frontière sont arrondies vers le haut. Des règles spéciales, comme par exemple l'arrondi au flottant de mantisse paire de la norme IEEE 754, s'appliquent si la valeur tombe exactement sur la frontière. Un test de borne d'arrondi est donc nécessaire même si l'arrondi des autres cas repose sur une information de pire cas.

Pour les fonctions usuelles transcendantes comme par exemple $\exp$, $\sin$, $\log_2$, cette détection est simple : les frontières d'arrondi sont des nombres rationnels tandis que les images de ces fonctions sur des arguments rationnels sont transcendantes sauf pour un nombre très petit d'arguments bien connus [101, 59]. Alors, seules quelques valeurs doivent être filtrées. En l'occurrence, $\exp(x)$ est rationnel seulement si $x = 0$ et $\log_2(x)$ est rationnel seulement pour des puissance entières de 2.

Pour la fonction $\mathrm{pow} : (x, y) \mapsto x^y$ la situation est différente. Premièrement, il s'agit d'une fonction bivariée. Ceci rend la recherche de pire cas pour un arrondi en double précision

infaisable en temps raisonnable. Les techniques actuelles sont confrontées à une explosion combinatoire [116, 85]. L'arrondi correct est donc basé seulement sur le processus itératif de Ziv [124, 98, 130]. Deuxièmement, les images de $\mathrm{pow}(x, y) = x^y$ sont des nombres flottants d'une certaine précision pour (un sous-ensemble des) flottants x et y. Elles tombent donc potentiellement sur des frontières d'arrondi. Pour s'en convaincre, il suffit de considérer par exemple $1296^{0.75} = 216$. L'ensemble des entrées (x, y) telles que x^y est une frontière d'arrondi a une structure complexe. Par exemple, x^y est rationnel si x est rationnel et y un entier ou, ce qui est plus difficile, si x est un carré parfait répété et y l'inverse de la puissance entière de 2 correspondante. Tout de même, tous les rationnels ne sont pas des frontières d'arrondi. La détection des cas de frontière d'arrondi de la fonction pow nécessite donc un algorithme particulier.

Dans cette section, nous nous intéressons à ce problème de détection de cas de frontière d'arrondi pour la fonction pow en arithmétique flottante IEEE 754 binaire. Nous nous réduisons au cas de la précision double mais nous nous occupons de tous les quatre mode d'arrondi : arrondi-au-plus-proche-pair et les trois modes dirigés [5].

Le problème n'est pas nouveau : divers algorithmes de détection ont déjà été proposés dans la bibliothèque MPFR [50, 124], dans la bibliothèque `libmcr` publiée par Sun [98] et dans la bibliothèque `libultim` publiée par Ziv chez IBM et intégrée dans la bibliothèque GNU C (`glibc`)[2] [130]. Dans ces trois approches, l'algorithme de détection effectue des calculs relativement chers au moment de l'exécution du programme. Des tests complexes assurent que tous ces calculs se font sans erreur d'arrondi. Un progrès dans notre approche est que ces calculs sont remplacés par des tests simples utilisant des constantes et des approximations qui sont déjà disponibles dans une implantation de la fonction pow. Typiquement, nous remplaçons des extractions répétées de racines carrées suivies d'un processus de test impliquant une boucle de multiplication et de mise au carré par huit comparaisons avec des constantes précalculées.

Une deuxième nouveauté dans notre travail est la technique du précalcul de ces constantes. Certes, il est impossible de calculer en temps raisonnable des bornes de pire cas pour la fonction pow sur tout son domaine de définition pour la précision double [116, 85]. Toutefois, de tels pire cas peuvent être calculés sur un sous-ensemble des nombres dans ce domaine [75]. Ce sous-ensemble contient principalement les arguments (x, n) des puissances entières x^n d'un flottant double précision x pour de petites valeurs n et les antécédents $(x, 2^{-F} \cdot n)$ de racines 2^F-ièmes d'un nombre double précision x élevées à une puissance entière très petite : $\left(x^{2^{-F}}\right)^n$. Nous observons et prouvons que tous les cas où x^y tombe sur une frontière d'arrondi doivent appartenir à un tel petit sous-ensemble des flottants double précision. En plus, nous montrons que nous pouvons détecter les cas de frontière d'arrondi en utilisant des approximations qui sont plus précises que le pire cas de l'arrondi : si une approximation de x^y est démontrablement deux fois plus proche d'une borne d'arrondi qu'un cas inexact peut jamais être, la vraie valeur de x^y est exactement sur la frontière d'arrondi.

Notre algorithme est donc une nouvelle application des résultats de pire cas mais pas seulement. Il a été conçu pour servir d'une brique de base dans une implantation correctement arrondie de pow. Nous montrons que sur des entrées aléatoires, les cas de frontière d'arrondi sont rares. Ceci est particulièrement vrai pour le mode d'arrondi par défaut, l'arrondi au plus proche. Notre algorithme permet d'obtenir une performance moyenne élevée en évitant tout ralentissement des cas non frontière d'arrondi sur le chemin critique. L'algo-

[2]Disponible sur `http://www.gnu.org/software/libc/`

rithme pioche les cas d'arrondi dans le flots de données essentiellement gratuitement après une deuxième itération dans le processus de Ziv.

2.2.2 Techniques développées pour l'arrondi correct de pow

Notations

Dans toute cette section 2.2, nous utiliserons la formalisation [8] des nombres flottants binaires suivante : nous noterons un flottant $2^E \cdot m$ avec une mantisse m et un exposant E entiers. Nous ne considérons pas les données flottantes comme l'infini ou les NaNs et dénoterons :

$$\mathbb{F}_k = \left\{ 2^E \cdot m \mid E \in \mathbb{Z},\ m \in \mathbb{Z},\ 2^{k-1} \leq |m| \leq 2^k - 1 \right\} \cup \{0\} \ .$$

Comme les nombres flottants double précision IEEE 754 ont un domaine d'exposant borné [5], nous les distinguerons de $\mathbb{F}_{53}$ et dénoterons leur ensemble :

$$\begin{array}{rcl} \mathbb{D} & = & \left\{ 2^E \cdot m \mid E \in \mathbb{Z},\ -1074 \leq E \leq 971,\ 2^{52} \leq |m| \leq 2^{53} - 1 \right\} \\ & \cup & \left\{ 2^{-1074} \cdot m \mid m \in \mathbb{Z},\ |m| \leq 2^{52} - 1 \right\} \ . \end{array}$$

Répétons que les nombres dans $\mathbb{D}$, pour lesquels m varie entre 2^{52} et $2^{53} - 1$ s'appellent des nombres normalisés et que les autres nombres s'appellent subnormalisés. On a $\mathbb{D} \subseteq \mathbb{F}_{53}$ [8].

Pour le test de frontière d'arrondi, nous considérons les quatre mode d'arrondi définies par la norme IEEE 754, en particulier le mode par défaut $\circ$. Nous noterons $\star_k : \mathbb{R} \to \mathbb{F}_k$ l'arrondi quelconque et $\circ_k : \mathbb{R} \to \mathbb{F}_k$ l'arrondi au plus près. Nous utilisons le symbole $\diamond_k : \mathbb{R} \to \mathbb{F}_k$ si une distinction entre deux arrondis est nécessaire.

Les fonctions d'arrondi $\star_k : \mathbb{R} \to \mathbb{F}_k$ sont discontinues. Les points de discontinuité d'une fonctions d'arrondi forment ce que l'on appelle les frontières d'arrondi dans le mode $\star_k$. Pour les modes $\star_k$ arrondi vers le bas, vers le haut et envers zéro, l'ensemble des frontières d'arrondi est égal à l'ensemble des flottants $\mathbb{F}_k$ [86]. Nous appellerons un tel cas de frontière d'arrondi un cas exact. Pour le mode d'arrondi au plus proche $\circ_k$, l'ensemble des frontières d'arrondi est formé par les milieux de nombres flottants consécutif dans $\mathbb{F}_k$ [86]. Nous appellerons un tel cas un cas mi-ulp.

Les milieux de nombres flottants dans $\mathbb{F}_k$ sont des nombres dans $\mathbb{F}_{k+1}$ ayant une mantisse entière m impaire. Comme $\mathbb{F}_k \subset \mathbb{F}_{k+1}$, les bornes de tous les modes d'arrondi $\star_k$ considérés appartiennent à $\mathbb{F}_{k+1}$. Le test en double précision si $\mathrm{pow}(x, y)$ est dans un cas de frontière d'arrondi se réduit donc à calculer le prédicat

$$RB(x, y) = (x^y \in \mathbb{F}_{54}) \ .$$

Le lien entre le dilemme du fabricant des tables et la détection de frontières d'arrondi

Comme on a déjà vu, il est impossible de décider l'arrondi correct d'une fonction, en l'occurrence x^y, avec le processus itératif de Ziv, calculant des approximations de plus en plus précises mais jamais exactes, sans filtrer les cas de frontière d'arrondi. Il est donc évidemment également impossible de se servir juste d'approximations et du processus de Ziv pour filtrer ces cas de frontière d'arrondi. En effet, il s'agit de décider, à base d'approximations de la fonction $f(x, y) = x^y$ si la fonction $g(x, y) = f(x, y) - x_0^{y_0}$ a un zéro en (x_0, y_0). Ceci est impossible à base d'approximations [128].

Ce qu'il est pourtant possible de calculer correctement à partir d'une approximation de x^y suffisamment précise, c'est l'arrondi de x^y au plus proche flottant $\mathbb{F}_{54}$ si l'arrondi à la précision de base, c'est-à-dire dans $\mathbb{F}_{53}$, présente un cas difficile. En effet, si l'arrondi $\star_{53}(x^y)$ est difficile, la mantisse de x^y présente après le 53-ième bit soit un bit à un ou à zéro suivi d'une suite de zéros ou de uns soit une suite de zéros ou de uns. Dans tous les cas, le 55-ième et le 56-ième bit de la mantisse sont égaux. L'arrondi au plus proche à 54 bits ne peut donc pas être un cas difficile. Donc on peut disposer de $\circ_{54}(x^y)$ pour filtrer les cas de frontière d'arrondi pour l'arrondi $\star_{53}(x^y)$. Comme on a trivialement $\circ_{54}(x^y) \in \mathbb{F}_{54}$, le prédicat de test de frontière d'arrondi $RB(x, y)$ devient $RB(x, y) = (x^y = \circ_{54}(x^y))$. La figure 2.2 illustre le phénomène décrit.

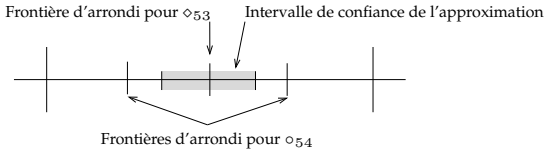


FIG. 2.2 – Utilisation d'une approximation de x^y dans le test de frontière d'arrondi

Le test de frontière d'arrondi peut donc être lancé après une première itération de Ziv. Il peut utiliser une approximation $x^y + \delta$ de x^y (entachée de l'erreur δ) pour calculer $\circ_{54}(x^y + \delta) = \circ_{54}(x^y)$. Cette information peut ensuite être exploitée pour le test si $x^y = \circ_{54}(x^y)$. Puis, une fois qu'une frontière d'arrondi a été détectée, c'est-à-dire si on a $x^y = \circ_{54}(x^y)$, la valeur correctement arrondie $\diamond_{53}(x^y)$ de la fonction x^y peut être déduite de l'approximation par un double arrondi $\diamond_{53}(\circ_{54}(x^y + \delta))$. Bien que le rapport [124] fasse allusion à cette technique, elle ne semble pas être utilisée dans une implantation précédente connue.

Techniques générales pour le test de frontière d'arrondi de pow

L'influence de la double précision Tous les tests de frontière d'arrondi précédemment connus pour la fonction pow : $(x, y) \mapsto x^y$ utilisent quelques propriétés de base des argument $x, y \in \mathbb{F}_{53}$ et de la frontière d'arrondi potentielle $x^y \in \mathbb{F}_{54}$. Une variante de l'algorithme correspondant est esquissée avec une idée de la démonstration dans [124]. Ces propriétés de base impliquent un schéma de branchement. Typiquement, les signes de quelques valeurs et de leurs exposants sont testés. Nous réutilisons ce schéma de branchements préliminaire dans notre algorithme. Puis, il convient de remarquer que les propriétés de base impliquent des bornes sur des valeurs particulières. Dans les approches précédentes, ces bornes sont utilisées afin de garantir la terminaison des algorithmes, qui sont itératifs. Notre algorithme teste explicitement pour ces bornes. Elles déterminent un domaine, dans lequel des constantes sont précalculées pour notre algorithme. Dans la section 2.2.4, nous étendons alors l'esquisse de preuve donnée dans [124], en particulier par la preuve de ces bornes.

Calcul des bornes et leur exploitation Considérons maintenant ces propriétés de base des cas de frontière d'arrondi x^y. Soit z une frontière d'arrondi proche de x^y, c'est-à-dire soit $z = \circ_{54}(x^y)$. Remarquons que les approches précédentes ne calculent pas explicitement z mais supposent juste qu'il existe. Les valeurs x, y et z sont des nombres virgule flottante. Sans nuire à la généralité, nous pouvons supposer que x est positif. Les nombres x, y et z

peuvent donc s'écrire

$$\begin{aligned} x &= 2^E \cdot m \\ y &= 2^F \cdot n \\ z &= 2^G \cdot k \end{aligned}$$

où $E, F, G \in \mathbb{Z}$, $m, k \in 2\mathbb{N}+1$ et $n \in 2\mathbb{Z}+1$.

Le test de frontière d'arrondi, c'est-à-dire la détermination de $RB(x,y) = (x^y = \circ_{54}(x^y))$, se réduit à tester si

$$2^{2^F \cdot E \cdot n} \cdot m^{2^F \cdot n} = 2^G \cdot k\,.$$

Comme m est impair (cf. section 2.2.4), ceci est équivalent aux deux conditions suivantes :

$$m^{2^F \cdot n} = k \tag{2.1}$$

$$2^F \cdot E \cdot n = G\,. \tag{2.2}$$

Distinguons maintenant deux cas en fonction du signe de n. Si n est négatif, $m^{2^F \cdot n} = k$ peut s'écrire

$$\left(m^{-n}\right)^{2^F} = \frac{1}{k}\,.$$

Comme m, n, k et F sont des entiers et $m, -n, k > 0$, cela implique avec la deuxième condition que x doit être une puissance entière de 2 et que $E \cdot y$ doit être un entier G (cf. section 2.2.4 pour plus de détails). Ce test est facile à réaliser sur des flottants IEEE 754 double précision. Inversement, si ces conditions sont remplies, alors x^y est une puissance entière de deux, c'est-à-dire on sait que x^y est un cas exact (ou un dépassement de capacité).

En revanche, si n est positif, la situation est plus compliquée. Il y a deux alternatives en fonction du signe de F. Si F est négatif, $m^{2^F \cdot n} = k$ peut s'écrire

$$\left(\sqrt[2^{-F}]{m}\right)^n = k\,.$$

Comme n est impair et 2^{-F} est pair (cf. section 2.2.4), ceci se réduit à tester s'il existe un entier $j \in \mathbb{N}$ tel que

$$\sqrt[2^{-F}]{m} = j \tag{2.3}$$

$$j^n = k\,. \tag{2.4}$$

Dans les approches précédentes, le test si $\sqrt[2^{-F}]{m} \in \mathbb{N}$ est réalisé par une extraction répétée de racine carrée et un test : comme m est impair, on a

$$\sqrt[2^{-F}]{m} \in \mathbb{N} \Rightarrow \sqrt[2^{-F-1}]{\sqrt{m}} \in \mathbb{N} \Rightarrow \sqrt[2^{-F-1}]{m'} \in \mathbb{N} \wedge m' = \sqrt{m} \in \mathbb{N}\,.$$

La condition $\sqrt{m} \in \mathbb{N}$ est testée principalement en prenant la racine carrée flottante de m et en testant si elle est une opération exacte, c'est-à-dire si elle ne produit pas d'arrondi. La terminaison de cette itération est assurée par une borne sur m : m a au plus 53 bits significatifs. Comme par chaque extraction de racine carrée, le nombre de bits significatifs de m est réduit à la moitié à chaque étape, il ne peut y avoir qu'au plus 5 itérations : $53 < 2^{5+1}$ (cf. section 2.2.4). Tandis que la borne $F \geq -5$ est toujours importante pour notre algorithme, la

boucle d'extractions de racines carrées n'est plus nécessaire. On verra dans la section 2.2.3 pourquoi c'est approprié et possible.

Si F est positif ou nul, $m^{2^F \cdot n} = k$ peut s'écrire

$$m^t = k$$

où $t = 2^F \cdot n = y \in \mathbb{N}$. En conséquence, indépendamment du signe de F, l'algorithme doit tester soit $m^t = k$ ou $j^n = k$ avec $m, n, j, t, k \in \mathbb{N}$. Soit $u^s = k$ le test à réaliser. Comme k est majoré par $2^{54} - 1$, la borne supérieure pour s dépend de u. Dans les approches précédentes, ces bornes sont stockées dans une table ou réalisées par une structure de branchements. Finalement, u^s est calculé par une boucle de multiplications. Dans notre approche, nous utilisons juste une borne plus faible : $s < 35$ parce que $u \geq 3$ (cf. Section 2.2.4).

D'autres implantations correctement arrondies de pow **existantes**

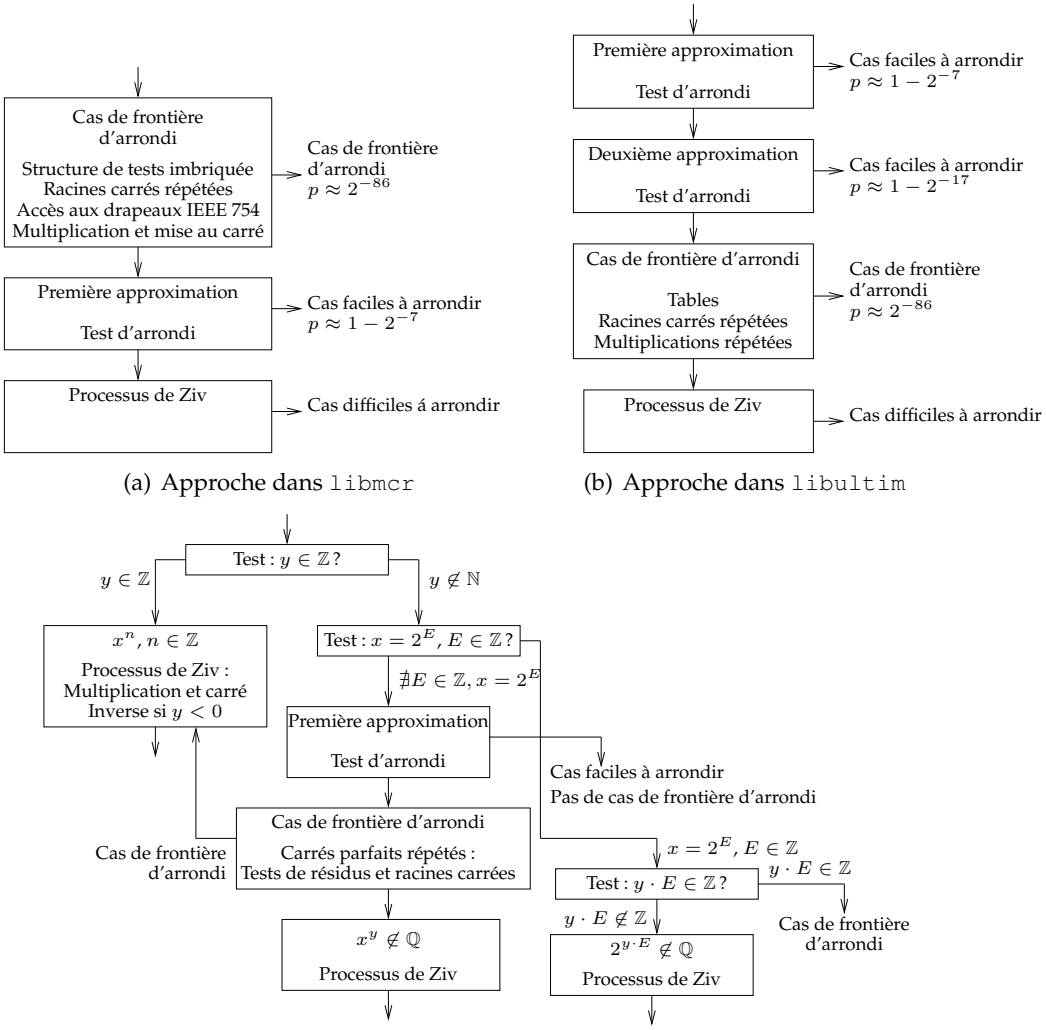
Toutes les implantations correctement arrondies de la fonction pow, dans la bibliothèque `libmcr` de Sun, dans la bibliothèque `libultim` de Ziv et dans la bibliothèque MPFR, utilisent une combinaison de la technique de Ziv et des techniques générales de détection de cas de frontière d'arrondi. Le flot de contrôle dans chacune des dites implantations est illustré par la figure 2.3. Ces illustrations ont été obtenues par une analyse du code source parce qu'une documentation du code n'est que partiellement disponible [130, 98, 50, 124]. La variable p figurant dans les illustrations indique la probabilité que le code prenne ce branchement sur des entrées aléatoires.

L'implantation dans la bibliothèque `libmcr` de Sun utilise l'approche la plus conservative pour le filtre de frontières d'arrondi (cf. figure 2.3(a)). Avant que le processus de Ziv ne soit lancé, tous les cas de frontière d'arrondi sont filtrés. Les arguments de la fonction traversent des branchements imbriqués avant d'être éventuellement testés par une boucle d'extractions de racines carrées. Le test si chacune des racines carrées est exact est réalisé en descendant et testant le drapeau IEEE 754 inexact. Dans le cas où une frontière d'arrondi est détectée, le résultat de pow est calculé par une boucle de multiplication et mise au carré. Les cas mi-ulp s'arrondissent correctement lors de la dernière multiplication de cette boucle [98]. Si des argument ne forment pas un cas de frontière d'arrondi, le processus itératif de Ziv est lancé sans que les résultats intermédiaires produits dans le test de frontière d'arrondi soient réutilisés.

Les branchements et, en particulier, l'accès aux drapeaux IEEE 754 sont des opérations chères sur les processeurs actuels à cause des blocages de pipeline (pipeline-stalls) qu'ils provoquent. Dans l'approche `libmcr`, quelques uns des branchements pour la détection de frontières d'arrondi sont exécutés indépendamment si le cas est une frontière d'arrondi ou non. Même si les cas non frontière d'arrondi moyens ne traversent pas tous les étages de la structure de branchements, le chemin critique devient plus long.

L'approche de Ziv dans la bibliothèque `libultim` essaie d'obtenir une performance accrue sur le chemin critique. La détection de bornes d'arrondi n'est faite qu'après deux itérations du processus d'arrondi correct de Ziv (cf. figure 2.3(b)). Comme des arguments de probabilité [130, 40, 47] montrent, les cas moyens sont alors traités plus rapidement. En conséquence, le test de frontières d'arrondi est exécuté en moyenne pour un nombre réduit d'entrées. Son coût relatif probabiliste décroît donc.

Bien que l'approche dans `libultim` permette d'obtenir une performance en moyenne plus élevée que l'approche dans `libmcr`, elle n'est toujours pas optimale : la détection de



(a) Approche dans `libmcr`

(b) Approche dans `libultim`

(c) Approche dans MPFR

FIG. 2.3 – Approches précédentes pour l'arrondi correct de x^y – cf. Fig. 2.5 pour la nôtre

frontière d'arrondi est retardée jusqu'après la deuxième étape d'approximation dans le processus de Ziv. Par contre, l'approximation n'est utilisée ni pour un rejet plus rapide des cas non frontière d'arrondi ni pour un calcul plus rapide de la valeur des cas de frontière d'arrondi. Bien qu'une approximation de x^y est disponible dont pourrait être déduit $\diamond_{53}(x^y)$ par $\diamond_{53}(\circ_{54}(x^y))$ comme expliqué ci-dessus, l'implantation de Ziv recalcule les valeurs de cas de frontière d'arrondi x^y par des multiplications répétées. Cette approche retarde donc les cas de frontière d'arrondi plus que nécessaire.

L'implantation de pow dans la bibliothèque MPFR modifie l'approche dans `libultim` [50, 124]. Quelques cas spéciaux, comme par exemple $y \in \mathbb{Z}$, $x = 2^E$, $E \in \mathbb{Z}$ et $x = 2^E$, $y \cdot E \in \mathbb{Z}$, sont filtrés avant que le processus de Ziv ne soit lancé. Le test de frontière d'arrondi est simple pour ces cas spéciaux. Pour les cas autres cas de frontière d'arrondi restants, une approche similaire à celle dans `libultim` est utilisée : une première étape de l'itération de Ziv est exécutée. Ceci permet d'obtenir une performance en moyenne élevée pour les cas non frontière d'arrondi. Le test de frontière d'arrondi est ensuite réalisé par un test répété de

carrés parfaits. La bibliothèque MPFR repose ici sur un test implanté dans la bibliothèque GNU Multi Precision (GMP)[3]. Une fois qu'un cas de frontière d'arrondi est détecté, la valeur de x^y est recalculé par la fonction de puissance entière, qui utilise un processus de multiplication et mise au carré.

Dans la comparaison des différentes approches, le point important suivant doit être pris en compte. Les bibliothèques `libmcr` et `libultim` sont ciblées pour la précision double [130, 98]. La bibliothèque MPFR supporte le calcul en précision arbitraire [50, 124]. Les algorithmes pour le calcul en précision arbitraire devraient avoir la meilleure complexité asymptotique connue dans le cas moyen. En revanche, les algorithmes pour une précision fixe donnée, comme la précision double, peuvent être optimisés non pour la complexité asymptotique mais pour leur latence en cycles. Dans notre cadre, nous essayons d'optimiser une implantation de pow pour la précision double. L'implantation dans la bibliothèque MPFR ne peut pas être comparée de façon sensée aux implantations en précision double dans tout aspect. Elle peut pourtant être une source d'inspiration.

2.2.3 Une nouvelle approche pour pow

Nombre de cas de frontière d'arrondi

Dans la continuation des améliorations apportées par les bibliothèques `libultim` et MPFR, nous voulons augmenter la performance en moyenne, et pour les cas non frontière d'arrondi, et pour les cas frontière d'arrondi. Pour l'analyse des cas en moyenne et l'optimisation en fonctions des raretés de cas, des informations sur les probabilités des différents types d'entrées sont nécessaires [130, 40]. Les décomptes de cas donnés dans cette section ont été effectués principalement sur les propriétés des frontières d'arrondi présentées à la section 2.2.2. Les cas ont été compté avec des algorithmes ad hoc rapidement prototypés dont la description dépasserait le cadre ici donné.

En précision double, la fonction $\mathrm{pow} : (x, y) \mapsto x^y$, $x, y \in \mathbb{D}$ a à peu près 2^{112} arguments réguliers. Nous appelons un argument régulier s'il correspond à une entrée (x, y) pour laquelle x^y est un nombre réel qui s'arrondit à la précision double sans qu'il n'y ait ni un dépassement de capacité vers $+\infty$ ou vers les subnormaux ni une perte totale de précision (par un arrondi à 1 en l'occurrence). Typiquement, les arguments irréguliers produisent NaN, $\pm\infty$, 0 ou 1. Ce nombre d'arguments réguliers peut être vérifié en considérant tous les exposants de x pour calculer des bornes pour y pour chacun de ces exposants.

Considérons maintenant le nombre de cas de frontière d'arrondi en les séparant dans les cas mi-ulp et les cas exacts. Quelques cas de frontière d'arrondi sont triviaux. Typiquement, les cas $y = 1$ ou $y = 2$ peuvent être gérés par un filtre ad hoc : $x^1 = x$ et $x^2 = x \cdot x$. Nous ne considérons donc pas ces deux cas triviaux. Le nombre de cas exacts est légèrement plus grand que le nombre de cas mi-ulps : il y a à peu près 2^{27} cas exacts non-triviaux et à peu près 2^{25} cas mi-ulps. Les conditions pour qu'une entrée (x, y) soit un cas de borne de frontière d'arrondi, détaillées à la section 2.2.2, sont légèrement moins fortes pour les cas exacts que pour les cas mi-ulps. Comme l'arrondi au plus proche est le mode d'arrondi par défaut IEEE 754 [5], nous nous concentrons alors sur les cas mi-ulps, qui sont associés à ce mode d'arrondi.

Il y a $37500822 \approx 2^{25.5}$ cas mi-ulps x^y, $y \neq 2$ en précision double. Quand on exclut additionnellement le cas $y = 3$, le nombre tombe à $19066760 \approx 2^{24.2}$ cas. L'exclusion du

[3] disponible sur `http://gmplib.org/`

cas $y = 4$ ne fait baisser le nombre de cas mi-ulps qu'à $18596893 \approx 2^{24.1}$. En revanche, remarquons que la plupart de ces cas, précisément 18431732, de ces cas sont formés par des arguments où $y = \frac{3}{2}$. Malheureusement, le rajout d'un filtre particulier pour $y = \frac{3}{2}$ est difficile, parce qu'il impliquerait l'utilisation d'une extraction d'une racine carrée, ce qui est une opération chère sur les processeurs actuels pipelinés. Il n'y a que 2330 cas mi-ulps x^y, $y \neq 2$, qui s'arrondissent – avec la règle de l'arrondi au flottant de mantisse paire – à des nombres subnormaux $\circ_{53}(x^y)$. Dans tous les cas subnormaux, x et y sont des nombres normalisés et y est différent de $3, 4$ ou $\frac{3}{2}$.

Pour des arguments réguliers flottants uniformément distribués (x, y), la probabilité qu'un argument soit un cas mi-ulp avec $y \neq 2$ est approximativement $p_{\text{mi-ulp}} = \frac{2^{25}}{2^{112}} = 2^{-87}$. Avec des arguments probabilistes [130, 40, 47], cette probabilité peut être mise en relation avec celle qu'un cas non frontière d'arrondi soit un cas si difficile à arrondir qu'une précision grande est nécessaire. Il s'agirait alors d'un cas pour lequel une précision correspondant à $53+1+87 = 141$ bits significatifs est nécessaire dans l'itération de Ziv pour garantir l'arrondi correct. Dans la bibliothèque `libultim` proposée par Ziv, le code de détection de frontières d'arrondi nous semble donc mal placé : invoqué après une approximation qui donne à peu près 80 bits significatifs [40, 130], il s'exécute trop souvent. Remarquons pourtant que cet argument est basé sur une hypothèse d'uniformité des entrées qui peut ne pas être vérifié en pratique.

L'exécution de la détection de cas de frontière d'arrondi après une étape dans l'itération de Ziv tardive a une influence négative sur la performance de réponse pour les cas de frontière d'arrondi. Par exemple, on peut s'imaginer une application qui utilise pow sur un ensemble d'entrées qui sont toutes des cas de frontière d'arrondi. Alors il convient de faire observation suivante : sur des arguments (x, y) uniformément distribués qui sont tous des cas mi-ulps non-triviaux, la probabilité que y soit $y = 3$ ou $y = 4$ est de $p_{\text{facile}} = \frac{37500822-18596893}{37500822} \approx 50.4\%$. Nous proposons donc de filtrer non seulement $y = 2$ mais aussi $y = 3$ et $y = 4$ avant de commencer le processus de Ziv. La détection des cas de frontière d'arrondi peut ensuite être faite après une étape qui donne une précision d'à peu près 120 bits valides. La gestion des cas $y = 2$, $y = 3$ et $y = 4$ peut être faite pratiquement gratuitement sur des processeurs super-scalaires. L'impact de la détection tardive de cas de frontière d'arrondi sur la performance seulement sur les cas de frontière tombe donc à la moitié. Nous allons observer exactement ce comportement par mesure à la section 2.2.5.

L'utilisation de bornes de pire cas pour la détection de cas de frontière d'arrondi

L'impact de la détection de frontières d'arrondi sur la performance totale d'une implantation correctement arrondie de pow peut être diminué par une simplification de l'algorithme de détection. Nous proposons dans la suite une méthode pour éviter le processus complète d'extraction répétée de racines carrées et de multiplication et mise au carré.

Supposons que l'on connaisse le pire cas de la fonction pow en double précision. Montrons alors comment les cas de frontière d'arrondi peuvent être discernés des cas non frontière d'arrondi à l'aide juste d'une approximation de la fonction. Une telle approximation est de toute façon nécessaire pour l'arrondi correct dans le processus de Ziv. Sur toutes les entrées $(x, y) \in \mathbb{D}^2$ double précision régulières, qui ne sont pas des cas de frontière d'arrondi, l'itération de Ziv peut fournir un résultat correctement arrondi dans tout mode d'arrondi après avoir approché $z = x^y$ par $\widehat{z} = x^y \cdot (1 + \varepsilon)$ avec une erreur relative ε inférieure à la borne $\overline{\varepsilon}$, c'est-à-dire $|\varepsilon| \leq \overline{\varepsilon}$, où $\overline{\varepsilon}$ est le pire cas du dilemme du fabricant de tables [101, 85, 116].

Autour des nombres flottants et leurs milieux, il y a donc des écarts dans lesquels aucun résultat x^y, $x, y \in \mathbb{D}$, peut tomber. La figure 2.4 illustre cette observation. Soit $z = x^y$ alors un cas de frontière d'arrondi, par exemple un cas mi-ulp. Soit $\widehat{z}$ une approximation de z avec une erreur relative ε inférieurs à la moitié d'erreur pire cas $\overline{\varepsilon}$, c'est-à-dire soit $\widehat{z} = x^y \cdot (1 + \varepsilon)$, avec $|\varepsilon| \leq \frac{1}{2} \cdot \overline{\varepsilon}$. L'approximation $\widehat{z}$ tombera donc dans cet écart – où aucun cas non frontière d'arrondi ne peut se trouver grâce à la borne de pire cas. Plus précisément, l'intervalle d'approximation Z autour de l'approximation $\widehat{z}$ du résultat $z = x^y$ d'un cas de frontière d'arrondi ne s'intersectera avec aucun intervalle d'approximation correspondant à un cas non frontière d'arrondi.

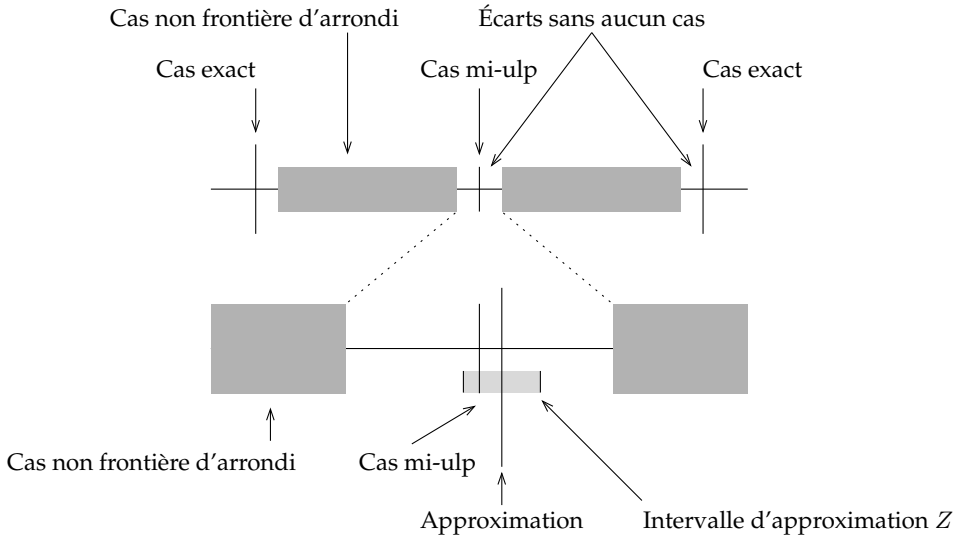


FIG. 2.4 – Utilisation de l'information pire cas pour la détection de frontières d'arrondi

La détection des cas de frontière d'arrondi peut donc être réalisée comme suit : après avoir approché x^y avec une précision légèrement plus grande que le pire cas, l'algorithme teste juste si l'approximation tombe dans l'écart ou non. Ce test est exactement le même que celui pour détecter si une approximation dans l'itération de Ziv permet déjà l'arrondi ou non [130, 33, 40]. Il doit donc être fait de toute façon dans une implantation correctement arrondie de pow basée sur ce processus itératif. Si l'information de pire cas était disponible pour la fonction pow sur tout son domaine d'implantation en précision double, la détection de frontières d'arrondi serait donc essentiellement gratuite.

La recherche de pire cas est actuellement infaisable pour pow en précision double [116, 85]. Mais nous n'avons pas besoin d'une information de pire cas complète pour que notre approche de détection de frontières d'arrondi marche. Nous définissons un sous-ensemble $\mathbb{S}$ des nombres double précision $\mathbb{D}^2$. Ce sous-ensemble correspond aux bornes sur les cas de frontière d'arrondi qui ont été décrites à la section 2.2.2.

$$\begin{aligned} \mathbb{S} &= \left\{(x,y) \in \mathbb{D}^2 \mid y \in \mathbb{N},\ 2 \leq y \leq 35\right\} \\ &\cup \left\{(m, 2^F n) \in \mathbb{D}^2 \mid F \in \mathbb{Z},\ -5 \leq F < 0,\ n \in 2\mathbb{N}+1,\ 3 \leq n \leq 35,\ m \in 2\mathbb{N}+1\right\}. \end{aligned}$$

Comme sera montré à la section 2.2.4, tous les cas de frontière d'arrondi de pow en précision double se trouvent dans $\mathbb{S}$. Le test de frontière d'arrondi peut alors réfuter des cas potentiellement frontière d'arrondi si quelque argument $(x, y) \in \mathbb{D}$ n'appartient pas à $\mathbb{S}$. Puis, il est

possible de calculer les pires cas de x^y pour toutes les entrées $(x, y) \in \mathbb{S}$ [85, 116]. Le domaine est suffisamment petit. Pour la première partie de $\mathbb{S}$, où $y \in \mathbb{N}$, l'information de pire cas, calculée pour l'arrondi correct des puissances entières x^n d'un nombre flottant double précision x peut être réutilisée [75, 74]. La deuxième partie consiste en à peu près $2^{58.4}$ couples de nombres ce qui est un nombre traitable d'entrées ; la section 2.2.3 décrira comment les recherches de pire cas ont été faites pour ce domaine. Pour les entrées de pow appartenant à ce sous-ensemble, $(x, y) \in \mathbb{S}$, la détection de cas de frontière d'arrondi peut donc se baser sur de l'information de pire cas selon l'approche décrite.

Le pire cas suivant pour l'arrondi correct en double précision a été trouvé dans le sous-ensemble $\mathbb{S}$: pour $x = 1988580363009869$, $y = 2^{-4} \cdot 5$, x^y s'écrit en binaire

$$x^y = 1110101111001110.0101001100001100010111001011000110001\ 1\ \underbrace{0\ldots0}_{60\text{ zéros}}111\ldots.$$

La précision du pire cas $\overline{\varepsilon}$ est donc $\overline{\varepsilon} = \left|\frac{\circ_{54}(x^y)-x^y}{x^y}\right| \geq 2^{-114}$.

Considérons maintenant la recherche de ces pires cas.

La recherche des pires cas pour $m^\alpha = m^{2^F \cdot n}$

La recherche des pires cas de la fonction $m^{2^F \cdot n}$, dont l'information est nécessaire pour notre approche s'est faite en utilisant les algorithmes de Muller-Lefèvre connus [86, 116, 85]. Elle est documenté dans [83]. Le coût de la recherche en termes de temps de calcul s'élève à 25 jours sur un petit ensemble de machines en réseau au Laboratoire de l'Informatique du Parallélisme.

L'algorithme de détection de frontières d'arrondi

Notre algorithme **détecteCasDeFrontière** combine tous les éléments décrits pour calculer le prédicat $RB(x, y) = (x^y \in \mathbb{F}_{54})$. Il est illustré par le listing Algorithme 1.

L'algorithme prend en entrée x, y et une approximation $\widehat{z} = x^y \cdot (1 + \varepsilon)$. La précision ε de cette approximation doit être légèrement plus grande que le pire cas de la fonction pow dans le sous-ensemble $\mathbb{S}$. Typiquement, nous choisissons $|\varepsilon| \leq 2^{-117}$. L'algorithme commence par arrondir $\widehat{z}$ au nombre flottant le plus proche dans $\mathbb{F}_{54}$, $2^G \cdot k = \circ_{54}(\widehat{z})$. Comme expliqué à la section 2.2.2, cet arrondi n'est pas assujetti au dilemme du fabricant des tables si l'arrondi $\star_{53}(\widehat{z})$ vers $\mathbb{D}$ l'est. L'algorithme effectue ensuite un test d'arrondi, c'est-à-dire il teste si $\widehat{z}$ est proche de ou sur une frontière d'arrondi. Si la condition de ce test, $\left|2^G \cdot k - \widehat{z}\right| \geq 2^{-116}$, est vérifiée, alors x^y est loin d'une frontière d'arrondi. Alors, il ne peut pas être un cas de frontière ; l'algorithme retourne *faux*. L'arrondi $\star_{53}(\widehat{z})$ de l'approximation $\widehat{z}$ donne alors déjà l'arrondi correct $\star_{53}(x^y)$. Ici, la valeur 2^{-116} est la constante précalculée qui a été la plus coûteuse à obtenir, à savoir par recherche de pire cas sur $\mathbb{S}$.

Après ce premier test, l'algorithme vérifie si x est une puissance entière de 2, c'est-à-dire si on a $x = 2^E$. Dans ce cas, x^y ne peut être un cas de frontière d'arrondi que si $E \cdot y \in \mathbb{Z}$. L'algorithme répond donc de façon appropriée dans ce cas. Le lecteur est prié de considérer la section 2.2.4 pour la preuve de correction dans ce cas particulier. Si x n'est pas une puissance entière de 2, l'algorithme teste si (x, y) appartient à $\mathbb{S}$ ou non. Dans le cas où (x, y) appartient à $\mathbb{S}$ mais y n'est pas entier, c'est-à-dire s'il se décompose en $y = 2^F \cdot n$ avec $n \in 2\mathbb{N}+1$ et un F négatif, l'algorithme vérifie en plus non seulement si $m^{2^F \cdot n} = k$ mais aussi

Algorithme :
Entrées : $x \in \mathbb{D}, x > 0, y \in \mathbb{D}, y \neq 0, y \neq 1$, $\widehat{z}$ tel que $\widehat{z} = x^y \cdot (1+\varepsilon)$ avec $|\varepsilon| \leq 2^{-117}$
Sorties : le prédicat $RB(x,y) = (x^y \in \mathbb{F}_{54})$
début
  Soit $2^G \cdot k = \circ_{54}(\widehat{z})$ tel que $G \in \mathbb{Z}, k \in \mathbb{N}$;
  si $\left|2^G \cdot k - \widehat{z}\right| \geq 2^{-116} \cdot z$ **alors renvoyer** *faux*;
  si $\exists E \in \mathbb{Z}, x = 2^E$ **alors**
    si $E \cdot y \in \mathbb{Z}$ **alors renvoyer** *vrai* **sinon renvoyer** *faux*;
  sinon
    si $y < 0 \vee y > 35$ **alors renvoyer** *faux*;
    Soit $F \in \mathbb{Z}$, $n \in 2\mathbb{N}+1$ tel que $2^F \cdot n = y$;
    si $n > 35 \vee F < -5$ **alors renvoyer** *faux*;
    si $F < 0$ **alors**
      Soit $E \in \mathbb{Z}$, $m \in 2\mathbb{N}+1$ tel que $2^E \cdot m = x$;
      si $E \cdot y \notin \mathbb{Z}$ **alors renvoyer** *faux*;
      si $2^{G-E\cdot y} \cdot k \notin 2\mathbb{Z}+1$ **alors renvoyer** *faux*;
    fin
    renvoyer *vrai*;
  fin
fin

Algorithme 1 : L'algorithme **détecteCasDeFrontière** pour filtrer les cas de frontière

si $E \cdot y = G$, respectivement si $k \in 2\mathbb{Z}+1$. Ce test est nécessaire dans ce cas parce que le test $x^y = 2^G \cdot k$ doit être décomposé en $E \cdot y = G$ et $m^{2^F \cdot n} = k$ (cf. section 2.2.2). Si au moins une des conditions, c'est-à-dire $(x,y) \in \mathbb{S}$ ou $E \cdot y = G$, reste insatisfaite, l'entrée donnée ne peut pas être un cas de frontière d'argument grâce à l'argument de l'écart. L'algorithme renvoie donc immédiatement *faux* dans ce cas. Si l'entrée doit être un cas de frontière d'arrondi ; l'algorithme renvoie *vrai*. Le test si $(x,y) \in \mathbb{S}$ implique des comparaisons avec des constantes précalculées simples 0, -5 et 35.

Notre algorithme de détection de frontières d'arrondi décompose ses entrées x et y en $x = 2^E \cdot m$ et $y = 2^F \cdot n$. Cette opération, qui est un calcul de la valuation dyadique d'un nombre en fait, semble nécessiter une boucle très dispendieuse pour compter les bits à zéros à partir du bit de poids faible dans la mantisse d'un nombre. Les approches précédentes, en l'occurrence le bibliothèque `libmcr` de Sun et la bibliothèque `libultim` de Ziv, utilisent une telle boucle. Mais en fait, des techniques sont connues[4] pour effectuer une telle décomposition juste par des opérations logiques ou une petite table [89].

Algorithme d'arrondi correct de pow

Sur la base de notre analyse des probabilités des cas de frontière d'arrondi, présentée à la section 2.2.3, en utilisant notre algorithme de détection de frontières d'arrondi, nous proposons l'approche représentée à la figure 2.5 pour l'arrondi correct de la fonction pow en précision double.

[4] voir par exemple `http://graphics.stanford.edu/~seander/bithacks.html`

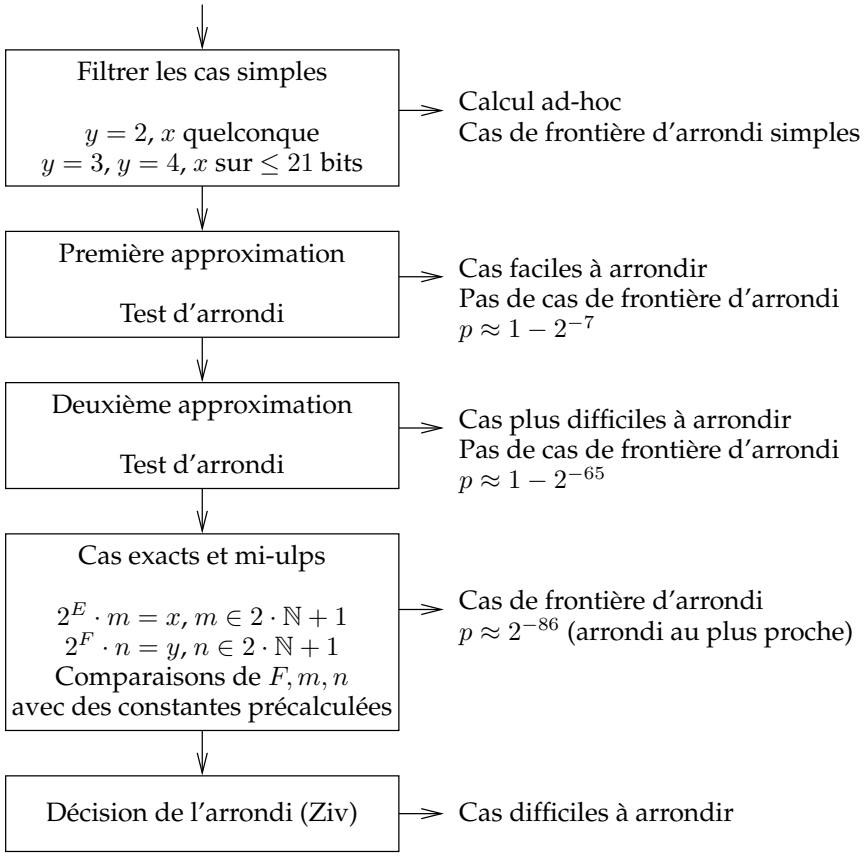


FIG. 2.5 – Nouvelle approche pour l'arrondi correct de pow

Dans notre approche, l'algorithme commence par filtrer des cas simples comme $y = 2$ pour un x flottant quelconque et $y = 3$ ou $y = 4$ pour x un flottant sur au plus 21 bits. Ce filtre peut être réalisé en parallèle avec le test pour les arguments irréguliers de la fonction pow (NaN, infinités etc.). Sur des processeurs super-scalaires, l'évaluation de x^y pour le cas général peut déjà commencer en parallèle. Le coût du filtre est donc caché dans le chemin critique. Les résultats de la fonction dans les cas spéciaux x^2, x^3 et x^4 peuvent être calculés d'une façon ad hoc. Comme à peu près la moitié des cas de frontière d'arrondi correspondent à ces cas spéciaux, la performance en moyenne sur des entrées purement frontière d'arrondi sera augmentée.

L'algorithme continue ensuite par deux itérations du processus d'arrondi correct de Ziv. La première étape sert à approcher x^y avec une précision équivalente à 60 bits significatifs [33, 40, 43]. Afin de répondre aux besoins de notre algorithme pour la détection de frontières d'arrondi, la deuxième étape approche ensuite x^y avec au moins 117 bits valides. Le test de frontière d'arrondi filtre ensuite les cas exacts et mi-ulps avant que d'autres itérations du processus de Ziv ne soient lancées. Dans notre approche, le test de frontière d'arrondi avec ses juste huit comparaisons devient une partie négligeable dans l'algorithme d'arrondi correct complet pour la fonction pow.

2.2.4 Preuve de correction de l'algorithme pour les frontières d'arrondi

La revendication de fournir l'arrondi correct d'une fonction ou, ce qui est plus important, la revendication qu'un code utilisant l'itération de Ziv termine, ne vaut rien s'il n'y a pas de preuve donnée. Démontrons alors notre algorithme de détection de frontières d'arrondi, tel qu'il est donné par le listing Algorithme 1 **détecteCasDeFrontière**. Nous allons d'abord démontrer une série de lemmes qui suit le schéma de l'argumentation donnée à la section 2.2.2. Cette preuve étend des concepts dont une ébauche est publiée dans [124].

Théorème 1
Soit

$$\begin{aligned}\mathbb{S} &= \left\{(x,y)\in\mathbb{D}^2 \mid y\in\mathbb{N},\ 2\le y\le 35\right\}\\ &\cup \left\{(m,2^F n)\in\mathbb{D}^2 \mid F\in\mathbb{Z},\ -5\le F<0,\ n\in 2\mathbb{N}+1,\ 3\le n\le 35,\ m\in 2\mathbb{N}+1\right\}.\end{aligned}$$

Soit $\circ_{54}$ l'arrondi au plus proche à 54 bits. Alors la propriété suivante est vérifiée :

$$\forall (x,y)\in\mathbb{S}, x^y\in\mathbb{F}_{54}\vee\left|\frac{\circ_{54}(x^y)-x^y}{x^y}\right|\ge 2^{-114}.$$

Démonstration (ésquisse)
La borne a été obtenue en utilisant un algorithme de recherche de pire cas [85, 75, 74]. La preuve de correction de cet algorithme dépasserait de loin le cadre de cette thèse.

Le test si $2^{E\cdot 2^F\cdot n}\cdot m^{2^F\cdot n}=2^G\cdot k$ peut être décomposé en deux tests séparés :

Lemme 1
Soit $E,F,G\in\mathbb{Z}$, $m,n,k\in 2\mathbb{N}+1$. Alors, on a

$$2^{E\cdot 2^F\cdot n}\cdot m^{2^F\cdot n}=2^G\cdot k \quad\Leftrightarrow\quad \begin{array}{l}E\cdot 2^F\cdot n=G\\ \wedge\ m^{2^F\cdot n}=k\,.\end{array}$$

Démonstration (ésquisse)
Considérons le fait que $m^{2^F\cdot n}$ et k ou m^n et $k^{2^{-F}}$ sont des entiers impairs et que $E\cdot 2^F\cdot n-G\neq 0$ ou $E\cdot n-G\cdot 2^{-F}\neq 0$. Alors les équations $2^{E\cdot 2^F\cdot n-G}\cdot m^{2^F\cdot n}=k$ ou $2^{E\cdot n-G\cdot 2^{-F}}\cdot m^n=k^{2^{-F}}$ impliquent des contradictions parce que leurs membres gauche sont pairs et leur membres droits sont impairs.

L'algorithme vérifie plusieurs conditions de minoration ou de majoration sur les entrées. Ces bornes se démontrent comme suit :

Lemme 2
Soit $m\in 2\mathbb{N}+1$ borné par $3\le m\le 2^{53}-1$. Soit $n\in 2\mathbb{N}+1$ borné par $1\le n\le 2^{53}-1$. Soit $k\in 2\mathbb{N}+1$ borné par $1\le k\le 2^{54}-1$. Soit $F\in\mathbb{Z}$ un entier. Supposons que $m^{2^F\cdot n}=k$. Alors $2^F\cdot n$ est borné par $2^F\cdot n\le 35$ et F est borné par $-5\le F\le 5$.

Démonstration
Montrons d'abord les majorations. Comme $k\le 2^{54}-1$, on sait que $m^{2^F\cdot n}\le 2^{54}$ et $2^F\cdot n\cdot\log_2(m)\le 54$. Comme $m\ge 3$, on a $\log_2(m)\ge\log_2(3)>0$. Alors $2^F\cdot n\le\frac{54}{\log_2(m)}\le\frac{54}{\log_2(3)}<$

$34.08 < 35$. Ceci est la borne supérieure à être démontré pour $2^F \cdot n$. Comme $n \geq 1$, on a $2^F \leq 35$ et donc $F \leq 5.13$. Comme F est entier, on a la borne donnée $F \leq 5$.

Montrons maintenant que $F \geq -5$. Sans nuire à la généralité, nous pouvons supposer que F est négatif. Soit p_i, q_i des nombres premiers tels que $i \neq i' \Rightarrow p_i \neq p_{i'} \wedge q_i \neq q_{i'}$. Soit $\alpha_i, \beta_i \in \mathbb{N}\backslash\{0\}$ des valuations telles que $m = \prod_i p_i^{\alpha_i}$ et $k = \prod_i q_i^{\beta_i}$. Comme $F \leq -1$, 2^{-F} est entier. Alors on a $m^n = k^{2^{-F}}$ où m^n et $k^{2^{-F}}$ sont entiers. En conséquence, l'équation $\prod_i p_i^{\alpha_i \cdot n} = \prod_i q_i^{\beta_i \cdot 2^{-F}}$ est vérifiée et il existe une permutation σ telle que $\forall i \,.\, p_i = q_{\sigma(i)} \wedge \alpha_i \cdot n = \beta_{\sigma(i)} \cdot 2^{-F}$. Comme m est un entier impair et $m \geq 3$, on a $\forall i \,.\, p_i \geq 3$. Puis, $m \leq 2^{53} - 1$ est vérifié et donc $3^{\alpha_i} \leq 2^{53}$ et $\alpha_i \leq 53 \cdot \frac{\log(2)}{\log(3)} \leq 34$. Soit $\kappa_i \in 2\mathbb{N} + 1$ des entier impairs et $\gamma_i \in \mathbb{N}$ des valuations tels que $\forall i \,.\, \alpha_i = 2^{\gamma_i} \cdot \kappa_i$. De tels κ_i et γ_i existent pour tous les α_i parce que l'on a $\alpha_i \in \mathbb{N}$. Comme $\forall i \,.\, \alpha_i \geq 1$, les majorations suivantes tiennent : $\forall i \,.\, \kappa_i \geq 1$. Comme $\forall i \,.\, \alpha_i \leq 34$, on a $2^{\gamma_i} \leq 34$ et $\gamma_i \leq \log_2(34) \leq 5.09$. Puisque $\gamma_i \in \mathbb{N}$, on a $\gamma_i \leq 5$. On a montré que l'égalité suivante est vérifiée : $2^{\gamma_i} \cdot (\kappa_i \cdot n) = \beta_{\sigma(i)} \cdot 2^{-F}$ Comme n et tous les κ_i sont des entiers impairs, $\kappa_i \cdot n$ est impair. Puis $\beta_{\sigma(i)}$ est un entier. En conséquence, $-F$ est majoré par γ_i qui est majoré par 5. Alors $F \geq -\gamma_i \geq -5$.

Le test si $m^{2^F \cdot n} = k$ pour $F < 0$ peut être décomposé en deux tests, en l'occurrence $m^{2^F} = j \in \mathbb{N}$ et $j^n = k$:

Lemme 3
Supposons que l'on ait $m, n, k \in 2\mathbb{N} + 1$, $F \in \mathbb{Z}$, $F \leq -1$.
Alors $m^{2^F \cdot n} = k \Leftrightarrow \exists j \in \mathbb{N} \,.\, \left(j = m^{2^F} \wedge j^n = k\right)$ *est vérifié.*

Démonstration (ésquisse)
Il suffit de remarquer que d'une part, la 2^{-F}-ième racine d'un entier m n'est entière que si toutes les valuations de la décomposition en facteurs premiers de m sont divisibles par 2^{-F} et que d'autre part, n est impair. Alors, si $j = m^{2^F}$ n'est pas entier, il existe une valuation dans la décomposition en facteurs premier à la fois de m et de m^n qui n'est pas divisible par 2^{-F} or que toutes les valuations de $k^{2^{-F}}$ sont divisibles par 2^{-F}.

La correction de l'algorithme 1 **détecteCasDeFrontière** doit également être démontrée pour le cas des valeurs y négatives que nous pouvons classifier comme suit :

Lemme 4
Supposons que l'on ait $x, y \in \mathbb{D}$ *tel que* $x > 0$, $y < 0$, $x^y \in \mathbb{R}$ *et* $2^{-1075} \leq |x^y| \leq 2^{-1024}$.
Alors on a $x^y \in \mathbb{F}_{54}$ *ssi* $\exists a \in \mathbb{Z} \,.\, (2^a = x \wedge a \cdot y \in \mathbb{Z})$.

Démonstration
L'existence de la valeur a énoncée implique clairement $x^y \in \mathbb{F}_{54}$. L'implication inverse peut se démontrer comme suit : Supposons que l'on ait $x^y \in \mathbb{F}_{54}$ mais que le contraire de ce qui est affirmé soit vérifié. Comme $x, y \in \mathbb{F}_{53}$ et $x^y \in \mathbb{F}_{54}$, il existe des entiers impairs $m, n, k \in 2\mathbb{N}+1$ et des entiers relatifs $E, F, G \in \mathbb{Z}$ tels que $x = 2^E \cdot m$, $y = -2^F \cdot n$ et $x^y = 2^G \cdot k$. On en déduit que $m^{2^F \cdot n} = 2^{-G-E \cdot 2^F \cdot n} \cdot \frac{1}{k}$. Il y a alors deux cas en fonction du signe de F. Si $F \geq 0$ alors 2^F est entier, $2^F \cdot n$ est entier et $-G - E \cdot 2^F \cdot n$ est un entier relatif. Il existe alors un entier $a \in \mathbb{N}$, un entier relatif $b \in \mathbb{Z}$ et un entier impair $c \in 2\mathbb{N}+1$ tels que $m^a = 2^b \cdot \frac{1}{c}$. On les obtient

en posant $a = 2^F \cdot n$, $b = -G - E \cdot 2^F \cdot n$ et $c = k$. Si $F < 0$, alors 2^{-F} est entier, $-G \cdot 2^{-F}$, $E \cdot n$ et $-G \cdot 2^{-F} - E \cdot n$ sont des entiers relatifs et $k^{2^{-F}}$ est un entier impair. Alors il existe un entier $a \in \mathbb{N}$, un entier relatif $b \in \mathbb{Z}$ est un entier impair $c \in 2\mathbb{N} + 1$ tels que $m^a = 2^b \cdot \frac{1}{c}$. On les obtient en posant $a = n$, $b = -G \cdot 2^{-F} - E \cdot n$ et $c = k^{2^{-F}}$.

Si $b \geq 0$ alors 2^b est entier. Comme m^a est entier, $\frac{2^b}{c}$ est entier. Comme c est impair, on a $\mathrm{pgcd}\left(2^b, c\right) = 1$. En conséquence, c est égal à 1, $c = 1$. Si $b < 0$ alors 2^{-b} et $2^{-b} \cdot c$ sont entiers. Comme m^a est entier, $\frac{1}{2^{-b} \cdot c}$ est entier. Alors, on a $2^{-b} \cdot c = 1$ et donc $c = 1$ parce que c est impair et 2^{-b} est entier.

Comme $c = 1$ et $c = k$ ou $c = k^{2^{-F}}$, k vaut 1 dans tous les cas. Cela implique que $x^y = 2^G \cdot 1$ est une puissance entière de 2. En conséquence, on a $x = 2^{\frac{G}{y}}$. Comme 2^ξ est transcendante pour toutes les valeurs ξ algébriques qui ne sont pas des entiers signés (cf. [59]), et comme x est un nombre algébrique, il existe un nombre $a = \frac{G}{y} \in \mathbb{Z}$. Alors $a \cdot y = G \in \mathbb{Z}$ et $x = 2^{\frac{G}{y}} = 2^{\frac{a \cdot y}{y}} = 2^a$. Ceci contredit les hypothèses.

Voilà enfin le théorème de correction pour notre algorithme pour la détection de frontières d'arrondi :

Théorème 2
L'algorithme 1 ***détecteCasDeFrontière*** *est correct.*
Cela veut dire que $\forall x \in \mathbb{D}, x > 0$ *et* $\forall y \in \mathbb{D}, y \neq 0, y \neq 1$ *tels que* $2^{-1075} \leq x^y \leq 2^{1024}$ *et* $\forall z = x^y \cdot (1 + \varepsilon)$ *pour quelque* ε, $|\varepsilon| \leq 2^{-116}$, *l'algorithme renvoie* vrai *ssi* $x^y \in \mathbb{F}_{54}$.

Démonstration (ésquisse)
En combinant le théorème 1, les lemmes 1, 2, 3 et 4 et en considérant les lignes auxquelles l'algorithme peut renvoyer *vrai* (respectivement *faux*), on vérifie trivialement la conséquence du théorème.

2.2.5 Résultats expérimentaux pour la détection des frontières d'arrondi

Nous avons implanté nos approches pour l'arrondi correct de la fonction pow et pour la détection de cas de frontière d'arrondi dans la bibliothèque CRLibm [33]. Nous avons comparé notre implantation à une bibliothèque `libm` non correctement arrondie, à la bibliothèque `libultim` de Ziv, à la bibliothèque `libmcr` de Sun et à la bibliothèque MPFR dans sa version 2.2.0. Les expérimentation ont été faites sur deux systèmes différents : premièrement sur un processeur Intel Xeon cadencé à 2.40 GHz avec GNU/Linux 2.6.19.2-server et `gcc` 3.3.5 comme compilateur et, deuxièmement, sur un processeur IBM Power5 cadencé à 1.66 GHz avec GNU/Linux 2.6.18.8-0.3-ppc64 et `gcc` 4.1.2. Remarquons que toutes les implantations ne sont pas disponibles partout. Le seul mode d'arrondi supporté par toutes les bibliothèques est l'arrondi au plus proche. Les temps d'exécution mesurés ont été normalisés, pour chaque processeur, à 1 avec CRLibm comme référence. Dans le cas du processeur Intel Xeon, l'unité flottante x87 est utilisée, `libmcr` ne se compilant pas pour l'unité SSE2.

Nous donnons des mesures différentes pour des entrées aléatoires et pour des entrées qui ne consistent qu'en cas de frontière d'arrondi non-triviaux ($y \neq 1, y \neq 2$). Nous indiquons les temps mesurés en moyenne et au pire cas. Les temps dans le pire cas sont les pires mesures observées sur des entrées aléatoire et non les mesures absolument pires observables (sur les pires cas inconnus de la fonction pow).

	Intel Xeon		IBM Power5	
	Entrées aléatoires moy./pire cas	$x^y \in \mathbb{F}_{54}$ moy./pire cas	Entrées aléatoires moy./pire cas	$x^y \in \mathbb{F}_{54}$ moy./pire cas
CRLibm	1/7.70	3.18/6.38	1/7.63	4.06/8.42
`libm`	1.20/134	0.633/0.899	-	-
`libultim`	-	-	1.65/8550	3.19/4.14
`libmcr`	3.54/172	0.636/1.61	-	-
MPFR	170/298	47.9/168	700/1090	188/534

Ces résultats montrent que la performance en moyenne sur des entrées aléatoires est augmentée au moins de 39% dans notre implantation par rapport aux implantations existantes, en l'occurrence `libultim`. Ces améliorations sont obtenues au détriment d'une légère décélération des très rares cas de frontière d'arrondi, en l'occurrence d'à peu près 21% par rapport à la bibliothèque `libultim`. La différence entre les temps de calcul pour les cas difficiles à arrondir et les cas de frontière d'arrondi peut être négligée : les cas de frontière d'arrondi ne sont qu'au plus 9% plus lents. Pour la bibliothèque `libmcr`, ce surcoût de la détection de frontières d'arrondi pouvait atteindre les 50%. Il semble raisonnable de penser qu'une application qui peut supporter un facteur de 7.70 entre le temps en moyenne et les pires cas au profit de l'arrondi correct, peut supporter 9% en plus pour une accélération de 39% en moyenne. Les différences observées sur l'architecture Xeon, indiquant que le pire cas de l'arrondi est plus lent que l'arrondi dans un cas de frontière d'arrondi, s'expliquent par la gestion compliquée et lente de l'arrondi subnormal sur cette architecture. Les cas de frontière d'arrondi sont à peu près 50% plus rapides en moyenne que dans leur pire cas. Ce dernier résultat expérimental valide parfaitement la pertinence de nos estimations théoriques à 49.6% (cf. section 2.2.3).

2.2.6 Conclusions et travaux futurs sur la fonction $\mathrm{pow}(x, y) = x^y$

Dans cette section, nous avons considéré la détection des cas de frontière d'arrondi pour la fonction $\mathrm{pow} : (x, y) \mapsto x^y$ en précision double. Nous avons présenté un algorithme pour la détection efficace de ces cas de frontière. L'algorithme, consistant juste en quelques comparaisons avec des constantes précalculées, permet d'obtenir une meilleur performance en moyenne d'une implantation de pow. Typiquement, le chemin critique n'y est plus ralenti par des opérations difficiles à paralléliser sur un processeur super-scalaire. Des boucles impliquant des tests et des extractions répétées de racines carrées sont remplacées par un programme linéaire.

Notre algorithme utilise des constantes précalculées. Ces constantes sont dérivées de la précision de pire cas pour l'arrondi de la fonction pow dans un sous-domaine particulier de l'ensemble des nombres double précision. Cette utilisation innovatrice des techniques développées à la base seul pour l'arrondi correct peut s'étendre à d'autres fonctions et peut être bénéfique en termes de performances. Nous allons entreprendre plus d'investigations dans cette direction à l'avenir.

Les avantages de notre approche et de notre implantation consistent en une amélioration de la performance en moyenne de pow d'à peu près 39% et dans la chute du surcoût de la détection de frontières d'arrondi à au plus 9%. L'inconvénient de l'approche est une décélération des cas de frontières d'arrondi qui sont pourtant très rares ($p = 2^{-87}$). La décélération

n'excède pourtant pas 21%. Par rapport à la performance en moyenne, cette décélération pourrait être diminuée si non seulement les cas triviaux $y = 2$, $y = 3$ et $y = 4$ pouvaient être gérés à part sans ralentir le chemin critique. En l'occurrence, encore 99.1% de nos cas mi-ulps non-triviaux sont formés par le cas $y = \frac{3}{2}$ et x sur au plus 36 bits significatifs. Il fait partie de nos projets de trouver un algorithme spécial pour ce cas. Avec les approches actuelles, il demanderait l'utilisation d'une extraction de racine carrée bloquant les pipelines sur un chemin spécial.

2.3 Conclusions sur l'arrondi correct des fonctions puissance

Dans les sections précédentes 2.1 et 2.2, nous avons présenté deux techniques qui contribuent à la solution du problème difficile qu'est l'arrondi correct des fonctions puissance.

En nous basant toujours sur des résultats de recherche de pire cas qui reste possible pour la famille des fonction x^n avec n un petit entier, nous avons proposé une implantation correctement arrondie. Cette implantation essaie d'utiliser au mieux le matériel Itanium donné. Elle cherche en plus à réutiliser le plus de valeurs déjà calculées à la phase rapide pour la phase précises. De bonnes performances peuvent donc être obtenues.

Ensuite, nous nous sommes intéressés à la détection rapide des cas de frontière d'arrondi pour la fonction bivariée x^y. Encore une fois, nous avons basé notre algorithme – très simple par rapport aux algorithmes connus dans la littérature – sur une information de pire cas, au moins partielle. Nous avons montré que la recherche de pire cas nécessaire peut être effectué en pratique car le domaine de recherche se réduit à un ensemble très réduit. Avec notre algorithme de détection de cas de frontière d'arrondi, conçu explicitement pour fonctionner en couple avec un algorithme de calcul de la fonction par approximations successives, le processus de Ziv devient une solution pour l'arrondi correct de la fonction x^y.

Cette situation peut être satisfaisante en pratique. Pourtant, elle me semble tout à fait insatisfaisante d'un point de vue généralité de l'arrondi correct. Pour presque toutes les autres fonctions – univariées bien sûr – définies dans la norme IEEE 754-2008 [67], nous savons maintenant fournir l'arrondi correct avec un temps de calcul et une utilisation mémoire bornés au pire cas. Pour x^y nous ne savons pas faire cela et sommes ainsi obligés de le remarquer à chaque fois pour le produit qu'est CRLibm. Le vrai problème derrière les fonctions puissance est donc la difficulté de faire des recherches de pire cas.

L'algorithme SLZ [116] semble être très prometteur non pour calculer, c'est-à-dire exhiber, des pires cas mais pour certifier qu'il n'y a pas de pire cas demandant plus de précision qu'une certaine valeur $\overline{\varepsilon}$. Stehlé démontre [115] que si cette borne est quadratique en la précision k du format vers lequel l'arrondi se fait, $\overline{\varepsilon} = \left(2^{-k}\right)^2$, la complexité de SLZ pour cette certification devient polynomiale en la longueur de la mantisse k. Comme une borne calculée sur $53^2 = 2809$ bits est tout à fait acceptable[5] comme recours final dans des implantations de la fonction puissance, l'algorithme semble être une vraie solution. Malheureusement, la situation est beaucoup moins claire en pratique à cause des problèmes suivants :

– L'algorithme est polynomial mais les constantes sont si grandes que l'on évalue le temps de calcul à plusieurs centaines de machines-années par combinaison d'exposant. Ceci est dû également au fait que l'algorithme SLZ [116] n'est pas encore aussi

[5]Des expérimentations durant cette thèse ont par exemple montré que le calcul de polynômes d'approximation de Remez est possible pour de telles précisions et que le degré nécessaire reste dans les alentours de 400.

optimisé que les algorithmes proposés par Muller et Lefèvre [85].
- L'algorithme est polynomial en la taille de la mantisse mais non en la taille du champ d'exposant. Celui est relativement réduit par rapport au domaine complet de la précision double à cause des dépassements de capacités mais toujours trop grand (à peu près 2^6 combinaisons pour les exposants de x et de y).
- Aucune réponse ne peut actuellement être trouvée au problème que l'implantation de l'algorithme SLZ pourrait avoir des bogues. Quand après un calcul de plusieurs années sur un ensemble non-négligeable de machines, l'algorithme ne répondrait rien d'autre que « *Oui,* 2809 *bits suffisent pour* x^y *en précision double.* » , qui croirait en cette réponse sybillique ? Seule la certification formelle peut y apporter une solution.

Toutefois, il est dans nos projets de pousser des investigations profondément dans cette direction.

CHAPITRE 3

Arithmétique multi-double

Dans toute statistique, l'inexactitude du nombre est compensée par la précision des décimales.

Alfred Sauvy, *économiste et sociologue français*

À cause du dilemme du fabricant des tables, nous sommes obligés de manipuler des approximations très précises à des fonctions mathématiques f afin d'en donner l'arrondi correct $\star(f(x))$. Pour la précision IEEE 754 double, des valeurs sur à peu près 120 bits doivent être gérées [88, 32]. La première étape dans le développement d'une bibliothèque de fonctions mathématiques correctement arrondies comme CRLibm consiste donc dans le travail le plus basique de l'arithméticien : se donner une arithmétique, c'est-à-dire concevoir un format de représentation et les opérations de base permettant de manipuler de telles précisions.

Dans ce chapitre nous proposons et défendons le format triple-double comme un format bien adapté aux besoins de l'arrondi correct des fonctions usuelles en précision double. Combiné avec les formats double et double-double, il donne l'arithmétique multi-double. Le format montrera ses avantages et nous servira également de base pour l'automatisation de l'implantation de fonctions, décrite au chapitre 6. Le chapitre 4.3 y fera également référence.

Ce chapitre résume nos travaux dans [81] et étend ceux dans [43].

3.1 Motivation et introduction au format triple-double

Comme on a déjà vu, le format les plus précis qui doit être obligatoirement être supporté par un système conforme à la norme IEEE 754-1985 [5] est le format double précision. Il offre 53 bits de précision mantisse. Dans notre cadre des fonctions mathématiques correctement arrondies [43] ainsi que par exemple dans le domaine de la géométrie calculatoire [108, 113], cette précision n'est pas suffisante. Même l'approche bien connue et prouvée de la double-double [100, 36, 9, 81, 106, 92] peut être insuffisante en termes de précision. Par exemple, l'arrondi correct des fonctions mathématiques avec comme format cible la précision double nécessite de manipuler des résultats intermédiaires sur 120 bits [88, 32]. Cela est à peu près 2.3 fois la précision du format double.

Différentes approches pour un calcul en grande précision ont été proposées. Il y a premièrement les bibliothèques de précision arbitraire comme MPFR [50] et les bibliothèques basées sur des expansions de nombres flottants [108, 113]. Deuxièmement, on trouve des bibliothèques à précision fixée comme la bibliothèque quad-double proposée par Bailey [63]

ou la bibliothèque SCSlib [33]. Finalement, des approches pour le calcul précis, n'ayant pas recours à des opérateurs de précision agrandie, ont été proposées [92, 55]. Elles restent pourtant spécifiques à certains domaines d'application ce qui empêche leur utilisation dans notre cadre ; on ne les considérera donc plus.

Les bibliothèques de précision fixée citées fournissent au moins 210 bits de précision, les bibliothèques de précision arbitraire peuvent supporter des précisions de milliers de bits, si nécessaire. Dans tous les cas, on observe un surcoût important pour l'émulation de la grande précision, ce qui affecte la performance. Les bibliothèques SCSlib et MPFR réalisent leurs calculs sur les unités entières du processeur. Des conversions sont donc nécessaires dans un environnement flottant. Les codes basés sur les expansions flottants et la quad-double contiennent beaucoup de branchements et utilisent des renormalisations chères après chaque opération [63, 78, 55].

Dans cette section, nous proposons un autre format, que nous appelons format triple-double. Il peut fournir jusqu'à 2.9 fois la précision du format double (cf. section 3.2.1). Il est nativement compatible avec le format double-double. Avec une approche particulière, que nous allons présenter, cela permet de mélanger du calcul en précision double avec de la double-double puis de la triple-double. Une telle *adaptation de la précision au juste minimum* est avantageuse en termes de performance : quelques parties de nos algorithmes n'auront besoin que d'un multiple modérément petit de la précision double de base. Seulement à quelques endroits clefs, une grande précision sera nécessaire.

Cette observation sur l'adaptabilité de la précision nécessaire s'explique bien à l'exemple d'une évaluation de polynômes sous schéma de Horner. Si dans $p(x) = \sum_{i=0}^{k} c_i \cdot x^i$, $|c_{i+1}| < |c_i|$, x est petit (c'est-à-dire si on a $|x| \leq 2^{-\mu}$, $\mu > 1$), alors l'erreur d'arrondi relative des étapes précédentes est pondérée par au moins la valeur absolue de x, c'est-à-dire par $2^{-\mu}$. L'évaluation d'un polynôme à une précision finale donnée peut donc se faire d'abord en format double (avec une erreur grande mais pondérée par quelque valeur très petite), ensuite en double-double (avec une erreur et une pondération moyenne) et finalement en précision triple-double (avec une erreur petite et un pondération proche de 1). Des effets similaires sont classiquement observées lors d'additions d'approximations d'ordre de grandeur très différente [64]. Le chapitre 6.2 donnera plus de détails à ce sujet.

Les opérations triple-double, que nous proposons (cf. section 3.2.3), peuvent être implantées sans aucun branchement. Cela est avantageux sur les processeurs super-scalaires actuels [78]. Le format triple-double est suffisamment efficace en termes de mémoire ce qui permet d'utiliser des techniques de tabulation dans les implantations de fonctions mathématiques [122, 33]. Comme on va voir, les données tabulées peuvent être partagées entre les différentes étapes.

L'utilisation des approches triple-double, présentées dans cette section, dans les implantations de fonctions mathématiques correctement arrondies en CRLibm a permis d'obtenir un gain en vitesse d'un facteur à peu près 10 par rapport au format de grande précision fixe SCSlib [43]. Plus de détails sur ce point sera discuté à la section 3.4.

Le format triple-double est un compromis naturel entre les précisions double-double et quad-double. La principale contribution de cette section par rapport aux approches existantes consiste en un cadre d'analyse d'erreur adaptive, qui sera motivé à la section 3.2.2 et détaillé à la section 3.3. Notre technique permet d'obtenir des bornes d'erreur d'arrondi fines tout en simplifiant les calculs pour des raisons de performance. Notre analyse de code sta-

tique indique où des renormalisations très chères peuvent être supprimées, qui, en d'autres approches comme la quad-double [63], étaient exécutées presque aveuglement après toute opération. Les preuves de précision associées ne seront pourtant pas mises en danger.

Le format triple-double, comme toutes les techniques d'expansion, n'étend que la précision de calcul et non le domaine d'exposant des nombres. Nous sommes conscient de ce problème plus théorique que pratique dans notre cadre. Nous ne l'adresserons plus ici.

3.2 Le format triple-double

3.2.1 Définition et cadre conceptuel

Il y a des techniques bien connues et bien prouvées pour étendre la précision d'un format virgule flottant jusqu'à presque le double de la précision [100, 36, 9, 81, 106, 92]. Dans ces approches, les nombres son représentés par une somme non-évaluée $x_{hi} + x_{lo}$ de deux nombres flottants $x_{hi}, x_{lo} \in \mathbb{F}$. Un tel format est généralement appelé double-double quand le format IEEE 754 double sert de base.

Deux opérations de base, appelées **Fast2Sum** et **Fast2Mult** et présentés plus en détail au chapitre 4.3, permettent de réaliser des additions et des multiplications exactes de nombres flottants $a, b \in \mathbb{F}$ avec un résultat stocké comme la somme non-évaluée de deux nombres flottants $c_{hi} + c_{lo}$ [36, 100, 106, 92]. Ces opérations garantissent que dans tous les cas $c_{hi} = \circ(c_{hi} + c_{lo})$, où, rappelons-le, $\circ_k$ dénote l'arrondi au plus proche. Ces opérations de base peuvent être implantées sans branchements [73, 106]. Les détails de leur implantation, ainsi que la preuve de leur correction ne sont pas intéressants pour l'analyse et un travail avec le format triple-double. On les considère, pour ainsi dire, comme des boîtes noires et dans l'implantation et dans la preuve. C'est quand on descend au niveau des opérations flottantes discrètes dans la preuve des implantations (cf. chapitre 4) que l'on doit analyser et prouver leur fonctionnement.

Les opérations d'addition et de multiplication exacte permettent de concevoir des opérations ayant en entrée et en sortie des nombres double-double. Comme cela, une précision proche du double de la précision de base peut être atteinte [36, 81, 92]. La performance engendrée par ces techniques est grande ; leur utilisation est donc courante [33, 73, 93, 9, 26, 106, 101, 92].

Le principe de représenter une mantisse de grande précision sur plusieurs mantisses plus courtes peut être étendu. Dans le cas général, un nombre x est représenté par une somme non-évaluée arbitrairement longue $x = \sum x_i$ de nombres flottants de précision de base $x_i \in \mathbb{F}$. Une telle représentation s'appelle une expansion [108, 113]. La gestion de sommes de longueur arbitraire implique un surcoût qui affecte la performance [48]. Il convient alors de se limiter à format de pseudo-expansions d'une taille fixe, en l'occurrence trois. Ceci est un compromis entre la précision et la performance adapté à nos besoins.

Définition 2 (Triple-double)
Soit $x_{hi}, x_{mi}, x_{lo} \in \mathbb{F}_{53}$ trois nombres flottants double précision tels que $|x_{hi}| > |x_{mi}| > |x_{lo}|$. Alors la somme non-évaluée $x_{hi} + x_{mi} + x_{lo}$ est un nombre triple-double.

Dans le cas idéal, un nombre triple-double peut représenter une mantisse qui est trois fois plus longue que celle d'un flottant double précision. Cela est illustré par la figure 3.1 ci-dessous.

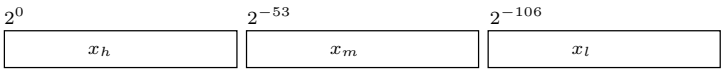


FIG. 3.1 – Une mantisse représentée sur un nombre triple-double

Le format triple-double est directement compatible avec les formats double-double et double. Par exemple, étant donné un triple-double $x = x_{hi} + x_{mi} + x_{lo}$, on peut extraire des approximations de x juste en négligeant des composantes. Le nombre x est approché à une certaine précision par le double-double $x_{hi} + x_{mi}$ et, à une précision encore moindre, par le double x_{hi}. Ce sera utilisé pour une évaluation sous schéma de Horner, comme déjà évoqué et explicité à la section 6.2. Inversement, dans des approches à base de tables [122, 43, 33] ou dans des algorithmes de raffinement de précision comme l'itération de Newton pour la division et la racine carrée [26, 93], il est possible de considérer d'abord une approximation double-double qui est ensuite raffinée et étendue à un triple-double.

La problématique associée à cette approche consiste dans la question de connaître la précision des approximations double et double-double obtenues en négligeant les composantes de poids faible d'un triple-double. Nous devons la résoudre pour la conception et la certification d'un algorithme adaptant la précision des valeurs intermédiaires. Principalement, il se peut que le triple-double $x = x_{hi} + x_{mi} + x_{lo}$ soit d'une forme telle que $x_{hi} + x_{mi}$ ne soit pas le double-double le plus proche de x ou que x_{hi} ne soit pas le double le plus proche de x.

3.2.2 Remarques sur la redondance des expansions flottantes

La triple-double est un système de numération redondant : il existe des valeurs x qui peuvent être représentées par deux nombres triple-double $x_{hi} + x_{mi} + x_{lo}$ et $x'_{hi} + x'_{mi} + x'_{lo}$ différents. Prenons par exemple $1.5+0.5-0.25 = 2-0.5+0.25$. En fait, les bits des mantisses de deux composantes, par exemple x_{mi} et x_{lo}, peuvent représenter des quantités du même poids et ainsi remplacer l'un l'autre. La figure 3.2 illustre cette redondance.

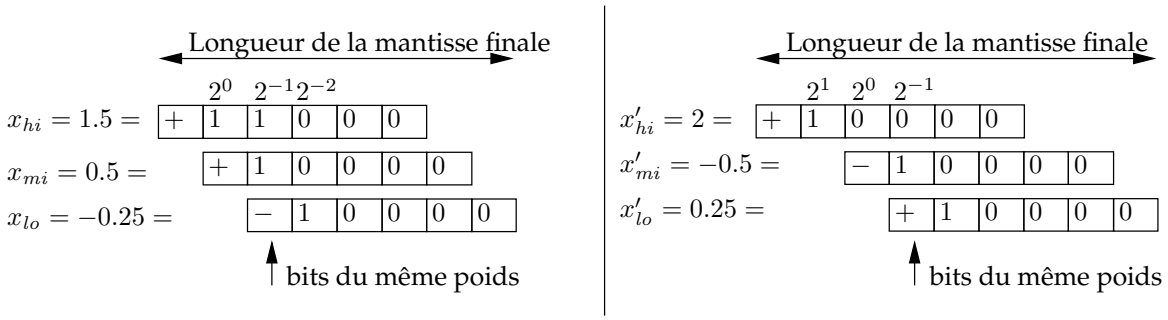


FIG. 3.2 – Chevauchements dans les flottants triple-double

Le format double-double est potentiellement aussi une représentation redondante mais en pratique, les nombres double-double sont tous produits par des opérateurs qui se terminent par une utilisation de l'une des opérations **Fast2Sum** ou **Fast2Mult** [33, 81]. Comme ces deux opérations garantissent que $x_{hi} = \circ(x_{hi} + x_{lo})$, il n'y pas de nombre double-double $x'_{hi} + x'_{lo}$ tel que $(x'_{hi}, x'_{lo}) \neq (x_{hi}, x_{lo})$, $x'_{hi} = \circ(x'_{hi} + x'_{lo})$ et $x'_{hi} + x'_{lo} = x_{hi} + x_{lo}$. Toute autre

somme de deux nombres flottants $a + b$ peut être renormalisée par un seul **Fast2Sum** à un coût très faible. Pour la triple-double, il n'y a pas d'opérateurs rapides, supporté en matériel [34], pour arrondir un nombre triple-double à la précision double, c'est-à-dire de calculer $\circ(x_{hi} + x_{mi} + x_{lo})$. La renormalisation est donc plus difficile et le résultat $x_{hi} + x_{mi} + x_{lo}$ d'une opération peut ne pas vérifier $x_{hi} = \circ(x_{hi} + x_{mi} + x_{lo})$ ou $x_{mi} = \circ(x_{mi} + x_{lo})$.

En principe, la redondance des nombres triple-double n'est pas un problème. Tout ce qui compte pour les additions et les multiplications dans nos codes, c'est la précision du résultat. C'est cette précision qui est clairement affectée par la redondance des nombres : tous les bit de deux composantes différentes qui représentent des quantités du même poids sont perdus pour les positions de poids faible. Toutefois tant que cette perte en précision n'est pas trop grande, les nombres triple-double peuvent stocker la précision dont on a besoin. Quand on cible une précision juste légèrement plus grande que deux fois (2.1 fois) la précision de la double précision, il serait même artificiel de forcer tous les nombres triple-double à être non-redondant. Des opérations de base qui garantiraient cela seraient juste plus coûteuses en termes de temps de calcul. Pourtant, dans les bibliothèques de calcul précis existantes [63], les renormalisations étaient faites après chaque opération. La contribution principal du travail présenté dans cette section consiste donc dans une approche de *l'adaptation de la précision au juste minimum* où la redondance peut avoir lieu jusqu'à un certain niveau lequel est statiquement analysé [81] :

Définition 3 (Chevauchement)
Soit $a_{hi} + a_{mi} + a_{lo} \in \mathbb{F}_{53}^3$ un nombre triple-double tel que $|a_{mi}| \leq 2^{-\alpha_o} \cdot |a_{hi}|$ et $|a_{lo}| \leq 2^{-\alpha_u} \cdot |a_{mi}|$. Si $\alpha_o \geq 52$ (respectivement $\alpha_u \geq 52$), a_{hi} et a_{mi} (respectivement a_{mi} et a_{lo}) ne se chevauchent pas. Sinon, ils se chevauchent d'au plus de $52 - \alpha_o$ (respectivement $52 - \alpha_u$) bits.

Cette définition du chevauchement implique que les nombres triple-double sont maximalement précis quand ils sont non-chevauchants. Remarquons que même pour des nombre non-chevauchants, on n'a toujours pas d'unicité, ce qui est sans importance pour notre cadre [81]. Si un arrondi d'un triple-double vers la précision double doit être effectué, une séquence logicielle est disponible [81].

3.2.3 Opérateurs triple-double

Renormalisation Un opérateur de renormalisation **Renormalize3** est toujours nécessaire quand le chevauchement des nombres devient trop grand. Dans notre approche, il n'est pourtant utilisé que quand l'analyse de précision et de chevauchement, expliquée à la section 3.3, indique que son utilisation est nécessaire pour maintenir la précision au niveau dont l'application a besoin. Tous les autres opérations triple-double peuvent retourner des résultats chevauchés. Dans d'autres approches, un renormalisation est faite à toute opération [63].

Algorithme : `Renormalisation`
Entrées : $a_{hi}, a_{mi}, a_{lo} \in \mathbb{F}$ vérifiant $|a_{mi}| \leq 2^{-2} \cdot |a_{hi}|$ et $|a_{lo}| \leq 2^{-2} \cdot |a_{mi}|$
Sorties : $r_{hi}, r_{mi}, r_{lo} \in \mathbb{F}$ tels que $r_{hi} + r_{mi} + r_{lo} = a_{hi} + a_{mi} + a_{lo}$ est non-chevauchant
début
  $(t_{1hi}, t_{1lo}) \leftarrow$ **Fast2Sum** (a_{mi}, a_{lo});
  $(r_{hi}, t_{2lo}) \leftarrow$ **Fast2Sum** (a_{hi}, t_{1hi});
  $(r_{mi}, r_{lo}) \leftarrow$ **Fast2Sum** (t_{2lo}, t_{1lo}) ;
fin

Algorithme 2 : Renormalisation

Une preuve de correction de cet algorithme et une discussion plus approfondie peut être trouvée dans [81].

L'algorithme de renormalisation pour l'arithmétique triple-double ne nécessite pas de branchements. Il ne demande que 9 opérations flottantes (additions) ce qui est significativement moins que l'algorithme de renormalisation proposé pour le format quad-double qui utilise au moins 24 additions et une boucle [63]. Le format triple-double donne donc une bien meilleure performance là où la précision complète de la quad-double n'est pas nécessaire.

Mais ce n'est pas que cette différence en termes de nombre d'opérations qui explique la bonne performance de la triple-double dans le cadre de l'implantation de fonctions mathématiques. C'est le fait que les renormalisations y sont employés très rarement. La section 3.4 donnera des mesures sur des cas pratiques.

Additions et multiplications Les opérateurs d'additions sont principalement basés sur l'addition exacte **Fast2Sum** [81]. Les composantes des opérandes sont additionnées exactement en ordre descendant de signifiance. Chaque étape produit une composante du triple-double résultat et un résidu qui est intégré dans la prochaine étape. À la dernière étape, les additions sont effectuées en arithmétique flottante double précision inexacte[1] [81].

Les opérateurs de multiplication commencent par calculer les produits partiels des composantes de leurs opérandes. Des multiplications exactes **Fast2Mult** [81] sont utilisées pour les produits de poids fort et milieu et des multiplications flottantes double précision sont employées pour les produits de poids faible. Les produits partiels de deux composantes de poids faible sont négligés. Les résultats intermédiaires sont ensuite sommés comme dans un opérateur d'addition [81].

L'adaptation de la précision au juste minimum et la compatibilité avec les formats double-double et double sont les avantages principaux de l'approche triple-double. Cette compatibilité doit se retrouver dans les signatures des opérateurs d'addition et de multiplication de la triple-double : ils sont définis sur toutes les combinaisons de formats d'arguments qui peuvent produire un triple-double. Les opérateurs les plus courants dans le cadre de l'implantation de fonction mathématiques sont donnés dans le tableau ci-dessous. Dans le tableau nous abrégeons précision double en D, double-double en DD et triple-double en TD.

[1] On l'appelle inexacte parce qu'elle engendre une erreur d'arrondi.

Opérateur	1^{er} arg^{t}	2^{nd} arg^{t}	res^{t}	flops - sans FMA		flops - avec FMA		
				$\oplus$	$\otimes$	$\oplus$	$\otimes$	FMA
Add133	D	TD	TD	16	0	16	0	0
Add233	DD	TD	TD	23	0	23	0	0
Add33	TD	TD	TD	24	0	24	0	0
Mul123	D	DD	TD	33	12	13	0	4
Mul133	D	TD	TD	34	13	14	1	4
Mul23	DD	DD	TD	55	19	19	1	6
Mul233	DD	TD	TD	97	31	35	1	10
Mul33	TD	TD	TD	79	28	31	4	8

Remarquons que les comptes d'opérations (en flops) dans la table ci-dessus ne reflète pas les vrais coûts des opérateurs sur machine à cause des effets de pipelinage dépendant des différents processeurs super-scalaires [78].

Une analyse de la latence minimale des opérateurs, c'est-à-dire de leur chemin critique, peut sembler plus pertinente qu'un compte d'opérations. Elle n'explique pourtant pas encore complètement la performance très bonne de l'application complète, qui est dans notre cas, l'implantation de fonctions mathématiques. En effet, cette latence est relativement grande pour les opérations triple-double : la partie basse d'un produit triple-double fois triple-double n'est par exemple disponible qu'après au moins 19 latences flottantes. Pourtant, nos codes utilisant la triple-double ne sont pas toujours 19 fois plus lentes qu'un code en précision double. La section 3.4 donnera plus de détails pour une réponse adéquate.

Toutefois, connaître les latences des opérateurs peut être intéressant pour d'autres applications que l'implantation de fonction mathématiques. Donnons donc le tableau suivant qui indique une comparaison pour quelques opérateurs d'addition et de multiplication. Nos indications supposent que l'opération FMA est disponible, qu'elle a une latence aussi grande que l'addition, qu'un nombre suffisamment grand d'unités parallèles est disponible pour qu'il n'y ait pas de dépendances de ressource et que toutes les composantes des opérandes (partie haute, moyenne et basse) sont disponibles au même cycle [93, 26, 78]. Sont indiquées les latences pour chacune des composantes du résultat.

Opérateur	1^{er} arg^{t}	2^{nd} arg^{t}	res^{t}	latence partie haute	latence partie moyenne	latence partie basse
Fast2Sum	D	D	DD	1	3	-
Add22	DD	DD	DD	5	7	-
Add33	TD	TD	TD	1	11	15
Fast2Mult	D	D	DD	1	2	-
Mul22	DD	DD	DD	5	7	-
Mul33	TD	TD	TD	1	17	19

Arrondi final Il ne suffit pas de calculer sur un triple-double pour l'arrondi correct, il faut aussi savoir l'arrondir correctement : $\star_{53}\left(x_{hi} + x_{mi} + x_{lo}\right)$ [81, 33]. À cause du manque de support matériel [34], ces opérateurs d'arrondi final sont implantés en logiciel. Les algorithmes sont principalement basés sur des tests et quelques manipulations de la représentation machine de nombres double précision. Les algorithmes, avec des preuves de correction

complètes, peuvent être trouvés dans [81]. À l'avenir, il convient de faire plus d'investigation s'il y a un moyen de rendre ces opérations libres de branchements. Une version optimisée pour l'arrondi au plus près peut être déduite des techniques présentées par Boldo et Melquiond dans [11].

Opérateurs divers Des opérateurs additionnels, comme une racine carrée avec un opérande double précision, produisant un résultat triple-double complètent notre suite de briques de base triple-double [33].

3.3 Une méthodologie pour l'analyse d'erreurs en triple-double

3.3.1 Bornes d'erreur d'arrondi relative pour les opérateurs triple-double

L'analyse traditionnelle d'erreurs d'arrondi en virgule flottante est basée sur une connaissance précise des bornes d'erreur de chacune des opérations de base dans l'application numérique sous analyse [64]. Dans quelques domaines applicatifs, des bornes très fines doivent être prouvées, comme par exemple pour l'arrondi correct des fonctions mathématiques [41] (cf. aussi chapitre 4).

Pour les précisions double et double-double, l'erreur ε de chaque opération est principalement bornée par une constante, la précision machine. Dans les preuves faites manuellement où à l'aide d'un outil de preuve comme Gappa [41] (cf. aussi chapitre 4), l'erreur maximale d'une opération de base ne dépend donc principalement pas de ces opérandes. Ceci simplifie l'analyse d'erreur.

En revanche, l'erreur (relative) ε d'un opérateur triple-double varie en fonction du chevauchement de ses opérandes triple-double. Dans les théorèmes de précision, l'erreur ε est donc bornée non par une constante mais par une expression en fonction du chevauchement maximal des opérandes. Nous fournissons un tel théorème de précision pour chacun de nos opérateurs triple-double [81, 33]. Par exemple pour la séquence **Mul233**, multipliant un double-double par un triple-double, le théorème se lit [81] :

Théorème 3 (Erreur relative de l'algorithme Mul233)
Supposons que les arguments $a_{hi} + a_{lo}$ *et* $b_{hi} + b_{mi} + b_{lo}$ *de l'algorithme* **Mul233** *satisfassent* $|a_{lo}| \leq 2^{-53} \cdot |a_{hi}|$, $|b_{mi}| \leq 2^{-\beta_o} \cdot |b_{hi}|$ *et* $|b_{lo}| \leq 2^{-\beta_u} \cdot |b_{mi}|$ *avec* $\beta_o \geq 4$ *et* $\beta_u \geq 4$.
Alors les valeurs renvoyées r_{hi}, r_{mi} *et* r_{lo} *vérifient*

$$r_{hi} + r_{mi} + r_{lo} = ((a_{hi} + a_{lo}) \cdot (b_{hi} + b_{mi} + b_{lo})) \cdot (1 + \varepsilon)$$

où

$$|\varepsilon| \leq 2^{-150} + 2^{-97-\beta_o} + 2^{-97-\beta_o-\beta_u}$$

Pour obtenir une valeur numérique pour la borne d'erreur d'une opération triple-double, qui peut être utilisée ensuite dans une analyse d'erreur traditionnelle, le théorème doit être instancié avec les bornes de chevauchement β_o, β_u du triple-double en opérande considéré. Il y a un théorème analogue pour chacun des opérateurs triple-double.

3.3.2 Théorèmes bornant le chevauchement

Conjointement avec les théorèmes de borne d'erreur, nous fournissons des théorèmes de majoration qui établissent une borne pour le chevauchement maximal dans un triple-double résultat d'une opérateur. Quand l'opérateur ne prend pas d'opérande triple-double, cette borne est statique, c'est-à-dire elle ne dépend pas des opérandes. Pour des opérateurs prenant au moins un argument triple-double, la borne de chevauchement est elle-même paramétrisée par la borne de chevauchement sur les opérandes triple-double. Par exemple pour l'opérateur **Mul233**, le théorème se lit [81] comme suit :

Théorème 4 (Chevauchement en sortie de l'opérateur Mul233)
Soit $a_{hi} + a_{lo}$ et $b_{hi} + b_{mi} + b_{lo}$ les argument de l'algorithme **Mul233**. *Soit $\beta_o, \beta_u \geq 4$ les bornes de chevauchement associées. Alors les valeurs retournées r_{mi} et r_{lo} ne se chevauchent pas du tout et le chevauchement de r_{hi} et r_{mi} est borné par $|r_{mi}| \leq 2^{-\gamma_o} \cdot |r_{hi}|$ avec*

$$\gamma_o \geq \min\left(48, \beta_o - 4, \beta_o + \beta_u - 4\right)$$

Pour les autres opérateurs, des théorèmes analogues sont disponibles. Le tableau suivant donne la liste complète des bornes de chevauchement et de précision pour les opérateurs prouvés dans [81] et implantés dans CR-Libm. Sont indiqués la borne d'erreur relative ε et les chevauchements γ_o et γ_u du résultat en fonction des chevauchements en entrée α_o et α_u du premier opérande respectivement de β_o et β_u du second opérande.

Opérateur	erreur $\lvert\varepsilon\rvert \leq$	chev^{t} $\gamma_o \geq$	chev^{t} $\gamma_u \geq$
Add133	$2^{-52-\beta_o-\beta_u} + 2^{-154}$	$\min\left(47, \beta_o - 2, \beta_o + \beta_u - 1\right)$	53
Add123	0	2^{-52}	53
Add233	$2^{-\beta_o-\beta_u-52} + 2^{-\beta_o-104} + 2^{-153}$	$\min\left(45, \beta_o - 4, \beta_o + \beta_u - 2\right)$	53
Add33	$2^{-\min(\alpha_o+\alpha_u,\beta_o+\beta_u)-47} + 2^{-\min(\alpha_o,\alpha_u)-98}$	$\min\left(\alpha_o, \beta_o\right) - 5$	53
Mul123	2^{-154}	47	53
Mul133	$2^{-49-\beta_o-\beta_u} + 2^{-101-\beta_o} + 2^{-156}$	$\min\left(47, \beta_o - 5, \beta_o + \beta_u - 4\right)$	53
Mul23	2^{-149}	48	53
Mul233	$2^{-97-\beta_o} + 2^{-97-\beta_o-\beta_u} + 2^{-150}$	$\min\left(48, \beta_o - 4, \beta_o + \beta_u - 4\right)$	53
Mul33	$2^{-151} + 2^{-99-\alpha_o} + 2^{-99-\beta_o}$ $+2^{-49-\alpha_o-\alpha_u} + 2^{-49-\beta_o-\beta_u}$ $+2^{50-\alpha_o-\beta_o-\beta_u} + 2^{-50-\alpha_o-\beta_o-\beta_u}$ $+2^{-101-\alpha_o-\beta_o} + 2^{-52-\alpha_o-\alpha_u-\beta_o-\beta_u}$	$\min\left(47, \beta_o - 4, \beta_o + \beta_u - 4\right)$	53

3.3.3 Analyse d'un code utilisant des opérateurs triple-double

L'analyse des erreurs d'arrondi d'un code donné utilisant des opérations triple-double consiste en deux étapes. Premièrement, des bornes de chevauchement sont récursivement calculées pour chaque triple-double dans le code. Ceci se fait à l'aide des théorème de chevauchement. Deuxièmement, les bornes d'erreur dans les théorèmes d'erreur relative sont instanciées et ensuite traitées comme dans l'analyse d'erreur d'arrondi traditionnelle [64, 41, 33, 81].

Illustrons ce calcul de chevauchements et de précisions à l'aide de l'extrait du code de la fonction $\text{expm1}(x) = e^x - 1$ dans CR-Libm donné au listing 3.1. Ce code évalue un polynôme

$p(x) = x + 1/2 \cdot x^2 + x^3 \cdot (c_3 + x \cdot (c_4 + t_6(x)))$ en arithmétique triple-double. On suppose que `t6h` et `t6l` contiennent une évaluation double-double de $t_6(x)$ ainsi que l'on ait `xSqh` + `xSql` $= x^2$ et `xSqHalfh` + `xSqHalfl` $= 1/2 \cdot x^2$. Dans l'exemple, l'évaluation du polynôme ne doit engendrer d'erreur plus grande que 2^{-130}.

```
1  Add123(&lPh,&lPm,&lPl,x,xSqHalfh,xSqHalfl);                                  // inf - 52/53 bstp
2  Mul123(&xCubeh,&xCubem,&xCubel,x,xSqh,xSql);                                 // 154 - 47/53 bstp
3
4  Mul123(&tt6h,&tt6m,&tt6l,x,t6h,t6l);                                         // 154 - 47/53 bstp
5  Add233(&t7h,&t7m,&t7l,C4h,C4m,tt6h,tt6m,tt6l);                               // 150 - 43/53
6
7  Mul133(&tt7h,&tt7m,&tt7l,x,t7h,t7m,t7l);                                     // 143 - 38/53
8  Add33(&t8h,&t8m,&t8l,C3h,C3m,C3l,tt7h,tt7m,tt7l);                            // 135 - 33/53
9
10 Mul33(&fHPhOver,&fHPmOver,&fHPlOver,xCubeh,xCubem,xCubel,t8h,t8m,t8l);       // 130 - 29/53
11
12 Renormalize3(&fHPh,&fHPm,&fHPl,fHPhOver,fHPmOver,fHPlOver);                  // inf - 52/53
13
14 Add33(&ph,&pm,&pl,lPh,lPm,lPl,fHPh,fHPm,fHPl);                               // 149 - 47/53
```

Listing 3.1 – Un extrait de CR-Libm.

L'analyse du code commence par le calcul de bornes de chevauchement pour toutes les valeurs triple-double produits par le code. Certaines de ces valeurs triple-double sont produites par des opérateurs triple-double qui ne prennent pas eux-mêmes des valeurs triple-double en opérande. Dans l'exemple, il y a trois de telles valeurs, `lPh` + `lPm` + `lPl`, `xCubeh`+`xCubem`+`xCubel` et `tt6h`+`tt6m`+`tt6l`. Pour ces valeurs, ou autrement dit, pour les opérateurs qui les produisent, la borne de chevauchement est une constante. Dans l'exemple, il est donc possible de lire les bornes de chevauchement, reportés en commentaire dans le programme d'ailleurs, directement dans le tableau donné à la section 3.3.2. Pour les autres valeurs triple-double, produites par des opérateurs prenant des opérandes triple-double en entrée, le calcul de la borne de chevauchement instantiée ne peut se faire qu'en connaissant déjà celle pour ses opérandes. Dans l'exemple, il est alors possible de borner le chevauchement supérieur γ_o de `t7h` + `t7m` + `t7l` par 43 qu'en connaissant celui de `t6h` + `t6m` + `t6l`. Celui-là, en revanche, a pu être calculé directement. Une passe récursive permet alors de propager l'information sur les bornes de chevauchement à l'aide des théorèmes de chevauchement. A la main (cf. aussi section 6.2.4), cette passe récursive se fait en pratique avant que des renormalisations explicites aient été rajoutées au code. Dans l'exemple, on obtient donc une valeur de 25 pour le chevauchement supérieur γ_o de `ph` + `pm` + `pl` après cette première passe.

Une fois qu'une borne de chevauchement est connue pour chacune des valeurs triple-double, il est possible calculer les borne d'erreur des opérateurs dont elles deviennent les opérandes. Dans l'exemple, on obtient donc, en instantiant le théorème correspondant, une borne d'erreur de 2^{-143} pour l'opérateur **Mul133** produisant `tt7h`+`tt7m`+`tt7l` en connaissant la borne de chevauchement $\gamma_o = 43, \gamma_u = 53$ pour `t7h` + `t7m` + `t7l`. Ce même calcul de bornes d'erreurs montre lors de la première passe, avant que des renormalisations explicites soient mises, si le chevauchement possible provoque des erreurs trop grandes. On obtient, par exemple, une erreur de 2^{-127} pour la dernière opération **Add33**si la renormalisation n'est pas faite ; ceci est une erreur trop grande dans l'exemple.

Si la borne d'erreur totale obtenue après cette analyse de précision est trop grande, des renormalisations explicites doivent être rajoutées au code. Ceci a été fait dans l'exemple. L'endroit où une renormalisation est rajoutée a une influence plus ou moins grande sur la borne d'erreur finale. Après l'avoir mise dans le code, un recalcula des bornes de chevauchement et des bornes d'erreur en aval de cette opération s'impose. Aucune étude n'a pour le présent

été faite sur l'optimisation du placement des renormalisations. Un algorithme glouton qui provoque un rajout d'une renormalisation le plus tard possible est actuellement utilisé en pratique manuelle et dans la génération automatique de codes (cf. section 6.2.4). Comme les précisions nécessaires pour l'arrondi correct dans CR-Libm sont avec 120 bits relativement faibles par rapport à ce que l'arithmétique triple-double peut fournir au maximum (à peu près 150 bits), le nombre des renormalisations est très faible dans les codes écrits dans le cadre de ce travail. Leur impact est donc minime, surtout quand la structure du code permet une parallélisation de la renormalisation avec d'autres opérations. La section 6.5.2 donnera des mesures pour expliciter ce point. Pour d'autre applications, ce nombre peut varier ; une étude reste à mener.

Remarquons toujours que si des valeurs doivent être tabulées dans une implantation de fonction mathématique, elles peuvent être précalculées de sorte qu'elle ne se chevauchent pas. Mentionnons aussi que les démonstrations des bornes de chevauchement sont généralement plus simples que celles pour les bornes d'erreur [81].

3.4 Gain en vitesse par l'arithmétique triple-double

Les implantations actuelles de fonctions mathématiques correctement arrondies dans notre bibliothèque CRLibm sont basées sur les arithmétiques double-double et double pour la première étape. Si l'arrondi nécessite un calcul plus précis, une étape précise est lancée qui, dans les approches précédentes était implantée en utilisant SCSlib. L'étape précise fournit alors une approximation d'à peu près 120 bits à la fonction, ce qui est suffisant pour l'arrondi correct [33, 40, 43].

L'utilisation de SCSlib empêchait non seulement la réutilisation de valeurs déjà calculées lors de la première étape mais aussi la tabulation de valeur ainsi que l'adaptation de la précision pendant l'évaluation du polynôme d'approximation.

L'approche nouvelle, que nous avons proposée, utilise l'arithmétique triple-double pour la phase précise. Il est ainsi possible d'utiliser le même algorithme dans les deux étapes, implanté une fois en double-double et une fois en triple-double. Une réutilisation de valeurs calculées et lues dans les tables est possible.

Le tableau suivant indique les temps mesurés pour la phase précise en triple-double par rapport à une implantation avec SCSlib. Les mesures sont données pour les fonctions `exp`, `log` et `asin` dans CRLibm. En plus, les mesures sont comparées à celles faites pour la bibliothèque `libultim` proposée par IBM [130]. Rappelons que cette dernière utilise de la multiprécision arbitraire. Les temporisations ont été faites sur un processeur IBM Power 5 en compilant avec gcc 3.3.3 sur un Linux 2.6.5.

Fonction	CRLibm triple-double	CRLibm SCSlib	`libultim` d'IBM
`exp`	1	10.3	109
`log`	1	7.66	857
`asin`	1	-	683

Dans le cadre ici donné, la performance de l'arithmétique triple-double a été uniquement évaluée dans une de ses applications, qui sont l'implantation de fonctions mathématiques. Ceci est un choix et il est discutable. Pourtant, il est difficile d'évaluer la performance propre

de nos opérateurs triple-double. Ceux-ci ont été écrit d'un façon qui permet une parallélisation de leurs opérations double précision de base tant que faire se peut par le compilateur employé. Employés dans des algorithmes plus ou moins parallélisables, les opérateurs s'exécuteront donc de façon plus ou moins parallèle dans les pipelines typiques actuels. Une mesure de temps d'une exécution d'un opérateur tombe en-dessous du seuil de précision de la temporisation. Quand il est employé dans une boucle, la mesure sera synthétique et nous doutons de sa représentativité, d'autant plus qu'une analyse statique de chevauchement sera difficile et que des renormalisations devraient être insérées dans un vrai code.

3.5 Conclusions sur l'arithmétique triple-double

Certaines applications nécessitent légèrement plus que deux fois la précision fournie par le format IEEE 754 double. Elles ne demandent pourtant pas non plus une précision très grande ni arbitraire et adaptable à l'exécution. Le format triple-double répond exactement à ces besoins. Par rapport à d'autres approches précédentes comme la quad-double, une nouveauté de notre approche triple-double consiste en une analyse statique du code pour économiser des renormalisation coûteuses. En plus, l'approche permet d'adapter la précision de calcul pour chaque endroit du code au strict minimum. Toutes les briques de base, incluant la renormalisation et l'arrondi final vers la double précision, sont prouvées. Pour des raisons de performances, elle n'utilisent pas de branchements sauf pour l'arrondi final.

Notre technique permet d'analyser des opérations triple-double avec des bornes d'erreurs variables. D'abord, chaque valeur est statiquement étiquetée avec une borne de chevauchement. Puis chaque opération est étiquetée par une borne d'erreur. Des renormalisations explicites sont insérées seulement là où la précision le demande. Pour chaque opérateur d'addition et de multiplication, un théorème de majoration du chevauchement et de l'erreur d'arrondi est disponible [33, 81].

L'arithmétique triple-double se comporte le mieux dans des applications où il est possible de profiter pleinement de son adaptabilité aux formats double-double et double ainsi que des économies de renormalisations par l'analyse statique. Elle a quelques désavantages comme son analyse d'erreur d'arrondi plus compliquée et fastidieuse (le chapitre 6 donnera des solutions) qui pourrait ne pas être utilisable sur tous les codes flottants. Il reste une question ouverte que de savoir si notre approche d'analyse statique de chevauchements et d'adaptation de la précision au juste minimum peut être appliquée à d'autres formats comme la quad-double. Peut-être les travaux comme [78, 92, 55], qui se rapprochent de nos résultats par une approche bien différente, y fournissent-ils des réponses plus appropriés.

Utilisée dans CRLibm, l'arithmétique triple-double a permis d'obtenir un gain en vitesse d'un facteur 10 par rapport aux approches précédentes utilisant la bibliothèque bien optimisée SCSlib [31]. Cette performance est maintenant comparable à celle obtenue en utilisant l'arithmétique double-étendue qui n'est pas disponible sur tous les systèmes [43]. Les phases précises pour l'arrondi correct des fonctions mathématiques usuelles ne sont donc jamais plus de 10 fois plus lentes que les phases rapides correspondantes.

CHAPITRE 4

Certification formelle de fonctions mathématiques

Mögen hätt' ich schon wollen, aber dürfen habe ich mich nicht getraut.

Karl Valentin, *dadaïste allemand*

Comme le chapitre 1 l'a montré, il est relativement simple de fournir l'arrondi correct si les informations de pire cas sont disponibles. Il suffit essentiellement d'approcher la fonction donnée à une précision plus grande que celle demandée par le pire cas. L'arrondi de l'approximation et celui de la fonction mathématique évaluée infiniment exactement sont alors égaux.

En pratique, la situation est plus compliquée. Avec peut-être un seul pire cas et une centaine de cas difficiles à arrondir demandant une grande précision entre approximativement 2^{53} à 2^{64} arguments à la fonction, comment pouvons-nous être sûrs que notre implantation fournit vraiment l'arrondi correct ? Avons-nous vraiment pris en compte toutes les éventualités et arcanes de l'arithmétique flottante ? Quelle erreur commettons-nous en remplaçant la fonction par un simple polynôme d'approximation ? Toutes ces questions demandent une réponse donnée avec une sûreté à la hauteur des exigences des applications bâties sur l'arrondi correct. Additionnellement, pour des raisons d'efficacité du développement des bibliothèques comme CRLibm, nous devons fournir ces réponses rapidement, donc quasiment automatiquement.

Ce chapitre discute les moyens que nous proposons afin de certifier des bornes sur les erreurs émanant de l'approximation et de l'évaluation dans nos implantations. La section 4.2 résume nos travaux sur la norme infini (norme sup) dans [24] ; la section 4.3 ceux sur la preuve formelle de code flottant et son automatisation, publiés dans [41, 42].

Avant de nous intéresser à la certification de bornes sur les deux sources d'erreurs principales séparément, définissons-les.

4.1 Introduction aux calculs d'erreur

4.1.1 Les deux sources d'erreur

Comme on a déjà vu à la section 1.1.3, dans une implantation d'une fonction mathématique f, cette fonction est – après réduction d'argument qui a réduit le domaine à un petit

intervalle I – remplacée par un polynôme d'approximation $p = c_0 + x \cdot (c_1 + x \dots)$ d'un certain degré avec des coefficients $c_i \in \mathbb{F}_t$. Ceci provoque l'erreur d'approximation $\varepsilon = \frac{p-f}{f}$.

Ce polynôme est ensuite implanté utilisant l'arithmétique flottante, par exemple comme $P = c_0 \oplus x \otimes (c_1 \oplus x \dots)$. Cette évaluation flottante du polynôme p est assujettie à des arrondis. Ceci provoque l'erreur d'arrondi $E = \frac{p-P}{p}$ [41, 42].

Les erreurs de ces deux étapes se combinent dans une erreur totale du nombre flottant retourné par le code sur l'argument x par rapport à la vraie valeur $f(x)$ [33, 41, 42].

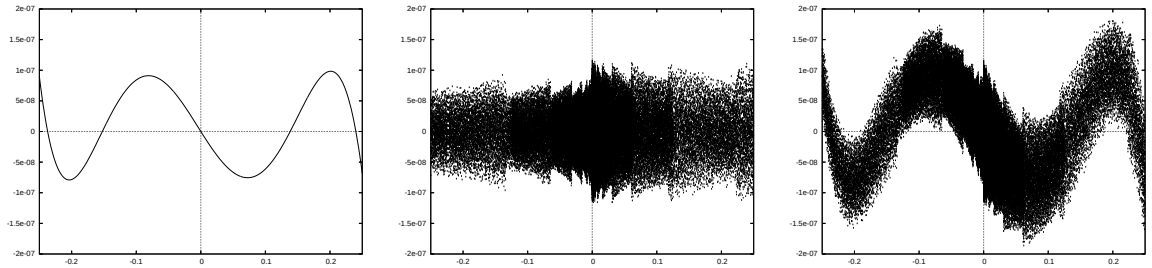
Prenons par exemple la fonction $f = \exp -1$. Supposons qu'une réduction d'argument a déjà réduit son domaine à $I = \left[-\frac{1}{4}; \frac{1}{4}\right]$ sans provoquer d'erreur. Cette fonction peut être approchée sur I par le polynôme

$$p(x) = x \cdot (1 + x \cdot (2097145 \cdot 2^{-22} + x \cdot (349527 \cdot 2^{-21} + x \cdot (87609 \cdot 2^{-21} + x \cdot 4369 \cdot 2^{-19}))))$$

Supposons alors que ce polynôme est implanté sous schéma de Horner en précision IEEE 754 simple :

$$P(x) = x \otimes (1 \oplus x \otimes (2097145 \cdot 2^{-22} \oplus x \otimes (349527 \cdot 2^{-21} \oplus x \otimes (87609 \cdot 2^{-21} \oplus x \otimes 4369 \cdot 2^{-19}))))$$

La figure 4.1 trace les erreurs d'approximation et d'arrondi ainsi que leur combinaison dans l'erreur totale.



(a) Erreur d'approximation ε (b) Erreur d'arrondi E (c) Erreur totale

FIG. 4.1 – Sources d'erreur en fonction de x

4.1.2 Défis liés aux calculs d'erreur

La qualité de l'implantation finale, correctement arrondie ou non, dépend principalement de son erreur totale [41, 33, 93, 42]. Les spécifications d'une implantation d'une fonction fixent souvent une borne pour l'erreur totale [41, 42, 40]. Dans le cas d'une implantation correctement arrondie, cette spécification découle directement de l'information de pire cas pour la fonction.

La démonstration que l'erreur totale d'une implantation est inférieure à une borne spécifiée est la clef de la preuve de correction de l'implantation. Comme la sûreté d'un système logiciel peut dépendre d'une unique implantation d'une fonction, la borne pour l'erreur totale doit être calculée d'une façon sûre. Pour une preuve valide, cette erreur totale ne doit en aucun cas être sous-estimée.

L'erreur totale est généralement majorée en considérant les différentes sources d'erreur séparément. Les erreurs liées à la réduction d'argument et à la reconstruction, qui ne seront pas considérées ici, peuvent être gérées de façon ad hoc [33, 101, 41, 42, 64].

L'erreur d'arrondi et l'erreur d'approximation restent le problème principal. Il s'agit de deux problèmes différents pour lesquels la situation en solutions disponibles n'est pas homogène.

- L'erreur d'arrondi E est une fonction discrète $\mathbb{F}_t \to \mathbb{R}$ ayant un comportement chaotique (cf. Figure 4.1(b)). Bien que l'analyse classique ne permette pas de la borner, plusieurs approches pratiques sont connues et peuvent être utilisées.
  Premièrement, une analyse manuelle des termes d'erreurs induites par chaque opération flottante [64] permet d'obtenir des bornes relativement fines et une sûreté assez satisfaisante. Deuxièmement, des approches utilisant des assistants de preuve formelle comme HOL, COQ ou PVS [60, 29] augmentent la sûreté au prix de preuves plus complexes et fastidieuses [41, 42]. Finalement, des outils comme Gappa permettent une analyse automatique des erreurs d'arrondi et leur vérification dans des assistants de preuve formelle [41, 42].
  Avec Gappa, l'analyse de la fonction d'erreur d'arrondi discrète devient relativement simple et extrêmement sûre. Nous allons expliciter l'utilisation de Gappa à la section 4.3.
- En revanche, l'erreur d'approximation $\varepsilon = \frac{p-f}{f}$ d'un polynôme p par rapport à une fonction continue et différentiable f donne également une fonction d'erreur continue et différentiable (cf. Figure 4.1(a)) dans les cas de la pratique. L'analyse classique est donc appropriée pour le calcul et la preuve d'une borne sur la norme infini (appelée aussi norme sup) $\|\varepsilon\|_\infty^I$. Ce problème peut alors sembler être simple.
  Toutefois, les approches et algorithmes précédents ne satisfont pas les besoins d'implantation sûre : la norme infini est soit sous-estimée soit l'approche est trop fastidieuse et spécifique à un cas.

4.2 L'approximation polynomiale et les normes infini certifiées

Nous allons donc nous intéresser d'abord aux approches connues pour borner une norme infini à la section 4.2.1. Cette étude permettra alors de spécifier les caractéristiques d'un nouvel algorithme dans la section 4.2.1. Nous proposons une implantation de cet algorithme répondant à ces spécifications dans la section 4.2.2. L'algorithme est capable de traiter des exemples rencontrés lors de l'implantation de fonctions mathématiques. La section 4.2.3 illustre ce point. Avant de donner nos conclusions sur l'algorithme à la section 4.2.5, nous monterons quelques de ses limitations à la section 4.2.4.

4.2.1 Analyse du problème et spécifications de l'algorithme

Travaux précédents

Les approches précédentes pour borner la norme infini $\|\varepsilon\|_\infty$ d'une fonction d'erreur $\varepsilon = \frac{p-f}{f}$ peuvent être classées dans deux catégories concernant leur incompatibilité avec une implantation sûre et complètement automatisée de fonctions mathématiques :

- Des techniques virgule flottantes proposées par Brent [13] peuvent retourner des sous-estimations à la norme infini. Nous les qualifierons alors d'*incertaines*. Les outils logiciels comme Maple, Matlab ou Mathematica implantent des telles algorithmes.
  Par exemple, reconsidérons la fonction $f = \exp -1$ et le polynôme p qui ont été donnés à la section 4.1.1.

Indépendamment de sa précision interne, Maple retourne $0.983491319532\cdots\cdot 10^{-7}$ pour $\left\|\frac{p-f}{f}\right\|_\infty^{\left[-\frac{1}{4},\frac{1}{4}\right]}$ (cf. section 4.2.3 pour plus de détails). Ceci est une sous-estimations d'au moins $1.8\cdot 10^{-17}$ parce que $\varepsilon(843485\cdot 2^{-22}) = 0.983491319722\cdots\cdot 10^{-7}$.
Les pannes logicielles importantes montrent que même des différences minimes ne doivent pas être négligées. Par le passé, nous avons utilisé Maple pour borner des fonctions d'erreur dans notre bibliothèque CRLibm pour laquelle nous prétendons assures l'arrondi correct [33] : c'était une faille (connue) de la preuve.

– D'autres approches augmentent la sûreté mais demandent plus de travail manuel fastidieux ou un temps de calcul très grand pour obtenir des bornes fines. Elles sont toutes basées sur un développement de Taylor p^* de la fonction f. La norme infini $\|p-f\|_\infty$ est alors majorée en utilisant l'inégalité triangulaire $|p(x)-f(x)| \leq |p(x)-p^*(x)| + |p^*(x)-f(x)|$. La borne sur $\|p^*-f\|_\infty$, c'est-à-dire sur le reste de troncature du développement p^*, est généralement montré à la main. Cela devient extrêmement difficile pour des composées des fonctions de base, d'autant plus quand ce processus doit être automatisé.
Krämer [76] donne une technique utilisée dans le développement de la bibliothèque FI_LIB. L'approche utilise l'arithmétique d'intervalle pour borner $\|p-p^*\|_\infty$. La borne sur le reste de troncature est démontré à la main. Le résultat peut être assujetti à un bogue dans l'implantation. Aucune preuve n'est produite. Les résultats ne sont pas très fins quand ils atteignent le niveau de la précision machine [65] : sur une machine double précision une certification d'une implantation double précision est donc impossible.
Harrison montre la correction d'une implantation de la fonction $\exp$ à l'aide de l'assistant de preuve HOL [60]. Sa preuve est très fastidieuse et ne résiste à aucun changement dans l'implantation. La borne sur le reste de troncature est démontrée manuellement et puis vérifiée automatiquement.
Des approches basées sur les modèles de Taylor permettent de calculer une borne sur le reste de troncature p^*-f. Des techniques ont été proposées pour PVS et d'autres outils basés sur les modèles de Taylor [29, 104]. Ils demandent souvent des calculs très coûteux [29].

Spécifications

Afin de pouvoir implanter une fonction f de façon sûre à l'aide d'approximation polynomiale p, on veut connaître une borne u telle que, pour point $x \in I$, l'erreur d'approximation $|\varepsilon(x)|$ n'est pas plus grande que u. La norme infini $\|\varepsilon\|_\infty$ serait meilleure réponse possible mais ce résultat peut rarement être atteint en pratique. Juste une valeur approchée sera disponible. On en déduit notre première requête concernant notre algorithme sûr pour le calcul de bornes de normes infini :

Requête 1 (Sûreté) *Quand l'algorithme ne peut pas retourner la valeur exacte de $\|\varepsilon\|_\infty$, il doit retourner une majoration u de cette valeur.*

Cette requête est essentielle pour la sûreté d'une implantation de fonction mathématique. En revanche, elle n'implique rien sur la qualité de la norme infini approchée u par rapport à la valeur réelle $\|\varepsilon\|_\infty$ qu'elle représente. Cela nous mène à notre deuxième requête :

Requête 2 (Qualité) *L'algorithme doit retourner une minoration ℓ de $\|\varepsilon\|_\infty$.*

Ainsi, on connaîtra un intervalle $[\ell, u]$ autour de la valeur exacte. L'algorithme implantant ces spécifications peut dépendre des quelques paramètres. Si l'intervalle $[\ell, u]$ est trop large, u peut être une majoration de $\|\varepsilon\|_\infty$ trop pessimiste. Dans ce cas, on peut relancer l'algorithme avec des paramètres mieux choisis afin d'obtenir une meilleure estimation de la valeur réelle de $\|\varepsilon\|_\infty$.

Remarquons que les techniques d'analyse numérique classique permettent d'exhiber un point x_0 où la norme infini est presque atteinte. Ainsi, la plupart du temps, n'est-il pas très difficile d'obtenir une minoration $\ell : |\varepsilon(x_0)|$ en est une bonne. La difficulté de notre problème vient donc du fait que nous voulons calculer une valeur $u \approx \|\varepsilon\|_\infty$ tout en garantissant qu'il s'y agit d'une majoration : $u \geq \|\varepsilon\|_\infty$.

L'algorithme pour le calcul de u sera probablement complexe et son implantation pourrait contenir des bogues. Pour cette raison, nous nous donnons une troisième charge pour garantir la sécurité du résultat :

Requête 3 (Preuve automatique) *En plus du résultat numérique $[\ell, u]$, l'algorithme doit retourner une preuve pour l'assertion $\|\varepsilon\|_\infty \in [\ell, u]$ laquelle pourra être vérifiée dans un outil indépendant (idéalement avec un assistant de preuve comme COQ ou PVS).*

À ce stade, il convient de faire deux remarques concernant la spécificité du contexte dans lequel nous calculons des normes infini. Premièrement, on remarque que l'algorithme devra soustraire $f(x)$ de $p(x)$, deux quantités qui, par construction, seront très proches l'une de l'autre :

Requête 4 (Cancellation importante) *L'algorithme devrait retourner des résultats d'une précision satisfaisante, même si p est une approximation excellente de f, c'est-à-dire quand $\varepsilon(x)$ est obtenu par une soustraction provoquant une cancellation importante.*

En pratique, il y a souvent un point $z \in I$ où l'expression $\varepsilon = (p - f)/f$ n'est pas définie parce que la fonction f a un zéro en z. Tout de même, le développeur d'une implantation pour f essaie toujours de maintenir l'erreur ε bornée dans un voisinage de z. Pour cette raison, p a presque toujours un zéro en z au moins du même ordre que f en pratique. Alors, la fonction ε peut être prolongée par continuité en z, même si l'expression n'est pas définie en z. Cela nous mène à notre dernière requête :

Requête 5 (Prolongement continu) *L'algorithme devrait être capable de retourner des résultats précis même si la fonction ε n'est définie en un point z que par continuité. Malgré cela, la sûreté de l'algorithme ne doit pas être compromise par cette requête : la valeur $+\infty$ est bien sûr meilleure qu'une valeur inférieure à $\|\varepsilon\|_\infty$.*

4.2.2 Un algorithme certifié de normes infini

Hypothèses

Afin de satisfaire ces requêtes, en particulier celle concernant la sûreté de l'algorithme, nous avons décidé d'utiliser une arithmétique d'intervalle avec précision ajustable [105, 3].

La propriété principale de l'arithmétique d'intervalle est appelée propriété d'inclusion : étant donné une fonction φ et un intervalle I, l'évaluation sur ordinateur de φ sur I en

arithmétique d'intervalle retourne un intervalle J tel que $\varphi(I) \subseteq J$. L'arithmétique prend les erreurs d'arrondi du calcul à sa charge pour garantir cette propriété d'inclusion. Elle produit ainsi des résultats mathématiquement valides. Plus loin, nous allons définir une procédure `eval` qui satisfera cette propriété d'inclusion pour l'évaluation de l'image d'une fonction sur un intervalle.

Dans la suite, nous allons toujours supposer que les intervalles manipulés sont des compacts. Quand I dénote un intervalle, $\overline{I}$ dénotera sa borne supérieure, $\underline{I}$ sa borne inférieure, $\mathrm{mid}(I) = (\overline{I} + \underline{I})/2$ son milieu et $\mathrm{diam}(I) = \overline{I} - \underline{I}$ sa largeur absolue.

De plus, nous allons supposer que f et $\varepsilon = (p - f)/f$ sont des fonctions continues et suffisamment de fois différentiables, données par des arbres d'expression. Notre algorithme manipule quelques premières dérivées des fonctions. Ces dérivées sont obtenues par différentiation symbolique. Ceci est choix de conception ; d'autres techniques peuvent être appropriées (cf. section 4.2.4).

Schéma général de l'algorithme

Notre algorithme est basé sur le théorème général suivant :

Théorème 5 *Soit φ une fonction différentiable sur un intervalle fermé $[a, b]$. Alors la fonction a un maximum sur $[a, b]$ et ce maximum est atteint*
- *soit dans une des bornes a ou b ;*
- *soit dans un point c tel que $\varphi'(c) = 0$.*

La même propriété est vérifiée pour le minimum.

Le principe de notre algorithme consiste en une application du théorème précédent à la fonction ε et en un encadrement sûr et rigoureux des zéros de la fonction ε' par une sous-procédure `boxZeros` (décrite à la section 4.2.2). Ce schéma général de l'algorithme est donné par le listing Algorithme 3.

```
1  Algorithme : CertifiedInfnorm
   Entrées : Une fonction ε et un intervalle fermé [a, b]
   un paramètre t pour contrôler la précision interne des calculs
   un paramètre Δ pour contrôler le diamètre maximal des encadrements des zéros
   un paramètre N pour le contrôle du degré maximal de récursion dans eval
   Sorties : Un intervalle [ℓ, u] tel que ‖ε‖∞ ∈ [ℓ, u]
2  début
3      Encadrer les zéros de ε' par B := boxZeros(ε', [a, b], Δ);
4      Rajouter les deux bornes : Z_left := [a, a] ; Z_right := [b, b] ; Z := {Z_left} ∪ B ∪ {Z_right};
5      pour tous les Z_i ∈ Z faire
6          Évaluer ε sur l'intervalle Z_i : Y_i := eval(ε, Z_i, t, N);
7      fin
8      Déduire un intervalle [ℓ, u] autour de la norme infini;
9      (Si demandé, produire une preuve du résultat);
10     renvoyer [ℓ, u];
11 fin
```

Algorithme 3 : Schéma général de l'algorithme de norme infini

Sous l'encadrement des zéros, nous comprenons le calcul d'une liste finie $\mathcal{B}$ d'intervalles disjoints $\mathcal{B} = \{Z_1, Z_2, \ldots\}$ tels que tout zéro de la fonction ε' appartient à un intervalle Z_i. Un paramètre Δ contrôle le diamètre maximal admissible pour un intervalle Z_i. Remarquons qu'il peut y avoir quelques intervalles $Z_i \in \mathcal{Z}$ qui ne contiennent aucun zéro de ε' et qu'un même intervalle Z_i peut contenir deux zéros distincts.

Le théorème 5 indique que les extrema de ε sont atteints soit aux zéros de ε' soit aux bornes de l'intervalle $[a, b]$. Pour cette raison, l'algorithme rajoute les deux intervalles points $[a, a]$ et $[b, b]$ à la liste $\mathcal{B}$ pour obtenir $\mathcal{Z}$.

Ensuite, l'algorithme évalue ε sur les encadrements. Cela veut dire que, pour tout $Z_i \in \mathcal{Z}$, l'algorithme appelle notre procédure `eval` qui retourne un intervalle Y_i tel que $\varepsilon(Z_i) \subseteq Y_i$. Deux paramètres, t et N, contrôlent la précision des résultats Y_i par rapport à $\varepsilon(Z_i)$. Les détails de cet algorithmes et la façon de laquelle les paramètres influencent sur le résultat final seront décrits à la section 4.2.2.

Voyons à présent comment les deux bornes ℓ et u, données pour encadrer le maximum de ε, peuvent être déduites de la liste des Y_i. La même méthode, non décrite, s'applique au minimum. Il est facile de déduire les bornes pour la norme infini à partir de celles pour le minimum et le maximum. Soit $x^\star$ un point où le maximum de ε est atteint. Par construction de la liste $\mathcal{Z}$, $x^\star$ est contenue dans un intervalle $Z \in \mathcal{Z}$.

Comme l'illustre la figure 4.2 et puisque $\varepsilon(Z) \subseteq Y$, la majoration $\overline{Y}$ est plus grande que le maximum de la fonction. Alors, le maximum de tous les $\overline{Y_i}$ est supérieur au maximum de ε sur l'intervalle $[a, b]$.

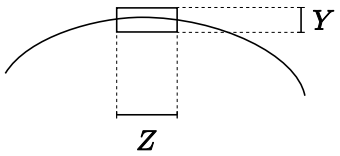


FIG. 4.2 – Autour du maximum

Remarquons que si $\mathrm{diam}(Z)$ est suffisamment petit, $\varepsilon(Z)$ l'est aussi. Alors, si le résultat Y de $\texttt{eval}(\varepsilon, Z, t, N)$ est suffisamment fin, $\mathrm{diam}(Y)$ est petit aussi. Alors la borne inférieure $\underline{Y}$ de Y est une bonne minoration du maximum (cf. Figure 4.2). Il s'ensuit que le maximum de tous les $\underline{Y_i}$ est une bonne minoration du maximum.

Ces techniques sont également connu comme le calcul d'un encadrement externe et interne de $\varepsilon(Z)$. Pour les détails techniques, Neumaier [105] est une référence.

En supposant que les procédures `eval` et `boxZeros` peuvent produire une preuve formelle de leur résultat (cf sections 4.2.2 et 4.2.2), on peut facilement obtenir une preuve pour le résultat de l'algorithme complet. Seule une preuve pour le théorème 5 et pour le calcul des encadrements externes et internes est nécessaire. La version actuelle de l'implantation de notre algorithme peut produire des preuves pour ses résultats. Ces preuves sont rédigées en langue anglaise seulement. Mais la méthode est conçue pour une génération future de preuves formelles.

Comme on verra, la procédure `boxZeros` a besoin d'évaluer des fonctions sur des intervalles. Nous expliquons donc d'abord la procédure `eval`.

Évaluation d'une fonction sur un intervalle

Dans la suite, nous allons présenter un algorithme `eval`(φ, I, t, N) qui calcule un intervalle J tel que $\varphi(I) \subseteq J$. Il prend en entrée la fonction φ, donnée sous forme d'expression. Il prend également comme argument l'intervalle I, une précision interne t (en bits) et un niveau de récursion N. Cet algorithme se base sur un autre algorithme `direval` d'évaluation directe d'intervalle, que nous détaillons d'abord.

Nous allons illustrer notre algorithme à l'exemple de la fonction définie par $\varphi(x) = \frac{\sin x^2}{x^2}$, de l'intervalle $I = [-0.5; 0.5]$, de la précision $t = 12$ et un niveau de récursion $N = 1$.

L'expression φ est construite de fonctions n-aires basiques ψ comme par exemple : $+$, $-$, $\times$, $/$, $\exp$, $\sin$, erf. Pour toutes ces fonctions ψ, nous disposons d'une procédure d'évaluation de base `baseeval`$(\psi, I_1, \ldots, I_n, t)$ qui calcule un intervalle J tel que $\psi(I_1, \ldots, I_n) \subseteq J$. Les bornes de l'intervalle J s'écrivent avec des nombres flottants, dont la précision est contrôlée par le paramètre t. Dans l'exemple, nous trouvons dans φ les fonctions de base $\psi_1 = \cdot^2, \psi_2 = \sin, \psi_3 = /$. Par exemple, `baseeval`$(\cdot^2, [-0.5; 0.5])$ donne l'intervalle $J = [0; 0.25]$ qui encadre l'image de la fonction carré sur l'intervalle $[-0.5; 0.5]$.

Une procédure `direval`(φ, I, t), qui calcule $J \supseteq \varphi(I)$, peut être construite comme suit : les arbres d'expression sont évalués récursivement d'en bas en haut en utilisant `baseeval` pour les fonctions de base. La correction de cet algorithme se déduit facilement de la propriété d'inclusion pour l'arithmétique d'intervalle [3]. Il suffit de faire une inductions sur l'arbre d'expressions φ.

Dans l'exemple, on a donc récursivement les évaluations suivantes :

$$\begin{aligned}
\texttt{baseeval}\left(\cdot^2, [-0.5; 0.5]\right) &= [0; 0.25] \\
\texttt{baseeval}\left(\sin, [0; 0.25]\right) &= \left[0; 2027 \cdot 2^{-13}\right] \\
\texttt{baseeval}\left(\cdot^2, [-0.5; 0.5]\right) &= [0; 0.25] \\
\texttt{baseeval}\left(/, \left[0; 2027 \cdot 2^{-13}\right], [0; 0.25]\right) &= [-\infty; \infty]
\end{aligned}$$

Chacun des intervalles J donnés en sortie encadre bien l'image de la fonction correspondante sur l'intervalle. Malheureusement, pour l'image de la fonction φ complète, l'encadrement est encore de taille infinie.

La bibliothèque MPFI[1] implante une procédure d'évaluation `baseeval` pour les fonctions usuelles. Nous utilisons cette bibliothèque.

Comme on a vu par l'exemple, cet algorithme naïf ne respecte pas la requête 5 donnée à la section 4.2.1. Dans le cas d'une division, `baseeval`$(/, J_1, J_2, t)$ retourne $[-\infty, +\infty]$ si l'intervalle dénominateur J_2 contient 0. Tout de même, dans une telle situation, il se peut que la fonction $\varphi = \theta_1/\theta_2$ puisse être prolongée par continuité dans un zéro $z \in I$ de θ_2 et qu'elle ait alors des bornes finies. Une variante de la règle de L'Hôpital pour l'arithmétique d'intervalle résout le problème. Nous verrons ensuite son application sur l'exemple.

Proposition 4.2.1 *Si θ_1 et θ_2 sont $\mathcal{C}^\infty$ sur I, qu'il existe $z \in I$ tel que $\theta_1(z) = \theta_2(z) = 0$ et que θ_1/θ_2 est tout de même $\mathcal{C}^\infty$ sur I alors*

$$\frac{\theta_1}{\theta_2}(I) \subseteq \left\{ \frac{\theta_1'(x)}{\theta_2'(y)}, \text{ avec } (x, y) \in I^2 \right\} = \frac{\theta_1'(I)}{\theta_2'(I)}.$$

[1] disponible sous `http://gforge.inria.fr/projects/mpfi/`

Démonstration Soit x un point dans I. Si $\theta_2(x) \neq 0$, par le théorème des valeurs intermédiaires, il est vérifié que :

$$\exists \xi_1 \in I, \exists \xi_2 \in I, \frac{\theta_1(x)}{\theta_2(x)} = \frac{\theta_1(z) + (x-z)\cdot\theta_1'(\xi_1)}{\theta_2(z) + (x-z)\cdot\theta_2'(\xi_2)} = \frac{\theta_1'(\xi_1)}{\theta_2'(\xi_2)}$$

Si $\theta_2(x) = 0$, on peut supposer que $x = z$. Si z est un point d'accumulation des zéros de θ_2, il est facile de voir qu'il est aussi un point d'accumulation des zéros de θ_2'. En particulier, par la continuité de θ_2', on a $\theta_2'(z) = 0$. Alors, $\theta_1'(I)/\theta_2'(I) = [-\infty, +\infty]$ et l'affirmation est vérifiée.

Si z n'est pas un point d'accumulation des zéros de θ_2, il existe un intervalle $\widetilde{I} \subseteq I$ où θ_2 n'a qu'un seul zéro z. Comme $\forall x \neq z \in \widetilde{I}$,

$$\frac{\theta_1(x)}{\theta_2(x)} \in \frac{\theta_1'(I)}{\theta_2'(I)}$$

par le théorème des valeurs intermédiaires et que $\theta_1'(I)/\theta_2'(I)$ est un intervalle fermée dans $\mathbb{R} \cup \{-\infty, +\infty\}$, on a

$$\left(\lim_{x\to z} \frac{\theta_1(x)}{\theta_2(x)}\right) \in \frac{\theta_1'(I)}{\theta_2'(I)}.$$

Dans l'exemple, nous avons $\theta_1(x) = \sin x^2$ et $\theta_2(x) = x^2$, dont la division a posé problème pour `baseeval`. Comme $z = 0 \in I$ est un zéro de θ_1 et θ_2, il est alors licite de remplacer $\varphi = \theta_1/\theta_2$ par $\varphi^* = \theta_1'/\theta_2'$, ce qui donne dans l'exemple $\varphi^*(x) = \frac{2\cdot x\cdot\cos x^2}{2\cdot x}$, qui – on verra plus loin – s'évalue déjà mieux.

L'application de la règle de L'Hôpital ne doit pas mettre en danger la sûreté de l'algorithme. Dans les cas usuels, il est facile de trouver un zéro $z \in \mathbb{F}_t$ par une itération de Newton-Raphson implantée en virgule flottante. Quoi qu'il en soit, comme ce zéro est calculé par un processus flottant non-certifié, le zéro z trouvé ne doit pas être utilisé toute de suite. Une preuve doit être établie que $\theta_1(z) = \theta_2(z) = 0$. La propriété d'inclusion de l'arithmétique d'intervalle en fournit la base : comme $\theta_i([z,z]) \subseteq \texttt{direval}(\theta_i, [z,z], t)$, si $\texttt{direval}(\theta_i, [z,z], t) = [0,0]$, il est vérifié que $\theta_i([z,z]) = [0,0]$ et que $\theta_i(z) = 0$. Si l'évaluation par intervalle de $\texttt{direval}(\theta_i, [z,z], t)$ ne permet pas de conclure, la règle ne sera pas appliquée. Dans ce cas, la borne sera infinie.

Dans l'exemple, le zéro $z = 0$ de $\theta_2(x) = x^2$ peut être trouvé sans problèmes par l'itération de Newton. L'évaluation de x^2 et $\sin$ étant exacte sur $[0;0]$ (l'intervalle de sortie est $[0;0]$), les conditions de la règle sont faciles à vérifier.

On peut critiquer notre approche pour l'application de la règle de L'Hôpital. Elle serait assujettie à trop de conditions qui sont toutes sous l'influence des surestimations engendrées par l'arithmétique d'intervalle sous-jacente. Sa justification est la suivante : bien qu'elle puisse ne pas marcher pour des fonctions arbitraires φ, cette heuristique est appropriée pour des fonctions $\varepsilon = \frac{p-f}{f}$ ou ε' que l'on rencontre lors de l'implantation de fonctions mathématiques f. Voir section 4.2.4 pour plus de détails.

L'algorithme complet `direval`, qui utilise la règle de L'Hôpital et qui permet d'évaluer des fonctions φ sur I est donné par l'algorithme 4.

Appliquons alors l'algorithme `direval` à notre exemple, pour l'illustrer. On a donc un appel $\texttt{direval}\left(\frac{\sin x^2}{x^2}, [-0.5; 0.5], 12\right)$. La fonction $\varphi(x) = \frac{\sin x^2}{x^2}$ étant composé en $\psi = /, \theta_1(x) =$

Algorithme : `direval`(φ, I, t)
Entrées : Une fonction φ donnée par un arbre d'expressions, un intervalle I et une précision t
Sorties : Un intervalle J tel que $\varphi(I) \subseteq J$
début
  si φ *est une feuille dans l'arbre* **alors renvoyer** `baseeval`(φ, I, t)
  sinon
    Soit $\theta_1, \ldots, \theta_n$ telles que $\varphi = \psi(\theta_1, \ldots, \theta_n)$ et soit $J_i =$ `direval`(θ_i, I, t)
    si ψ *est une division et* $0 \in J_2$ **alors**
      Calculer un zéro approximatif $z \in \mathbb{F}_t$ de $\theta_2(z)$ par Newton-Raphson
      Soit $T_1 =$ `direval`$(\theta_1, [z, z], t)$ et $T_2 =$ `direval`$(\theta_2, [z, z], t)$
      si $T_1 = [0, 0]$ *et* $T_2 = [0, 0]$ **alors renvoyer** `direval` $(\psi(\theta_1', \theta_2'), I, t)$
      sinon renvoyer $[-\infty, +\infty]$
    fin
    sinon renvoyer `baseeval`$(\psi, J_1, \ldots, J_n, t)$
  fin
fin

Algorithme 4 : `direval` - Évaluation de fonctions sur intervalles directe

$\sin x^2$ et $\theta_2(x) = x^2$, l'algorithme calcule d'abord $J_1 =$ `direval` $\left(\sin x^2, [-0.5; 0.5], 12\right) = \left[0; 2027 \cdot 2^{-13}\right]$ et $J_2 =$ `direval` $\left(x^2, [-0.5; 0.5], 12\right) = [0; 0.25]$. Ensuite, comme ψ est une division et que $0 \in [0; 0.25] = J_2$ l'algorithme cherche un zéro z de $\theta_2(x)$. En pratique, on a $z = 0$ pour l'exemple $\theta_2(x) = x^2$. L'algorithme continue par deux évaluations

$$T_1 = \texttt{direval}\left(\sin x^2, [0; 0], 12\right) = \texttt{direval}\left(\sin, \texttt{direval}\left(\cdot^2, [0; 0], 12\right), 12\right) = [0; 0]$$

$$\text{et } T_2 = \texttt{direval}\left(\cdot^2, [0; 0], 12\right) = [0; 0].$$

Comme T_1 et T_2 vérifient alors la condition d'être des intervalles points zéro, l'appel récursif, appliquant la proposition 4.2.1, peut se faire. L'algorithme s'appelle alors sur

$$\texttt{direval}\left(\frac{2 \cdot x \cdot \cos x^2}{2 \cdot x}, [-0.5; 0.5], 12\right).$$

Pendant l'évaluation de cet appel, le cas d'une division par un intervalle contenant zéro se reproduit encore une fois ; en effet, `direval` $(2 \cdot x, [-0.5; 0.5], 12) = [-1; 1]$. L'algorithme vérifie alors encore une fois la condition et s'appelle récursivement sur

$$\texttt{direval}\left(\frac{2 \cdot \cos x^2 - 4 \cdot x^2 \cdot \sin x^2}{2}, [-0.5; 0.5], 12\right).$$

Là, l'évaluation du dénominateur donne l'intervalle point $[2; 2]$ qui ne contient plus zéro. La division est donc possible : avec

$$\texttt{direval}\left(2 \cdot \cos x^2 - 4 \cdot x^2 \cdot \sin x^2, [-0.5; 0.5], 12\right) = \left[3461 \cdot 2^{-11}; 2\right],$$

on obtient $J = \left[3461 \cdot 2^{-12}; 1\right]$ comme encadrement sûr de l'image

$$\varphi\left([-0.5; 0.5]\right) = [0.989615...; 1].$$

La procédure `direval` ne respecte pas toujours la requête 4 : des effets de cancellation et de décorrélation [3] peuvent conduire à des surestimations importantes de $\varphi(I)$ en résultat de la procédure $\texttt{direval}(\varphi, I, t)$. En utilisant le théorème des valeurs intermédiaires, nous pouvons utiliser une forme centrée [3] dans une approche d'évaluation sur intervalle par développement de Taylor : en choisissant un centre $m \in I$, on a

$$\varphi(I) \subseteq \varphi([m,m]) + (I - [m,m]) \cdot \varphi'(I).$$

La procédure $\texttt{eval}(\varphi, I, t, N)$ évalue alors φ sur l'intervalle point $[m,m]$ par `direval` et s'appelle récursivement sur $\texttt{eval}(\varphi', I, t, N-1)$ jusqu'à ce que $N = 0$, auquel cas φ' est évaluée aussi par `direval`. Comme démontré dans [3], si le diamètre $\mathrm{diam}(I)$ de l'intervalle I est inférieur à 1, la surestimation de l'intervalle J retourné par rapport à $\varphi(I)$ décroît exponentiellement avec un N croissant.

Dans l'exemple, avec $N = 1$, on a donc une évaluation de $\varphi = \frac{\sin x^2}{x^2}$ sur $I = [-0.5; 0.5]$ comme

$$\varphi(I) \subseteq \left(\frac{\sin x^2}{x^2}\right)([0;0]) + [-0.5;0.5] \cdot \left(\frac{2 \cdot x^3 \cdot \cos x^2 - 2 \cdot x \cdot \sin x^2}{x^4}\right)([-0.5;0.5]).$$

A l'exemple, on voit déjà très bien que cette technique peut affiner les encadrement calculés mais que ceci a un coût et des limites : les expressions à manipuler grossissent rapidement.

Afin de permettre la génération d'une preuve, une trace des calculs dans `eval` est sauvegardée. Cette trace contient également toutes les informations concernant l'utilisation du Taylor sur intervalles et de la règle de L'Hôpital.

Encadrer les zéros d'une fonction

Pour encadrer les zéros d'une fonction φ dans un intervalle I, nous utilisons une dichotomie : nous évaluons d'abord ϕ sur I avec notre procédure `eval` et obtenons donc un intervalle J. Comme $\varphi(I) \subseteq J$, si 0 n'appartient pas à J alors φ n'a aucun zéro dans l'intervalle I.

Si, en revanche, on a $0 \in J$, ceci ne veut pas forcément dire que $0 \in \varphi(I)$, mais il y a un soupçon. Alors, dans ce cas, un pas de dichotomie est fait : nous coupons I en deux moitiés $I_1 \cup I_2$ et appelons la procédure `boxZeros` récursivement sur I_1 et I_2. Nous arrêtons ce processus quand le diamètre de l'entrée est plus petit qu'un paramètre Δ.

Afin de pouvoir générer une preuve du résultat, l'algorithme retient tout simplement les décisions prises au cours de l'algorithme. Il écrit alors des théorèmes de la forme $(\varphi(I) \subseteq J) \wedge (0 \notin J) \Rightarrow 0 \notin \varphi(I)$. La preuve que $\varphi(I) \subseteq J$ est donnée par `eval`.

Remarquons que l'algorithme dichotomique est un peu naïf. Un algorithme plus sophistiqué comme la version intervalle de l'itération de Newton (cf. [3]) pourrait également être utilisé mais nous ne l'avons pas encore implanté.

4.2.3 Exemples pour l'algorithme de norme infini

Présentons maintenant deux exemples qui montrent les résultats pratiques de notre algorithme. L'algorithme est intégré dans l'outil Sollya (cf. chapitre 5). Nous comparons les résultats de la version 1.0 de Sollya à Maple 10 (Build ID 190196). Nos expérimentations ont été faites sur un ordinateur équipé d'un processeur Intel Pentium 4 cadencé à 2.5 GHz avec GNU/Linux (kernel 2.6.19.2-ws #1 SMP i686).

Algorithme : `boxZeros`
Entrées : Une fonction φ ; un intervalle I ; un paramètre Δ
Sorties : Une liste $\mathcal{B}$ d'intervalles encadrant les zéros de φ
début
  $J := \texttt{eval}(\varphi, I, t, N)$;
  si $0 \in J$ **alors**
    si $\mathrm{diam}(I) < \Delta$ **alors renvoyer** $\{I\}$;
    sinon $I_1 := [\underline{I}, \mathrm{mid}(I)]$; $I_2 := [\mathrm{mid}(I), \overline{I}]$;
    renvoyer $\texttt{boxZeros}(\varphi, I_1, \Delta) \cup \texttt{boxZeros}(\varphi, I_2, \Delta)$;
  fin
  sinon renvoyer $\{\}$;
fin

Algorithme 5 : Comment encadrer les zéros

Nous avons utilisé la procédure `infnorm` de la bibliothèque `numapprox` de Maple. Nous avons fixé `Digits` à 100.

Nous utilisons une notation abrégée pour les résultats $[\ell, u]$ de notre algorithme : nous notons les décimales communes de ℓ et u suivies de l'intervalle de deux décimales suivantes possibles. Par exemple, quand le résultat de notre algorithme est $[0.123456, 0.1234789]$, nous écrivons $0.1234[5-8]$.

Exemple fil rouge : Considérons l'exemple décrit à la section 4.1.1. Soit f la fonction $\exp -1$, $I = [-0.25, 0.25]$ et p le polynôme donné ci-dessus. On obtient pour $\|\varepsilon\|_\infty^I$:

Maple	$0.9834913195329190\ldots\mathrm{e}-7$
Notre algorithme ($N=0$, $t=165$, $\Delta=2^{-27}$)	$0.98349131972[1-3]\mathrm{e}-7$
Valeur exacte	$0.9834913197221\ldots\mathrm{e}-7$

Comme on peut voir, le résultat donné par Maple est une sous-estimation. Cette estimation ne s'améliore pas quand `Digits` augmente. Ceci peut être vérifié en augmentant `Digits` au-delà des 100 chiffres décimaux. Contrairement à la croyance générale, les résultats de Maple ne semblent pas converger vers la valeur exacte quand `Digits` tend vers l'infini.

Cette sous-estimation peut dépraver la correction de l'implantation de la fonction f. Le résultat de notre algorithme donne à peu près le même nombre de chiffres décimaux corrects que Maple, mais il borne rigoureusement la valeur exacte de la norme infini ce qui permet d'avoir confiance en le résultat. En plus, en augmentant t et en diminuant Δ, nous pouvons obtenir des bornes de plus en plus fines de la valeur exacte.

Logarithme de CRLibm : Dans le deuxième exemple, nous considérons l'implantation de la fonction $f : x \mapsto \log_2(1+x)$ dans la bibliothèque CRLibm [33]. La norme infini de l'erreur

d'approximation $\varepsilon = (p - f)/f$ doit être calculée sur $[-1/512, 1/512]$ où le polynôme est

$$\begin{aligned} p(x) \quad = \quad & x \cdot \left(\frac{11704532700986780303630157415754 5}{2^{106}} + x \cdot \left(\frac{58522663504933901606981166592605}{-2^{106}} + \right. \right. \\ & x \cdot \left(\frac{8663094464742397}{2^{54}} + x \cdot \left(\frac{-6497320848515433}{2^{54}} + x \cdot \left(\frac{2598928339549937}{2^{53}} + \right. \right. \right. \\ & \left. \left. \left. \left. x \cdot \left(\frac{-541446114948727}{2^{51}} + x \cdot \frac{3712726891772213}{2^{54}} \right) \right) \right) \right) \right) \right) . \end{aligned}$$

Maple	$0.21506063319877\ldots\mathrm{e}-21$
Notre algorithme ($N = 2$, $t = 165$, $\Delta = 2^{-88}$)	$0.215060633232252001406277045[72-80]\mathrm{e}-21$
Valeur exacte	$0.215060633232252001406277045737382\ldots\mathrm{e}-21$

Maple retourne quasiment instantanément mais sous-estime la valeur exacte. Notre algorithme tourne pendant à peu près 320 secondes et produit un résultat sûr.

4.2.4 Limitations de l'algorithme

Notre algorithme de norme infini donné à la section 4.2.2 précédente souffre de quelques limitations. Ces limitations sont de différents types :

- Sur quelques instances pour $\varepsilon = \frac{p-f}{f}$, l'algorithme ne réussit pas à donner une borne finie pour la norme infini $\|\varepsilon\|_\infty$ à cause d'un manque de simplification symbolique. Les dérivées symboliques utilisées dans l'algorithme pour l'évaluation de Taylor et la règle de L'Hôpital peuvent contenir des sous-expressions qui se simplifieraient à zéro symboliquement mais qui, numériquement, sont la source d'instabilités et décorrélations. Un exemple typique est la sous-expression $\frac{\sin\sqrt{x}}{\sqrt{x}}$ évaluée sur un intervalle contenant 0. L'utilisation de la règle de Horner sans recourir à de la simplification symbolique ne permet pas de calculer des bornes finies.
- En revanche, l'utilisation de la différentiation symbolique peut être inappropriée. La taille des expressions qui représentent les dérivées successives d'une fonction peut croître exponentiellement. En particulier des termes fractionnels empêchent l'utilisation de dérivées de haut degré. Les fonctions $\varepsilon = \frac{p-f}{f}$ contiennent de telles fractions. La différentiation automatique [56] devrait être considérée pour résoudre ce problème.
- L'algorithme présenté a plusieurs paramètres : la précision t de l'arithmétique d'intervalle, la diamètre Δ maximal des encadrements des zéros et le niveau N de récursions dans l'évaluation intervalle de Taylor. Ces paramètres ont tous une certaine influence sur la finesse du résultat $R = [l, u]$. Bien que des utilisateurs expérimentés sachent trouver des valeurs pour les paramètres pour lesquelles l'algorithme se comporte bien, leur influence peut être trop imprévisible en général. En particulier, on peut observer des changements abrupts de la finesse de R en fonction de Δ.
- La génération de preuves souffre encore de quelques problèmes. À part le fait qu'elle ne s'interface pas encore directement avec un assistant de preuve formelle, la taille des preuves peut être trop importante. Les preuves énumèrent explicitement toutes les évaluations par arithmétique d'intervalle de toutes les fonctions de base, toutes les différentiations symboliques, toute simplification etc. En l'occurrence, pour l'exemple fil rouge (cf. section 4.2.3), à peu près $45\,000$ théorèmes et lemmes sont écrits. Il manque un moyen de simplifier cette preuve a posteriori.

4.2.5 Conclusions sur l'algorithme de norme infini

L'implantation de fonctions mathématiques f demande de borner l'erreur d'approximation $\varepsilon = \frac{p-f}{f}$ d'un polynôme p par rapport à f. Les approches préexistantes avant sont insatisfaisantes. Comme solution, nous avons donné un algorithme sûr pour borner la norme infini $\|\varepsilon\|_\infty$ d'une fonction continue et différentiable ε.

Dans notre cadre, les fonctions $\varepsilon = \frac{p-f}{f}$ ont souvent un conditionnement très mauvais et présentent des difficultés autour des zéros de f. Notre algorithme peut surmonter les deux problèmes dans les cas pratiques. En particulier, il utilise un algorithme d'évaluation par intervalle basé sur une arithmétique à précision ajustable. Il combine l'évaluation par intervalle de Taylor avec des heuristiques pour l'utilisation de la règle de L'Hôpital. Ces heuristiques ne mettent pas en danger sa sûreté : l'algorithme démontre automatiquement les conditions nécessaires.

Notre algorithme peut générer une preuve en langue anglaise pour chaque instance. De cette façon, il se valide soi-même. La preuve certifie qu'aucun bogue éventuel n'a compromis sa correction. Il s'y agit d'une première étape : notre but à terme est l'interfaçage avec un assistant de preuve formelle.

Notre algorithme a quelques limitations. En l'occurrence, quelques fonctions demandent l'utilisation de simplifications symboliques. Des travaux futurs attaqueront cet angle.

L'implantation de notre algorithme a déjà été utilisé sur des problèmes réels. Il a pu être utilisé sur tous les problèmes apparus lors du développement de fonctions de CRLibm sans difficulté particulière.

4.3 Certification de bornes d'erreur d'arrondi à l'aide de Gappa

Nous venons de voir un algorithme automatique qui permet le calcul certifié de bornes sur l'erreur d'approximation dans une implantation. Dans ce qui suit, nous allons motiver et utiliser l'outil Gappa[2] développé par Guillaume Melquiond durant sa thèse [96] pour le calcul et la preuve de bornes sur la deuxième source d'erreurs, les erreurs d'arrondi. Nous allons voir aussi comment Gappa sert de support pour intégrer toutes les sources d'erreur afin d'arriver à une preuve complète de la correction d'une implantation.

Comme déjà dit, cette section reprend nos travaux publiés dans [41, 42] ; certains aspects sont résumés, d'autres sont mieux explicités.

4.3.1 Preuves de propriétés de codes flottants

Le comportement des nombres flottants

Comme nous avons déjà vu, le comportement des nombres en virgule flottante et des nombres réels diffèrent souvent. Les nombres flottants ne sont pas une simple approximation des réels qui ne peut jamais être exacte. Illustrons ce point avec la séquence d'opérations flottantes donné ci-dessous, due à Dekker [36]. Nous en avons déjà fait usage au chapitre 3. Nous l'y avons considéré comme une boîte noire vérifiant certaines propriétés, définies quasiment par intention. Regardons maintenant comment ces propriétés peuvent être exhibées et démontrées pour une séquence d'opération flottantes maintenant explicitement donnée.

[2] `http://lipforge.ens-lyon.fr/www/gappa/`

L'approche pour cette petite preuve nous permettra ensuite d'attaquer des codes plus complexes.

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

Listing 4.1 – L'algorithme **Fast2Sum**.

Cette séquence ne consiste qu'en trois opérations. La première calcule la somme flottante de deux nombres `a` et `b`. Si cette somme était exacte, la deuxième retournerait toujours `b` et la troisième 0. À cause de l'arrondi, la somme est pourtant souvent inexacte. En arithmétique IEEE754 avec l'arrondi au plus proche pair et sous certaines conditions, cet algorithme calcule en `r` l'erreur commise lors de ce premier arrondi. Autrement dit, il vérifie que $\mathtt{r} + \mathtt{s} = \mathtt{a} + \mathtt{b}$ en plus du fait que `s` est le nombre flottant le plus proche de $\mathtt{a} + \mathtt{b}$. L'algorithme **Fast2Sum** nous fournit donc un moyen de représenter exactement la somme de deux nombres flottants sur une paire de deux nombre flottants. Cela est une opération très utile.

Cet exemple illustre un point important qui est en fait le point clef pour la preuve de propriétés sur les flottants : les nombres flottants peuvent être une approximation des réels dont le comportement peut différer des propriétés les plus basiques des réels, mais ils forment aussi un ensemble très bien défini des nombres rationnels. Cet ensemble est ainsi régi par d'autres propriétés bien définies sur lesquelles il est possible de construire des preuves mathématiques comme la preuve de correction de l'algorithme **Fast2Sum**.

Alors, revenons maintenant à la condition qui doit être satisfaite pour que l'algorithme **Fast2Sum** marche. Cette condition est que l'exposant de `a` est plus grand ou égal à celui de `b`, ce qui est en l'occurrence vérifié par exemple si $|\mathtt{a}| \geq |\mathtt{b}|$. Afin de pouvoir utiliser cet algorithme, on doit donc d'abord prouver cette condition. Remarquons que des alternatives pour l'algorithme **Fast2Sum** existent pour le cas où on ne peut pas prouver cette condition. La version de Knuth [73] demande 6 opérations au lieu de 3 pour cela. Ainsi, savoir prouver la condition, qui est une propriété des valeurs flottantes du code, permet d'améliorer la performance du code.

La preuve des propriétés de la séquence **Fast2Sum**, c'est-à-dire de trois opérations flottantes, demande plusieurs pages de démonstrations [36], et est, en fait, actuellement hors de la portée de l'outil Gappa. Cela est typiquement dû au fait qu'elle ne peut pas être réduite à une manipulation de domaines et d'erreurs. Ceci n'est pas un problème car cet algorithme a déjà été prouvé avec des assistants de preuve [30]. Nous la considérons alors comme une brique de base pour des programmes flottants plus larges et nous nous intéressons à la preuve de ces derniers. Dans le cas d'une utilisation du **Fast2Sum**, ceci implique une preuve de ses conditions.

Dans notre cadre, la classe de programmes flottants complexes, qui a motivée ce travail, est facile à définir : l'implantation de fonction mathématiques.

Fonctions mathématiques

Les implantations actuelles de fonctions mathématiques usuelles en virgule flottante [52, 121, 120, 91, 101] ont plusieurs particularités qui rendent leur preuve difficile :

– La taille du code est trop importante pour qu'une preuve manuelle soit possible. Dans les premières versions de la bibliothèque CRLibm [33], la preuve complète manuellement rédigée d'une seule fonction demandait une dizaine de pages. Il est difficile d'avoir confiance en de telles preuves.

– Le code étant optimisé pour la performance, avec une utilisation extensive d'astuces virgule flottante comme le **Fast2Sum** ci-dessus, les outils classiques d'analyse réelle et flottante ne peuvent pas être appliqués directement. Souvent, considérer les mêmes opérations comme sur des nombres réels serait complètement dépourvu de sens.
– Le code est censé évoluer pour des raisons d'optimisations futures. D'une part de meilleurs algorithmes peuvent être trouvés. D'autre part, la technologie des processeurs flottants ne cesse de se développer. De tels changement impliqueront la nécessité de re-rédiger de nouveau la preuve, ce qui à la fois fastidieux et source d'erreurs.
– La plupart des connaissances nécessaires pour prouver une borne d'erreur sur le code sont implicites ou cachées, soit derrière la sémantique du langage de programmation utilisé – qui définit du parenthésage implicite par exemple – soit dans les différentes approximations faites dans le code. Alors, la traduction seule d'un bout de code en un ensemble de variables mathématiques qui représentent les valeurs manipulées par ce code est fastidieux et également source d'erreurs si faite à la main.

Heureusement, les implantations flottantes de fonctions mathématiques ont aussi des propriétés qui sont faciles à appréhender et qui rendent leurs preuves faisables :

– Il y a une définition simple et claire de l'objet mathématique que le code flottant est censé approcher. Ceci ne sera pas toujours le cas pour d'autres codes, comme par exemple des codes pour la simulation numérique.
– La taille du code est suffisamment petit pour être traitable, typiquement de l'ordre d'une centaine d'opérations flottantes.
– Le flot de contrôle de ces codes est simple et consiste principalement d'un code linéaire avec quelques tests supplémentaires. Les boucles sont évitées.

Le problème avec le code efficace

Un code efficace est particulièrement difficile à analyser et à prouver à cause des techniques et astuces utilisées par les développeurs aguerris.

Beaucoup d'opérations flottantes sont exactes et le concepteur averti essaie de les utiliser. Comme exemple, on cite la multiplication par une puissance de deux, la soustraction de deux nombres d'ordre de grandeur similaires exacte grâce au lemme de Sterbenz [118], les algorithmes d'addition et de multiplication exacte retournant un double-double, la multiplication d'un petit entier par un flottant dont la mantisse contient suffisamment de zéros dans les bits de poids faibles, etc.

Le programmeur expert fera également de son mieux pour éviter des calculs plus précis que strictement nécessaire. Il enlèvera une opération flottante qui n'améliore pas substantiellement la précision du résultat final. Ce faisant, il commet ainsi une approximation supplémentaire qui devra être prise en compte dans la preuve. Pourtant, il est quelquefois difficile de savoir quelle valeur est une approximation de quelle autre valeur, en particulier puisque les calculs sont reparenthésés afin de maximiser la précision flottante.

Pour illustrer l'obfuscation de code qui en résulte, introduisons un bout de code qui nous servira d'exemple fil rouge pour le reste de cette section.

Exemple : une évaluation de polynôme en précision double-double

Le listing 4.2 est un extrait du code pour la fonction sin dans la bibliothèque CRLibm. Ces trois lignes calculent la valeur du polynôme impair

$$p(y) = y + s_3 \times y^3 + s_5 \times y^5 + s_7 \times y^7.$$

Ce polynôme, avec un coefficient de degré 1 à 1, ce polynôme ressemble à l'approximation de Taylor pour sin. Dans les équations pour notre implantation de sin, l'argument réduit y idéal est obtenu en soustrayant à l'argument flottant x un multiple entier de $\pi/256$. L'argument réduit est borné par $y \in [-\pi/512, \pi/512] \subset [-2^{-7}, 2^{-7}]$.

Pourtant, comme y est irrationnel, l'implantation de cette réduction d'argument doit retourner un nombre approchant y plus précis qu'un flottant double précision, sinon il n'y a pas d'espoir d'obtenir une précision plus grande que la double précision pour l'implantation complète de sin. Dans notre code, la réduction d'argument retourne donc un double-double $\texttt{yh} + \texttt{yl}$.

Afin de minimiser le nombre d'opérations, nous utilisons le schéma de Horner pour l'évaluation du polynôme :

$$p(y) = y + y^3 \times (s_3 + y^2 \times (s_5 + y^2 \times s_7)).$$

Pour un argument sur double-double, l'expression à calculer est donc

$$(\texttt{yh} + \texttt{yl}) + (\texttt{yh} + \texttt{yl})^3 \times (s_3 + (\texttt{yh} + \texttt{yl})^2 \times (s_5 + (\texttt{yh} + \texttt{yl})^2 \times s_7)).$$

Mais le code réel utilise une approximation de cette expression : le calcul est suffisamment précis si toutes les étapes de Horner sauf la dernière sont effectuées en précision double. Donc y_l sera négligé pour ces itérations et les coefficients s_3, s_5 et s_7 seront stockés sur des nombres double précision notés `s3`, `s5` et `s7`. L'expression précédente devient :

$$(\texttt{yh} + \texttt{yl}) + \texttt{yh}^3 \times (\texttt{s3} + \texttt{yh}^2 \times (\texttt{s5} + \texttt{yh}^2 \times \texttt{s7})).$$

Si cette expression est évaluée avec le parenthésage ci-dessus, une précision très pauvre en résultera. Spécifiquement, l'addition flottante `yh+yl` retourne, par définition d'un double-double, juste `yh`. Alors l'information contenue dans `yl` sera complètement perdue. Heureusement, l'autre partie de l'évaluation de Horner a également un ordre de grandeur beaucoup plus petit que `yh`. Cela se déduit facilement du fait que $|y| \leq 2^{-7}$ et donc $|y^3| \leq 2^{-21}$. Le parenthésage suivant donne donc un algorithme plus précis :

$$\texttt{yh} + \left(\texttt{yl} + \texttt{yh} \times \texttt{yh}^2 \times (\texttt{s3} + \texttt{yh}^2 \times (\texttt{s5} + \texttt{yh}^2 \times \texttt{s7}))\right).$$

Ceci est la dernière version de l'expression ; seule l'addition la plus à gauche doit être précise. Nous utilisons alors pour elle la brique de base **Fast2Sum**, qui retourne la somme exacte sous la forme d'un double-double. Pour toutes les autres opérations, nous utilisons l'arithmétique machine de base, c'est-à-dire la double précision. Nous obtenons finalement le code donné au listing 4.2.

```
1 yh2 = yh * yh;
2 ts = yh2 * (s3 + yh2 * (s5 + yh2 * s7));
3 Fast2Sum(sh, sl,   yh, yl + yh * ts);
```

Listing 4.2 – Trois lignes de C

Résumons. Ce code implante l'évaluation d'un polynôme avec plusieurs couches d'approximation. En l'occurrence, la variable $\mathtt{yh2}$ approche y^2 à travers les couches suivantes :

- y est approché par $\mathtt{yh} + \mathtt{yl}$ avec une erreur relative de $\varepsilon_{\mathrm{argred}}$
- $\mathtt{yh} + \mathtt{yl}$ est approché par $\mathtt{yh}$ dans la plupart du calcul,
- $\mathtt{yh}^2$ est approché par $\mathtt{yh2}$ avec une erreur d'arrondi correspondant à la double précision.

En plus, le polynôme représente une approximation de la fonction $\sin$ avec une erreur d'approximation relative bornée par $\overline{\varepsilon}_{\mathrm{approx}}$ que nous supposons connue dans cette section. Nous avons vu à la section 4.2 comment cette borne a pu être calculée de façon sûre.

Alors la difficulté dans le calcul d'une borne fine de l'erreur totale commise dans une implantation d'une fonction mathématique consiste dans la nécessité de combiner toutes ces sources d'erreur sans en oublier et sans utiliser des bornes trop pessimistes lors de la combinaison des erreurs de plusieurs couches d'approximation. Le compromis typique ici est qu'une borne d'erreur plus fine représente un travail beaucoup plus considérable que celui nécessaire pour une borne moins précise. Remarquons que la preuve complexe d'une borne fine peut en revanche inspirer moins de confiance. Le lecteur peut se faire une idée de ce compromis en essayant de mettre en relation toutes les valeurs intermédiaires entachées d'erreur avec des intervalles et de propager ces bornes avec l'arithmétique d'intervalles. Dans beaucoup de cas, des bornes d'erreur plus fines seront obtenues quand les intervalles seront coupés en plusieurs sous-intervalles et traités séparément. En contrepartie, une explosion des cas à considérer fera en augmenter considérablement le coût. Ceci est une des tâches que l'outil Gappa, présenté ici, automatise.

Travaux précédents et reliés

Dans notre cadre l'implantation de fonctions mathématiques avec arrondi correct, la finesse de la borne d'erreur total obtenue pour un code flottant a une influence directe sur la performance finale. Plus précise est la borne, plus rapide sera le code. Dans notre approche à deux étapes, rapide et précise, ce n'est pas l'erreur effective de la étape rapide qui entre dans le test d'arrondi. C'est la borne d'erreur qui est codée dans ce test ; quand elle est trop imprécise, le test produira trop de faux négatifs : la phase précise sera lancée pour des cas où l'arrondi aurait pu se faire [43].

Dans le cadre de l'implantation de fonctions d'arithmétique d'intervalles, une borne imprécise provoque une croissance supplémentaire des intervalles [45].

Historiquement, les preuves écrites pour les versions de la bibliothèque CRLibm jusqu'à la version 0.8 [33] consistent typiquement, pour juste quelques lignes de code, en plusieurs pages de preuves écrites à la main et plusieurs pages de calcul en Maple. Ceci fournit bien sûr une documentation excellente et aide à maintenir le code mais l'expérience a montré que ce style de démonstration provoque un nombre non négligeable d'erreurs humaines. Bien que l'implantation du calcul d'erreur en Maple soit une première étape vers une automatisation du processus et aide à éviter des erreurs de calcul mental, elle n'empêche pas les fautes méthodologiques.

Il y a eu d'autres essais de preuve assistée par ordinateur d'implantations de fonctions mathématiques ou des codes flottants similaires. L'approche purement formelle de la preuve proposée par Harrison [60, 61, 62] est plus profonde que l'approche Gappa, puisqu'elle prend en compte, par exemple, également les erreurs d'approximation. Elle est pourtant accessible seulement aux experts de la preuve formelle et assez fragile quant aux change-

ments dans un code. L'approche proposée par Krämer et al. [65, 66] repose sur la surcharge d'opérateur et ne fournit pas de preuve formelle.

4.3.2 L'outil Gappa

L'outil Gappa a été développé par Guillaume Melquiond en même temps que cette thèse se faisait. Il a été conçu pour remplir le vide observé entre les méthodes manuelles et les méthodes purement formelles, demandant un travail de spécialiste avec un assistant de preuve formelle. L'outil Gappa a été très amélioré sous notre pression de trouver des moyens pour passer de codes très simples (quelques lignes) à des implantations de fonctions mathématiques.

Gappa étend le paradigme de l'arithmétique d'intervalle au domaine de la certification de codes numériques [28, 97]. Étant donné une description de la propriété logique sur les bornes d'expressions mathématiques, l'outil essaie de montrer la validité de cette propriété. Quand la propriété contient des expressions non bornées, l'outil calcul des intervalles-bornes tels que la propriété soit vraie. Par exemple, la propriété incomplète $x + 1 \in [2, 3] \Rightarrow x \in [?, ?]$ peut être donnée en entrée à Gappa. L'outil répond que $[1, 2]$ est un intervalle pour l'expression x tel que la propriété entière soit vérifiée.

Une fois que Gappa a atteint l'étape où il considère la propriété comme vérifiée, il génère une preuve formelle qui peut être vérifié par un assistant de preuve indépendant. Cette preuve est complètement indépendante de Gappa et sa validité ne dépend pas de la validité propre de Gappa lui-même. Elle peut être vérifiée mécaniquement en externe et être incluse dans des développement de preuves plus conséquents.

Représentation de la virgule flottante dans Gappa

La section 4.3.3 donnera des exemples pour la syntaxe de Gappa. Elle montera également que Gappa peut être appliqué à des expressions mathématiques plus compliquées que juste $x + 1$, en particulier à des approximations flottantes de fonctions mathématiques. Ici, nous nous intéressons d'abord à la question de savoir comment des expressions contenant des opérateurs flottants peuvent être exprimées en Gappa.

Gappa ne manipule que des expressions sur des nombre réels. Dans la propriété $x + 1 \in [2, 3]$, x est juste un nombre réel universellement quantifié et l'opérateur $+$ est l'addition usuelle sur les réels $\mathbb{R}$. L'arithmétique flottant est exprimée à travers des *opérateurs d'arrondi* ; ce sont des fonctions de $\mathbb{R}$ dans $\mathbb{R}$ qui associent à chaque nombre réel x sa valeur arrondie $\star(x)$ dans un format spécifique $\star$. Ces opérateurs suffisent pour exprimer des propriétés de codes basés sur la plupart des arithmétiques flottantes ou virgule fixe.

Les infinités et les NaNs ne font pas partie de cet formalisme : les opérateurs d'arrondi retournent une valeur réelle sans qu'il y ait une borne supérieure sur la valeur absolue des nombres flottants. Cela veut dire que ni des NaNs ni des dépassements de capacité vers le haut ne seront générés ni propagés comme ils le seraient dans l'arithmétique IEEE754. Pourtant, on peut toujours utiliser Gappa pour prouver des propriétés apparentées. En effet, il est facile de montrer qu'aucun dépassement de capacité et aucun NaN ne sera produit par une division par 0 dans le code : il suffit d'exprimer termes d'intervalles le fait que dénominateur ne peut pas devenir 0. Ce qui ne peut pas être prouvé, c'est la propagation correcte des infinités ou NaNs dans un code.

Le moteur de Gappa

L'évaluation basique intervalle d'une expression ne prend pas en compte les corrélations des différents termes dans l'expression. En conséquence, les encadrements intervalles sont trop larges pour être utiles. Ceci est particulièrement vrai pour des encadrements d'expressions d'erreurs, par exemple quand l'erreur absolue d'une expression $\star(a) - b$ d'une approximation $\star(a)$ par rapport à une valeur exacte b est à borner. L'option la plus simple consiste dans un calcul séparé d'intervalles encadrant $\star(a)$ et b suivi d'une soustraction en arithmétique d'intervalle. Pourtant, en réécrivant l'expression comme $(\star(a) - a) + (a - b)$ et en bornant $\star(a) - a$, ce qui est une simple erreur d'arrondi, et $a - b$ séparément avant de les sommer, un résultat beaucoup plus précis est souvent obtenu. Ces règles s'inspirent des techniques que les concepteurs de codes appliquent à la main pour certifier leurs implantations numériques.

Gappa inclut une base de données remplie de telles règles de réécriture. La section 4.3.2 montre comment l'utilisateur peut étendre cette base de règles avec des indications de preuve spécifiques à son problème. L'outil applique ces règles automatiquement, ce qui veut dire qu'il peut borner des expressions le long de plusieurs chemins d'évaluation. Comme chacun des intervalles obtenus pour chaque façon d'évaluer l'expression encadre l'expression initiale, leur intersection est aussi un encadrement sûr. Gappa maintient une structure de données représentant les chemins qui produisent, par intersection, l'encadrement le plus fin et abandonne les autres possibilités afin de réduire la taille de la preuve finale. Il peut arriver que l'intersection finale est vide. Ceci implique qu'il y a une contradiction entre les hypothèses de la proposition logique ce qui donne à Gappa la possibilité d'en déduire tous les autres propositions.

Une fois que les bornes demandées par l'utilisateur ont été prouvées, une preuve formelle est générée en suivant les chemins de calcul retenus.

Indications de preuve

Quand Gappa n'est pas capable de satisfaire le but de la propriété logique donnée, cela ne veut pas nécessairement dire que la propriété est fausse. Il peut juste vouloir dire que Gappa n'a pas assez d'informations sur les expressions qu'il essaie d'encadrer.

Dans ces situations, il est possible d'aider Gappa en lui fournissant des *indications de réécriture pour la preuve*. Ces règles de réécriture ressemblent celles contenues dans la base de données de Gappa pour les arrondis. En revanche, leur utilité est spécifique au problème particulier étudié.

Indications de preuve explicites Une indication de preuve explicite a la forme suivante :

```
Expr1 -> Expr2;
```

Elle est utilisée pour donner l'information suivante à Gappa : « Je crois pour une certaine raison que, si jamais tu avais à calculer un encadrement intervalle de l'expression `Expr1`, tu pourrais obtenir un intervalle plus fin en essayant d'encadrer l'expression mathématiquement équivalente `Expr2` ». Cette formulation floue s'explique mieux à l'aide des deux exemples suivants :

1. La « certaine raison » ci-dessus représente typiquement la connaissance qu'a le spécialiste humain. Il sait par exemple que les expressions `A`, `B` et `C` sont des approximations

différentes de la même valeur et qu'en plus, A est une approximation de B, qui, lui, est une approximation de C. Comme avant, cela veut dire que les expressions sont corrélées et que l'indication de preuve adéquate se lit

```
A - C -> (A - B) + (B - C);
```

L'indication préconise à Gappa de calculer d'abord des intervalles pour $A - B$ et $B - C$ et de les sommer pour obtenir un encadrement pour $A - C$ – il sera plus fin.

Comme il y a un nombre infini d'expressions de type B qui pourraient être insérées dans le membre droit de la règle de réécriture, Gappa ne peut pas juste essayer toute règle de réécriture possible quand il doit encadrer $A - C$. Donc Gappa n'essaie que quelques expressions pour B et l'utilisateur doit donner celles qui manquent. Heureusement, comme on le montera plus loin, Gappa infère d'habitude quelques expressions B utiles et il applique les règles de réécriture automatiquement. Une indication de preuve explicite de ce type n'est donc nécessaire que dans des cas très inhabituels. Ceci n'a pas toujours été le cas : les différentes versions de Gappa ont pour ainsi dire dû *découvrir* les indications de preuve habituelles que l'on fait constamment. Notre travail pratique avec Gappa a donc contribué profondément à cette base de donnée de réécritures automatiques.

2. Les erreurs relatives peuvent être manipulées de façon similaire. L'indication à utiliser dans ce case est

```
(A-C)/C -> (A-B)/B + (B-C)/C + ((A-B)/B)*((B-C)/C) {B <> 0, C <> 0};
```

Il s'y agit toujours d'une identité mathématique, comme on peut facilement vérifier. Les propriétés données en accolades indiquent à Gappa quand la règle est vérifiée : à chaque fois que Gappa veut l'appliquer, il doit d'abord monter ces propriétés.

Comme la première indication, cette deuxième est souvent nécessaire dans une preuve. Alors Gappa essaie encore d'inférer quelques expressions B utiles et les applique où possible sans aide le l'utilisateur.

3. Quand x est une approximation de MX et que l'erreur relative $\varepsilon = \frac{x-MX}{MX}$ de x par rapport à MX est connue à l'outil, x peut être réécrit comme $MX \cdot (1 + \varepsilon)$. Ce type d'indication est particulièrement utile en combinaison avec la suivante.

4. Quand des termes fractionnels comme $\frac{Expr1}{Expr2}$ sont manipulés où Expr1 et Expr2 sont intimement corrélés parce qu'il approchent l'un l'autre, la division par intervalle ne produit pas de résultat utilisable sur des intervalles larges. Dans ce cas, il est mieux d'extraire une partie corrélée A des deux expressions en écrivant $Expr1 = A \cdot Expr3$ et $Expr2 = A \cdot Expr4$. Comme on espère, Expr3 et Expr4 sont faiblement (ou pas du tout) corrélés, ce qui donne un encadrement fin $\frac{Expr1}{Expr2}$. L'indication se lit :

```
Expr1 / Expr2 -> Expr3 / Expr4  { A <> 0 };
```

Une indication de preuve fournie par l'utilisateur n'est correcte que si ses deux membres sont mathématiquement équivalents. Gappa teste cela automatiquement. Si le test échoue, Gappa en avertit l'utilisateur qui doit revoir la règle pour une erreur, par exemple une faute de frappe dans un nom de variable. Pour cette raison, il est même sûr d'écrire des indications de preuve très complexes : on ne peut pas introduire d'erreur tant qu'aucun avertissement n'est émis. D'ailleurs, des indications de preuve inutiles sont ignorées en tout silence dans

preuve formelle produite. D'écrire des indications de preuves inutile n'a donc d'autre conséquence que d'augmenter le temps d'exécution de Gappa qui essaie les appliquer en vain ; cela est complètement sans risque.

Remarquons que quand une expression représente une erreur entre deux termes, comme par exemple `x` $-$ `Mx`, le terme moins précis doit être écrit à gauche et le terme plus précis à droite. La raison principale pour cette règle est que les théorèmes de la base de données incluse dans Gappa sont écrits de cette façon. Cette convention empêche une explosion combinatoire du nombre de chemins de calcul à explorer pour un encadrement.

Indications de preuve automatiques Après avoir démontré plusieurs implantations de fonction mathématiques, nous avons découvert que certaines indications de preuves se répétaient dans chaque preuve. Il s'y agissait typiquement des trois premiers types énumérés dans ce qui précède.

Une nouvelle syntaxe d'indications de preuves fut alors introduite dans Gappa :

```
Expr1 ~ Expr2;
```

Elle se lit « `Expr1` approche `Expr2` ». Elle a comme effet d'insérer automatiquement des règles de réécriture pour les erreurs absolue et relatives entre `Expr1` et `Expr2`. Il se peut qu'alors que des indications inutiles soient introduites, ce qui est – nous le répétons – sans conséquences.

Étant donné cette méta-indication de preuve, Gappa applique, à chaque fois qu'il rencontre une expression de la forme `Expr1` $-$ `Expr3` pour quelque expression `Expr3`, la règle `Expr1` $-$ `Expr3` $\rightarrow$ (`Expr1` $-$ `Expr2`) $+$ (`Expr2` $-$ `Expr3`). De même, quand il rencontre une expression `Expr3` $-$ `Expr2`, il essaie de la réécrire comme (`Expr3` $-$ `Expr1`) $+$ (`Expr1` $-$ `Expr2`).

Remarquons que même si les méta-indications de preuve sont là pour indiquer à Gappa d'insérer automatiquement des règles de réécriture, elles sont elles-mêmes insérées automatiquement dans quelques cas. Par exemple, l'utilisateur n'est pas obligé de dire que `Expr1` approche `Expr2` si `Expr1` est l'arrondi de `Expr2`. Gappa insère également une méta-indication automatiquement quand une expression d'erreur absolue ou relative apparaît dans l'encadrement que Gappa est censé prouver.

Indications de preuves de dichotomie Finalement, il est possible d'instruire Gappa à découper quelques intervalles et à explorer ses chemins sur les sous-intervalles correspondants. Il y a plusieurs possibilités. Par exemple, l'indication de preuve suivante

```
$ z in (-1,2);
```

se lit « De meilleurs encadrements pourraient s'obtenir en séparant les sous-cas $\mathtt{z} \leq -1$, $-1 \leq \mathtt{z} \leq 2$ et $2 \leq \mathtt{z}$ ».

L'indication suivante trouve les points de découpage automatiquement par une dichotomie sur l'intervalle de `z` jusqu'à ce que le but de preuve correspondant à `Expr` soit satisfait pour tous les sous-intervalles de `z`.

```
Expr $ z;
```

Une approche interactive pour écrire des indications de preuve Gappa a évolué et inclut de plus en plus d'indication de preuve automatiques. Pourtant, le plupart des preuves applicatives de la vrai vie demandent toujours d'écrire des indications de preuve complexes et

spécifiques au problème. La tâche de trouver la bonne indication dont Gappa a besoin pourrait être très complexe et demander une maîtrise parfaite de sa base de données de théorèmes et des algorithmes de recherche de chemins. Heureusement, une façon beaucoup plus simple consiste à construire la preuve petit à petit et d'interroger l'outil en rajoutant et enlevant des buts de preuves intermédiaires à la preuve. La section 4.3.3 suivante montre cette approche à l'aide d'un exemple étendu. Mais avant cela, décrivons les contours de la méthodologie que nous utilisons pour prouver des implantations de fonctions mathématiques.

4.3.3 Une approche pour la preuve d'implantations de fonctions en Gappa

Comme dans toute preuve, le style est important dans le travail avec Gappa. Dans une preuve vérifiée formellement sur machine, un mauvais style ne mettra pas en danger la validité de la preuve mais il peut empêcher l'auteur de la preuve de venir à sa fin. Dans notre cadre de la bibliothèque CRLibm, il peut également empêcher l'acceptation des preuves formelles assistées par ordinateurs dans la communauté de concepteurs de codes.

Gappa n'impose aucun style et quand nous avons commencé sur ce sujet, aucune expérience précédente ne pouvait nous donner d'inspiration. Seulement après quelques mois d'utilisation de Gappa, nous avons trouvé notre *style de programmation* dans Gappa afin de rendre les preuves plus concises et lisibles. Notre méthodologie s'applique bien à la preuve d'implantations de fonctions mathématiques. Cette section sera dédiée à la décrire.

Cette méthodologie consiste en trois étapes auxquelles correspondent trois sections dans le fichier source Gappa.

- Premièrement, le code C est traduit en équations Gappa. La traduction se fait d'une façon qui assure que la preuve Gappa démontrera vraiment quelques propriété de ce programme et non d'un programme similaire. Ensuite des équations sont rajoutées qui décrivent ce que le programme est censé implanter. D'habitude, ces équations sont en correspondance avec le code.
- Puis, la propriété à démontrer est rajouté. Elle est habituellement de la forme

  ```
  hypothese -> proprietes,
  ```

  où les hypothèses sont des bornes connues sur les entrées ou sur des contributions d'erreur déterminées à l'extérieur de Gappa, comme les erreurs d'approximation (cf. aussi section 4.2).
- Finalement, les indications de preuve doivent être rajoutées pour aider Gappa. Cette partie est construite petit à petit de façon interactive.

Les sections suivantes détaillent ces trois étapes.

Traduction d'un code C virgule flottante en Gappa

Reconsidérons notre code C, auquel nous avons rajouté les constantes pour les coefficients du polynôme :

```
1 s3 = -1.6666666666666665741e-01;
2 s5 =  8.3333333333333332177e-03;
3 s7 = -1.9841269841269841253e-04;
4 yh2 = yh * yh;
5 ts = yh2 * (s3 + yh2 * (s5 + yh2 * s7));
6 Fast2Sum(sh, sl,   yh, yl + yh * ts);
```

Il y a un grand nombre d'arrondis dans ce code, donc la première chose à faire est de définir des opérateurs d'arrondi Gappa pour le mode d'arrondi utilisé dans le programme.

Dans notre exemple, nous utilisons la ligne suivante pour définir `IEEEdouble` comme abréviation pour l'arrondi-au-plus-proche-égalités-paires IEEE 754 ; ceci est le mode d'arrondi utilisé en interne dans notre bibliothèque CRLibm.

```
@IEEEdouble = float<ieee_64,ne>;
```

Alors, si le code C est lui-même suffisamment simple et propre, l'étape de traduction consiste seulement en une explicitation des opérations d'arrondi qui sont implicites dans le code source C. Pour commencer, considérons les constantes pour les coefficients `s3`, `s5` et `s7`. Elles sont données par des chaînes de caractères en base décimale que les compilateurs C que nous utilisons convertissent en des nombres flottants binaires double précision avec un arrondi au plus proche. Nous nous assurons alors que Gappa travaille sur les mêmes constantes que le code C compilé en insérant les arrondis explicites suivants :

```
s3 = IEEEdouble(-1.6666666666666741e-01);
s5 = IEEEdouble( 8.3333333333333177e-03);
s7 = IEEEdouble(-1.9841269841269841253e-04);
```

Puis, nous devons faire la même chose pour tous les arrondis derrière les opérations arithmétiques en C. Le rajout manuel de tous ces opérateurs d'arrondi serait pourtant fastidieux et source d'erreurs. La syntaxe Gappa serait en plus tellement différente de la syntaxe C ce qui dégraderait la confiance et maintenabilité. D'ailleurs, on devrait appliquer sans fautes les règles de parenthésage implicite bien définies par la norme C99 [68]. Gappa a donc une syntaxe particulière qui permet de faire cette tâche automatiquement. Les lignes Gappa suivantes :

```
yh2 IEEEdouble= yh * yh;
ts  IEEEdouble= yh2 * (s3 + yh2 * (s5 + yh2 * s7));
```

définissent les mêmes relations mathématiques entre leurs membres droit et gauche que le programme C ci-dessus. Bien sûr, cela n'est vrai que sous les conditions suivantes :

- toutes les variables C sont des variables double précision,
- la combinaison compilateur/ système d'exploitation/ processeur utilisé respectent les normes C99 et IEEE754 pour la double précision.

Finalement, nous devons exprimer le comportement de l'algorithme **Fast2Sum** dans Gappa. Pendant qu'en C il s'agit d'une macro ou d'un appel de fonction, nous préférons dans notre but d'une preuve d'ignorer cette complexité et d'exprimer tout simplement son comportement en Gappa. Il calcule – on l'a vu – la somme exact de deux nombres sur un double-double. Ici encore, nous devons faire confiance à une preuve externe de ce comportement [36, 73] :

```
r IEEEdouble= yl + yh*ts;
s = yh + r;                 #  le Fast2Sum est exact, donc sh + sl = yh + r
```

Remarquons ici que nous nous intéressons à l'erreur relative de la somme `s` par rapport à la valeur exacte de la fonction sin. Pour cela, le fait que la somme `s` doit être exprimée sur deux flottants double précision n'importe.

Il est plus important de voir que cela rajoute une condition supplémentaire pour que cette traduction de code soit fidèle : comme le **Fast2Sum** nécessite que l'exposant de `yh` soit plus grand ou égal que celui de `r`, nous devons prouver cela. Nous déléguons cette sous-preuve à Gappa en rajoutant simplement la précondition du **Fast2Sum** dans la conclusion du théorème à démontrer.

En somme, des segments de codes qui forment des séquences d'opérations sans branchements (straight-line programs) avec principalement des variables double précision, un ensemble de définitions Gappa correspondantes peut être obtenu facilement en remplaçant l'opérateur d'affectation C par `IEEEdouble=` dans le script Gappa. Ceci est une opération très sûre.

Définir des valeurs idéales

Le code donné est censé évaluer le sinus de son entrée. En effet, la propriété que nous voulons démontrer porte sur l'erreur relative de la valeur `s` par rapport à $\sin y$: en syntaxe Gappa, cela s'exprime par `(s - SinY)/SinY in ?`. Nous devons donc définir `SinY` formellement. En plus, afin de pouvoir dérouler la preuve, nous devons définir formellement ce que chacune des variables est censée calculer.

Dans ce but, nous essayons de définir une valeur « mathématiquement idéale » pour chacune des variables du code. Elle servira de référence par rapport à laquelle les erreurs seront calculées. Il y a quelque choix dans cette notion de l'idéal mathématique. En l'occurrence, quelle est l'idéal mathématique de la variable `yh2` ? Il pourrait être

- le carré exact de `yh` sans arrondi ou bien
- le carré exact de `yh` + `yl`, que `yh` approche, ou bien
- le carré exact de l'argument réduit idéal y, qui est d'ailleurs souvent irrationnel.

Dans notre preuve, nous choisissons comme valeur idéale à la fois pour `yh` + `yl` et `yh`, l'argument réduit idéal y que nous dénotons `My` dans le script Gappa. Il s'agit du nombre réel défini par la fonction de réduction d'argument de l'argument du code `x` et l'irrationnel π. De façon analogue, la valeur mathématique la plus pure que `yh2` approche sera notée `My2`, définie comme `My2` $=$ `My`^{2}.

Pour l'approximation flottante du polynôme, donnée par `s`, nous pourrions choisir, comme idéal mathématique, soit la valeur de la fonction $\sin$ que le polynôme approche, soit la valeur du polynôme évalué sur `My` sans les erreurs d'arrondi. Nous choisissons la deuxième façon puisqu'elle est syntaxiquement plus proche de `s`.

Considérons maintenant quelques conventions sur les noms des variables Gappa, qui représentent des valeurs arrondies ou idéales ou encore des erreurs. La première règle est évidemment de nommer les expressions Gappa qui représentent des variables C avec le même nom. Nous imposons dans cette convention également que ces nom commencent par une minuscule. Puis, les noms des variables Gappa qui représentent des termes mathématiquement idéaux commenceront par la lettre "`M`". Nous utilisons des noms de variables commençant par une majuscule pour les autres valeurs intermédiaires. Bien sûr, des variables correspondantes devraient avoir de nom apparentés et, si possible, explicites. Répétons qu'il s'y agit de conventions qui font partie du style de preuve et non de la syntaxe Gappa. En effet, le fait qu'un nom de variable commence par une majuscule ne donne aucune information à l'outil. Les noms apparentés non plus.

En l'occurrence, il sera pratique de définir une variable Gappa égale à `yh` + `yl` :

```
Yhl = yh + yl;
```

Définition de ce que le code est censé calculer

La définition de valeurs mathématiquement idéales se résume à indiquer à Gappa ce que le code C est censé mettre en œuvre. En l'occurrence la ligne pour `ts` peut être vue comme

une évaluation approchée d'un polynôme au point `My` :

```
My2 = My * My;
Mts = My2 * (s3 + My2 * (s5 + My2 * s7));
```

Là, nous avons gardé les coefficients du polynôme en minuscules : comme déjà discuté à la section 4.3.1, le polynôme ainsi définie appartient tout de même à l'ensemble des polynômes à coefficients réels et nous avons des techniques pour calculer à l'extérieur de Gappa une borne pour son erreur relative par rapport à la fonction qu'il approche (cf. section 4.2).

Le lien entre `ts` et le polynôme approchant la fonction sin s'exprime le mieux à l'aide de valeurs mathématiques idéales :

```
PolySinY = My + My * Mts;
```

En somme toute, `PolySinY` est le polynôme implanté sur machine avec les même coefficients `s3` à `s7` comme dans le code C mais évalué sans erreur d'arrondi et évalué dans `My` et non dans `yh` + `yl`, qui l'approche.

Une autre approche serait d'utiliser un polynôme de Taylor. Dans ce cas, l'erreur d'approximation serait donnée par le reste de Lagrange. Le polynôme idéal serait le polynôme de Taylor avec des coefficients de Taylor idéaux, notés avec un M, dont certains auraient des correspondants arrondis qui apparaîtraient dans le code. Gappa pourrait gérer également cette technique, qui semble moins pratique.

Une autre question cruciale est de savoir comment nous pouvons définir la fonction mathématique idéale réelle que nous approchons par notre code ? Gappa ne connaît pas de fonctions transcendantes, comme sin ou log en interne. Notre approche actuelle peut se formuler en français comme « $\sin(\texttt{My})$ est une valeur dont, tant que `My` est plus petit que $6.29 \cdot 10^{-3}$, notre polynôme idéal ne s'éloigne relativement pas plus que de $2.26 \cdot 10^{-24}$ ». En Gappa, ceci se traduit en quelques hypothèses dans la propriété à démontrer comme suit :

```
   |Yhl| in [0, 6.29e-03]
/\ |(PolySinY - SinY)/SinY| <= 2.26e-24
/\ ... # (plus d'hypotheses, voir ci-dessous)
-> epstotal in ?
```

Ici, l'intervalle pour `Yhl` est défini par la réduction d'argument. Cette borne a été calculée séparément ; elle vaut $\pi/512$ plus quelques marges pour l'inexactitude de la réduction d'argument.

De la même façon, le distance relative maximale entre la fonction sin et le polynôme sur cet intervalle a été calculée à l'extérieur de Gappa. Nous utilisons pour cela notre algorithme de norme infini avec génération d'une preuve présenté à la section 4.2. Il nous fournit une borne sûre exprimé par un théorème de la forme

$$|\texttt{Yhl}| \leq 6.29 \cdot 10^{-3} \Rightarrow \left| \frac{\texttt{PolySinY} - \texttt{SinY}}{\texttt{SinY}} \right| \leq 2.26 \cdot 10^{-24}$$

Nous l'injectons juste dans la preuve Gappa.

Quant au style de preuve, il est plus approprié de ne manipuler que des expressions nommées. Ceci est particulièrement vrai pour les termes d'erreur. Nous mettons donc des définitions de termes d'erreur comme les suivantes dans notre preuve Gappa :

```
Epsargred = (Yhl - My)/My;           # erreur de reduction d'argument
Epsapprox = (PolySinY - SinY)/SinY;  # erreur d'approximation polynomiale
Epsround = (s - PolySinY)/PolySinY;  # erreur d'arrondi dans l'evaluation polynomiale
Epstotal = (s - SinY)/SinY;          # erreur totale
```

Définition de la propriété à démontrer

Dans l'introduction précédente, le théorème à démonter, exprimé comme des implications en logique classique du premier ordre, se lit comme suit :

```
{
# Hypotheses
   |My| in [1b-200, 6.4e-03]
/\ |Epsargred| <= 2.53e-23
/\ |Epsapprox| <= 3.7e-22
# Buts de preuve
-> Epstotal in ?  # but principale de notre preuve
/\ |r/yh| <= 1    # condition pour la validite du Fast2Sum
}
```

Le script de preuve initial complet Gappa est donné ci-dessous. Il rajoute une définition plus précise de `yh` et `yl` qui déclare qu'il s'agit de deux nombres double précision formant une double-double non-chevauché. Le script rajoute également une borne inférieure à la valeur absolue de l'argument réduit. Elle a été obtenu à l'aide de l'algorithme Kahan [72, 114, 101]. Cette borne inférieure est importante elle nous permettra de garder la plupart des valeur loin de zéro, alors loin du domaine de subnormalisation ; les erreurs relatives resteront donc toutes petites et ne pourront pas devenir arbitrairement grandes.

```
@IEEEdouble = float<ieee_64,ne>;

# yh+yl est un double-double (appelons-le Yhl)
yh = IEEEdouble(dummy1);
yl = IEEEdouble(dummy2);
Yhl = yh + yl;       # Ci-dessous, il y a aussi une hypothese indiquant que yl<ulp(yh)

#--------------- Traduction du code C  --------------------------

s3 = IEEEdouble(-1.6666666666666665741e-01);
s5 = IEEEdouble( 8.3333333333333332177e-03);
s7 = IEEEdouble(-1.9841269841269841253e-04);
yh2 IEEEdouble=  yh * yh;
ts  IEEEdouble=  yh2 * (s3 + yh2*(s5 + yh2*s7));
r   IEEEdouble=  yl + yh*ts;
s             =  yh + r;   # pas d'arrondi, il s'agit du Fast2Sum

#-------- Definition mathematique de ce que nous approchons --------

My2 = My*My;
Mts = My2 * (s3 + My2*(s5 + My2*s7));
PolySinY = My + My*Mts;

Epsargred = (Yhl - My)/My;           # erreur de reduction d'argument
Epsapprox = (PolySinY - SinY)/SinY;  # erreur d'approximation polynomiale
Epsround = (s - PolySinY)/PolySinY;  # erreurs d'arrondi dans l'evaluation du polynome
Epstotal = (s - SinY)/SinY;          # erreur totale

#----------------------  Theoreme a demontrer -------------------------
{
 # Hypotheses
    |yl / yh| <= 1b-53
 /\ |My| in [1b-200, 6.3e-03] # borne inferieure garantie par l'algorithme de Kahan
 /\ |Epsargred| <= 2.53e-23
 /\ |Epsapprox| <= 2.26e-24

 ->

 # Buts de preuve
    |Epstotal| < 1b-67
 /\ |r/yh| <= 1
}
```

Listing 4.3 – Le fichier Gappa initial.

Avec une petite aide par l'utilisateur

Quand le script de preuve ci-dessus est donné en entrée à Gappa, il produira la sortie suivante :

```
Warning: no path was found for Epstotal.
Warning: no path was found for |r / yh|.

Results for |yl / yh| in [0, 1.11022e-16] and |Yhl| in [6.22302e-61, 0.00629]
        and |Epsargred| in [0, 2.53e-23] and |Epsapprox| in [0, 2.26e-24]:
Warning: some enclosures were not satisfied.
```

Cela veut dire que Gappa a besoin d'aide, sous la forme d'indications de preuve. Par où commencer ? Il y a plusieurs façons d'interagir avec l'outil afin de comprendre pourquoi il échoue.

- Nous pouvons rajouter des buts de preuves supplémentaires pour obtenir des encadrement des variables intermédiaires. En l'occurrence, quand nous rajoutons le but « `|My| in ?` », nous obtenons la réponse « `|My| in [0, 0.00629]` ».
  Gappa a su déduire cet encadrement de celui donné en hypothèse pour `Yhl` et de l'arbre d'expression pour `Epsargred` (cf. section 4.3.2). De manière analogue, nous pouvons vérifier que Gappa sait borner finement la valeur de `PolySinY` mais nous découvrons qu'il échoue avec un encadrement de `s`.
- Nous pouvons rajouter des hypothèses additionnelles et regarder quel progrès ils provoquent. En l'occurrence, en rajoutant un encadrement fictif pour `Epsround`, nous permettons à Gappa de terminer la preuve, grâce à ses indications de preuve automatiques.

De cette manière, il est possible de détecter l'endroit où le moteur de Gappa est perdu dans la preuve afin de lui fournir des indications de preuve pour continuer.

Dans notre cas, la meilleure chose à faire est d'exprimer toutes les couches d'approximation détaillées à la section 4.3.1. En équations Gappa, nous obtenons :

```
# Couches d'approximation sur s
S1 = yh + (yl + IEEEdouble(yh*ts));   # s sans l'arrondi final
S2 = yh + (yl + yh*ts);               # suppression de l'arrondi penultime
S3 = (yh+yl) + (yh+yl)*ts;            # rajout de yl qui avait ete neglige

Eps1 = (s-S1)/S1;
Eps2 = (S1-S2)/S2;
Eps3 = (S2-S3)/S3;
Eps4 = (S3-PolySinY)/PolySinY;
```

Remarquons encore une fois que toutes ces erreurs sont définies par rapport aux valeurs idéales et non l'inverse.

Nous pouvons ensuite rajouter des buts de preuve pour ces erreurs relatives nouvellement déclarées : Gappa ne saura borner aucune d'entre elles. Nous devons fournir des indications de preuve. Tout d'abord, nous pouvons aveuglement rajouter des indications qui expriment les couches d'approximation dans notre calcul :

```
yh ~ Yhl;
s~S1;
S1~S2;
S2~S3;
S3~PolySinY;
```

Une autre indication consiste à demander une dichotomie en zéro qui, en pratique, permettra à Gappa de travailler seulement sur des intervalles ne contenant pas zéro. Ceci évite des problèmes reliés à la division par zéro dans les termes d'erreur relative :

```
$ My in (0);
```

Ces indications supplémentaires n'améliorent pourtant pas la situation. Analysons-la alors plus en détail.

Considérons d'abord seulement Eps4, qui est le l'erreur relative entre S3 et PolySinY. Nous l'avons vu – Gappa est capable de borner ces deux valeurs. Et S3 et PolySinY sont des expressions polynomiales sans arrondi et avec des coefficients identiques. Alors la différences entre elles se réduit à la différence entre Yhl = yh+yl, utilisé dans S3 et My, utilisé dans PolySinY. Nous avons une mesure précise de cette différence : c'est Epsargred. L'indication que nous devons fournir à Gappa devrait donc exprimer Eps4 en fonction de Epsargred, laquelle, évaluée en arithmétique d'intervalle, donne un encadrement fin.

Voici une technique générique pour obtenir une telle indication. En commençant par l'indication Eps4 → (S3 − PolySinY)/PolySinY, qui ne fait que développer la définition de Eps4, nous la réécrivons petit à petir jusqu'à obtenir une expression qui fait intervenir Epsargred. Dans le listing Gappa reflétant cette indication de preuve, nous avons laissé, dans le but de ce tutoriel, les étapes de réécriture intermédiaires tout en les commentant.

```
Eps4 ->
#  (S3-PolySinY)/PolySinY;
#  S3/PolySinY - 1;
#  ((yh+yl) + (yh+yl)*ts) / (My + My*Mts)  - 1;
#  ((yh+yl)/My) * (1+ts)/(1+Mts)  - 1;
#  (Epsargred+1) * (1+ts)/(1+Mts)  - 1;
#  Epsargred * (1+ts)/(1+Mts)   +  1 * (1+ts)/(1+Mts)  - 1;
#  Epsargred * (1+ts)/(1+Mts)   +  (ts-Mts)/(1+Mts);
   Epsargred * (1+ts)/(1+Mts)   +  Mts*((ts-Mts)/Mts) / (1+Mts);
```

En considérant les ordres de grandeurs (cf. section 4.3.1), nous voyons que même une évaluation naïve de cette dernière expression par arithmétique d'intervalles sera très précise. En effet, Mts ainsi que ts sont très petits par rapport à 1. Donc la première expression est proche de Epsargred. La deuxième est l'erreur relative de ts par rapport à Mts, que nous suspectons être de l'ordre de 2^{-52}, multiplié par Mts$/(1 +$ Mts$)$, qui est borné par 2^{-14}. Nous avons donc une somme de termes petits qui devrait fournir un encadrement fin.

Même avec cette indication de preuve, Gappa n'est toujours pas capable de donner un encadrement fin pour Eps4. En rajoutant les buts de preuve supplémentaires « Mts in ? » et « ts in ? », nous observons que Mts est correctement encadré mais que ts ne l'est pas. Nous rajoutons donc une définition de l'erreur relative de ts par rapport à Mts :

```
EpstsMts = (ts - Mts) / Mts;
```

et nous répétons l'analyse de la situation et la rédaction d'une indication de preuve. Nous décrivons les différentes couches d'approximation entre ts et Mts, définissons des termes d'erreur intermédiaires pour eux et fournissons une indication pour mieux les borner. Ce processus nécessite alors, quant à lui, d'expliquer à Gappa comment on passe de My2 à yh2 à travers trois couches d'approximation.

Il dépasserait le cadre ici donné que de détailler ce processus ligne par ligne. Le script de preuve final Gappa est reproduit dans [42] et disponible dans les distributions actuelles de CRLibm [3].

[3] téléchargeable sous http://lipforge.ens-lyon.fr/www/crlibm/

En résumé

Le développement d'indications de preuve adaptées est la partie la plus gourmande en temps dans la rédaction d'une preuve en Gappa. Ceci est dû au fait que c'est là que l'intelligence du concepteur de code est nécessaire. Pourtant, nous espérons d'avoir montré que ceci peut être fait de façon incrémentale.

L'exemple choisi pour ce tutoriel sur Gappa est relativement complexe : sa preuve Gappa finale consiste de 100 lignes, dont la moitié sont des indications de preuve. La borne d'erreur totale trouvée pour `Epstotal` est de $2^{-67.18}$. Ici, le temps d'exécution est d'une demi-seconde sur une machine récente. Ce temps augmente considérablement quand des dichotomies plus profondes sont utilisées.

Quelques autres implantations de fonctions mathématiques sont plus simples. Nous avons pu écrire une preuve pour une implantation de la fonction logarithme [43] avec seulement quelques indications de preuve [41]. Une raison pour cela est le fait que la fonction logarithme sur la grille flottante ne s'approche jamais de très près de 0 ; seules des erreurs absolues doivent alors être manipulées dans la preuve qui en dévient plus simple.

Une fois Gappa a pu prouver tous les but d'une preuve, il peut générer une preuve formelle pour l'assistant de preuve Coq. Dans le cas de l'implantation présentée ici, cette preuve consiste en plus de 3000 lignes et peut être vérifiée par Coq en quelques minutes.

4.3.4 Conclusions sur l'outil Gappa

La validation de bornes d'erreur d'arrondi à un niveau d'abstraction très bas de l'arithmétique flottante, typique pour l'implantation prouvée de fonction mathématiques, a toujours été un défi puisque plusieurs sources d'erreur s'accumulent dans une erreur totale. Gappa est un assistant de preuve de très haut niveau spécifique et bien adapté à ce type de preuves.

À l'aide de Gappa, il est facile de traduire une fonction écrite en C en une description mathématique des opérations impliquées. La confiance que cette traduction est fidèle est haute. Il est également aisé d'exprimer de la connaissance mathématique implicite que l'on peut avoir sur un code et son contexte. Gappa utilise l'arithmétique d'intervalle pour manipuler des minorations et majorations de plages de valeurs et de bornes d'erreur d'un code numérique. Il gère la plupart des problèmes de décorrelation automatiquement grâce à des règles de réécriture connues et à un moteur de recherche explorant toutes des réécritures possibles d'une expression afin de minimiser la taille des intervalles encadrants obtenus. Toutes les étapes d'une preuve sont basées sur une bibliothèque de théorèmes qui permet à Gappa de traduire son processus de calcul en un script de preuve formelle qui peut être vérifiée mécaniquement par un assistant de preuve de bas niveau comme Coq.

Si Gappa manque d'une information pour terminer une preuve, l'utilisateur peut rajouter de nouvelles règles de réécriture dans le script Gappa. Ces règles peuvent être trouvées en interrogeant l'outil pendant le développement d'un script de preuve. Aussi est-il possible d'obtenir rapidement une preuve complètement validée inspirant une bonne confiance que la preuve prouve en effet le code C initial. Ce processus n'est pourtant pas instantané : il demande autant de connaissance et ruse d'écrire une preuve Gappa qu'il le faudrait pour une preuve sur papier. En revanche, beaucoup moins de travail est nécessaire puisque Gappa tient compte de tous les détails fastidieux (comme par exemple des termes négligés, des opérations exactes, des expressions reparenthésées etc.) une fois que le problème a été cor-

rectement décrit.

La distribution actuelle de notre bibliothèque de fonctions mathématiques correctement arrondie CRLibm contient plusieurs bouts de preuves Gappa reflétant plusieurs étapes de son développement [96]. Bien que ce développement ne soit toujours pas terminé, la version actuelle (0.9) est très stable et nous considérons une généralisation de l'outil pour les développements futurs dans CRLibm. En l'occurrence, la section 6.2.5 décrira une méthode de générer automatiquement la preuve Gappa d'un code généré en même temps.

4.4 Conclusions sur la certification de bornes d'erreur

Dans les sections précédentes, nous avons proposé des algorithmes et méthodologies pour attaquer le problème du calcul et de la cerification de bornes sur les deux sources principales d'erreur dans une implantation de fonction mathématique. Notre algorithme de norme infini résout le problème de la certification d'une borne sur l'erreur d'approximation qui a longtemps laissé ouverte par les bibliothèques, y compris CRLibm. Notre méthodolgie d'utilisation de l'outil Gappa permet de fournir rapidement une preuve formelle du calcul d'erreur d'arrondi. Nos connaissances dans l'exploitation de l'outil Gappa sont si avancées maintentant qu'ils nous est même possible d'automatiser la rédaction de preuves Gappa, comme le montrera la section 6.2.5.

La situation avant cette thèse et celle maintenant ne sont pas comparables : des preuves complexes, n'inspirant aucune confiance, plus ou moins liées au vrai code et publiées dans des documents partiellement illisibles [33, 31] ont été remplacées par des preuves complètement basées sur un calcul sur ordinateur. Elles accompagnent tout naturellement les implantations, voire sont générées en même temps qu'elles. Bien sûr, en pratique, on peut se demander ce qui est mieux : disposer d'une preuve à la main, bien rédigées en langage mathématique habituel et expliquées aussi bien que faire se peut et surtout bien visibles dans la documentation, ou bien avoir des preuves Gappa dans un sous-répertoire de la distribution dans lequel personne ne se perd jamais...

Certaines problématiques attendent encore à être traitées à l'avenir. Premièrement, il semble actuellement très difficile d'exprimer le fonctionnement d'une réduction d'argument en Gappa. Par exemple, les implantations utilisent souvent quelques manipulations de motifs de bit du stockage des nombres flottants afin d'aller plus vite. Il se peut que ce problème reste sans espoir. Deuxièmement, même si le temps d'exécution de l'algorithme de norme infini ne joue pas de rôle important tant qu'il reste en dessous de quelques jours, une amélioration de cette performance serait la bienvenue. Nous investigons déjà dans des techniques de différentiation automatique, qui semblent être très prometteuses.

CHAPITRE 5

Sollya - un outil pour le développement de codes numériques

Ces vérités d'une recherche si délicate et qui semblaient se dérober à la vue humaine, méritent bien d'être suivies de près ; cette partie de la philosophie est un microscope avec lequel notre esprit découvre des grandeurs infiniment petites.

Voltaire, *Newton II, 10*

Les chapitres qui précèdent, en particulier le chapitre 1, ont montré que le processus d'implantation de fonctions mathématique comporte une composante importante liée à l'expérimentation. L'espace de recherche que l'on peut parcourir avant d'arriver à un code satisfaisant est vaste. Les questions qui se posent nécessitent souvent une analyse manuelle, par exemple de fonctions et de bornes d'erreurs. Toujours dans le souci de la rapidité du développement, un outil de travail adapté et intégré, qui permette d'accéder rapidement à l'information analysée, s'avère nécessaire.

Une partie des travaux pour cette thèse a été consacrée au développement du logiciel Sollya. Sollya répond aux besoins d'assistance dans l'implantation. Il permet également son automatisation, décrite au chapitre 6. Il intègre une implantation de l'algorithme de calcul de norme infini présenté dans la partie 4.2.

Sollya est un projet logiciel partagé avec d'autres personnes de l'équipe Arénaire. Leurs sujets de recherche diffèrent de celui de cette thèse. C'est l'effort en développement logiciel qui est mis à leur disposition. Cela mène à une bonne réutilisation de code pour des problèmes résolus lors de cette thèse. Certains points, notamment concernant le langage Sollya, ont été établis conjointement avec les personnes intéressés. L'outil connaît déjà quelques utilisateurs extérieurs à l'équipe Arénaire, notamment à l'École Polytechnique et dans l'équipe DALI à Perpignan.

Dans ce chapitre, nous analysons d'abord en détail les besoins en assistance lors de l'implantation de fonctions mathématiques. Nous décrivons ensuite les principales fonctionnalités de l'outil avant de jeter un regard critique sur l'implantation de Sollya.

5.1 Analyse des besoins

La tâche d'implanter une fonction mathématique en arithmétique flottant consiste en plusieurs étapes. Ces sous-tâches peuvent être résumées comme suit.

1. Analyse du comportement de la fonction dans son domaine de définition. On s'y intéresse principalement à sa dynamique, ses zéros et pôles, ses valeurs particulières pour l'arrondi. Ceci implique de calculer et de tracer – sur ordinateur – la fonction que l'on est en train d'implanter. On retient les fonctionnalités évaluation, recherche de zéros et traçage.
2. Construction d'une réduction d'argument appropriée. On essaie d'exhiber une propriété algébrique de la fonction utile par rapport au comportement des flottants. Techniquement, des valeurs de tables doivent être calculés et formatés pour une implantation sur machine. Pour garantir une bonne précision des valeurs tabulées, l'utilisation d'une arithmétique multiprécision s'impose : on retient les fonctionnalités d'évaluation multiprécision et de langage de script pour un formatage adapté.
3. Calcul d'une approximation polynomiale avec une erreur cible. On doit déterminer le degré approprié du polynôme d'approximation et calculer ce polynôme. L'approximation polynomiale se fait d'abord avec des coefficients réels et doit passer ensuite aux polynômes sur les flottants. L'erreur d'approximation induite doit être analysée et bornée par des normes infini. On retient les fonctionnalités de calcul de polynômes minimax, de normes infini ainsi que du support pour les arrondis machine. Une extension vers les techniques d'utilisation de l'algorithme LLL pour déterminer des coefficients flottants [17] est souhaitable.
4. Conception d'un code évaluateur pour le polynôme d'approximation. Il y faut déterminer le schéma de l'algorithme et la précision nécessaire pour les différentes opérations arithmétiques. Une borne d'erreur d'arrondi pour le code produit doit être calculée et prouvée, de préférence à l'aide de l'outil Gappa. Même si la détermination du schéma est un travail manuel d'abord, l'outil pourrait fournir la base pour son automatisation.

Un support est nécessaire pour procéder facilement et rapidement à chacune de ces étapes. Cet outil doit être relativement flexible car même si les étapes se ressemblent, leur réalisation ne se fait pas toujours de la même façon à cause des comportements divers des fonctions considérées. L'aide logicielle doit permettre l'intégration de toute la chaîne d'étapes. Invoquer plusieurs outils divers, avec des formats d'entrée et de sortie différents, provoque une perte d'efficacité dans le développement.

Historiquement, pour la bibliothèque CRLibm, Maple remplissait le rôle de ce support. Maple offre une bonne partie des algorithmes numériques nécessaires, en particulier pour le calcul de valeurs de fonctions en grande précision, l'approximation polynomiale et le calcul de normes infini. Il permet de tracer des fonctions et il propose un langage de script pour les expérimentations. Tout de même, Maple n'est pas la solution idéale. Cela a trois raisons principales :

1. À la base, Maple est un logiciel de calcul formel. Bien sûr, quelques fonctionnalités de manipulation d'expressions de fonction peuvent être utiles, par exemple pour la détermination d'une réduction d'argument appropriée. Toutefois, la plupart des algorithmes de base nécessaires pour l'implantation de fonction sont de nature numérique.

Maple fournit ces briques de base numériques, qui sont l'évaluation de fonctions composées, la norme infini et l'approximation polynomiale. Même l'arithmétique multiprécision est disponible. Mais ces algorithmes sont souvent très mal spécifiés quant à leurs résultats, aux erreurs commises et à la certitude que l'on peut avoir dans leur implantation correcte. Souvent, leur seule documentation disponible est l'aide intégrée du système Maple. Leur performances laisse quelquefois à désirer. Par exemple, le temps de calcul d'un polynôme minimax en Maple croît énormément avec le degré du polynôme demandé ; traiter des cas de degré plus grand que 60 relève de l'exploit. Le support de Maple pour l'arithmétique flottante binaire est restreint ; Maple se base sur une arithmétique décimale – au moins dans son interface utilisateur. Comme Maple est un logiciel propriétaire, il n'est pas possible de remédier aux problèmes rencontrés en modifiant les algorithmes existants.

2. Maple est un logiciel propriétaire. Nous ciblons un support pour développer des implantations de fonctions mathématiques qui, elles, sont censées constituer des logiciels libres. Le script ayant servi à produire l'implantation peut être vu comme une validation. Il est distribué avec l'implantation mais alors exploitable seulement après l'acquisition d'un logiciel propriétaire. A fortiori, les approches d'implantation présentées ne peuvent être promues auprès d'un public plus large si elles dépendent d'un logiciel payant.
3. L'intégration d'une chaîne de travail complète pour l'implantation de fonctions est difficile en Maple. Le moyen principal pour étendre ses fonctionnalités est son langage script. Une intégration d'autres outils peut s'avérer difficile et inefficace en termes de performances.

Une fois Maple écarté comme support utilisable à long terme, l'utilisation et l'extension d'autres logiciels pourraient être envisagées. En listant les possibles candidats, cela s'avère tout de même difficile :

- Mathematica est aussi un logiciel propriétaire dont les algorithmes ne peuvent pas être soumis à une vérification.
- Matlab, avec son correspondant logiciel libre GNU Octave[1], ne supporte que de la virgule flottante double précision. La multiprécision est indispensable pour l'implantation de fonctions au-delà de la double précision. En plus, Matlab est spécialisé dans le calcul sur matrices. Il ne supporte pas de notion claire de fonctions en tant qu'objet dans son langage de script. Ceci pose des problèmes dans notre cas d'application.
- Macsyma, avec son homologue libre Maxima[2], ne fournit pas assez de support pour la virgule flottante non plus. Seule la double précision est supportée de base. Des extensions comme EULER[3] ne semblent pas suffire ; aucun algorithme de haut niveau, comme par exemple de calcul de norme infini ou d'approximation polynomiale n'est disponible.
- GiNaC fournit un support considérable pour la manipulation d'expression de fonction, le calcul formel. Les fonctions mathématiques transcendantes de base sont disponibles avec une arithmétique multiprécision à travers CLN[4]. Tout de même, CLN n'est pas aussi répandu et par conséquence testé que MPFR.

[1] `http://www.gnu.org/software/octave/`
[2] `http://maxima.sourceforge.net/`
[3] `http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/euler/index.html`
[4] `http://www.ginac.de/CLN/`

GiNaC est une bibliothèque C++ dont l'extensibilité a été prévue dès le départ. Son extensibilité s'arrête pourtant à ce niveau. Cet outil n'a pas d'interface interactive de haut niveau fournissant un vrai langage de script. Cela est nécessaire pour l'expérimentation autour d'une implantation.

– PARI/GP[5] est un système de calcul formel pour l'arithmétique. GP est un langage de script permettant l'expérimentation. Il est pourtant difficile à appréhender. La notion de fonction comme objet de base, qui peut être passé en argument à une fonction, n'y existe guère. PARI/GP fournit de l'arithmétique flottante multiprécision avec un support pour les fonctions mathématiques de base. L'évaluation de fonction composées y est possible. En revanche, le modèle de virgule flottante sous-jacent ne permet pas de gérer correctement la précision finale obtenue. Aucun algorithme de calcul de norme infini ou d'approximation polynomiale n'est disponible.
– SAGE[6] réunit plusieurs outils de calcul mathématique à travers des interfaces avec le langage Python. On y compte principalement les outils déjà énoncés, PARI/GP, Maxima, GNU Octave et encore Maple. Même si l'utilisation du langage de script évolué Python promet de premier abord une extensibilité facile, la structure complexe des interactions dans SAGE rend l'expérimentation très difficile. SAGE cible l'uniformisation de l'interface. Les problèmes rencontrés avec les arithmétiques flottantes de ses composantes ne sont pas résolus par SAGE.

Aucun de ces outils connus listés ne répond donc complètement aux besoins exhibés pour l'implantation de fonctions mathématiques. C'est pour répondre à ces besoins qu'a été développé Sollya.

5.2 Les principales fonctionnalités de l'outil Sollya

Le logiciel Sollya a déjà atteint un certain niveau de maturité. La description de toutes ses fonctionnalités dépasserait alors le cadre de ce document. Elles ont été documentées en détail dans [25]. La documentation est également disponible à l'intérieur de l'outil sous forme d'une commande d'aide.

L'utilisation de l'outil Sollya se fait habituellement dans une session interactive. À l'invite, l'utilisateur peut donner des commandes et expressions dans le langage de script Sollya. Ses instructions sont immédiatement interprétées par l'outil qui fournit des réponses dans le même langage. Ceci rend l'expérimentation dans l'analyse de fonctions à implanter agréable et rapide. Le script suivant en est un exemple. Il illustre le début d'analyse quand il s'agit d'implanter la fonction $f = \arcsin$ sur le domaine $dom = [-1; 1]$.

```
> f = asin(x);x
> dom = [-1;1];
> plot(f,dom);
> dirtyfindzeros(f,dom);
[|0|]
> difficultpoints = dirtyfindzeros(f/(diff(f) * x),dom);
> for X in difficultpoints do print(X,",",f(X));
-1 , (pi) / (-2)
1 , (pi) / 2
```

[5] http://pari.math.u-bordeaux.fr/

[6] http://www.sagemath.org/

Les fonctionnalités de Sollya peuvent être classées principalement en cinq catégories décrites dans la suite :

Langage de script Sollya est un interpréteur pour un langage de script Turing-complet. Les styles de programmation impératif et fonctionnel sont supportés. Le langage est typé au moment de son interprétation. Dans la liste de type de base figurent d'abord les types classiques des langages haut niveau comme les booléens, les chaînes de caractères, les listes et valeurs constantes. Puis, comme Sollya est un outil pour l'implantation de fonctions mathématiques, celles-ci forment également un type de base. Le typage est polymorphe, c'est-à-dire certaines commandes opèrent sur des objets de type différent et ajustent le type de leur résultat en fonction de ces entrées. Aussi, l'opérateur d'évaluation d'une fonction par exemple peut travailler soit sur un point donné soit sur tout un intervalle. Les listes sont également polymorphes. Elles sont formées d'objets de nature et type différents. Le typage implicite effectué seulement au moment de l'évaluation ainsi que sa polymorphie suivent une philosophie de langage de script adapté à des expérimentations rapides.

Manipulations formelles Bien que l'outil Sollya se veuille un outil de calcul numérique, il intègre quelques fonctionnalités nécessitant la manipulation formelle d'expressions mathématiques. Les expressions, vue comme fonctions, peuvent alors être par exemple dérivées formellement ou simplifiées dans leurs sous-expressions constantes. D'autres manipulations sont possibles sur des expressions définissant des polynômes. Entre autres, il est possible de les réécrire sous forme de Horner ou canonique, d'en calculer le degré et d'en extraire un coefficient particulier ou un sous-polynôme. Ces fonctionnalités formelles sont nécessaires pour une analyse de polynômes d'approximation et des fonctions d'erreur d'approximation associées.

Manipulation de valeurs en virgule flottante Nos implantations de fonctions mathématiques sont basées sur la virgule flottante normalisée IEEE 754. Les arithmétiques double précision et multi-double y jouent un rôle particulier. L'outil Sollya est lui-même basé sur de la virgule flottante multi-précision basée sur la bibliothèque MPFR [50, 124]. Il supporte nativement l'arrondi double précision, la construction de double-doubles et de triple-doubles ainsi qu'un arrondi générique à une précision donnée. D'autres commandes permettent d'arrondir tous les coefficients d'un polynôme à des formats flottants donnés. Des commandes d'affichage sont prévues pour pouvoir correctement transmettre les valeurs de tels coefficients flottants dans un code d'implantation de fonction ou la preuve Gappa associée.

Communication avec d'autres outils La communication de données avec d'autres logiciels externes à Sollya est supportée aussi. Premièrement, Sollya peut échanger des données sous forme de texte. Ces textes, produits en sortie par Sollya ou relus en entrée, peuvent être formatés dans des modes d'affichage divers. Ces modes sont adaptés aux logiciels comme Gappa, Pari/GP, Maple etc. Au formatage texte de base se rajoute la possibilité d'échanger des données dans le format XML normalisé Math-ML Content description [6]. Deuxièmement, Sollya supporte un mode de communication avec des logiciels externes qui consiste dans leur chargement dynamique. Ainsi, avec seulement quelques commandes, une fonctionnalité qui n'est pas disponible dans Sollya mais implantée par un programme externe

peut être intégrée dans l'outil. Ce support est également disponible pour des fonctions mathématiques manquantes à Sollya. Il est facile de les implanter en arithmétique d'intervalle à l'extérieur de l'outil et de les lier dynamiquement à Sollya.

Algorithmes numériques Les fonctionnalités numériques constituent la plupart des algorithmes implantés dans Sollya. Les plus importantes en sont les suivantes :

- Sollya permet la définition de fonctions composées à partir de fonctions de base ou d'opérateurs externes liés dynamiquement. Une opération de base sur ces fonctions composées est leur évaluation en un point. Sollya supporte cette évaluation avec arrondi fidèle en multi-précision arbitraire. Pour une fonction composée quelconque et un point donné qui n'est pas un zéro de la fonction, Sollya peut calculer une évaluation à un nombre de bits fixé d'avance. Si l'expression de la fonction provoque des cancellations qui réduisent le nombre de bits significatifs du résultat, l'outil adapte automatiquement la précision intermédiaire pour garantir que le résultat est un des deux flottants encadrant la valeur exacte de la fonction au point donné à la précision relative demandée.
  Cette garantie que, par défaut, toute fonction s'évalue avec une précision effective égale à celle demandée, est un avantage de l'outil Sollya en termes de sécurité. On est souvent amené à analyser par exemple des fonctions d'erreur, qui, par construction, provoquent des cancellations entre la fonction approchée et son approchant. Ces analyses peuvent être complexes à faire. Plusieurs phénomènes doivent y être pris en compte en même temps. L'évaluation avec arrondi fidèle aide alors à se concentrer sur ces points importants sans faire de mauvaises appréciations suite à une précision trop faible des calculs. Dans une certaine mesure, la manipulation de fonctions et de leurs valeurs dans Sollya s'approche alors d'une évaluation à valeurs réelles et non numériques. C'est cette fonctionnalité qui fait vraiment une différence avec les outils comme Maple et Mathematica. L'utilisateur ressent l'arithmétique non comme flottante mais comme réelle.
  L'algorithme derrière cette évaluation avec arrondi fidèle est simple. Il se base sur l'évaluation de fonctions composées en arithmétique d'intervalles. La fonction est évaluée avec une précision intermédiaire croissante jusqu'à ce que l'encadrement fourni par l'arithmétique d'intervalle soit suffisamment fin pour garantir un arrondi fidèle. Cette adaptation de la précision est faite de façon ad-hoc dans Sollya et peut être améliorée [77].
- La capacité de produire des résultats précis en adaptant automatiquement la précision de l'outil Sollya ne se restreint pas à l'évaluation de fonction. L'outil peut tracer des fonctions composées, assujetties à des cancellations numériques ou non, dans un domaine donné. Cette fonctionnalité, pour laquelle l'outil a été créé dans un premier temps d'ailleurs, est indispensable pour un développement manuel rapide et sûr d'implantation de fonctions mathématiques.
- L'outil intègre des implantations pour les différents algorithmes de calcul de norme infini présentés au chapitre 4.2. L'outil intègre donc un algorithme certifié et lent ainsi qu'un algorithme numérique approximatif et rapide.
- Sollya intègre une implantation de l'algorithme de Remez [110, 23, 112, 101] pour permettre l'approximation polynomiale. Cette implantation a été faite par Sylvain Chevillard dans le cadre de la coopération autour de l'outil. L'algorithme est constamment

utilisé dans le développement d'implantations de fonctions mathématiques. L'implantation de cet algorithme dans l'outil profite pleinement d'autres fonctionnalités de l'outil, en particulier de l'évaluation de fonctions composées avec arrondi fidèle. Une analyse plus précise de cette brique algorithmique importante se trouve au chapitre 6.3.
– L'outil intègre une commande spécialisée qui permet de produire un code C évaluateur de polynôme pour un polynôme, un domaine et une précision donnée. Cette commande fait partie des fonctionnalités de Sollya qui tendent à l'automatisation de l'implantation de fonctions mathématiques. Elle sera décrite en détail au chapitre 6.2. La commande est capable de générer automatiquement une preuve Gappa en même temps qu'une séquence d'évaluation est produite.

L'outil Sollya est toujours en développement. Certaines fonctionnalités manquent encore. La liste suivante en donne celles qui méritent probablement d'être rajoutées à l'outil en premier.
– Sollya devrait intégrer du support pour l'algèbre linéaire. Le logiciel se veut un outil numérique pour la résolution de problèmes mathématiques. Doté d'un langage complet et avancé, il donne la base pour ressoude des problèmes de plus en plus complexes avec des algorithmes implantés directement dans le langage de script. Plusieurs problèmes, par exemple l'interpolation, nécessitent un support pour l'algèbre linaire qui manque actuellement.
– Toute constante est considéré – et typée – comme une valeur flottante. Ceci crée de problèmes pour la gestion de nombres entiers. Par exemple, certains entiers très grands ne sont pas représentables dans Sollya quand la précision est choisie trop petite. Une boucle additionnant itérativement 1 à une variable compteur jusqu'à ce qu'une certaine limite soit atteinte peut ne pas terminer pour des raisons de précision flottante. Un support pour une arithmétique entière exacte, indépendante du calcul flottant, devrait être rajouté à l'outil.
– La syntaxe, le typage et la conception des algorithmes de base de l'outil ne permettent actuellement que de considérer des fonctions univariées. Pour étendre encore l'espace de recherche ou bien de traiter ne serait-ce que des fonctions comme x^y, Sollya devrait être modifié pour une gestion de fonctions multivariées. Cette modification devra éventuellement remettre en cause le cœur de l'architecture logicielle actuelle.

5.3 L'intégration logicielle de Sollya

Le logiciel Sollya est entièrement écrit en langage C. Le choix de langage est principalement dû à l'utilisation des bibliothèques suivantes, pour lesquelles un support dans des langages de plus haut niveau manque :
– la bibliothèque standard POSIX,
– les bibliothèques multiprécision MPFR et MPFI pour les arithmétiques flottante et d'intervalles,
– la bibliothèque de calcul sur entiers longs GMP,
– la bibliothèque libxml2 pour le support Math-ML et
– les outils flex et bison pour l'analyse lexicale et syntaxique du langage Sollya.

La structure logicielle de Sollya peut principalement être découpée en trois grandes parties, même si le code reste relativement monolithique au stade actuel de son développement. Sollya consiste en :

1. *Un module d'analyse lexicale et syntaxique du langage Sollya.* Ce module gère également l'interactivité du logiciel, c'est-à-dire l'affichage de l'invite, la gestion de commandes d'interruption de calcul, etc.

2. *Un module d'interprétation implantant la machine virtuelle nécessaire pour exécuter un script Sollya.* Cette machine gère les variables Sollya dans leurs portées, exécute les commandes de contrôle du langage (branchements, boucles etc.), effectue le typage des expressions et lance les commandes numériques supportés. L'affichage des résultats calculés par les algorithmes numériques est principalement fait par ce module aussi.

3. *Un module implantant les fonctionnalités numériques de l'outil.* Il se décompose en plusieurs sous-unités dont chacune correspond à une des fonctionnalités de l'outil. Il intègre l'évaluation de fonctions composées, la simplification d'expressions de fonction etc.

La communication entre les différents modules de l'outil n'est pas ou peu unifiée. Seuls les modules d'analyse lexicale et d'interprétation utilisent une structure de donnée unique. Cette structure sert à représenter tout élément du langage Sollya, comme fonctions, domaines, chaînes de caractères, boucles, affectations etc. Les autres modules et unités de l'outil communiquent avec des structures de données ad-hoc représentant des informations diverses. La communication se fait par appel de fonction.

Les expressions de fonction sont partout représentées sous forme d'un arbre dont les feuilles sont soit des constantes, soit une seule variable libre dont le nom est stocké dans une variable globale accessible à tout l'outil. Cette représentation, conçue au tout départ du développement de l'outil, facilite certaines manipulations d'expressions, par exemple la dérivation formelle. Elle a le désavantage important de rendre l'implantation de la gestion de fonctions multivariées très difficile.

L'outil Sollya est capable de produire des preuves pour le résultat de calcul de norme infini certifiées (cf. chapitre 4.2) et pour la génération d'un code évaluateur d'un polynôme d'approximation (cf. chapitre 6.2). Dans les deux cas, la génération de la preuve nécessite la mémorisation des étapes de calcul intermédiaires produisant un résultat. Dans la réalisation logicielle, cela implique que les fonctions de calcul ou de génération de code doivent non seulement manipuler les données qui relèvent du calcul ou de la génération mais aussi celles correspondant à la preuve. Cet entrelacement n'est pas résolu de façon satisfaisante dans la mise en œuvre logicielle actuelle. Il empêche la maintenance facile des parties de code concernées.

L'interface des modules de calcul numérique et d'interprétation n'a pas été conçue dans l'optique d'une réutilisation des fonctionnalités numériques dans d'autre applications. Même si ces fonctionnalités sont actuellement exportées dans une bibliothèque Sollya, utilisée par exemple par le projet FloPoCo[7], l'interface reste restreinte et devrait être remodelée.

[7] `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

5.4 Conclusions et perspectives de l'outil Sollya

Sollya est un premier outil libre particulièrement destiné et adapté au développement de fonctions mathématiques. Le langage Sollya a une syntaxe ciblée sur la tâche donnée, qui essaie de faciliter au maximum la manipulation de fonctions univariées, d'arrondis et d'autres objets flottants. Sa philosophie, qui combine la manipulation mathématiquement exacte avec un arrondi fidèle contrôlable, donne une sécurité importante à ses utilisateurs. Ceux-ci, qui travaillent quotidiennement avec l'outil, soit dans l'équipe Arénaire soit extérieurement, temoignent de l'ergonomie importante de cet outil interactif.

Toutes les approches pour l'implantation de fonction mathématiques présentées dans cette thèse sont réalisées à la base du logiciel Sollya. Ceci crée une certaine dépendance de l'outil et de sa continuation dans l'avenir. Les approches pour l'implantation de fonction mathématiques présentées ici ont été réalisées et implantées dans le langage Sollya. À l'avenir, un compromis entre la continuation de l'outil Sollya et le coût de porter ces approches en un autre langage devra être trouvé. Tout de même, le langage Sollya reste un langage script spécialisé. Son temps d'interprétation reste important. Si les techniques d'implantation de fonctions mathématiques devaient un jour intégrer des logiciels où la performance compte, comme par exemple des compilateurs, il faudra trouver un moyen de passer du langage script Sollya à un binaire efficace.

Au début, la structure logicielle de Sollya était adaptée à la seule fonctionnalité de l'outil : tracer des fonctions d'erreurs. Le besoin et l'utilité de l'outil grandissant, cette structure logicielle s'est développé presque « organiquement » par le rajout de fonctionnalités. Des projets futurs concernés par l'implantation de fonctions mathématiques, comme le projet EVA-Flo[8], auront certainement des besoins similaires concernant le calcul flottant, l'approximation de fonctions et un langage de script pour le contrôle. Même si l'outil Sollya ne pourra peut-être pas continué et développé plus pour ces projets d'une certaine envergure, sa réalisation algorithmique et logicielle pourra leur servir d'inspiration. Une réécriture s'imposera pourtant dès qu'un support de fonctions multi-variée se fera vraiment sentir.

Sollya est un outil de calcul numérique qui intègre un interpréteur pour un langage de manipulation de données numériques. Développer et implanter cet interpréteur de langage de script a eu un impact relativement important sur le temps de développement total de l'outil. Cela est dû, on l'a vu au chapitre 5.1, au manque d'outils libres intégrant déjà un interpréteur que l'on pourrait étendre. Pour les projets futurs d'approximation, d'évaluation et de génération de codes flottants, le coût d'implantation associé à l'approche ou langage de contrôle risque encore de croître. Une réflexion poussée, basée sur les expériences avec Sollya, semble alors nécessaire.

[8] `http://www.ens-lyon.fr/LIP/Arenaire/EVA-Flo/EVA-Flo.html`

CHAPITRE 6

Automatisation de l'implantation de fonctions mathématiques

L'automatisation : système simplifiant tellement le travail qu'on finira par avoir besoin d'un cerveau électronique pour se tourner les pouces.

Noctuel, *écrivain français*

L'implantation de fonctions mathématiques, pour l'arrondi correct ou non, a longtemps été le travail fastidieux de spécialistes. Leur savoir-faire était illustré par leur travail fourni mais jamais bien explicité de façon algorithmique. La problématique de ce chapitre est l'algorithmique de ce savoir-faire, et, au passage, sa remise en question. Algorithmiquement, le niveau change : ce n'est plus l'algorithme d'approximation d'une fonction qui est au centre d'intérêt mais l'algorithme générateur qui produit cet algorithme d'approximation.

Cette automatisation n'a pas seulement l'avantage d'accélérer l'implantation de bibliothèques mathématiques. Elle ouvre aussi la voie vers de nouvelles optimisations numériques, par exemple dans les compilateurs, qui trouveraient un meilleur compromis entre la précision et la performance. C'est cette approche du calcul, jamais trop précis, donc jamais trop lent, qui est un point important de ce chapitre.

Dans ce qui suit, nous analysons d'abord pourquoi l'automatisation est sensée. Puis, nous nous intéressons aux deux sous-problèmes principaux : la génération d'un code évaluateur de polynômes et la recherche de tels polynômes d'approximation de fonctions. Ensuite, nous présentons brièvement une ébauche pour la gestion automatique de la réduction d'argument. Finalement, nous donnons des résultats concrets de notre générateur prototypé d'implantations de fonctions. Une courte discussion établissant un lien entre l'implantation automatique de fonctions mathématiques et la transformation de codes numériques clôt ce chapitre.

6.1 Les raisons d'automatiser l'implantation de fonctions

L'automatisation de l'implantation de fonctions mathématiques consiste à donner un algorithme pour la tâche suivante : à partir d'une description d'une fonction f, d'un domaine d'implantation dom et d'une erreur cible $\overline{\varepsilon}$, générer une séquence F d'opérations machine telle que, pour toute valeur x représentable dans le domaine, l'erreur entre $F(x)$ et $f(x)$ est

plus petite que $\bar{\varepsilon}$. Ce problème algorithmique est résolu dans la mesure qu'un spécialiste de l'implantation de fonctions mathématiques, avec son savoir-faire, est capable d'implanter une fonction. Les approches qu'il utilise, pour pas dire l'algorithme de génération qu'il déroule à la main, sont l'objet des livres de référence par Muller [101], Markstein [93] ou Cornea et al. [26]. Tout de même, si les approches sont décrites, l'algorithme n'est donné dans aucun de ces livres. Il reste du savoir-faire caché et non explicité. Par le passage au niveau méta, cet algorithmique est intellectuellement intéressante et mérite d'être considérée.

Sous un angle plus pratique, il y a principalement deux raisons d'automatiser l'implantation de fonctions mathématiques.

Complexité de la tâche La rapidité, l'habileté et l'assiduité humaine sont trop faibles par rapport à la complexité de la tâche. Cela nuit à la qualité des implantations obtenues.

L'expérience avec le développement de la bibliothèque CRLibm montre que le temps nécessaire à un spécialiste humain pour trouver un code évaluateur d'une fonction est élevé. Typiquement pour une fonction comme asin, on compte un à deux mois de temps de développement. Cette lenteur est due à plusieurs facteurs :

- La chaîne d'outils pour l'approximation polynomiale n'est pas suffisamment intégrée. Le développeur doit calculer un polynôme d'approximation dans un outil, par exemple Maple. Il doit le transférer dans un outil qui permettait de déterminer les précisions nécessaires pour le polynôme à coefficients flottants. Puis, il doit utiliser un troisième outil pour calculer ce polynôme d'approximation à coefficients flottants. Finalement, il doit intégrer les valeurs de coefficients, écrites sous forme de multi-doubles, dans un programme C. Bien sûr, tous les algorithmes nécessaires pourraient être implantés en Maple ; ceci n'est pourtant pas le cas en pratique.
- La détermination du schéma d'évaluation polynomiale nécessite l'analyse de l'importance des erreurs d'arrondi lors des opérations d'addition et de multiplication concernant l'erreur totale. Pour cela, les ordres de grandeur respectives de toutes les valeurs intermédiaires intervenant dans des additions doivent être estimés et bornées. Les précisions des opérateurs peuvent ensuite être choisie en fonction de l'importance de leurs erreurs d'arrondi. La traduction de ce choix de précisions en noms de briques de base de calcul multi-double pour l'implantation en langage C doit se faire à la main. Comme décrit au chapitre 3, elle est fastidieuse à cause de la multitude des combinaisons entre la précision, la taille du premier opérande et la taille du deuxième opérande. Des erreurs humaines lors de l'implantation du polynôme en langage C, comme l'oubli de déclaration d'une des dizaines de variables intermédiaires, se rajoutent à la difficulté.
- L'utilisation de l'arithmétique triple-double, décrite au chapitre 3, nécessite le calcul de bornes de chevauchement pour permettre un insertion de briques de renormalisation. Ce calcul de chevauchement est fait manuellement et souvent de tête. Il faut instancier, pour chaque opérateur triple-double, le théorème de borne de chevauchement associée et évaluer la nouvelle borne de chevauchement.
- La preuve de la borne d'erreur, suivant l'approche présenté au chapitre 4.3, nécessite la traduction du code C en syntaxe Gappa. Aucun arrondi ne doit être oublié pour assurer que Gappa prouve bien le code que l'on veut prouver et non un code qui y ressemble. Un rapport correct entre toute valeur flottante calculée, son équivalent exact sans arrondi et l'erreur associée doit être établie en syntaxe Gappa. Même si les aides à la preuve que nécessite Gappa se ressemblent toujours, elles doivent être instanciées

à la main.

Ce processus d'implantation manuel a des influences négatives sur la qualité du code et de la preuve associée :

- Dans l'implantation d'une fonction mathématique, il y a un lien étroit entre la réduction d'argument qui a un coût en taille de tables et en nombres d'accès à la mémoire et le polynôme d'approximation et le coût de son évaluation. Un compromis entre ces deux coûts doit être trouvé. Avec une approche d'implantation manuelle, l'échantillonnage de l'espace de recherche est très restreint. Le spécialiste humain reposera son travail sur son expérience et ses heuristiques. Les résultats ne peuvent donc pas être considérés comme optimaux en termes de performance.
- Comme l'expérience avec la bibliothèque CRLibm le montre, quasiment la moitié du temps de développement d'une implantation de fonction est passée à rédiger la preuve d'une borne d'erreur à l'aide de l'outil Gappa (et encore plus si l'on n'avait pas Gappa). Comme déjà mentionné au chapitre 4.3, les preuves Gappa ne résistent pas à des changements de l'algorithme d'évaluation. Le lien qui les relie au code de l'implantation n'est pas suffisamment étroit. Souvent, le coût en termes de temps de développement de preuve ne justifie donc pas de reporter une amélioration nouvellement trouvée dans une implantation de fonction. Cela nuit à la qualité d'une implantation au long terme.
- Comme déjà discuté à la section 4.3, une preuve Gappa pour la borne d'erreur d'arrondi survenue lors de l'évaluation du polynôme d'approximation contient des hypothèses additionnelles dès que l'arithmétique multi-double est utilisée. Principalement, ces hypothèses correspondent aux bornes de chevauchement et de précision des opérateurs triple-double. Si le calcul de ces bornes est fait à la main, des erreurs humaines peuvent survenir. Cela nuit à la qualité du certificat de borne d'erreur donné pour une implantation de fonction mathématique. L'automatisation, en particulier du calcul des bornes de chevauchement, augmente cette qualité. En effet, il suffit de prouver la correction de cet algorithme pour le chevauchement.

La qualité des implantations souffre donc à cause de la lenteur du développement manuel mais cette lenteur a également un impact économique. Les coûts de développement et de maintenance d'une bibliothèque de fonctions mathématique sont élevés et restent élevés durant tout le cycle de vie de la bibliothèque. En effet, à cause des changements technologiques du matériel, comme la longueur des pipelines, la taille des caches etc., une maintenance continuelle d'une bibliothèque est nécessaire. En plus, une multitude de caractéristiques de systèmes différents est à considérer à tout moment. Ce défi ne semble pouvoir être affronté qu'avec une génération et optimisation automatique des implantations.

Gestion des fonctions composées La gestion de fonctions composées, comme par exemple $\log_2(1+2^x)$, dans les systèmes actuels n'est pas optimale mais permet d'être améliorée par l'automatisation.

Actuellement, les codes numériques utilisant des fonctions composées se basent sur les bibliothèques `libm` qui implantent un certain nombre de fonctions usuelles. La composition des fonctions se fait dynamiquement par des appels séquentiels aux implantations bibliothèques. Dans l'exemple, $\log_2(1+2^x)$ résulte dans un premier appel à la fonction `exp2(x)`, implantant 2^x, suivi d'une addition avec 1 et d'un deuxième appel à la fonction `log2(x)`, approximant $\log_2 x$. Comme les implantations dans la bibliothèque mathématique doivent être génériques, un test de cas spéciaux pour les infinités, Not-A-Number, ou arguments in-

valides est exécuté à chaque appel. Dans l'exemple, le code de `log2(x)` teste toujours si son argument, $1 + 2^x$, est positif. Ceci est évidemment le cas.

La précision du résultat de cette composition dynamique de fonction dépend du comportement flottant sous-jacent. Elle peut en souffrir de façon conséquente. Une borne d'erreur n'est pas connue a priori ; elle doit être établie par une analyse d'erreur. Dans l'exemple, on observe que l'erreur devient rapidement importante dès que x devient négatif : 2^x devient petit par rapport à 1 ce qui implique une erreur importante lors de l'addition $1 + 2^x$.

Non seulement la précision souffre par ce mode de composition de fonctions. Des dépassements de capacité parasites peuvent survenir dans des valeurs intermédiaires de l'évaluation. Dans l'exemple, 2^x provoque un dépassement pour des valeurs de x plus grands que la largeur de l'exposant du format flottant. Tout de même, les valeurs réelles de $\log_2(1 + 2^x)$ ne provoquent de dépassement de capacité que pour des valeurs de x très proches de la valeur maximale représentable.

Comme cette évaluation de fonctions est basée sur une bibliothèque de fonctions usuelles statiquement implantée, la précision de fonctions composées est non seulement inconnue a priori mais aussi difficilement adaptable aux besoins de l'application. Le choix de précisions d'implantation des fonctions dans la bibliothèque est typiquement limité à la simple et double précision. Souvent les applications veulent favoriser la performance devant la précision sans pour autant calculer faux à cause d'une précision trop dégradée. Avec deux précisions, on ne peut pas parler d'une vraie adaptation. L'approche qui consiste à proposer des implantations dans les bibliothèques pour plus de précisions différentes connaît des limites évidentes. L'impact mémoire des bibliothèque augmenterait rapidement, ainsi que le coût de maintenance de ces bibliothèques. En plus, le choix de la bonne précision pour chaque fonctions de base serait difficile pour l'utilisateur car il demanderait une analyse d'erreur manuelle poussée.

L'automatisation de l'implantation de fonctions quelconques, c'est-à-dire composées, est un pas vers une amélioration de cette situation. Au lieu de proposer des bibliothèques génériques, les systèmes pourraient générer des implantations spécifiques. Celles-ci seraient insérées en ligne dans le code numérique. Elles auraient une précision fixée a priori par l'utilisateur et précisément adaptable. Les surcoûts dus à des appels de fonction ou à des tests de cas spéciaux seraient évités. Avec une connaissance fine des domaines des fonctions composées dans de tels compilateurs numériques, une optimisation des performances serait possible : le code ne devrait pas être générique mais juste couvrir ce domaine.

Une telle génération de codes évaluateurs de fonctions mathématiques réduirait également les besoins en analyse d'erreur a posteriori. L'analyse d'erreur se ferait en interne dans l'algorithme générateur.

En aparté, remarquons que de telles implantations dynamiques de fonctions composées semblent contredire les idées derrière l'arrondi correct. L'arrondi correct des fonctions élémentaires fait en sorte qu'une expression mathématique contenant de telles fonctions s'évalue bit à bit identiquement indépendamment du système. Ceci vient de ce qu'un arrondi se fait effectivement après toute évaluation de fonction mathématique. Pour l'implantation dynamique automatique, les fonctions de base intervenant dans une expression de fonction composée sont considérées comme exactes, c'est-à-dire sans arrondi. La génération de codes évaluateurs optimisés pour des fonctions composées mènerait alors effectivement à des résultats différents de ceux obtenus avec l'arrondi correct. Avec juste une borne d'erreur spécifiée, les résultats diffèreraient d'un système à l'autre. Il s'agit ici d'un compromis entre portabilité et qualité numérique. D'une part, le changement de l'interprétation sémantique

d'une expression de fonction composée est clair et doit être annoncé comme tel. D'autre part, il est toujours possible de songer à spécifier l'arrondi correct pour des fonctions composées entières, auquel cas les résultats seraient bit à bits exacts.

Les parties suivantes de ce chapitre essaient de répondre à ces besoins en automatisation. Un algorithme complet de génération d'implantation de fonctions mathématiques fait appel à deux algorithmes spécialisés particuliers : l'algorithme de génération de codes évaluateurs de polynômes et l'algorithme d'approximation de fonctions par des polynômes appropriés. Considérons alors d'abord ces deux algorithmes. Bien que ceci contredise l'ordre naturel dans lequel les choses se font dans une implantation, commençons par la génération de codes d'évaluation pour des raison de clareté des explications.

6.2 Génération automatique de codes évaluateurs de polynômes

6.2.1 Un cadre d'étude

Par l'approximation polynomiale, l'implantation d'une fonction f se réduit à l'implantation d'un polynôme p approchant f. Il faut alors générer une séquence P d'additions $\oplus$ et multiplications $\otimes$ flottantes qui évaluent p. Pour un x donné, la séquence calcule $P(x)$ une valeur flottante proche de la valeur $p(x)$ réelle. De façon un peu plus formelle, la séquence est un programme *straight-line flottant* où x est une variable d'entrée et $P(x)$ une des variables affectées.

Un contrôle de l'erreur relative d'arrondi, $\varepsilon(x) = \frac{P(x)-p(x)}{p(x)}$, est nécessaire. La génération du code évaluateur se fait sous la prémisse d'assurer que cette erreur d'arrondi reste plus petite qu'une borne donnée, l'erreur cible $\overline{\varepsilon} \in \mathbb{R}^+$. Le code généré doit alors vérifier que pour tout flottant x dans un domaine d'entrée dom, $|\varepsilon(x)| \leq \overline{\varepsilon}$.

Ce contrôle d'erreur n'est pas une simple vérification que la précision des opérations n'induit pas une erreur d'arrondi totale $\varepsilon(x)$ plus grande que $\overline{\varepsilon}$; c'est un choix d'opérateurs. Le chapitre 3 a montré que l'arithmétique multi-double est un moyen performant pour des erreurs cibles au-delà de la double précision. L'arithmétique multi-double permet alors de choisir non seulement une précision de calcul mais une précision pour chacune des opérations d'une séquence d'évaluation.

Une génération de codes multi-double pour des erreurs cibles arbitrairement petites semble difficile à ce stade. Pour fixer un cadre, on ne considérera l'arithmétique qu'à l'ordre 3, c'est-à-dire la double, double-double et triple-double. L'erreur cible minimale possible sera alors de 2^{-140}. Ces 140 bits suffiront pour toutes les applications pratiques pour l'arrondi correct en double précision. Pourtant, le cadre est déjà suffisamment large pour parler d'un vrai prototype pour la génération de codes avec un espace de recherche dans les erreurs cibles $\overline{\varepsilon}$ intéressant.

L'utilisation de valeurs multi-double ne se restreint pas aux valeurs produites par les opérations de la séquence générée. Il est important de considérer également le point d'évaluation x donné sur une variable double, double-double ou triple-double. Il faut accommoder les réductions d'argument qui ne peuvent pas produire un argument réduit sur un seul flottant double. De la même façon, les coefficients p_i du polynôme $p(x) = \sum_{i=0}^{n} p_i \cdot x^i$ peuvent également être des flottants double, double-double ou triple-double. De cette façon seule il est possible de trouver des polynômes d'approximation dont l'erreur d'approximation va au-delà de la double précision.

Pour un polynôme donné, il y a plusieurs séquences d'évaluation. Par exemple le polynôme $p(x) = p_0 + p_1 \cdot x + p_2 \cdot x^2$ peut être évalué comme $p(x) = p_0 + x \times (p_1 + x \times p_2)$, comme $p(x) = (p_0 + p_1 \times x) + p_2 \times (x \times x)$ ou encore d'autres façons. Les différentes séquences se distinguent par le temps d'exécution sur les processeurs superscalaires courants et par les erreurs d'arrondi différentes. Les paramètres importants y sont le nombre d'opérations à exécuter et le parallélisme possible entre elles.

Lors de la génération d'un code optimisé, l'espace de recherche à parcourir est large. Toute séquence pour un polynôme p consiste en deux sous-séquences pour des polynômes r et s, tels que $p(x) = r(x) \square s(x)$, où $\square$ est la dernière addition ou multiplication. Les sous-séquences pour r et s se décomposent récursivement de la même façon. Il est clair qu'en toute généralité, il y a une infinité de tels polynômes r et s. Seulement sous certaines contraintes, ce nombre devient fini. Comme les coefficients p_i du polynôme p donné sont des nombres flottants et que la séquence ne pourra utiliser que des flottants, il est raisonnable d'interdire toute modification de la valeur des coefficients. Elle provoquerait des arrondis. Dans ce cas, les polynômes r et s satisfont la propriété suivante : les ensembles formés par les coefficients de r et s sont une partition de l'ensemble des coefficients de p. Comme ni r ni s ne sont le polynôme nul, il y a $\sum_{k=1}^{m-1} \binom{m}{k} = 2^m - 2$ telles partitions pour un polynôme p ayant m coefficients non nuls. Même si ce nombre est fini parce que r et s se décomposent de la même façon, il est exponentiellement grand dans m.

À cause de cette explosion combinatoire, la génération d'une séquence d'évaluation est un problème difficile dans toute sa généralité. Il dépasserait le cadre de cette thèse de le traiter.

On décide donc de ne considérer que les séquences d'évaluation de type schéma de Horner. Ce schéma repose sur une réécriture du polynôme sous la forme :

$$p(x) = p_0 + x \cdot q(x)$$

où q est également un polynôme sous cette forme. La séquence d'opérations correspondante est alors composée de la séquence Q évaluant q, d'une multiplication par x et d'une addition avec p_0. L'utilisation du schéma de Horner simplifie également l'adaptation de la précision des opérations (cf. section 6.2.2), le calcul d'une borne de l'erreur d'arrondi $\varepsilon(x)$ [10] et la génération d'une preuve Gappa (cf. section 6.2.5) pour cette borne.

Évidemment, la restriction aux séquences de Horner a l'inconvénient que la performance des codes produits peut être non optimale sur les machines superscalaires. Une séquence de Horner est intrinsèquement séquentielle et n'utilise donc pas toutes les capacités des machines actuelles. Par exemple, plus de parallélisme est présent dans les schémas d'évaluation proposés par Knuth et Eve ainsi que par Estrin [73, 111]. Toutefois le parallélisme interne aux opérations multi-double réduit cet effet.

Pour certaines valeurs des coefficients du polynôme, il est sous-optimal qu'une séquence de Horner charge toujours tous les coefficients p_i. Deux cas spéciaux se présentent : soit un des coefficients vaut 1, soit un ou plusieurs coefficients sont nuls. Dans le premier cas, on observe une multiplication par la constante 1. Comme la séquence correspondante à $p_i + (x + (x \times x) \times q(x))$ nécessite plus d'additions et de multiplications que la séquence de Horner $p_i + x \times (1 + x \times q(x))$, l'effet peut être négligeable. Il ne sera donc pas considéré dans notre cadre. Dans le deuxième cas, où un coefficient p_i vaut 0, il faut pourtant prévoir une gestion spéciale dans l'algorithme de génération de séquence d'évaluation. On observerait sinon l'addition de la constante 0 chargée explicitement à la place d'une recopie simple :

$p_{i-1} + x \times (0 + x \cdot q(x))$ nécessite clairement plus d'opérations que $p_{i-1} + x \times (x \times q(x))$ ou $p_{i-1} + (x \times x) \times q(x)$. Cette optimisation est d'autant plus importante que des polynômes peuvent contenir plusieurs coefficients nuls ce qui implique la possibilité d'une réutilisation de valeurs déjà calculés. Par exemple, dans $p_{i-1} + (x \times x) \times (p_{i+1} + (x \times x) \times q(x))$ le résultat de $x \times x$ peut être réutilisé. Le cas se présente souvent comme les polynômes à implanter sont souvent pairs ou impairs. Plus de détails sur la gestion de ce cas seront donnés dans la section 6.2.3.

Le cadre pour l'étude d'un algorithme pour générer des séquences d'évaluation étant fixé, considérons maintenant d'abord comment la précision des opérations intermédiaires peut être automatiquement adaptée.

6.2.2 Adaptation de la précision

Pour concevoir un algorithme d'adaptation de la précision des opérations dans une séquence de Horner, une analyse de l'importance des erreurs d'arrondi des opérations est nécessaire. Cette analyse peut se faire comme suit. Il est facile d'en dériver un algorithme qui distribue l'erreur maximale cible $\overline{\varepsilon}$ permise pour l'évaluation du polynôme sur les différentes opérations flottantes.

Il convient d'abord de fixer les notations :

- On considère le polynôme $p(x) = p_0 + x \cdot q(x)$ défini sur l'intervalle dom.
- Le polynôme sera approché en flottant par la séquence $P(x) = p_0 \oplus x \otimes Q(x)$. Ici, $\oplus$ et $\otimes$ dénotent l'addition et la multiplication flottante de l'étape, indépendamment de leur précision. C'est-à-dire, par un léger abus de langage, on note $\oplus$ toute opération d'addition, serait-elle de précision double, double-double ou triple-double. Le sous-polynôme q est approché par la sous-séquence Q.
- Notons les différentes erreurs comme suit :

$$\begin{aligned}
\varepsilon(x) &= \frac{P(x) - p(x)}{p(x)} \\
\varepsilon_Q(x) &= \frac{Q(x) - q(x)}{q(x)} \\
\varepsilon_M(x) &= \frac{x \otimes Q(x) - x \cdot Q(x)}{x \cdot Q(x)} \\
\varepsilon_A(x) &= \frac{p_0 \oplus x \otimes Q(x) - (p_0 + x \otimes Q(x))}{p_0 + x \otimes Q(x)} \\
\varepsilon_T(x) &= \frac{x \otimes Q(x) - x \cdot q(x)}{x \cdot q(x)}
\end{aligned}$$

  Ici, $\varepsilon(x)$ représente l'erreur relative totale entre le résultat de la séquence $P(x)$ et la valeur exacte $p(x)$ du polynôme. Pareillement, $\varepsilon_Q(x)$ est l'erreur relative entre le résultat de la sous-séquence $Q(x)$ et le polynôme $q(x)$. Les erreurs $\varepsilon_M(x)$ et $\varepsilon_A(x)$ représentent l'erreur relative élémentaire respectivement de la multiplication $\otimes$ et de l'addition $\oplus$. Pour des raisons de preuve, il convient de définir également l'erreur combinée $\varepsilon_T(x)$ entre $x \otimes Q(x)$ et son équivalent mathématique $x \cdot q(x)$.
- Un point important dans l'analyse d'une étape de Horner est l'amplification de l'erreur

des étapes précédentes. La quantité suivante y joue un rôle important :

$$\beta(x) = \frac{x \cdot q(x)}{p_0 + x \cdot q(x)} = \frac{x \cdot q(x)}{p(x)}$$

On notera

$$\overline{\beta} = \|\beta\|_\infty^{dom}.$$

On remarque que $\beta(x) = 1 - \frac{1}{1+\alpha(x)}$, où

$$\alpha(x) = \frac{x \cdot q(x)}{p_0}$$

est le rapport entre les deux opérandes de l'addition dans l'étape considérée. On note pareillement

$$\overline{\alpha} = \|\alpha\|_\infty^{dom}.$$

Analysons maintenant l'erreur d'arrondi totale d'un séquence de Horner. On observe que

$$\begin{aligned}
\varepsilon_T(x) &= \frac{x \otimes Q(x) - x \cdot q(x)}{x \cdot q(x)} \\
&= \frac{x \cdot Q(x) \cdot (1 + \varepsilon_M(x)) - x \cdot q(x)}{x \cdot q(x)} \\
&= \frac{x \cdot q(x) \cdot (1 + \varepsilon_Q(x)) \cdot (1 + \varepsilon_M(x)) - x \cdot q(x)}{x \cdot q(x)} \\
&= \varepsilon_Q(x) + \varepsilon_M(x) + \varepsilon_Q(x) \cdot \varepsilon_M(x)
\end{aligned}$$

Cela veut dire que l'erreur de la multiplication d'une étape et l'erreur accumulée des étapes précédentes ont la même influence sur le résultat intermédiaire de l'étape $x \cdot q(x)$. Il suffit alors de ne considérer d'abord que l'erreur combinée $\varepsilon_T(x)$.

Alors le résultat de la séquence flottante $P(x)$ peut s'exprimer en fonction de la valeur exacte du polynôme $p(x)$, de $\beta(x)$ et des erreurs comme suit :

$$\begin{aligned}
P(x) &= p_0 \oplus x \otimes Q(x) \\
&= (p_0 + x \otimes Q(x)) \cdot (1 + \varepsilon_A(x)) \\
&= (p_0 + x \cdot q(x) \cdot (1 + \varepsilon_T(x))) \cdot (1 + \varepsilon_A(x)) \\
&= (p(x) + x \cdot q(x) \cdot \varepsilon_T(x)) \cdot (1 + \varepsilon_A(x)) \\
&= (p(x) + p(x) \cdot \beta(x) \cdot \varepsilon_T(x)) \cdot (1 + \varepsilon_A(x)) \\
&= p(x) \cdot (1 + \beta(x) \cdot \varepsilon_T(x)) \cdot (1 + \varepsilon_A(x))
\end{aligned}$$

L'erreur totale $\varepsilon(x)$ est donc

$$\begin{aligned}
\varepsilon(x) &= \frac{P(x) - p(x)}{p(x)} \\
&= \frac{p(x) \cdot (1 + \beta(x) \cdot \varepsilon_T(x)) \cdot (1 + \varepsilon_A(x)) - p(x)}{p(x)} \\
&= (1 + \beta(x) \cdot \varepsilon_T(x)) \cdot (1 + \varepsilon_A(x)) - 1 \\
&= \varepsilon_A(x) + \beta(x) \cdot \varepsilon_T(x) + \beta(x) \cdot \varepsilon_A(x) \cdot \varepsilon_T(x) \qquad (6.1)
\end{aligned}$$

Les conclusions suivantes sont à tirer de ce résultat. Elles permettent ensuite de concevoir un algorithme distribuant l'erreur cible $\overline{\varepsilon}$ sur les différentes opérations.

– L'erreur d'arrondi de l'addition dans une étape, $\varepsilon_A(x)$ entre complètement dans l'erreur totale du polynôme $P(x)$. Autrement dit pour évaluer un polynôme avec une erreur cible $\overline{\varepsilon}$ correspondante à 60 bits, la précision de l'addition finale doit être au moins de 60 bits, voire un peu plus grande à cause des erreurs quadratiques.
– L'influence de l'erreur $\varepsilon_T(x)$, qui consiste en parties égales de l'erreur de la multiplication et de l'erreur accumulée des étapes suivantes, est pondérée par $\beta(x)$. Quand cette quantité reste petite, $\overline{\beta} \ll 1$, on observe un effet positif sur les précisions nécessaires pour la multiplication et pour l'évaluation du sous-polynôme $q(x)$. Leur erreur d'arrondi peut être plus grande que l'erreur cible car elle est réduite par cet effet d'échelle. En première approximation, on a un gain en précision de $\log_2 \overline{\beta}$ bits à l'étape de Horner considérée.
  Dans le cas où $\beta(x)$ deviendrait plus grand que 1 en valeur absolue, l'effet inverse serait à constater. L'erreur accumulée du sous-polynôme $Q(x)$ et de la multiplication serait amplifiée lors de l'addition. Cela veut dire qu'une cancellation aurait lieu, ce qui est à éviter, particulièrement dans le cadre de l'arithmétique multi-double. Une discussion plus profonde de cette problématique est donnée à la Section 6.3.2.
  On fixera donc la borne $\overline{\beta} \leq 1$ et on fait en sorte qu'elle soit vérifiée par l'algorithme de génération.

Cette analyse du comportement des erreurs dans une étape de Horner étant faite, il est facile de concevoir un algorithme d'adaptation de la précision pour une séquence de Horner. On propose un algorithme récursif. Il prend en entrée un polynôme $p(x) = p_0 + x \cdot q(x)$, un domaine dom et une erreur cible $\overline{\varepsilon}$. À chaque récursion, il détermine deux bornes $\overline{\varepsilon}_A$ et $\overline{\varepsilon}_M$ pour l'addition et multiplication d'une étape ainsi qu'une nouvelle erreur cible $\overline{\varepsilon}_Q$ pour l'évaluation du sous-polynôme $q(x)$. Ces bornes $\overline{\varepsilon}_A$, $\overline{\varepsilon}_M$ et $\overline{\varepsilon}_Q$ sont telles que si $|\varepsilon_A| \leq \overline{\varepsilon}_A$, $|\varepsilon_M| \leq \overline{\varepsilon}_M$ et $|\varepsilon_Q| \leq \overline{\varepsilon}_Q$ alors $|\varepsilon(x)| \leq \overline{\varepsilon}$. L'algorithme suit exactement l'analyse d'erreur : la borne $\overline{\varepsilon}_A$ est égale à l'erreur cible $\overline{\varepsilon}$ diminué d'un facteur correspondant à un peu plus d'un bit pour les erreurs quadratiques – on multiplie par $7/16$. Les bornes $\overline{\varepsilon}_M$ et $\overline{\varepsilon}_Q$ sont égales à un $\overline{\varepsilon}$ diminué et divisé par $\overline{\beta}$ pour compenser l'effet de pondération. Pour un bon comportement de l'algorithme, il convient de fixer une borne supérieure pour l'erreur cible : $\overline{\varepsilon} \leq 1/8$. Interdire des précisions cibles inférieures à 3 bit ne nuit pas à l'utilité pratique de l'algorithme. Cette condition est assurée par l'algorithme également pour les appels récursifs. Le listing 6 donne l'algorithme.

L'analyse d'erreur de Horner étant faite, la preuve de correction de l'algorithme 6 est facile. Dans le cas de base, où le degré de q est 0, on a $\varepsilon_M = 7/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}$. Comme la séquence d'évaluation de q consistera dans ce cas seulement en une affectation du seul coefficient constant de q, ce qui ne provoque pas d'arrondi, on a alors $\varepsilon_Q(x) = 0$. En conséquence,

$$|\varepsilon_T(x)| \leq \overline{\varepsilon}_M + \overline{\varepsilon}_Q + \overline{\varepsilon}_M \cdot \overline{\varepsilon}_Q = 7/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}$$

Dans le cas de récurrence, on a $\overline{\varepsilon}_M = 3/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}$ et $\overline{\varepsilon}_Q = 3/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} \leq 1/8$. Ceci donne par hypothèse de récurrence

$$\begin{aligned} |\varepsilon_T(x)| &\leq \overline{\varepsilon}_M + \overline{\varepsilon}_Q + \overline{\varepsilon}_M \cdot \overline{\varepsilon}_Q \\ &\leq 3/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} + 3/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} + 3/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} \cdot 1/8 \\ &= \frac{\overline{\varepsilon}}{\overline{\beta}} \cdot (3/16 + 3/16 + 3/128) \leq 7/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} \end{aligned}$$

Entrées : un polynôme $p(x) = p_0 + x \cdot q(x)$, un domaine $dom \subseteq \mathbb{R}$, une erreur cible $\overline{\varepsilon} \in \mathbb{R}^+, \overline{\varepsilon} \leq 1/8$
Sorties : deux listes de précisions pour les additions et multiplications, $\perp$ si erreur

Algorithme `déterminePrécisions` (p, dom, $\overline{\varepsilon}$) :
début
  $\overline{\beta} \leftarrow \left\| \frac{x \cdot q(x)}{p_0 + x \cdot q(x)} \right\|_\infty^{dom}$;
  si $\overline{\beta} > 1$ **alors renvoyer** $\perp$;
  $\overline{\varepsilon}_A \leftarrow 7/16 \cdot \overline{\varepsilon}$;
  si `degré` $(q) = 0$ **alors**
    $\overline{\varepsilon}_M \leftarrow 7/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}$;
    renvoyer $([\overline{\varepsilon}_A], [\overline{\varepsilon}_M])$;
  sinon
    $\overline{\varepsilon}_M \leftarrow 3/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}$; $\overline{\varepsilon}_Q \leftarrow \min\left(3/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}, 1/8\right)$; /* min *pour la récurrence* */
    precQ $\leftarrow$ `déterminePrécisions` $(q, dom, \overline{\varepsilon}_Q)$;
    si precQ $= \perp$ **alors renvoyer** $\perp$;
    (precAdd , precMul) $\leftarrow$ precQ ;
    renvoyer (ε_A :: precAdd , ε_M :: precMul); /* :: *est la concaténation* */
  fin
fin

Algorithme 6 : L'algorithme `déterminePrécisions` pour adapter la précision

Alors, dans tous les cas, on a

$$\begin{aligned} |\varepsilon(x)| &\leq \overline{\varepsilon}_A + \overline{\beta} \cdot \overline{\varepsilon}_T + \overline{\beta} \cdot \overline{\varepsilon}_A \cdot \overline{\varepsilon}_T \\ &= 7/16 \cdot \overline{\varepsilon} + \overline{\beta} \cdot 7/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} + \overline{\beta} \cdot 7/16 \cdot \overline{\varepsilon} \cdot 7/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} \\ &= \overline{\varepsilon} \cdot (7/16 + 7/16 + 49/256 \cdot \overline{\varepsilon}) \\ &\leq \overline{\varepsilon} \cdot (7/16 + 7/16 + 49/256 \cdot 1/8) \\ &= \overline{\varepsilon} \cdot \frac{1841}{2048} \leq \overline{\varepsilon} \end{aligned}$$

L'algorithme pour le calcul des précisions nécessaires est donc correct.

Tout en étant bien spécifié et correct, l'algorithme 6 est basé sur une heuristique et peut donner des résultats sous-optimaux. En effet, rien n'assure que la répartition de l'erreur cible en les différentes erreurs est optimale. Dans certains cas, il vaudrait certainement mieux fixer $\overline{\varepsilon}_A$ à une valeur plus grande que $7/16 \cdot \overline{\varepsilon}$ et de compenser cette hausse par une erreur plus petite lors de la multiplication et l'erreur du sous-polynôme. Il est clair qu'il telle compensation d'erreurs demande des algorithmes dont la complexité est bien plus grande que celle du l'algorithme heuristique donné. En pratique, on observe rarement un cas où les précisions choisies par l'algorithme sont trop grandes ou trop petites. Ce n'est pas à cause d'une répartition sous-optimale des erreurs élémentaires que la borne d'erreur finale, c'est-à-dire une borne fine pour $|\varepsilon(x)|$, observée en pratique est beaucoup plus petite que l'erreur cible $\overline{\varepsilon}$. C'est l'effet suivant qui y joue un rôle beaucoup plus important.

L'algorithme 6 détermine des bornes d'erreur pour les différentes opérations d'addition et de multiplication. Dans le cadre de l'arithmétique multi-double, ces bornes doivent être

traduites en des noms d'opérateurs double, double-double ou triple-double. Cette traduction est simple : on choisit l'opérateur sur un nombre minimal de doubles qui ne commet pas d'erreur plus grande que le $\overline{\varepsilon}_A$ (ou $\overline{\varepsilon}_M$) donné. Comme, dans notre cadre, il n'y a qu'un choix entre trois précisions possibles, on observe un effet important de discrétisation. La précision des opérateurs est alors souvent non seulement légèrement plus grande pour satisfaire les bornes, mais beaucoup plus grande. Certes, cet effet peut provoquer la génération d'une séquence avec une erreur totale $\varepsilon(x)$ beaucoup plus petite que l'erreur cible $\overline{\varepsilon}$ de départ. En revanche, le caractère discret du choix des opérateurs cache les possibles insuffisances de la répartition heuristique de l'erreur cible. En effet, que $\overline{\varepsilon}_A$ soit $7/16 \cdot \overline{\varepsilon}$ ou $3/4 \cdot \overline{\varepsilon}$ n'a quasiment d'influence pour un choix entre $\overline{\varepsilon}_A < 2^{-53}$ pour un opérateur double ou $\overline{\varepsilon}_A < 2^{-103} = 2^{-50} \cdot 2^{-53}$ pour un opérateur double-double. Remarquons que cette surestimation des erreurs d'erreurs peut être diminuée si une analyse de l'erreur totale $\varepsilon(x)$ est faite a posteriori, par exemple en Gappa (cf. Section 6.2.5 ci-après). Dans le cas où la borne a posteriori indique une erreur trop petite par rapport à l'erreur cible $\overline{\varepsilon}$, une heuristique consiste à diminuer l'erreur cible donnée au générateur de séquences jusqu'à ce que la borne d'erreur a posteriori ne tienne plus.

De premier abord, il est simple de donner un algorithme de traduction des bornes en opérateurs. Il suffit de les comparer aux bornes d'erreur des opérateurs double et double-double (cf. Section 3). On obtient l'algorithme donné par le listing 7.

Entrées : une précision cible $\overline{\varepsilon} \in \mathbb{R}^+$, $\overline{\varepsilon} \leq 1/8$, pour une opération multi-double
Sorties : un entier $k \in [\![1, 3]\!]$ donnant le nombre de doubles en résultat de l'opération

Algorithme `précisionMultidouble` ($\overline{\varepsilon}$) :
début
  si $\overline{\varepsilon} \leq 2^{-103}$ **alors renvoyer** 3;
  si $\overline{\varepsilon} \leq 2^{-53}$ **alors renvoyer** 2 ; **sinon renvoyer** 1;
fin

Algorithme 7 : L'algorithme `précisionMultidouble`

Il s'avère que cet algorithme 7 est encore trop simple. Pour l'arithmétique multi-double, il ne suffit pas bon seulement de déterminer le nombre ρ de doubles par lesquels sera représenté le résultat de l'opération. Il faut également connaître le nombre de flottants doubles pour les deux opérandes, σ et τ. Par exemple, on a $\sigma = 2$ pour un double-double en premier opérande. Pour une séquence d'évaluation de polynôme avec le schéma de Horner, aucun choix n'est à faire pour ces nombres de doubles σ et τ dans les opérandes :

- Dans les opérations de multiplication, $x \otimes Q(x)$, le nombre de doubles σ du premier opérande est celui nécessaire pour écrire x. Cette valeur est prise en argument de l'algorithme générant des séquences d'évaluation. Le deuxième opérande de la multiplication est le résultat de la sous-séquence Q. Celle-ci se termine par une addition produisant t doubles. On a alors $\tau = t$ doubles pour le deuxième opérande.
- Dans les opérations d'addition, $p_i \oplus (x \otimes Q(x))$, le nombre de doubles σ du premier opérande est donné par le nombre de doubles du coefficient p_i. Comme avant, le deuxième opérande s'écrit sur un nombre de double τ fixé par l'opération qui précède. Si celle-ci produit t doubles, on a $\tau = t$.

En revanche, certaines combinaisons de nombre de doubles (σ, τ, ρ) dans les opérandes et le résultats sont impossibles.

- Les opérateurs multi-doubles ne produisent jamais plus de nombres double précision qu'ils ne prennent en argument. Cela veut dire que $\sigma + \tau \leq \rho$. Il est facile de s'en convaincre : un opérateur d'addition de σ et de τ nombres double précision produisant plus de $\sigma+\tau$ nombres doubles « inventerait » au moins un flottant double précision : les $\sigma + \tau$ nombres en entrée forment déjà un nombre multi-double somme des opérandes. De même, comme le produit de deux nombres tenant sur n et m bits tient sur $n+m$ bits ($2^n \cdot 2^m = 2^{n+m}$), un opérateur de multiplication multi-double produit au plus $\sigma + \tau$ nombres double précision.
  Dans le cas où un opérateur multi-double produit $\rho = \sigma + \tau$ flottants double précision qu'il prend en entrée, il devient exact. Il est alors possible de produire une valeur sur un nombre de double inférieur à celui qui inféré par l'analyse d'erreur et l'algorithme 7. Remarquons que cet effet provoque encore une surestimation des erreurs d'arrondi des opérateurs.
- Les opérateurs d'addition multi-doubles produisent au moins autant de nombres double précision qu'il prennent en argument pour chacun de leurs arguments. Cela veut dire que $\rho \geq \max(\sigma, \tau)$. Dans le cas contraire, il y aurait soit une partie négligeable dans un des opérandes soit une cancellation. Les deux cas sont à exclure dans le cadre donné. Dans le cas où une partie des opérandes serait négligée, soit le polynôme d'approximation serait changé par l'altération du coefficient p_i concerné, soit l'opération de multiplication produirait un résultat sur un nombre trop élevé de flottants double, donc trop précis.
  L'algorithme doit surtout assurer qu'il n'y ait pas d'altération du polynôme. Elle briserait le bon comportement de l'erreur d'approximation du polynôme (cf. Section 6.3.3).

Il est facile de calculer le nombre de doubles nécessaires pour écrire un coefficient du polynôme. Le listing 8 donne l'algorithme.

Entrées : un nombre flottant $x \in \mathbb{F}_{53} + \mathbb{F}_{53} + \mathbb{F}_{53}$, maximalement un triple-double
Sorties : le nombre minimal $k \in [\![1, 3]\!]$ de doubles nécessaires pour écrire x

Algorithme `longueurMultidouble` ($\overline{\varepsilon}$) :
début
  si $x \in \mathbb{F}_{53}$ **alors renvoyer** 1;
  si $x - \circ_{53}(x) \in \mathbb{F}_{53}$ **alors renvoyer** 2 ; **sinon renvoyer** 3;
fin

Algorithme 8 : L'algorithme `précisionMultidouble`

Ce dernier algorithme 8 nous permet de donner l'algorithme 9 complet qui détermine pour un polynôme p, un domaine dom et une erreur cible $\overline{\varepsilon}$, deux listes contenant les combinaisons (σ, τ, ρ) de nombres de doubles en opérande et résultat de chaque addition et multiplication d'une séquence de Horner. Cet algorithme simule statiquement chaque opération de la séquence de Horner en appliquant les règles donnés ci-dessus. Le listing 9 donne l'algorithme.

Entrées : un polynôme $p(x) = \sum_{i=0}^{n} p_i \cdot x^i$, un domaine $dom \subseteq \mathbb{R}$, une erreur cible $\overline{\varepsilon} \in \mathbb{R}^+, \overline{\varepsilon} \leq 1/8$, un entier $r \in [\![1, 3]\!]$ de doubles utilisés pour écrire x
Sorties : deux listes de triplets (σ, τ, ρ) donnant le nombre de doubles σ et τ pour le premier respectivement second opérande et le nombre de double ρ pour le résultat de chaque opération d'addition et de multiplication, $\perp$ si erreur

Algorithme `déterminecombinaisonsMultidouble` (p, dom, $\overline{\varepsilon}$, r) :
début
  precP $\leftarrow$ `déterminePrécisions`(p, dom, $\overline{\varepsilon}$) ;
  si precP $= \perp$ **alors renvoyer** $\perp$;
  (precAdd , precMul) $\leftarrow$ precP ; $n \leftarrow$ `degré`(p) ;
  $t \leftarrow$ `longueurMultidouble`(p_n) ;
  listeAdd $\leftarrow$ [] ; listeMul $\leftarrow$ [];
  pour $i = n - 1$ **à** 0 **faire**
    $a \leftarrow$ `précisionMultidouble`(precAdd $[i]$) ;
    $m \leftarrow$ `précisionMultidouble`(precMul $[i]$) ;
    $c \leftarrow$ `longueurMultidouble`(p_i) ;
    si $t + r < m$ **alors** $\rho \leftarrow t + r$ **sinon** $\rho \leftarrow m$;
    listeMul $\leftarrow (r, t, \rho)$ ::listeMul ; $t \leftarrow \rho$;
    si $t + c < a$ **alors** $\rho \leftarrow t + c$ **sinon** $\rho \leftarrow a$;
    si $c > a$ **alors** $\rho \leftarrow c$;
    listeAdd $\leftarrow (c, t, \rho)$ ::listeAdd ; $t \leftarrow \rho$;
  fin
  renvoyer (listeAdd , listeMul);
fin

Algorithme 9 : L'algorithme `déterminecombinaisonsMultidouble`

Ce dernier algorithme 9 résout en principe le problème posé : déterminer une séquence de Horner pour évaluer un polynôme p avec une arithmétique multi-double tout en garantissant une erreur d'arrondi totale $\varepsilon(x)$ inférieure à une erreur cible $\overline{\varepsilon}$ donnée. En effet, il suffit d'afficher les listes de combinaisons de nombre de doubles en opérande et résultat pour les additions et multiplication dans la syntaxe de langage d'implantation, dans notre cadre le C.

Tout de même, la séquence ainsi implantée peut être sous-optimale : comme déjà discuté dans la Section 6.2.1, le polynôme p peut avoir des coefficients nuls. Comme l'algorithme 9 ne les traite pas d'une façon spéciale, la séquence implantée contient des chargements de la constante 0 suivis d'additions inutiles. La prochaine section s'intéresse à ce problème.

6.2.3 Le schéma de Horner et polynômes à coefficients nuls

Il serait facile de traiter le cas où certains coefficients p_i du polynôme sont nuls par un simple test supprimant les additions avec zéro dans la séquence implantée. Ceci résulterait dans des séquences contenant des suites de multiplications par la variables x. Dans le cas où il n'y a pas qu'un coefficient nul, ces suites de multiplications se répéteraient. Ceci serait sous-optimal.

Le cas où plusieurs coefficients du polynôme sont nuls est relativement fréquent. Le polynôme p approche une fonction f dans un petit intervalle contenant souvent 0. Si la fonction est symétrique autour de 0 (paire ou impaire), le polynôme d'approximation suit souvent

cette symétrie [101, 82]. Du coup, seul un coefficient sur deux n'est pas nul. Le nombre de vraies additions avec des coefficients non-nuls dans une séquence de Horner diminue de moitié. Plus de détails sur ce sujet sont donnés à la Section 6.3.2.

Il convient alors de proposer une approche pour des séquences de Horner modifiés. L'évaluation se fait en deux étapes. D'abord, toutes les puissances nécessaires x^{k_i} de la variable x sont précalculées, mais une seule fois. Ensuite, la séquence évalue p comme

$$p(x) = p_{\kappa(0)} + x^{k_1} \cdot \left(p_{\kappa(1)} + x^{k_2} \cdot \left(\dots \left(p_{\kappa(t-1)} + x^{k_t} \cdot p_{\kappa(t)}\right) \dots\right)\right)$$

où le coefficient constant de q n'est pas nul.

La technique à mettre en œuvre nécessite de résoudre les problèmes algorithmiques suivants :

1. Calcul de la liste des puissances de x nécessaires pour une évaluation de Horner.
2. Détermination de bornes d'erreur cible pour le précalcul de ces puissances de x. En effet, ce précalcul n'est pas exact et doit se fait également en arithmétique multi-double. Il engendre une erreur qui doit être bornée.
3. Génération d'une séquence de multiplications calculant toutes les puissances de x nécessaires avec une erreur inférieure à l'erreur cible déterminée. Il s'agit de calculer non seulement une puissance entière de x mais d'en calculer plusieurs. Les techniques traditionnelles [73] ne suffisent plus.
4. Modification de l'algorithme de génération de séquences de Horner avec arithmétique multi-double pour l'utilisation des puissances de x précalculées. Il y est nécessaire de provoquer un arrondi de x^{k_i} aux étapes de Horner où la précision de cette puissance est trop élevée, donc le nombre de doubles pour la représenter trop grand. Ceci se fait à un coût nul dans le code produit ; il suffit de ne pas prendre en compte les parties de poids faible du flottant multi-double.

Intéressons-nous alors à chacun de ces sous-problèmes. Le calcul de la liste des puissances de x nécessaires est simple. Il suffit de simuler une évaluation de Horner traditionnelle. La longueur des sous-séquences d'additions avec zéro et de multiplications répétées est équivalente aux exposants des puissances de x. Pour éviter un calcul double d'une même puissance, cette information est alors condensée dans une représentation de l'ensemble des puissances nécessaires de x. L'algorithme 10 implante cette simulation du schéma de Horner.

Entrées : un polynôme $p(x) = \sum_{i=0}^{n} p_i \cdot x^i, \quad n$ degré de p

Sorties : une liste ℓ minimale de m entiers non-nuls distincts, triés dans l'ordre décroissant, tels qu'il existe un entier minimal t et

$$k_1, \ldots, k_t \in \ell, \quad \ell = \bigcup_{i=1}^{t} k_i$$

tels qu'on peut écrire p comme

$$p(x) = p_{\kappa(0)} + x^{k_1} \cdot \left(p_{\kappa(1)} + x^{k_2} \cdot \left(\ldots \left(p_{\kappa(t-1)} + x^{k_t} \cdot p_{\kappa(t)} \right) \ldots \right) \right)$$

avec

$$\kappa(0) = 0, \quad \kappa(1) = k_1, \quad \kappa(2) = k_1 + k_2, \quad \ldots$$

et

$$\forall i \neq 0, p_{\kappa(i)} \neq 0$$

Algorithme `déterminePuissancesNécessaires` (p) :
début
  allouer L, *un tableau de* n *booléens*;
  pour $k = 1$ **à** n **faire** $L[k] \leftarrow$ faux;
  $i \leftarrow n$;
  tant que $i > 0$ **faire**
    $k \leftarrow 0$;
    tant que $i - k - 1 \geq 0 \wedge p_{i-k-1} = 0$ **faire** $k \leftarrow k + 1$;
    si $i - k - 1 \geq 0$ **alors** $k \leftarrow k + 1$;
    $L[k] \leftarrow$ vrai;
    $i \leftarrow i - k$;
  fin
  $\ell \leftarrow []$;
  pour $k = 1$ **à** n **faire** **si** $L[k]$ **alors** $\ell \leftarrow k :: \ell$;
  renvoyer ℓ;
fin

Algorithme 10 : L'algorithme `déterminePuissancesNécessaires`

Remarquons que ce calcul de la liste des puissances de x nécessaire, réalisé par l'algorithme 10, peut être intégré à l'algorithme 6, déterminant les erreurs cibles pour les additions et multiplications dans une séquence de Horner. Cette intégration permet alors de déterminer non seulement la liste ℓ de puissances nécessaires mais aussi une liste associée de bornes d'erreur cible pour leur calcul. En effet, il est possible de modifier l'analyse d'erreur d'une séquence de Horner, donnée à la Section 6.2.2, comme suit :

Notons $X_k(x)$ le résultat du calcul flottant approchant x^k. Notons $\varepsilon_{X_k}(x)$ l'erreur associée :

$$\varepsilon_{X_k}(x) = \frac{X_k(x) - x^k}{x^k}$$

Dans le cas où $k \neq 0$, à la place de calculer $x \otimes Q(x)$ approchant $x \cdot q(x)$, l'étape de la séquence de Horner modifiée, évaluera $X_k(x) \otimes Q(x)$ approchant $x^k \cdot q(x)$. L'erreur entachant cette

valeur, notons-la toujours $\varepsilon_T(x)$, peut alors s'analyser comme :

$$\begin{aligned}
\varepsilon_T(x) &= \frac{X_k(x) \otimes Q(x) - x^k \cdot q(x)}{q(x)} \\
&= \frac{x^k \cdot (1+\varepsilon_{X_k}(x)) \otimes Q(x) - x^k \cdot q(x)}{q(x)} \\
&= \frac{x^k \cdot (1+\varepsilon_{X_k}(x)) \cdot Q(x) \cdot (1+\varepsilon_M(x)) - x^k \cdot q(x)}{q(x)} \\
&= \frac{x^k \cdot (1+\varepsilon_{X_k}(x)) \cdot q(x) \cdot (1+\varepsilon_Q(x)) \cdot (1+\varepsilon_M(x)) - x^k \cdot q(x)}{q(x)} \\
&= \varepsilon_{X_k}(x) + \varepsilon_Q(x) + \varepsilon_M(x) + \\
&\quad \varepsilon_{X_k}(x) \cdot \varepsilon_Q(x) + \varepsilon_{X_k}(x) \cdot \varepsilon_M(x) + \varepsilon_Q(x) \cdot \varepsilon_M(x) + \varepsilon_{X_k}(x) \cdot \varepsilon_M(x) \cdot \varepsilon_Q(x)
\end{aligned}$$

Cette nouvelle analyse d'erreur montre que les erreurs de multiplication $\otimes$, d'évaluation du sous-polynôme q et de la nouvelle erreur, dont est entachée $X_k(x)$, entrent dans l'erreur $\varepsilon_T(x)$ de la même façon. Il est alors possible de modifier l'algorithme 6 pour fixer $\overline{\varepsilon}_M$, $\overline{\varepsilon}_Q$ et la borne $\overline{\varepsilon}_{X_k}$ pour $\varepsilon_{X_k}(x)$ comme suit :

$$\begin{aligned}
\overline{\varepsilon}_M &\leftarrow 1/8 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} \\
\overline{\varepsilon}_Q &\leftarrow \min\left(1/8 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}, 1/8\right) \\
\overline{\varepsilon}_{X_k} &\leftarrow \min\left(\overline{\varepsilon}_{X_k}, 1/8 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}, 1/8\right)
\end{aligned}$$

où tous les $\overline{\varepsilon}_{X_k}$ sont d'abord initialisés à $1/2$, c'est-à-dire à 1 bit de précision. Il est évident que la modification ne doit être faite que si la puissance x^k en question n'est pas x, c'est-à-dire $k \neq 1$.

Cette modification est locale. Elle n'interfère pas avec le reste de la preuve de l'algorithme 6. En effet, on a

$$\begin{aligned}
|\varepsilon_T(x)| &\leq \varepsilon_{X_k}(x) + \varepsilon_Q(x) + \varepsilon_M(x) + \\
&\quad \varepsilon_{X_k}(x) \cdot \varepsilon_Q(x) + \varepsilon_{X_k}(x) \cdot \varepsilon_M(x) + \varepsilon_Q(x) \cdot \varepsilon_M(x) + \varepsilon_{X_k}(x) \cdot \varepsilon_M(x) \cdot \varepsilon_Q(x) \\
&\leq \frac{\overline{\varepsilon}}{\overline{\beta}} \cdot (3/8 + 3/64 + 1/512) \\
&= 217/512 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}} \\
&\leq 7/16 \cdot \frac{\overline{\varepsilon}}{\overline{\beta}}
\end{aligned}$$

La borne pour $\varepsilon_T(x)$ n'est donc pas modifiée.

Deux des sous-problèmes pour une génération de séquences de Horner avec précalcul de puissances de x sont traités. L'algorithme 6 modifié en lui intégrant l'algorithme 10 et en adaptant le calcul des erreurs cibles calcule quatre listes : la liste ℓ des exposants k des puissances x^k nécessaires et les listes d'erreurs cibles pour les additions et multiplications dans l'évaluation de Horner et pour les puissances précalculées.

Intéressons-nous alors au troisième problème à résoudre : générer une séquences de multiplication pour calculer toutes ces puissances x^k de x pour $k \in \ell$ avec une erreur bornée par l'erreur cible $\overline{\varepsilon}_{X_k}$. Faisons d'abord abstraction du caractère flottant des calcul et des erreurs d'arrondi.

Le calcul d'une k-ième puissance d'un nombre x avec k un entier fixé est un problème connu dans la littérature [73, 54]. L'algorithme qui consiste à calculer x^k par k multiplications avec x est toute de suite écarté. C'est la méthode des paysans russes qui donne une complexité logarithmique en k. La méthode épelle k bit par bit et fait soit une mise au carré pour un bit à 0 soit une mise au carré et une multiplication pour un bit à 1. Cette approche est appelé exponentiation rapide et peut être encore améliorée. L'optimal en complexité est obtenu pour la plus courte chaîne d'addition $2 = c_1 < c_2 < \cdots < c_\lambda = k$, $c_i \in \mathbb{N}$ et $\forall i, \exists j, k < i, c_i = c_j + c_k$. Calculer la plus courte chaîne d'addition pour un entier donné est un problème difficile mais abordable pour de petites valeurs de k, typiquement pour $k \leq 1000$ [125].

En revanche, le calcul simultané de plusieurs puissances x^k, $k \in \ell$, connu sous le nom de multi-exponentiation, n'a pas été largement traité dans la littérature. Les travaux précédents [38, 99] ne considèrent le problème que dans une arithmétique modulaire et seulement pour des différences $k_1 - k_2$, $k_1, k_2 \in \ell$, très petites. Ils se concentrent plus sur le cas où plusieurs valeurs différentes doivent être élevées à un petit nombre de puissances proches.

Notre cadre génère des problèmes de multi-exponentiation différents :

- Seule une valeur, la variable x, doit être mise à de diverses puissances x^k, $k \in \ell$. Pour obtenir la complexité optimale théorique, l'algorithme devrait exhiber les plus courtes chaînes d'addition ayant, deux à deux, les préfixes les plus longs possibles, ce qui est difficile en toute généralité.
- Comme sera discuté plus en détail à la Section 6.3.2, des coefficients nuls répétés dans un polynôme approchant une fonction reflètent, comme dans le polynôme de Mac Laurin, des symétries de la fonction. De telles symétries sont fréquentes. Cela implique que le problème de multi-exponentiation dégénère souvent en un problème de calcul d'une seule puissance avec exposant pair. L'algorithme de multi-exponentiation doit donc se dégrader gracieusement pour traiter ce cas spécial.
- Toujours à cause des propriétés de symétrie de la fonction approchée, les exposants de différentes puissances à précalculer sont souvent eux-mêmes des puissances entières de 2. Par exemple, $k_1 = 2$, $k_2 = 4$ et $k_3 = 8$. L'algorithme de calcul des puissances pourra alors favoriser les mises au carré devant les multiplications.
- Dans l'arithmétique multi-double, une mise au carré est légèrement moins chère qu'une multiplication. Cela rejoint le point qui précède.
- Même si notre algorithme de génération de séquences de Horner est censé traiter tout polynôme et de tout degré, en pratique la puissance maximale à calculer sera bornée par à peu près 32, voire moins.

L'algorithme glouton suivant permet de traiter le problème de la détermination des puissances intermédiaires nécessaires. L'algorithme donne une liste de puissances intermédiaires de type exponentiation rapide si seule une puissance est nécessaire au départ. Le traitement de ce cas spécial favorise également les mises au carré devant les multiplications.

Entrées : une liste ℓ d'entiers non-nuls distincts, triés dans l'ordre décroissant
Sorties : une liste s d'entiers non-nuls distincts, triés dans l'ordre décroissant, telle que

$$\ell \subseteq s, \quad \max(\ell) = \max(s)$$

et

$$\forall k \in s, (k = 1 \vee \exists i, j \in s, (i, j < k \wedge k = i + j))$$

Algorithme `déterminePuissancesIntermédiaires` (ℓ) :
début
  si $\ell = []$ **alors renvoyer** ℓ;
  décomposer $k :: \ell' \leftarrow \ell$;
  si $k = 1$ **alors renvoyer** $[k]$;
7 $s' \leftarrow$ `déterminePuissancesIntermédiaires(`ℓ'`);`
  pour tous les $i \in s'$ **faire**
    pour tous les $j \in s'$ **faire**
11   **si** $k = i + j$ **alors renvoyer** $k :: s'$;
    fin
  fin
  si k *est pair* **alors**
16    $\ell'' \leftarrow$ `insérerDécroissant(`s',$\frac{k}{2}$`);`
  sinon
    si $s' = []$ **alors**
20      $\ell'' \leftarrow [k - 1, 1]$;
    sinon
      décomposer $i :: r \leftarrow s'$;
      $j \leftarrow k - i$;
25      $\ell'' \leftarrow$ `insérerDécroissant(`s',j`);`
    fin
  fin
  $s'' \leftarrow$ `déterminePuissancesIntermédiaires(`ℓ''`);`
30 **renvoyer** $k :: s''$;
fin

Algorithme 11 : L'algorithme `déterminePuissancesIntermediaires`

Ce dernier algorithme 11 utilise la procédure `insérerDécroissant` qui insère un entier dans une liste tout en gardant son caractère trié.

La preuve de correction de l'algorithme 11 passe d'abord par une preuve de sa terminaison. Celle-ci devient aisée en considérant l'ordre lexicographique des ordres bien-fondés suivants : l'ordre sur le cardinal de l'ensemble des nombres $k \in [\![1; \max(\ell)]\!]$ qui n'appartiennent pas à ℓ, formellement $|\{k \in [\![1; \max(\ell)]\!] \, | k \notin \ell\}|$, et l'ordre sur la longueur de la liste ℓ. En effet, à chaque appel récursif, il y a soit au moins un rajout d'une puissance intermédiaire (lignes 16, 20 et 25) soit raccourcissement de la liste ℓ (ligne 7). La terminaison étant établie, une preuve par induction est possible pour la correction de l'algorithme. Si l'algorithme renvoie un résultat à la ligne 11, il est clair par le test qui précède que $k = i + j$ avec i, j contenus dans la liste renvoyé et plus petits que k. En effet, k est le plus grand élément de la liste ℓ parce que celle-ci est triée dans l'ordre décroissant. Par hypothèse de récurrence, il

existe une décomposition pour les autres éléments de la liste renvoyée. Si l'algorithme se termine à la ligne 30, l'hypothèse de récurrence assure que tout élément de la liste s'' appartient à ℓ''. Cette dernière liste ℓ'' contient par construction deux éléments i, j tels que $i + j = k$: on a soit $i = \frac{k}{2}$ et $j = \frac{k}{2}$, soit $i = k - 1$ et $j = 1$ soit i l'élément maximum dans la sous-liste s' et $j = k - i$.

La complexité de l'algorithme 11 peut être assez importante. En effet, l'algorithme peut faire jusqu'à $\mathcal{O}(n)$ appels récursifs à la ligne 30 avec une liste de valeur maximale n de plus en plus remplie. Ces appels sont suivies de $\mathcal{O}(n)$ appels récursif à la ligne 7 dont chacun coûte au plus $\mathcal{O}(n^2)$ opérations pour les deux boucles sur les éléments de s'. Ceci donne une complexité de $\mathcal{O}(n^4)$. L'algorithme est tout de même très performant en pratique où la puissance maximale x^n à calculer ne dépasse guère x^8. Dans notre cadre pratique, on observe qu'aucun polynôme d'approximation d'une fonction mathématique ne contient plus de 8 coefficients consécutifs nuls.

Considérons maintenant le rajout de la gestion des erreurs d'arrondi à l'algorithme 11. Pour cela, analysons d'abord l'erreur d'arrondi entachant une puissance $X_k(x)$ calculée à partir de $X_i(x)$ et $X_j(x)$ avec $k = i + j$.

Notons

$$\varepsilon_{M_{ij}}(x) = \frac{X_i(x) \otimes X_j(x) - X_i(x) \cdot X_j}{X_i(x) \cdot X_j(x)}$$

l'erreur relative de la multiplication entre les approximations $X_i(x)$ et $X_j(x)$. On a alors

$$\begin{aligned}
\varepsilon_{X_k}(x) &= \frac{X_k(x) - x^k}{x^k} \\
&= \frac{X_i(x) \otimes X_j(x) - x^i \cdot x^j}{x^i \cdot x^j} \\
&= \frac{x^i \cdot (1 + \varepsilon_{X_i}(x)) \cdot x^j \cdot \big(1 + \varepsilon_{X_j}(x)\big) \cdot \big(1 + \varepsilon_{M_{ij}}\big) - x^i \cdot x^j}{x^i \cdot x^j} \\
&= \varepsilon_{X_i}(x) + \varepsilon_{X_j}(x) + \varepsilon_{M_{ij}}(x) + \\
&\quad \varepsilon_{X_i}(x) \cdot \varepsilon_{X_j}(x) + \varepsilon_{X_i}(x) \cdot \varepsilon_{M_{ij}}(x) + \varepsilon_{X_j}(x) \cdot \varepsilon_{M_{ij}}(x) + \\
&\quad \varepsilon_{X_i}(x) \cdot \varepsilon_{X_j}(x) \cdot \varepsilon_{M_{ij}}(x)
\end{aligned}$$

Cette analyse montre donc que l'erreur de multiplication et les erreurs dont sont entachés $X_i(x)$ et $X_j(x)$ entrent de la même façon dans l'erreur de $X_k(x)$ par rapport à x^k.

En se basant sur cette analyse, il est donc possible de rajouter un calcul de bornes d'erreur cibles à l'algorithme 11. L'algorithme prend en deuxième argument la liste des erreurs cibles $\overline{\varepsilon}_{X_k}$ bornant $\varepsilon_{X_k}(x)$. Cette liste est lui est fourni au départ par l'algorithme 7 modifié. Les erreurs cibles $\overline{\varepsilon}_{X_i}$ n'appartenant pas à cette liste sont initialisées à 1, c'est-à-dire à 0 bits de précision. L'algorithme adapte ensuite les erreurs cibles $\overline{\varepsilon}_{X_i}$ et $\overline{\varepsilon}_{X_j}$ quand, aux lignes 11, 16, 20 ou 25, il exhibe i et j tels que $k = i + j$: à partir de la borne $\overline{\varepsilon}_{X_k}$, algorithme calcule

$$\begin{aligned}
\overline{\varepsilon}_{X_i} &\leftarrow \min\left(\overline{\varepsilon}_{X_i}, 1/4 \cdot \overline{\varepsilon}_{X_k}\right) \\
\overline{\varepsilon}_{X_j} &\leftarrow \min\left(\overline{\varepsilon}_{X_j}, 1/4 \cdot \overline{\varepsilon}_{X_k}\right) \\
\overline{\varepsilon}_{M_{ij}} &\leftarrow 1/4 \cdot \overline{\varepsilon}_{X_k}
\end{aligned}$$

L'analyse d'erreur étant faite, il est facile de vérifier que cette répartition garantit que les erreurs ε_{X_k} entachant les puissances de x calculées sont inférieurs aux bornes $\overline{\varepsilon}_{X_k}$ détermi-

nées :

$$\begin{aligned}
|\varepsilon_{X_k}(x)| &\leq 1/4 \cdot \overline{\varepsilon}_{X_k} + 1/4 \cdot \overline{\varepsilon}_{X_k} + 1/4 \cdot \overline{\varepsilon}_{X_k} + \\
&\quad 1/4 \cdot \overline{\varepsilon}_{X_k} \cdot 1/4 \cdot \overline{\varepsilon}_{X_k} + 1/4 \cdot \overline{\varepsilon}_{X_k} \cdot 1/4 \cdot \overline{\varepsilon}_{X_k} + 1/4 \cdot \overline{\varepsilon}_{X_k} \cdot 1/4 \cdot \overline{\varepsilon}_{X_k} + \\
&\quad 1/4 \cdot \overline{\varepsilon}_{X_k} \cdot 1/4 \cdot \overline{\varepsilon}_{X_k} \cdot 1/4 \cdot \overline{\varepsilon}_{X_k} \\
&\leq \overline{\varepsilon}_{X_k} \cdot (3/4 + 3/16 + 1/64) \\
&= \overline{\varepsilon}_{X_k} \cdot 61/64 \\
&\leq \overline{\varepsilon}_{X_k}
\end{aligned}$$

L'intégration des modifications nécessaires pour le calcul d'erreur aux algorithmes 6, 10 et 11, nous permet donc de déterminer, pour un polynôme p et un domaine dom donné, une séquence de multiplications flottantes à précision $\overline{\varepsilon}_{M_{ij}}$ pour précalculer les puissances nécessaires dans une séquence de Horner. Il est facile de traduire la séquence de ces erreurs cibles $\overline{\varepsilon}_{M_{ij}}$ en une séquence de largeurs multi-doubles (σ, τ, ρ) des opérandes et résultats des multiplications impliquées. En effet, il suffit toujours de simuler statiquement les exponentiations. L'algorithme 7 y sert de base. Comme dans l'algorithme 9, des tests supplémentaires vérifient les conditions additionnelles découlant du fait qu'une multiplication de deux nombres sur n et m doubles ne peut produire plus de $n + m$ doubles. Il est donc possible de générer une séquence d'opérations multi-double pour le précalcul de puissances pour une séquence de Horner évaluant le polynôme p.

Ce qui manque finalement est l'adaptation de l'algorithme 9 pour utiliser ces puissances précalculées dans la génération d'une séquence de Horner. Cette adaptation est en principe simple : au lieu de passer sur tous les coefficients du polynôme, l'algorithme itère sur ses coefficients non-nuls. À la place de la variable x, l'algorithme considère une multiplication par la bonne puissance précalculée $X_k(x)$ à chaque étape.

Ici, on observe une petite complication : les puissances $X_k(x)$ sont précalculées à une précision satisfaisant le pire des cas dans le schéma de Horner. En effet, comme la précision nécessaire au cours d'une évaluation de Horner augmente dans notre cadre, une même puissance peut être nécessaire deux fois et avec deux précisions différentes. Elle n'est évidemment précalculée qu'une fois et avec la plus grande précisions de toutes ses occurrences. Dans le cadre de la multi-double, cela implique que les puissances précalculées $X_k(x)$ sont stockées sur un nombre de flottants double précision correspondant à cette pire précision.

Au moment de leur utilisation à un endroit où une précision moindre suffit, l'utilisation directe de la valeur de $X_k(x)$ n'est pas possible. Le résultat d'une opération multi-double s'écrit toujours sur un nombre de flottants double précision au moins égal au nombre de flottants pour chacun des opérandes. L'utilisation directe d'une puissance précalculée sur un nombre de flottants double précision trop élevé et l'application de cette dernière règle anéantirait les avantages de l'adaptation de le précision : le reste de l'évaluation de Horner se ferait avec la précision multi-double de la puissance précalculée. Il convient alors d'arrondir une puissance précalculée, en négligeant certains flottants double précision (troncature), si elle est trop précise pour une étape de Horner. Cela provoque évidemment une erreur d'arrondi supplémentaire qui doit être prise en compte.

Il est important d'observer que cette source d'erreur supplémentaire ne peut pas être intégrée dans la répartition des erreurs cibles pour les différentes opérations que font les algorithmes proposés ci-dessus. En effet, ces algorithmes déterminent, par la répartition des erreurs cibles, le nombre de flottants double précision en résultat de chaque opération. Mais

c'est justement l'incompatibilité de ce nombre de flottants en résultat avec le nombre de flottants pour la puissance précalculée qui provoque le rajout d'une nouvelle source d'erreur. Il est alors nécessaire de trouver une modification locale de l'algorithme de génération de séquences de Horner pour prendre en compte cette erreur d'arrondi supplémentaire.

Notons $\circ\,(X_k(x))$ le résultat de la troncature du nombre multi-double $X_k(x)$. Désignons par $\varepsilon_R(x)$ l'erreur associée :

$$\varepsilon_R(x) = \frac{\circ\,(X_k(x)) - X_k(x)}{X_k(x)}$$

Il est facile de voir que cette erreur $\varepsilon_R(x)$ et l'erreur $\varepsilon_M(x)$ de la multiplication entre la puissance arrondi et la valeur $Q(x)$ du sous-polynôme se comportent de la même façon part rapport à leur importance dans l'erreur combinée suivante :

$$\begin{aligned}
\varepsilon_C(x) &= \frac{\circ\,(X_k(x)) \otimes Q(x) - X_k(x)\cdot Q(x)}{X_k(x)\cdot Q(x)} \\
&= \frac{X_k(x)\cdot Q(x)\cdot(1+\varepsilon_M(x))\cdot(1+\varepsilon_R(x)) - X_k(x)\cdot Q(x)}{X_k(x)\cdot Q(x)} \\
&= \varepsilon_R(x) + \varepsilon_M(x) + \varepsilon_R(x)\cdot\varepsilon_M(x)
\end{aligned}$$

En effet, s'il n'y a pas de nécessité d'arrondir $X_k(x)$, on a $\varepsilon_R(x) = 0$ et donc $\varepsilon_C(x) = \varepsilon_M(x)$.

En tout cas, il est possible de remplacer l'erreur de la multiplication seule par l'erreur combinée $\varepsilon_C(x)$. C'est ce terme d'erreur qui doit satisfaire

$$|\varepsilon_C(x)| \leq \overline{\varepsilon}_M$$

où $\overline{\varepsilon}_M$ est la borne déterminée par l'algorithme 6 de répartition des erreurs cibles. L'algorithme 9, simulant les différentes étapes de Horner et déterminant le nombre de flottants double précision (σ, τ, ρ) dans les opérations, peut être modifié comme suit :

1. Étant donné $\overline{\varepsilon}_M$, une valeur préliminaire ρ est déterminée à l'aide de l'algorithme 7. Ce choix implique une borne d'erreur pour la multiplication a posteriori ; notons-la $\widetilde{\overline{\varepsilon}}_M \in \mathbb{R}^+$. On a

   $$\widetilde{\overline{\varepsilon}}_M \leq \overline{\varepsilon}_M \quad \text{et } |\varepsilon_M(x)| \leq \widetilde{\overline{\varepsilon}}_M$$

   Ce choix de ρ donne également une borne pour le nombre σ de flottants double précision de la valeur de $\circ\,(X_k(x))$.

   Si $X_k(x)$ est stocké sur au plus $\sigma \leq \rho$ flottants double précision, aucun arrondi n'est à faire : $\varepsilon_R(x) = 0$ et $|\varepsilon_C(x)| = |\varepsilon_M(x)| \leq \overline{\varepsilon}_M$.

   Sinon, une troncature de $X_k(x)$ à $\sigma = \rho$ flottants double précision et faite. Une borne d'erreur a posteriori $\widetilde{\overline{\varepsilon}}_R \in \mathbb{R}^+$ est déterminée pour l'erreur d'arrondi $\varepsilon_R(x)$:

   $$|\varepsilon_R(x)| \leq \widetilde{\overline{\varepsilon}}_R$$

   Les deux bornes a posteriori $\widetilde{\overline{\varepsilon}}_M$ et $\widetilde{\overline{\varepsilon}}_R$ sont intégrées pour obtenir une borne a posteriori $\widetilde{\overline{\varepsilon}}_C$ pour l'erreur combinée $\varepsilon_C(x)$:

   $$\begin{aligned}
   \widetilde{\overline{\varepsilon}}_C &= \widetilde{\overline{\varepsilon}}_R + \widetilde{\overline{\varepsilon}}_M + \widetilde{\overline{\varepsilon}}_R\cdot\widetilde{\overline{\varepsilon}}_M \\
   |\varepsilon_C(x)| &\leq \widetilde{\overline{\varepsilon}}_C
   \end{aligned}$$

   Si cette borne $\widetilde{\overline{\varepsilon}}_C$ est plus petite que l'erreur cible $\overline{\varepsilon}_M$, le choix de σ et ρ satisfait les bornes de l'analyse d'erreur pour cette étape de Horner et peut être maintenu.

2. Sinon, il suffit d'augmenter ρ et σ de 1. En effet, comme $\sigma = \rho$, $\widetilde{\overline{\varepsilon}}_R$ et $\widetilde{\overline{\varepsilon}}_M$ sont dans un rapport 4 à cause des propriétés de l'arithmétique multi-double. Comme $\widetilde{\overline{\varepsilon}}_M \leq \overline{\varepsilon}_M$ et $\widetilde{\overline{\varepsilon}} \leq 2^{-51}$, cela implique que la borne de l'erreur combinée a posteriori $\widetilde{\overline{\varepsilon}}_C$ n'est jamais plus grande que $6 \cdot \overline{\varepsilon}_M$. Grâce au caractère discret de l'arithmétique multi-double, le rajout d'un flottant double précision à l'opérande et au résultat de la multiplication divise l'erreur combinée par au moins 8. Donc $1/8 \cdot \widetilde{\overline{\varepsilon}}_C \leq 6/8 \cdot \overline{\varepsilon}_M \leq \overline{\varepsilon}_M$.

Observons que le fait qu'un arrondi supplémentaire peut être fait dans le premier cas, sans que ρ doive être modifié, est dû au caractère discret de l'arithmétique multi-double et à la nature de la répartition des erreurs dans le pire cas.

Il nous est alors possible d'intégrer toutes ces modifications pour obtenir un algorithme pour la génération de séquences de Horner avec précalcul des puissances nécessaire et adaptation de la précision multi-double.

6.2.4 Gestion du chevauchement

Malheureusement, l'algorithme, proposé aux sections 6.2.2 et 6.2.3 qui précèdent, n'est toujours pas complet. Dans l'analyse d'erreur de l'évaluation Horner, nous avons toujours supposé que l'arithmétique triple-double fournit une précision fixe, plus grande que 102 bits, et que cette précision suffisait dans le cadre donné, où l'erreur cible $\overline{\varepsilon}$ est plus grande que 2^{-140}, c'est-à-dire 140 bits. Comme le chapitre 3 le montre, l'arithmétique triple-double peut répondre à ces besoins. En revanche, une analyse du chevauchement des valeurs triple-double et une insertion d'opérations de renormalisation sont nécessaires pour cela. Dirigeons donc notre intérêt sur l'automatisation de ces deux tâches pour compléter l'algorithme de génération de séquences Horner.

La majoration automatique des chevauchements des valeurs triple-double dans une séquence de Horner (ou dans le précalcul des puissances) passe encore une fois par une simulation statique. L'algorithme de simulation attache une borne de chevauchement à toute valeur triple-double. Cette borne est calculée en fonction de la borne de chevauchement des opérandes de l'opération produisant la valeur et le type d'opération. Ce dernier est fixé par la combinaison (σ, τ, ρ) de nombres de flottants double précision en opérandes et résultat. Cette fonction de transfert pour bornes de chevauchement a été établie explicitement pour tout type d'opération sous forme d'un théorème correspondant (cf. chapitre 3). Codée dans l'algorithme, elle peut être récursivement appliquée pour déterminer une borne pour toute valeur triple-double.

Le chevauchement d'une valeur triple-double en opérande n'a pas seulement une influence sur le chevauchement de la valeur triple-double en résultat d'une opération. Il détermine surtout la précision de l'opération, c'est-à-dire l'erreur d'arrondi provoquée. Plus le chevauchement est grand, plus l'erreur d'arrondi est grande. C'est ces erreurs d'arrondi $\varepsilon_A(x)$ ou $\varepsilon_M(x)$ que l'algorithme de génération de séquences doit vérifier : les bornes d'erreur cible $\overline{\varepsilon}_A$ et $\overline{\varepsilon}_M$ doivent être satisfaites. Si une borne d'erreur ne peut pas être satisfaite a posteriori pour une séquence de Horner, une opération de renormalisation doit être inséré dans la séquence avant l'opération en question. La valeur triple-double ne montre plus de chevauchement ce qui fait revenir l'erreur d'arrondi sous la borne donnée.

Le lien entre une borne pour l'erreur d'une opération d'un certain type et la borne de chevauchement de ces opérandes est explicitement connu. Un théorème correspondant a été prouvé manuellement (cf. chapitre 3). Le lien est une fonction de transfert qui peut être

codée dans l'algorithme. La vérification de l'erreur cible et l'insertion d'une opération de renormalisation dans la séquence sont donc algorithmiquement simples.

L'insertion de renormalisation provoque un changement des bornes de chevauchement. Le calcul des bornes de chevauchement et des bornes d'erreur qui en découlent ainsi que l'insertion des renormalisation aux endroits nécessaires doivent être effectués en même temps.

L'implantation pratique de ce calcul est fastidieuse. Le nombre de combinaisons pour le type d'opération (σ, τ, ρ) est élevé. Pour chaque type, la fonction de transfert du chevauchement et de la borne d'erreur doit être codée. L'évaluation de cette fonction se faisant également en virgule flottantes, des erreurs d'arrondi peuvent survenir et doivent être majorées correctement. L'opération de renormalisation est une opération exacte. Statiquement, la valeur avant renormalisation et celle renormalisée sont égales. Pourtant les deux valeurs ne comportent pas pareillement en opérande d'une opération triple-double. La gestion des renormalisation demande donc de maintenir un numéro de version pour toute valeur qui peut être probablement réutilisée. Dans notre cadre de séquences de Horner avec optimisation pour des coefficients nuls, une valeur peut être réutilisée si elle est une puissance précalculée ou intervient dans cette multi-exponentiation.

6.2.5 Génération automatique de preuves Gappa

Bien que l'on ait confiance que l'analyse d'erreur faite pour borner et répartir les erreurs cibles lors de la génération d'une séquence pour un polynôme soit juste, une erreur d'implantation pourrait mettre une implantation en danger. Une preuve formelle, établie a posteriori sur le code produit, est souhaitée. Si elle devait être faite manuellement après analyse d'un programme automatiquement généré, l'automatisation n'aurait plus grand intérêt. On tâchera donc de générer une preuve formelle lors de la production automatique de code.

Comme l'utilisation manuelle de Gappa l'a montré (cf. section 4.3), cet outil de preuve formelle est suffisamment avancé dans son automatisation de génération de preuve pour être utilisé pour des séquences de Horner. Typiquement, pour ce schéma d'évaluation, le moteur de Gappa peut appliquer une technique d'analyse d'erreur récursive comme celle donnée à la section 6.2.2.

Il est donc relativement simple de produire une preuve Gappa pour une implantation automatique d'une évaluation de Horner. La génération de la preuve consiste, dans ce cas, dans un affichage en syntaxe Gappa (pretty-print) du programme émis déjà en syntaxe C. Pour chaque opération, on retient, en plus de la suite des opérations, les informations suivantes qui sont traduites ensuite en Gappa :

- Le but de l'opérateur : addition, multiplication, chargement de constante ou renormalisation.
- L'erreur produite par l'opérateur ou son exactitude dans le cas d'une addition ou multiplication exacte produisant un double-double.
- Le type de l'opérateur (σ, τ, ρ) correspondant au nombre de flottants double précision en opérandes et résultat.
- Les bornes de chevauchements pour les valeurs triple-double produite par l'opération.

Ces informations sont transformées en des preuves Gappa de structure traditionnelle, dictée par la syntaxe Gappa. On distingue cinq parties :

1. Dans la première partie, la suite des opérations est affichée avec les différents arrondis flottants. Les opérations multi-double sont traduites par des opérateurs à erreur

relative `add_rel<>` et `mul_rel<>`. Pour la précision triple-double, la précision effective, obtenue après calcul de chevauchement, est indiquée dans l'instantiation de ces opérateurs.

Dans cette partie, une décomposition de valeurs multi-doubles $v_h + v_m + v_l$ en leurs composantes $v_h + v_m$ et v_h approchant la valeur est également émise. Cela implique une variable de chevauchement pour la triple-double. Elle sera bornée par hypothèse.

Les variables Gappa utilisées portent le même nom que les variables du code C produit. Cette partie définit $P(x)$.

2. Ensuite, la suite des opérations est affichée encore une fois en omettant les arrondis flottants. Les variables Gappa employées ici sont préfixés d'un identifiant caractéristique. Cette partie définit $p(x)$.
3. Une équation Gappa déclare l'erreur d'arrondi totale $\varepsilon(x) = \frac{P(x)-p(x)}{p(x)}$.
4. L'implication à prouver est ensuite donnée : sous les hypothèses suivantes, $\varepsilon(x)$ est borné en valeur absolue par $\overline{\varepsilon}$.

   Les hypothèse comprennent les bornes pour l'argument x, qui doit être dans le domaine $x \in dom$ et les bornes de chevauchement. Pour permettre à Gappa de comprendre la décomposition de valeurs multi-doubles leur composantes, il y est nécessaire de faire l'hypothèse que les valeurs décomposées sont comprises en valeur absolue dans le domaine de représentation de la double précision $\left[2^{-1074}; 2^{1024}\right[$.

   Pour des raisons techniques, le domaine de définition dom doit être découpé en sa partie positive et négative s'il contient 0. L'implication à prouver se décompose alors en deux parties également.
5. Des indications au moteur Gappa reliant les valeurs calculées avec arrondi à leurs correspondants mathématiquement exacts complètent le fichier. Cela permet à Gappa de composer les erreurs de façon récursive dans son analyse d'erreur.

   Des indications supplémentaires sont nécessaires pour exprimer certaines propriétés de la gestion du chevauchement lors de la décomposition de valeurs multi-doubles. À la lecture humaine de la preuve Gappa, ces règles de réécriture semblent complexes mais sont de simple instantiations d'une même règle émise pour toute décomposition. Remarquons tout de même qu'il est difficile de prévoir le comportement de Gappa. Ainsi, dans une preuve automatiquement générée, trop d'indications sont souvent données à Gappa. Cela augmente le temps nécessaire pour le moteur Gappa de gérer la preuve par rapport à une preuve écrite à la main.

6.2.6 Résultats, conclusions et extensions possibles

L'algorithme de génération de séquences de Horner, que l'on vient de voir, a été implanté à l'intérieur de l'outil Sollya. Il y est disponible à travers une commande Sollya spéciale, appelée `implementpoly`.

Il y a deux raisons pourquoi l'implantation s'est faite à l'intérieur de Sollya, c'est-à-dire en langage C, et non dans le langage haut niveau Sollya. La première est historique : au moment de cette mise en œuvre, le langage Sollya était encore en train de se créer et ne pouvait nullement satisfaire les besoins pour un tel algorithme. Le code comportant à peu près 6000 lignes, il n'a pas été porté dans le langage Sollya dans la suite. La deuxième raison pour le choix du langage C et non Sollya est plus importante. La génération automatique

du programme C pour la séquence et de la preuve Gappa demande de respecter la syntaxe C et Gappa. Des structures de données ainsi que des manipulation de chaînes de caractères doivent donc être supportées. Le langage de Sollya, un outil numérique, n'est pas adapté à ce type d'application.

L'implantation de polynôme automatisée a servie lors de la mise en œuvre de plusieurs fonctions dans la bibliothèque CRLibm. C'est surtout la possibilité de générer automatiquement des preuves de bornes pour les erreur d'arrondi en Gappa qui est apprécié. Elle diminue considérablement le temps de développement. Pour des polynômes typiques utilisés dans CRLibm avec un degré inférieur à 50, la commande `implementpoly` génère le code C et la preuve Gappa en 1 à 3 secondes sur des machines actuelles.

Évidemment, la rapidité du développement d'une implantation ne vaut rien si la qualité du code est faible. Comme premier paramètre pour la qualité du code, le temps d'évaluation du polynôme par la séquence est important. La génération automatique peut être considérée mure quand elle produit un code de même qualité qu'un spécialiste humain.

Pour juger le générateur automatique, comparons alors, à titre d'exemple, une implantation manuelle d'un polynôme dans CRLibm à un code synthétisé automatiquement. La tâche consiste à donner une séquence d'évaluation pour un polynôme p de degré 37 approchant la fonction asin dans le domaine $dom = [-0.185; 0.185]$. La fonction étant impaire, le polynôme est choisi pour n'avoir que de monômes impairs. L'implantation ne doit pas engendrer d'erreur d'arrondi $\varepsilon_P(x)$ plus grande que $\overline{\varepsilon}_P = 2^{-118}$.

Les deux codes, manuellement écrit ou généré, se ressemblent : comme le polynôme est impair, le carré de la variable x est précalculé. L'évaluation de Horner se fait d'abord en double précision pour passer ensuite aux précisions double-double et triple-double. Dans les deux codes, le calcul des chevauchements dans arithmétique triple-double provoque l'utilisation de deux renormalisations.

Le code écrit manuellement pour la bibliothèque CRLibm se lit comme suit :

```
Mul12(&xSqh,&xSql,x,x);

highPoly = PolyLowC27h + xSqh * (PolyLowC29h + xSqh * (PolyLowC31h +
           xSqh * (PolyLowC33h + xSqh * (PolyLowC35h + xSqh * PolyLowC37h))));
Mul12(&tt1h,&tt1l,xSqh,highPoly);
Add22(&t1h,&t1l,PolyLowC25h,0,tt1h,tt1l);
MulAdd22(&t2h,&t2l,PolyLowC23h,PolyLowC23m,xSqh,xSql,t1h,t1l);
MulAdd22(&t3h,&t3l,PolyLowC21h,PolyLowC21m,xSqh,xSql,t2h,t2l);
MulAdd22(&t4h,&t4l,PolyLowC19h,PolyLowC19m,xSqh,xSql,t3h,t3l);
MulAdd22(&t5h,&t5l,PolyLowC17h,PolyLowC17m,xSqh,xSql,t4h,t4l);
MulAdd22(&t6h,&t6l,PolyLowC15h,PolyLowC15m,xSqh,xSql,t5h,t5l);
MulAdd22(&t7h,&t7l,PolyLowC13h,PolyLowC13m,xSqh,xSql,t6h,t6l);
Mul23(&tt8h,&tt8m,&tt8l,xSqh,xSql,t7h,t7l);
Add33(&t8h,&t8m,&t8l,PolyLowC11h,PolyLowC11m,PolyLowC11l,tt8h,tt8m,tt8l);
Mul233(&tt9h,&tt9m,&tt9l,xSqh,xSql,t8h,t8m,t8l);
Add33(&t9h,&t9m,&t9l,PolyLowC9h,PolyLowC9m,PolyLowC9l,tt9h,tt9m,tt9l);
Mul233(&tt10h,&tt10m,&tt10l,xSqh,xSql,t9h,t9m,t9l);
Add33(&t10h,&t10m,&t10l,PolyLowC7h,PolyLowC7m,PolyLowC7l,tt10h,tt10m,tt10l);
Mul233(&tt11hover,&tt11mover,&tt11lover,xSqh,xSql,t10h,t10m,t10l);
Renormalize3(&tt11h,&tt11m,&tt11l,tt11hover,tt11mover,tt11lover);
Add33(&t11h,&t11m,&t11l,PolyLowC5h,PolyLowC5m,PolyLowC5l,tt11h,tt11m,tt11l);
Mul233(&tt12h,&tt12m,&tt12l,xSqh,xSql,t11h,t11m,t11l);
Add33(&t12h,&t12m,&t12l,PolyLowC3h,PolyLowC3m,PolyLowC3l,tt12h,tt12m,tt12l);
Mul123(&xCubeh,&xCubem,&xCubel,x,xSqh,xSql);
Mul33(&tt13h,&tt13m,&tt13l,xCubeh,xCubem,xCubel,t12h,t12m,t12l);
Add133(&t13h,&t13m,&t13l,x,tt13h,tt13m,tt13l);
Renormalize3(polyh,polym,polyl,t13h,t13m,t13l);
```

Listing 6.1 – Une séquence d'évaluation pour p approchant $\mathrm{asin}\, x$ implantée à la main.

On remarque que l'auteur du code se rend compte du fait que le polynôme commence par

$p(x) = x \cdot \big(1 + x^2 \cdot q(x)\big)$. Il essaie d'optimiser l'évaluation en évitant le chargement de la constante $1 : p(x) = x + x^3 \cdot q(x)$.

L'algorithme de génération de séquence de Horner n'est pas capable de faire cette optimisation. En revanche, les borne d'erreur cibles calculées par l'algorithme sont plus fines que celles utilisées par l'humain. En conséquence, moins d'opérations sont faites en double-double et triple-double. En effet, le passage de la double-double à la triple-double, par exemple, ne se fait qu'à l'addition du coefficient p_7 au lieu de celle du coefficient p_{11}.

Voilà le code produit automatiquement :

```
Mul12(&p_x_0_p2h,&p_x_0_p2m,x,x);

p_t_1_0h = p_c_37h;
p_t_2_0h = p_t_1_0h * p_x_0_p2h;
p_t_3_0h = p_c_35h + p_t_2_0h;
p_t_4_0h = p_t_3_0h * p_x_0_p2h;
p_t_5_0h = p_c_33h + p_t_4_0h;
p_t_6_0h = p_t_5_0h * p_x_0_p2h;
p_t_7_0h = p_c_31h + p_t_6_0h;
p_t_8_0h = p_t_7_0h * p_x_0_p2h;
p_t_9_0h = p_c_29h + p_t_8_0h;
p_t_10_0h = p_t_9_0h * p_x_0_p2h;
p_t_11_0h = p_c_27h + p_t_10_0h;
p_t_12_0h = p_t_11_0h * p_x_0_p2h;
Add12(p_t_13_0h,p_t_13_0m,p_c_25h,p_t_12_0h);
MulAdd22(&p_t_14_0h,&p_t_14_0m,p_c_23h,p_c_23m,p_x_0_p2h,p_x_0_p2m,p_t_13_0h,p_t_13_0m);
MulAdd22(&p_t_15_0h,&p_t_15_0m,p_c_21h,p_c_21m,p_x_0_p2h,p_x_0_p2m,p_t_14_0h,p_t_14_0m);
MulAdd22(&p_t_16_0h,&p_t_16_0m,p_c_19h,p_c_19m,p_x_0_p2h,p_x_0_p2m,p_t_15_0h,p_t_15_0m);
MulAdd22(&p_t_17_0h,&p_t_17_0m,p_c_17h,p_c_17m,p_x_0_p2h,p_x_0_p2m,p_t_16_0h,p_t_16_0m);
MulAdd22(&p_t_18_0h,&p_t_18_0m,p_c_15h,p_c_15m,p_x_0_p2h,p_x_0_p2m,p_t_17_0h,p_t_17_0m);
MulAdd22(&p_t_19_0h,&p_t_19_0m,p_c_13h,p_c_13m,p_x_0_p2h,p_x_0_p2m,p_t_18_0h,p_t_18_0m);
MulAdd22(&p_t_20_0h,&p_t_20_0m,p_c_11h,p_c_11m,p_x_0_p2h,p_x_0_p2m,p_t_19_0h,p_t_19_0m);
MulAdd22(&p_t_21_0h,&p_t_21_0m,p_c_9h,p_c_9m,p_x_0_p2h,p_x_0_p2m,p_t_20_0h,p_t_20_0m);
Mul22(&p_t_22_0h,&p_t_22_0m,p_t_21_0h,p_t_21_0m,p_x_0_p2h,p_x_0_p2m);
Add233Cond(&p_t_23_0h,&p_t_23_0m,&p_t_23_0l,p_t_22_0h,p_t_22_0m,p_c_7h,p_c_7m,p_c_7l);
Mul233(&p_t_24_0h,&p_t_24_0m,&p_t_24_0l,p_x_0_p2h,p_x_0_p2m,p_t_23_0h,p_t_23_0m,p_t_23_0l);
Add33(&p_t_25_0h,&p_t_25_0m,&p_t_25_0l,p_c_5h,p_c_5m,p_c_5l,p_t_24_0h,p_t_24_0m,p_t_24_0l);
Mul233(&p_t_26_0h,&p_t_26_0m,&p_t_26_0l,p_x_0_p2h,p_x_0_p2m,p_t_25_0h,p_t_25_0m,p_t_25_0l);
Add33(&p_t_27_0h,&p_t_27_0m,&p_t_27_0l,p_c_3h,p_c_3m,p_c_3l,p_t_26_0h,p_t_26_0m,p_t_26_0l);
Mul233(&p_t_28_0h,&p_t_28_0m,&p_t_28_0l,p_x_0_p2h,p_x_0_p2m,p_t_27_0h,p_t_27_0m,p_t_27_0l);
Add133(&p_t_29_0h,&p_t_29_0m,&p_t_29_0l,p_c_1h,p_t_28_0h,p_t_28_0m,p_t_28_0l);
Renormalize3(&p_t_29_1h,&p_t_29_1m,&p_t_29_1l,p_t_29_0h,p_t_29_0m,p_t_29_0l);
Mul133(&p_t_30_0h,&p_t_30_0m,&p_t_30_0l,x,p_t_29_1h,p_t_29_1m,p_t_29_1l);
Renormalize3(p_resh,p_resm,p_resl,p_t_30_0h,p_t_30_0m,p_t_30_0l);
```

Listing 6.2 – Une séquence d'évaluation pour p générée automatiquement.

C'est la mesure du temps d'évaluation des deux codes qui marque la différence. On mesure[1] un temps d'évaluation de 367 unités de temps pour le code manuellement optimisé et de 270 unités pour le code généré. Donc, le code automatique est non seulement de qualité comparable mais même 26% meilleur ! Évidemment, il est possible de trouver des exemples où le code généré est de qualité inférieure au code manuellement optimisé. Pour les fonctions dans CRLibm pourtant, on n'observe pas de changement important : il s'agit souvent de quelques 2% de ralentissement, comme pour le polynôme approchant $\log(1 + x)$ en CRLibm. Plus de résultats de mesure seront donnés dans la section 6.5.2.

L'algorithme de génération de séquences d'évaluation de polynômes est basé sur un certain nombre d'heuristiques. Seules des séquences de Horner peuvent être générées. Comme on a vu, la qualité du code produit s'apparente de celle qui est attendue pour une bibliothèque comme CRLibm. Pour aller encore plus loin, les extensions suivantes à l'algorithme pourraient être faites :

[1] sur un AMD Opteron bi-processeur cadencé à 1,8 GHz avec Linux 2.6.22-2-amd64 #1 SMP

- Optimisations pour des cas spéciaux de l'arithmétique flottante, dont les spécialistes humains tirent profit. Il s'y agit en premier lieu des cas où une multiplication par un des coefficients est exacte parce qu'il s'agit d'une puissance entière de 2. Le cas d'une soustraction exacte grâce au lemme de Sterbenz ne peut pas arriver dans notre cadre où $\overline{\beta} \leq 1$ (cf. Section 6.3.2).
- Gestion du cas où $\overline{\beta} > 1$, c'est-à-dire où les précisions nécessaires au cours d'une évaluation de Horner n'augmentent pas à chaque étape. Pour cela, il faut une approche plus fine pour l'analyse d'erreur et une approche plus complexe pour la répartition des erreurs cibles. Le schéma de Horner peut être inadapté à ce type de problème, au moins tant que l'on ne s'autorise pas de changement de base par une translation. Ce point mérite d'être analysé et comparé aux travaux dans [111, 55, 92].
- Gestion d'autres précisions dans l'arithmétique utilisée. On y considérera en particulier les autres précisions offertes par les machines actuelles : précision simple et précision double-étendue. Leur gestion permettrait d'une part de réduire l'effet de discrétisation des erreurs cibles pour des cas où la précision double (respectivement double-double) est déjà trop précise. D'autre part, elle pourrait apporter un gain en rapidité d'évaluation sur des systèmes où les opérations simples ont une latence plus faible que celles de la double précision. Évidemment, dans l'optique de l'arithmétique multi-double, une nouvelle question se posera : doit-on considérer l'utilisation d'une arithmétique multi-simple, multi-double-étendue voir mixte comme la double-simple par exemple ?
- Gestion d'autres schémas d'évaluation comme l'évaluation d'Estrin [111, 101]. Cela demande un parcours déjà plus large de possibilités. In fine, l'espace de toutes les séquences évaluant le polynôme complet pourrait être considéré. L'explosion combinatoire correspondante semble être traitable seulement si des heuristiques peuvent être trouvées qui orientent la recherche très tôt vers l'optimal. L'optimalité reste à définir : les phénomènes de discrétisation observés rendent minimales et chaotiques les différences dans le temps d'évaluation de deux séquences.
- Génération de séquences d'évaluation en arithmétique multi-double sans la modularisation en opérateurs prouvés à part. Dans une certaine mesure, l'adaptation des erreurs cibles s'arrête au niveau des opérateurs d'addition et de multiplication de base. Cela provoque une discrétisation supplémentaire. Tout de même, après application des macros pour les opérateurs de base, le code produit se réduit à un code flottant en précision double. Avec une analyse d'ordres de grandeur et d'erreur plus poussée et un support de preuve par Gappa plus sophistiqué, il devrait être possible de générer des opérations de base à la volée et avec une erreur plus finement adaptée. Les travaux de [55, 92] vont également dans ce sens.

6.3 Automatisation de l'approximation polynomiale

La génération automatique de séquences d'évaluation de polynômes est une brique importante pour l'implantation automatique de fonctions mathématiques. Pourtant, on est toujours restreint aux polynômes. Il convient alors de regarder comment des fonctions mathématiques usuelles quelconques peuvent être approchées par des polynômes. Notre intérêt portera surtout sur des algorithmes automatiques pouvant calculer un polynôme d'approximation de bonne qualité.

Les analyses et résultats décrits aux deux prochaines Sections 6.3.1 et 6.3.2 sont une version étendue de l'article [82] accepté pour publication dans les actes de la conférence RNC'08.

6.3.1 État de l'art et techniques disponibles

Dans cette section, nous analysons les diverses méthodes et surtout les différents algorithmes pour calculer un polynôme d'approximation $p(x) = \sum_{i=0}^{n} p_i \cdot x^i$ pour une fonction donnée $f : \mathbb{R} \to \mathbb{R}$. Comme cette approximation engendre une erreur relative

$$\varepsilon(x) = \frac{p(x) - f(x)}{f(x)}$$

et qu'elle n'est valide que sur un petit intervalle, nous fixons – pareillement aux sections précédentes – une erreur cible $\overline{\varepsilon} \in \mathbb{R}^+$ et un domaine $dom \subset \mathbb{R}$. L'algorithme d'approximation polynomiale recherché devra alors vérifier que

$$\forall x \in dom, |\varepsilon(x)| \leq \overline{\varepsilon}$$

c'est-à-dire que

$$\|\varepsilon\|_\infty^{dom} \leq \overline{\varepsilon}$$

Remarquons que par ces hypothèses d'entrée, nous dévions légèrement du principe d'approximation polynomiale habituel [110, 23, 112, 101] : nous ne fixons pas a priori le degré n du polynôme et cherchons une approximation de la fonction f parmi les polynômes de degré au plus n. L'algorithme doit déterminer ce degré minimal n à partir d'une autre information donnée : l'erreur cible $\overline{\varepsilon}$.

La littérature existante propose plusieurs méthodes, c'est-à-dire algorithmes, pour l'approximation polynomiale [110, 23, 112, 1, 101]. D'entre eux, on cite les polynômes obtenus par troncature des séries de Taylor ou de Tchebychev[2], les polynômes d'interpolation dans les points de Tchebychev, les polynômes minimax ou les polynômes calculés par l'algorithme de Remez[3]. On définira et discutera ces types de polynômes plus loin. Remarquons déjà que l'on fera effectivement une différence entre les polynômes de Remez et les polynômes minimax – dans la littérature, un abus de langage les confond souvent. Dans notre cadre de l'implantation automatique optimisée de fonctions, quatre aspects sont importants dans l'analyse de chacun de ces types de polynômes :

1. La forme d'entrée de la fonction à approcher. Certains algorithmes d'approximation polynomiale ne fonctionnent efficacement que sur des fonctions définies par des expressions consistant de fonctions (élémentaires) de base. Ils manipulent l'expression par des méthodes trouvées dans des systèmes de calcul formel ; par exemple, pour la dériver.

   Pour d'autres algorithmes d'approximation, la fonction peut être définie comme une boîte noire. Dans ce cas, la fonction peut être un code, appelé par l'algorithme d'approximation sur des points où la fonction est à évaluer.

[2]Les translittérations du nom du grand mathématicien Пафнутий Львович Чебышёв varient : Chebychev, Tchebycheff, Tchebychev, etc. Nous choisissons la version française approximative Tchebychev et ignorons ainsi la translittération scientifique Čebyšëv.

[3]Pareil, nous translitérons le nom d'Евгений Яковлевич Ремез par Remez. C'est ce que préconisent les linguistes.

Dans le souci de la généralité de l'implantation automatique, il est certainement préférable d'avoir des algorithmes travaillant sur des boîtes noires. Tout de même, les algorithmes peuvent souvent être optimisés par l'information supplémentaire contenue dans une expression de fonctions de base. Un juste équilibre devra être trouvé.

Remarquons que la plupart des algorithmes d'approximation font également l'hypothèse de quelques propriétés mathématiques des fonctions en entrée, comme la continuité jusqu'à un certain ordre.

2. La complexité de l'algorithme d'approximation, si connue, et, à défaut, l'efficacité d'une implantation pratique de l'algorithme. Si l'implantation automatique est censée devenir un outil de production employé à la volée, seuls de algorithmes de complexité polynomiale peuvent être considérés. En plus, leur implantation pratique doit fournir un résultat dans un temps assez limité, de l'ordre de quelques secondes à minutes sur des machines actuelles.

3. La qualité de l'approximation obtenue. La performance d'une évaluation polynomiale sur machine dépend avant tout du degré du polynôme. Celui-ci dépend de l'erreur maximale $\widehat{\varepsilon} = \|\varepsilon\|_\infty^{dom}$ d'un polynôme p obtenu par approximation à un degré donné : plus grand le degré, moins grande l'erreur. Les divers algorithmes d'approximation optimisent différemment cette erreur $\widehat{\varepsilon}$ et ainsi le degré minimal du polynôme tel que son erreur satisfait l'erreur cible, $\widehat{\varepsilon} \leq \overline{\varepsilon}$.

   Formellement parlant, soit $p^* \in \mathbb{R}_n[x]$ le polynôme de degré n^* minimisant l'erreur maximale $\widehat{\varepsilon}^* = \|\varepsilon^*\|_\infty^{dom}$ avec $\varepsilon^*(x) = \frac{p^*(x)}{f(x)} - 1$. Donc

$$\forall p \in \mathbb{R}[x], \left\| \frac{p}{f} - 1 \right\|_\infty^{dom} \geq \widehat{\varepsilon}^*.$$

   Ce polynôme p^* est communément appelé *polynôme minimax* pour la fonction f, le domaine dom et le degré donné [23, 112, 101].

   Soit p le polynôme renvoyé par un des algorithmes d'approximation ; soit $\widehat{\varepsilon} = \|\varepsilon\|_\infty^{dom}$ le maximum de son erreur $\varepsilon(x) = \frac{p(x)}{f(x)} - 1$. On a donc

$$\widehat{\varepsilon} \geq \widehat{\varepsilon}^* \quad \text{et} \quad \eta = \frac{\widehat{\varepsilon}}{\widehat{\varepsilon}^*} \geq 1.$$

   Ici, η est une mesure de la qualité de l'algorithme d'approximation pour une fonction, un domaine et un degré donné.

   De la même façon, il est possible de définir le polynôme $p^* \in \mathbb{R}[x]$ de degré n^* minimal qui soit un polynôme minimax pour ce degré et dont l'erreur maximale $\widehat{\varepsilon}^*$ satisfasse l'erreur cible, $\widehat{\varepsilon}^* \leq \overline{\varepsilon}$. De la même façon, un algorithme d'approximation calcule alors un polynôme p de degré n minimal (pour tous les polynômes en sortie de cet algorithme) satisfaisant également l'erreur cible, $\widehat{\varepsilon} \leq \overline{\varepsilon}$. Sa qualité en termes de degré est alors

$$\gamma = \frac{n}{n^*} \geq 1.$$

   Les deux quantités mesurant la qualité relative d'une approximation, η et γ, sont reliées. Avec η grandissant, γ croît pour compenser la plus mauvaise approximation à un

degré par un degré plus haut. La relation entre η et γ dépend de la fonction f à approcher dans le domaine dom. Elle est en générale inconnue. Seul Cheney [23] donne des moyens d'estimer n^* et Li [90] analyse η dans un cas spécial.

Alors qu'il est impossible d'atteindre le minimum $\eta = 1$ avec un algorithme d'approximation dans le cas général d'une fonction quelconque – cela impliquerait d'écrire les coefficients potentiellement transcendants du polynôme minimax – il est possible d'atteindre un degré minimal $\gamma = 1$ en pratique. En effet, le degré optimal et le degré du polynôme produit par l'algorithme d'approximation sont des valeurs entières discrètes. Il suffit alors que la qualité du polynôme produit par l'algorithme d'approximation en η soit suffisamment bonne pour que l'algorithme ne doivent pas augmenter le degré pour compenser la non-optimalité de l'approximation. Pour des fonctions usuelles et des domaines de définition petits, ceci est souvent le cas.

4. La conservation de propriétés de symétrie de la fonction f par le polynôme approximateur p. En effet, pour une évaluation du polynôme d'approximation sous forme de Horner avec précalcul des puissances nécessaires, l'optimalité du degré du polynôme n'est pas le seul critère. Quand le polynôme est symétrique, l'évaluation se fait en la puissance x^2 précalculée, donc en un nombre moitié d'étapes. Ce qui importe est alors que pour une fonction symétrique, le polynôme d'approximation soit également symétrique.

   Certains algorithmes d'approximation peuvent conserver la symétrie d'une fonction. D'autres génèrent des polynômes avec, à la place de coefficients nuls reflétant la symétrie, des coefficients très petits. L'évaluation en une puissance précalculée n'est donc pas directement possible. D'autres techniques présentés à la section 6.3.2 sont nécessaire. Elles reposent pourtant sur la possibilité de calculer des approximations polynomiales respectant les symétrie par au moins un algorithme.

Il convient alors de définir et de considérer un par un les types d'approximations polynomiales et leurs algorithmes déjà évoqués. On regarde les quatre aspects donnés.

Approximation minimax

Comme déjà mentionné, le polynôme minimax pour un degré n est le polynôme $p^* \in \mathbb{R}_n[x]$ qui minimise le maximum de l'erreur $\varepsilon^*(x) = \frac{p^* x}{f(x)} - 1$. C'est-à-dire

$$\forall p \in \mathbb{R}_n[x], \left\| \frac{p}{f} - 1 \right\|_\infty^{dom} \geq \|\varepsilon^*\|_\infty^{dom} .$$

Il n'existe pas d'algorithme qui puisse calculer ce polynôme exactement dans le cas général. En effet, comme déjà indiqué, ses coefficients peuvent être transcendants. Il existe l'algorithme de Remez [110, 23, 112] qui peut produire un polynôme aussi proche du polynôme minimax que voulu. Il sera discuté plus loin.

Il est vain de considérer les aspects algorithmiques de la forme d'entrée de la fonction et de complexité pour le calcul impossible du polynôme minimax. Ce polynôme est considéré pour des fins de comparaison. Évidemment sa qualité en η et γ est optimale : on a $\eta = 1$ et $\gamma = 1$.

Pour une fonction symétrique autour de l'origine, le polynôme minimax pour un domaine $dom = [-a; a]$ symétrique conserve la symétrie. Le polynôme est pair pour une fonction paire et impair pour une fonction impaire. Un preuve pour cette propriété peut être

trouvée dans [102, 51]. Elle est basé sur les sens directs et indirects du théorème de Tchebychev [23], qui montre qu'un polynôme est le polynôme minimax de degré n si et seulement si son erreur atteint ses extrema en $n+2$ point et que ses extrema alternent dans leur signe.

Polynômes par série de Taylor tronquée

Il est possible de développer toute fonction (analytique) autour d'un point t par la série de Taylor :

$$f(x) = \sum_{i=0}^{\infty} \frac{f^{(i)}(t)}{i!} (x-t)^i$$

En tronquant cette série à un ordre n, on obtient un polynôme d'approximation dont le reste $p(x) - f(x)$ peut être analytiquement borné de deux façons :

$$\begin{aligned} p(x) &= \sum_{i=0}^{n} \frac{f^{(i)}(t)}{i!} (x-t)^i \\ p(x) - f(x) &= -\int_t^x \frac{f^{(n+1)}(\tau)}{n!} (x-\tau)^n \, d\tau = -\frac{f^{(n+1)}(\xi)}{(n+1)!} (x-t)^{n+1}, \quad t < \xi < x \end{aligned}$$

Pour des intervalles d'approximation $dom = [a, b]$ petits, c'est-à-dire de diamètre $d = b - a$ inférieur à 1, ce terme de reste est petit tant que la dérivée $n+1$-ième de la fonction reste bornée par b :

$$|p(x) - f(x)| \leq \left| f^{(n+1)}(\xi) \right| \cdot d^{n+1} \leq b \cdot d^{n+1}.$$

Ceci est le cas pour les fonctions usuelles. Dans ce cas, la série tronqué de Taylor fournit un polynôme d'approximation de la fonction.

Les séries de Taylor des fonctions mathématiques usuelles comme e^x, $\sin$, $\cos$ etc. sont connues explicitement dans des points de développement t particuliers [1]. De là, donner un algorithme qui calcule, au moins approximativement, les coefficients de la série de Taylor d'une fonction f quelconque autour d'un point t quelconque est plus difficile.

En toute généralité, pour une fonction donnée par une boîte noire, même pouvant fournir un encadrement de la fonction à toute précision, le calcul des coefficients de la série de Taylor est impossible car indécidable [128]. C'est seulement avec un oracle assurant que les rapport des dérivées successives de la fonction restent bornés dans un intervalle I considéré que l'on peut utiliser une technique d'approximation par différence divisée. Si

$$\begin{aligned} \forall \xi, x \in I \quad , \quad \left| \frac{f''(\xi)}{f'(x)} \right| \leq c_1 \\ \forall x \in I \quad , \quad \left| \frac{f(x)}{f'(x)} \right| \leq c_2 \end{aligned}$$

alors la dérivée $f'(x)$ est approchée par

$$\frac{f(x+h) - f(x)}{h} = f'(x) \cdot (1 + \varepsilon)$$

Par la récurrence sur la technique, il est alors possible de calculer les n premières dérivées d'une fonction f boîte noire. On obtient une approximation du polynôme de Taylor.

L'analyse de l'erreur relative montre tout de même que la précision de calcul nécessaire double à chaque étape ce qui rend la technique très chère en termes de complexité et efficacité pratique. On a des erreurs ε_1 et ε_2 sur l'évaluation de f en $x+h$ et x. Ces erreurs sont amplifiées par la cancellation lors de la soustraction.

En fixant une précision $k \in \mathbb{Z}$ de base et en évaluant $f(x+h)$ et $f(x)$ au double de cette précision, on obtient :

$$\begin{aligned} |\varepsilon_1| &\leq 2^{-2k} \\ |\varepsilon_2| &\leq 2^{-2k} \\ h &= 2^{-k} \end{aligned}$$

et alors

$$\frac{f(x+h)\cdot(1+\varepsilon_1) - f(x)\cdot(1+\varepsilon_2)}{h} = f'(x)\cdot(1+\varepsilon)$$

avec

$$\varepsilon = \frac{f(x)}{f'(x)}\cdot\frac{\varepsilon_1-\varepsilon_2}{h} + \frac{f''(\xi)}{f'(x)}\cdot\frac{h}{2}\cdot(1+\varepsilon_1) + \varepsilon_1$$

Ce qui effectivement donne une borne en $\mathcal{O}(2^{-k})$ pour l'erreur sur $f'(x)$:

$$|\varepsilon| \leq c_2\cdot 2^{-2k}\cdot 2^k + c_1\cdot 2^{-k} + 2^{-2k} = 2^{-k}\cdot\left(c_1 + c_2 + 2^{-k}\right)$$

Pour le calcul des polynômes de Taylor, l'approche de la fonction f donnée sous forme de boîte noire est donc à écarter : l'oracle pour les bornes c_1 et c_2 est non-constructif et le calcul est cher en pratique. Remarquons en plus que 2^n évaluations de la fonction f sont nécessaires pour un polynôme de Taylor de degré n.

En abandonnant le calcul numérique pour une approche de calcul formel, il est alors possible de ne considérer que les fonctions définies par des expressions sur une liste de fonctions de base. On peut alors faire l'hypothèse que l'expression de la dérivée de toute fonction de base est connue. Il est alors possible de générer l'expression de la dérivée de toute expression fonction en appliquant simplement les règles de Leibniz. On appellera ce processus différentiation formelle. Par hypothèse toujours, on dispose également d'une représentation de la série de Taylor pour toute fonction de base dans certains points particuliers.

Il y a l'algorithme de calcul d'un polynôme de Taylor de degré n qui consiste à écrire formellement les n expressions des dérivées f, f', …, $f^{(n)}$ et de les évaluer en un point t. Cet algorithme peut produire des résultats (exacts) pour de petits degrés et certaines fonctions de base qui se comportent bien, comme par exemple $f = \sin$. En toute généralité, l'algorithme ne passe pas à l'échelle parce que l'on observe une explosion combinatoire pour la taille de l'expression des dérivées successives. En effet, les règles de Leibniz pour la multiplication et division doublent (au moins) la taille de l'expression :

$$\begin{aligned} (f\cdot g)' &= f'\cdot g + f\cdot g' \\ \left(\frac{f}{g}\right)' &= \frac{f'\cdot g - f\cdot g'}{g^2} \end{aligned}$$

Il est évident que sans simplification formelle et en toute généralité, les tailles des expressions des dérivées f' et g' sont aussi complexes celles des fonctions f et g.

Cet algorithme de calcul formel des dérivées n'est donc efficace ni en théorie ni en pratique. Berz et Hoffstätter proposent donc une autre technique pour calculer des polynômes

de Taylor [7]. Leur approche permet même de borner le reste de la série tronquée $p(x) - f(x)$. L'algorithme sous-jacent peut être adapté et amélioré pour satisfaire les besoins de la preuve formelle [131, 21].

Leur technique est basée sur l'observation que le polynôme de Taylor d'ordre n pour une fonction composée $f = g + h$, $f = g \cdot h$ ou $f = g \circ h$ en un point t peut être calculé à partir de polynômes de Taylor d'ordre n pour g et h en t ou $\widetilde{t} = h(t)$. Notons p le polynôme pour f en t, q celui pour g en t, r celui pour h en t et s celui pour g en $\widetilde{t}$. On obtient pour $f = g + h$

$$p(x) = \sum_{i=0}^{n} \frac{(g+h)^{(i)}(t)}{i!} (x-t)^i = \sum_{i=0}^{n} \frac{g^{(i)}(t) + h^{(i)}(t)}{i!} (x-t)^i = g(x) + h(x).$$

De la même façon, comme le théorème binomial est valide pour la multiplication et la différentiation :

$$\begin{aligned} p(x) &= \sum_{i=0}^{n} \frac{(g \cdot h)^{(i)}(t)}{i!} (x-t)^i \\ &= \sum_{i=0}^{n} \frac{1}{i!} \sum_{k=0}^{i} \binom{i}{k} \left(g^{(k)} \cdot h^{(k-i)}\right)(t) \cdot (x-t)^i \\ &= \sum_{i=0}^{n} \sum_{k=0}^{i} \frac{g^{(k)}(t)}{k!} \cdot \frac{h^{(i-k)}(t)}{(i-k)!} (x-t)^i \\ &= q(x) \cdot r(x) \bmod x^n \end{aligned}$$

Ici, on écrit $\bmod x^n$ pour noter la troncature du polynôme $q(x) \cdot r(x)$ à l'ordre n.

Il s'avère également que le polynôme de Taylor à l'ordre n pour une fonction composée $= g \circ h$ est la composition des polynômes de Taylor pour g et h. Clairement, par double application du théorème de Taylor, on a pour la série complète pour $f = g \circ h$:

$$f(x) = \sum_{i=0}^{\infty} \frac{g^{(i)}(h(t))}{i!} \left(\sum_{k=0}^{\infty} \frac{h^{(k)}(t)}{k!} (x-t)^k - h(t) \right)^i = \sum_{i=0}^{\infty} \frac{g^{(i)}(h(x))}{i!} \left(\sum_{k=1}^{\infty} \frac{h^{(k)}(t)}{k!} (x-t)^k \right)^i$$

Comme la série $\sum_{k=1}^{\infty} \frac{h^{(k)}(t)}{k!} (x-t)^k$ n'a pas de coefficient constant, les n premiers coefficients de la série pour f ne dépendent que des n premiers coefficients de la série pour g en $\widetilde{t} = h(t)$ et de celle pour h en t. On obtient donc

$$p(x) = s(r(x)) \bmod x^n.$$

Cette technique permet alors de calculer un polynôme de Taylor pour toute fonction f composée de fonctions de base récursivement en tout point t. Il est tout de même difficile de l'implanter dans un algorithme efficace en pratique. En effet, l'approche part de l'hypothèse qu'il est possible de calculer un polynôme de Taylor pour toute fonction de base en tout point t. Ceci est en général possible pour toute fonction analytique par morceaux et tout point où la fonction n'a pas de pôle. En pratique, l'algorithme doit passer par un dictionnaire de séries pour les fonctions usuelles en quelques points t' connus. Pour effectuer le décalage de la série de t' en t, l'algorithme doit s'assurer que ce décalage est valide, c'est-à-dire que t est dans le rayon de convergence de la série autour de t'. Cette opération est donc très délicate

pour un algorithme automatique et correct, qui, du coup, doit faire appel à des techniques poussées de calcul formel. Cháves [21, 22] analyse ses problèmes et propose des solutions pour trois fonctions de base : exponentielle, sinus et arc-tangente. Elles sont tout de même fastidieuses à mettre en œuvre.

Intéressons-nous alors à l'aspect de la qualité de l'approximation d'une fonction par un polynôme de Taylor. Comme on a déjà vu, le reste de troncature, donc l'erreur absolue d'un polynôme de Taylor de degré n s'écrit de la forme :

$$p(x) - f(x) = -\frac{f^{(n+1)}(\xi)}{(n+1)!}\,(x-t)^{n+1}$$

Pour les fonctions usuelles comme e^x, $\sin$ etc., la dérivée $n+1$ de f ne varie que très peu dans l'intervalle dom donné[4]. En conséquence, l'erreur se comporte principalement comme la fonction $(x-t)^{n+1}$ dans l'intervalle : l'erreur atteint deux ou trois extrema globaux dans le domaine dom. Le théorème de Tchebychev montre que l'erreur d'une approximation polynomiale de degré n a forcément $n+1$ extrema dans l'intervalle [23]. Le polynôme de Taylor n'est donc clairement pas optimal, c'est-à-dire un polynôme minimax ou proche du polynôme minimax. En effet, le polynôme de Taylor ne dépend que d'un point t de l'intervalle et non de l'intervalle complet. L'approximation est optimale autour de t mais de moindre qualité dans un intervalle dom relativement large.

Il est difficile d'analyser théoriquement la qualité de l'approximation par polynôme de Taylor par rapport aux polynômes minimax. En pratique, on observe une qualité mauvaise pour la finesse de l'erreur maximale, quantifiée par η. Utiliser une approximation de Taylor à la place d'une approximation minimax peut faire perdre des dizaines de bits de précision, surtout pour les petites précisions cibles. Par des effets de discrétisation toujours, la qualité en degré nécessaire pour satisfaire une erreur cible, mesurée par γ, est souvent très bonne, voire optimale tant que le diamètre des domaines dom considérés restent plus petit que 1. Dans la pratique du cadre de l'implantation de fonctions mathématiques, cette condition est toujours remplie.

Les deux tableaux suivants illustrent le comportement des paramètres de qualité η et γ pour les fonctions $\exp$ et $\cos$ et des combinaisons de diamètres de domaines dom, centrés en 0, et degré n du polynôme de Taylor. Le comportement pour d'autres fonction usuelles, comme $\log(1+x)$, $\arccos$ etc. est comparable.

Pour des combinaisons intéressantes, le tableau indique l'erreur maximale $\widehat{\varepsilon}$ provoquée par le polynôme de Taylor, le degré $\widetilde{n}$ nécessaire pour une approximation minimax d'erreur au plus aussi grande que $\widehat{\varepsilon}$ et les valeurs η et γ. Le polynôme minimax n'étant pas calculable, il est remplacé par un polynôme de Remez dont la qualité peut être majorée (cf. section 6.3.1 plus loin pour plus de détails).

[4] On observe évidemment aussi des variations linéaires en $x-t$ pour des dérivées $n+1$ impaires autour de t. Il suffit de factoriser $x-t$ pour que le raisonnement tienne.

Fonction exponentielle

$\log_2$ du diamètre	degré n	$\log_2 \widehat{\varepsilon}$ (Taylor)	degré $\widetilde{n}$ minimax	qualité η – bits perdus	qualité γ
-6	11	-112.8	11	2048 – 11	1
-6	10	-102.3	10	1024 – 10	1
-5	10	-91.25	10	1024 – 10	1
-6	9	-91.75	9	512 – 9	1
-5	9	-81.75	9	512 – 9	1
-4	9	-71.75	9	514 – 9	1
-1	9	-21.63	7	1.52 – 1	1.28
-1	4	-6.656	3	1.80 – 1	1.33

Fonction cosinus

$\log_2$ du diamètre	degré n	$\log_2 \widehat{\varepsilon}$ (Taylor)	degré $\widetilde{n}$ minimax	qualité η – bits perdus	qualité γ
-6	15	-156	14	32768 – 15	1.07
-6	14	-156	14	32768 – 15	1
-6	13	-134.5	12	8192 – 13	1.08
0	13	-50.38	10	2.83 – 1	1.30
-6	12	-134.5	12	8192 – 13	1
-1	11	-52.88	10	2048 – 11	1.10
-1	9	-21.81	6	1.45 – 1	1.50

Quant au quatrième aspect à considérer pour un algorithme d'approximation polynomiale, la conservation de symétries, il y a peu à dire. Il est clair que la série de Taylor n'a que de coefficients pairs pour une fonction paire autour du point de développement t et que de coefficients impairs pour une fonction impaire. Le polynôme de Taylor suit donc les symétries de la fonction. Ceci n'est plus vrai si seule une approximation au polynôme de Taylor peut être calculée : les coefficients nuls deviennent alors des valeurs très petites mais non-nulles. En pratique, on observe cet inconvénient pour les algorithmes de calcul de séries de Taylor présentés.

On conclusion, les polynômes d'approximation obtenus en tronquant la série de Taylor ne sont de bonne qualité autour du point de développement t. Leur qualité dans un domaine dom est donc sous-optimale et en pratique souvent mauvaise : plusieurs dizaines de bits de précision sont perdus. On cherche donc des polynômes d'approximation qui minimisent l'erreur sur tout un intervalle. Les polynômes obtenus par série de Tchebychev donnent une première solution.

Polynômes par série de Tchebychev tronquée

Une série de Taylor fournit un polynôme dans la base canonique des monômes x^i. Il est également possible de développer une fonction f, définie sur $dom = [-1; 1]$, dans la base des

polynômes de Tchebychev $T_i(x)$, dont la définition suivra. On obtient par ce développement la série de Tchebychev :

$$f(x) = \sum_{i=0}^{\infty} a_i T_i(x)$$

En tronquant cette série à l'ordre n, on obtient un polynôme approximateur p de la fonction f dans le domaine $dom = [-1; 1]$:

$$p(x) = \sum_{i=0}^{n} a_i T_i(x)$$

On appellera p *le polynôme de série de Tchebychev tronquée*. Il ne doit pas être confondu avec les polynômes de Tchebychev ni avec le polynôme de meilleure approximation minimax, même si celle-ci est souvent appelé approximation au sens de Tchebychev [112, 101].

Les polynômes de Tchebychev sont définis par une récurrence [123, 23] :

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_i(x) &= 2 \cdot x \cdot T_{i-1}(x) - T_{i-2}(x) \end{aligned}$$

Il est facile de montrer que $T_i(\cos x) = \cos(i \cdot x)$. Les $T_i(x)$ forment une base orthogonale pour le produit scalaire suivant [23, 101] :

$$\langle f, g \rangle = \int_{-1}^{1} \frac{f(x)g(x)}{\sqrt{1-x^2}} dx$$

Il est donc possible de déterminer les coefficients a_i de la série de Tchebychev et donc d'un polynôme de série de Tchebychev tronquée par intégration [101] :

$$\begin{aligned} f(x) &= \sum_{i=0}^{\infty} a_i T_i(x) \\ a_i &= \begin{cases} \frac{1}{\pi} \langle f, T_i \rangle & \text{si } i = 0 \\ \frac{2}{\pi} \langle f, T_i \rangle & \text{sinon} \end{cases} \end{aligned}$$

Étant donné un polynôme de série de Tchebychev tronquée $p(x) = \sum_{i=0}^{n} a_i \cdot T_i(x)$, il est évidemment possible de faire un changement de base pour obtenir un polynôme dans la base des monômes x^i :

$$p(x) = \sum_{i=0}^{n} a_i \cdot T_i(x) = \sum_{i=0}^{n} p_i x^i$$

Ici, il est important de remarquer que le i-ième polynôme de Tchebychev est de degré i et que la troncature de la série de Tchebychev à l'ordre n fournit donc un polynôme d'approximation de degré n. Cette propriété se démontre facilement par induction sur la définition des $T_i(x)$. De la même façon, il est facile de faire un changement de variable affine sur une fonction f donnée sur un domaine $dom = [a; b]$ pour satisfaire la contrainte que l'approximation par série de Tchebychev se fait toujours sur le domaine $[-1; 1]$.

L'approximation par polynôme de série de Tchebychev tronquée à l'ordre n se fait donc par le calcul de $n+1$ intégrales et un changement de base, dont le coût est négligeable. Avec le changement de variables $x = \cos t$ pour éviter d'intégrer sur des pôles en -1 et 1, le calcul des intégrales

$$\langle f, T_i \rangle = \int_{-1}^{1} \frac{f(x) \cdot T_i(x)}{\sqrt{1-x^2}} dx = \int_{0}^{\pi} f(\cos t) \cdot \cos(i \cdot t) dt$$

peut se fait numériquement par une des différentes techniques de quadrature. Ces techniques peuvent assez facilement accommoder des fonctions données par boîte noire sous condition de disposer de quelques hypothèses sur la décroissance des dérivées successives des fonctions intégrandes [49]. Dans le cadre donné des fonctions usuelles, ces hypothèses sont souvent remplies. Le calcul de polynômes de série de Tchebychev tronquée est alors facilement automatisable.

Le coût de l'intégration numérique se mesure principalement par le nombre d'évaluation de la fonction intégrande, donc de f. Pour un polynôme de degré n, $n+1$ intégrales sont à calculer. Dans le cadre donné, où les intégrales doivent être déterminées à une centaine de bits de précision, chacune des intégrales nécessite une centaine d'évaluations de f [49]. Une analyse de la complexité plus précise reste un travail à faire.

La qualité des approximations par polynômes de série de Tchebychev tronquée est très bonne. Une borne pour la qualité $\eta = \frac{\widehat{\varepsilon}}{\widehat{\varepsilon}^*}$, rapport entre l'erreur maximale $\widehat{\varepsilon}$ de l'approximation trouvée et l'erreur maximale $\widehat{\varepsilon}^*$ du polynôme de meilleure approximation minimax, est connue :

$$1 \leq \eta \leq 4 + \ln n$$

où n est le degré du polynôme [23]. La perte v de précision en bits, engendrée par l'utilisation d'un polynôme approximateur imparfait à la place du minimax, se mesure par le logarithme de η, $v = \log_2 \eta$. Dans notre cadre, le degré des polynômes reste quasiment toujours en dessous de 55, valeur pour laquelle la perte en bits v dépasse la valeur 3. Le tableau suivant illustre même par des exemples pratiques que la valeur de η reste bornée par 1.1 :

Fonction	domaine	degré	η
$\exp(x)$	$\left[-2^{-14}; 2^{-14}\right]$	8	1.000
$\exp(x)$	$[-1/4; 1/4]$	6	1.0139
$\exp(x)$	$[-1; 1]$	5	1.070
$\frac{\ln(1+x)}{x}$	$\left[-2^{-8}; 2^{-8}\right]$	12	1.002
$\exp(\cos(x^2)+1)$	$\left[-2^{-5}; 2^{-5}\right]$	9	1.076
$\mathrm{asin}(x)$	$[-1/4; 1/4]$	9	1.003
$\mathrm{acos}(x)$	$[1/4; 1/2]$	9	1.085
$\cos(x)$	$[-\pi/64; \pi/64]$	5	1.000

Puisque la qualité en termes de sous-optimalité de l'erreur maximale η est très bonne, les effets de discrétisation font en pratique que la qualité γ en termes de degré nécessaire est quasiment toujours $\gamma = 1$.

Les polynômes de série de Tchebychev tronquée ont les mêmes propriétés de symétrie que la fonction f. Il est facile de voir que les polynômes de Tchebychev $T_i(x)$ de degré i

impair sont des fonctions impaires et que les $T_i(x)$ pour i pair sont paires. En effet, il suffit de considérer leur définition récursive. Pour une fonction f paire, les fonctions intégrandes $\frac{f(x)\cdot T_{2k+1}(x)}{\sqrt{1-x^2}}$ sont toutes impaires. L'intégrale étant symétrique, les coefficients a_i impaires sont donc nulles, ainsi que le sont les coefficients p_i impaires. Tout de même, la propriété n'est souvent plus vérifiée si une intégration numérique est utilisée. L'integrande n'étant pas identiquement nulle, l'algorithme de quadrature sera sujet à de fortes cancellations qui ne produiront pas un vrai zéro.

Polynômes interpolateurs dans les points de Tchebychev

Comme on a vu, les polynômes de série de Tchebychev tronquée fournissent de bons approximants polynomiaux. Leur calcul nécessite pourtant une intégration numérique coûteuse en évaluations de fonction. Pour éviter cette intégration numérique tout en gardant certains avantages des polynômes de série de Tchebychev, un autre type de polynôme peut être défini. L'algorithme sous-jacent est un pur algorithme d'interpolation en $n+1$ points. Il s'avère qu'il peut être vu également comme un algorithme de quadrature [12].

D'après le théorème de Tchebychev [23], un polynôme est le minimax pour un degré n donné si son erreur atteint $n+2$ extrema globaux dans l'intervalle dom donné et que les signes de ces extrema alternent. L'erreur d'un tel polynôme a donc $n+1$ zéros dans l'intervalle. En plus, si les coefficients de la série de Tchebychev décroissent rapidement, l'erreur est approximativement égale au premier terme du reste de la série tronquée [93, 101] :

$$f(x) = \sum_{i=0}^{\infty} a_i \cdot T_i(x) = \underbrace{\sum_{i=0}^{n} a_i \cdot T_i}_{p(x)} \quad + \quad a_{n+1} \cdot T_{n+1}(x) + \underbrace{\sum_{i=n+2}^{\infty} a_i T_i(x)}_{\text{négligé}}$$

Il est facile de montrer que le $n+1$-ième polynôme de Tchebychev

$$T_{n+1}(x) = \cos\left((n+1) \cdot \operatorname{acos} x\right)$$

admet $n+1$ zéros $x_j = \cos\left(\frac{2j+1}{n+1} \cdot \frac{\pi}{2}\right)$ dans l'intervalle $[-1;1]$ [93].

Pour les fonctions usuelles, dont les coefficients de Tchebychev décroissent rapidement, il convient alors de déterminer un polynôme d'approximation $p(x)$ de degré n dont l'erreur s'annule dans les $n+1$ zéros du polynôme $T_{n+1}(x)$. Cela veut dire que $p(x)$ interpole $f(x)$ en ces points. L'erreur de ce polynôme oscillera alors avec $n+2$ extrema globaux de signes alternés. Il satisfera alors *presque* les conditions du théorème de Tchebychev. Donc il est supposé être proche du minimax.

On nommera points de Tchebychev les zéros x_i, $i = 0, \ldots, n$, du $n+1$-ième polynôme de Tchebychev $T_{n+1}(x)$. Et finalement, on appellera polynôme interpolateur dans le points de Tchebychev le polynôme

$$p(x) = \sum_{i=0}^{n} c_i \cdot T_i(x) = \sum_{i=0}^{n} p_i \cdot x^i$$

qui interpole f dans ces points, c'est-à-dire le polynôme tel que $p(x_i) = f(x_i)$.

L'algorithme de détermination de ce polynôme interpolateur pourrait être un algorithme classique d'interpolation. Il peut pourtant toujours être simplifié en utilisant la propriété

d'orthogonalité des polynômes de Tchebychev $T_i(x)$ [93]. Aux points d'interpolation $x_j = \cos\left(\frac{2j+1}{n+1} \cdot \frac{\pi}{2}\right)$, f et p coïncident, donc aussi la somme de leurs produits avec les $T_k(x_j)$:

$$\begin{aligned} f(x_j) &= \sum_{i=0}^{n} c_i T_i(x) \\ \sum_{j=0}^{n} T_k(x_j) \cdot f(x_j) &= \sum_{j=0}^{n} \sum_{i=0}^{n} c_i \cdot T_k(x_j) \cdot T_i(x_j) \end{aligned}$$

Il est facile de montrer [93] que la somme sur les $T_i(x_j) \cdot T_k(x_j)$ dans les zéros x_j de $T_{n+1}(x)$ satisfait :

$$\sum_{j=0}^{n} T_i(x_j) \cdot T_k(x_j) = \begin{cases} n+1 & \text{si } i = k = 0 \\ 1/2 \cdot (n+1) & \text{si } i = k > 0 \\ 0 & \text{si } i \neq k \end{cases}$$

On obtient alors la formule suivante qui se transforme toute de suite en algorithme :

$$c_i = \begin{cases} \frac{1}{n+1} \cdot \sum_{j=0}^{n} f(x_j) & \text{si } i = 0 \\ \frac{2}{n+1} \cdot \sum_{j=0}^{n} T_i(x_j) \cdot f(x_j) & \text{sinon} \end{cases}$$

L'algorithme calcule donc ces coefficients c_k du polynôme interpolateur dans la base de Tchebychev et effectue ensuite un simple changement de base [93].

Le coût de l'algorithme en termes d'évaluations de la fonction f est très faible : juste $n+1$ valeurs $f(x_j)$ sont nécessaires. Dans le coût complet, s'y rajoutent le calcul des $n+1$ points d'interpolation x_j et l'évaluation des $n \cdot (n+1)$ valeurs $T_i(x_j)$ dans les sommes. Dans les deux cas, il s'agit d'évaluations de la fonction cosinus. Cet algorithme de calcul de polynômes d'approximation de fonctions est donc le moins cher de tous les algorithmes présentés ici. Comme il s'agit d'un algorithme d'interpolation, qui nécessite juste $n+1$ évaluations de la fonction à approcher, cette fonction peut être donnée par une boîte noire sans hypothèses supplémentaires.

Pour les fonctions f considérées dans le cadre donné, la qualité et en termes d'erreur maximale relative à l'erreur du minimax, η, et en termes d'optimalité du degré du polynôme pour satisfaire une erreur cible, γ, est très bonne.

Des bornes explicites sont connues pour $\eta = \frac{\hat{\varepsilon}}{\hat{\varepsilon}^*}$. Sans faire d'hypothèse particulière sur f, Powell [107] donne la borne suivante pour des polynômes de degré n :

$$1 \leq \eta \leq 1 + \frac{1}{n+1} \sum_{i=0}^{n} \tan\left(\frac{(i+1/2) \cdot \pi}{2 \cdot (n+1)}\right)$$

Cette borne ne croît que très lentement avec le degré n des polynômes. Dans le cadre donné, où le degré des polynômes est borné par au maximum 80, on observe donc une qualité η bornée par

$$1 \leq \eta \leq 4.761 \quad \text{pour } 2 \leq n \leq 80$$

Cela implique que l'utilisation du polynôme d'interpolation dans les points de Tchebychev à la place du polynôme optimal minimax comme approximateur n'entraîne qu'une perte de précision au plus $\log_2(\eta) \leq 3$ bits.

Pour des fonctions élémentaires usuelles, comme $\exp$, $\sin$, $\cos$ etc. ces bornes sont même encore trop pessimistes. Li démontre que pour cette classe de fonctions, *se comportant gentiment*, la perte de précision en bit est encore moindre et de l'ordre d'un bit de précision [90].

Cette bonne qualité en termes d'erreur maximale induit l'optimalité ou quasi-optimalité du degré du polynôme approximateur pour satisfaire une erreur cible en pratique si des polynômes d'interpolation dans les points de Tchebychev sont utilisés. Même si on a $\eta \leq 1.1$ en pratique, on a quasiment toujours $\gamma = 1$, comme l'illustre le tableau suivant :

Fonction	Domaine	degré n	précision $\log_2(\widehat{\varepsilon})$	degré $\widetilde{n}$ du minimax de même erreur	qualité γ
$\exp(x)$	$[-1; 1]$	7	22.13	7	1
$\exp(x)$	$[-1/4; 1/4]$	7	38.25	7	1
$\exp(x)$	$\left[-2^{-14}; 2^{-14}\right]$	8	152.5	8	1
$\cos(x)$	$[-\pi/64; \pi/64]$	8	73.25	8	1
$\cos(\sqrt{x})$	$[0; \pi/64]$	6	79.75	6	1
$\mathrm{acos}(x)$	$[1/4; 1/2]$	9	38.50	9	1
$\mathrm{asin}(x)$	$[-1/8; 1/8]$	9	47.38	9	1
$\exp(\cos(x^2)+1)$	$\left[-2^{-5}; 2^{-5}\right]$	9	68.75	8	1.125
$\frac{\log(1+x)}{x}$	$\left[-2^{-8}; 2^{-8}\right]$	12	119.8	12	1

Il reste à remarquer que le polynôme p interpolateur dans les points de Tchebychev suit les propriétés de symétrie de la fonction f. Cela veut dire que p est pair si f est paire et impair si f est impaire [90]. Tout de même, le polynôme d'approximation p est calculé en pratique par l'algorithme numérique esquissé ci-dessus. En conséquence, les coefficients nuls du polynôme exact pair ou impair sont remplacés par des valeurs très petites mais non-nulles.

Comme on a vu, l'algorithme de polynômes approximateurs interpolateurs dans les points de Tchebychev est simple, efficace et donne des approximations polynomiales de bonne qualité en pratique. L'algorithme peut travailler sur des fonctions définies par des boîtes noires, ce qui facilite l'automatisation du processus de génération de bon approximants polynomiaux. Une amélioration des résultats est pourtant toujours possible :

- Les algorithmes présentés ci-dessus minimisent, plus ou moins bien, l'erreur absolue $p(x)-f(x)$ du polynôme p par rapport à la fonction f. Dans le cadre donné de la virgule flottante, une optimisation de l'erreur relative $\varepsilon(x) = \frac{p(x)-f(x)}{f(x)} = \frac{p(x)}{f(x)} - 1$ semble plus appropriée.
  Il y a des heuristiques pratiques pour le contourner, qui consistent à diviser $f(x)$ par son premier monôme non nul dans son développement de Taylor [93, 26]. Cette approche mène à une fonction $g(x) = \frac{f(x)}{x^k}$ qui ne varie que peu dans l'intervalle dom donné. En conséquence $\varepsilon(x) = \frac{x^k \cdot q(x)}{f(x)} - 1 = \frac{q(x)-g(x)}{g(x)} \approx c \cdot (q(x) - g(x))$. La technique reste une heuristique et il convient de la remplacer par un processus plus exact.
- En pratique, aucun des algorithmes d'approximation présentés ci-dessus ne satisfait le but de conserver les symétries de la fonction f à approcher. Toujours des valeurs nulles sont remplacées par de petites valeurs par le processus numérique.
  Une technique pour réussir à fournir des approximants symétriques pour des fonctions symétriques consiste à détecter ces zéros devenus petites valeurs et à recalculer

ensuite une approximation polynomiale symétrique par construction. Ce calcul demande pourtant de spécifier le calcul d'un polynôme dans une base de monômes symétrique.

L'algorithme de Remez permet de répondre à ces deux besoins.

Polynômes de Remez

L'algorithme de Remez [110, 23, 112, 101] est un procédé convergent qui calcule un polynôme approximateur d'une fonction proche du polynôme optimal minimax. On appellera polynôme de Remez un tel polynôme itérativement calculé.

Une bibliographie importante sur cet algorithme est disponible [110, 126, 119, 51, 23, 53, 112, 79, 101]. Il dépasserait alors le cadre de cette thèse que donner une explication complète de cet algorithme ; on se restreindra donc aux aspects relatifs aux problèmes posés dans cette thèse. Indiquons toujours qu'une bonne introduction au fonctionnement de l'algorithme est proposée par Muller [101].

Définitions et convergence Pour une fonction $f : \mathbb{R} \to \mathbb{R}$, un domaine $dom \subset \mathbb{R}$ et un degré n donnés, l'algorithme de Remez calcule une suite $\left(p^{[j]}\right)_j$ de polynômes

$$p^{[j]} \in \mathbb{R}_n[x].$$

Sous certaines hypothèses détaillées plus loin, les polynômes de cette suite tendent vers le polynôme minimax p^* pour f, dom et n. On associe à cette suite de polynômes une suite $\left(\varepsilon^{[j]}\right)_j$ de fonctions d'erreur[5]

$$\varepsilon^{[j]}(x) = \frac{p^{[j]}(x)}{f(x)} - 1$$

avec leurs erreurs maximales

$$\widehat{\varepsilon}^{[j]} = \left\|\varepsilon^{[j]}\right\|_\infty^{dom}.$$

Il convient également de considérer la suite des qualités relatives des $\widehat{\varepsilon}^{[j]}$ par rapport à l'erreur maximale $\widehat{\varepsilon}^* = \|\varepsilon^*\|_\infty^{dom}$ du polynôme minimax p^*. On fixe

$$\eta^{[j]} = \frac{\widehat{\varepsilon}^{[j]}}{\widehat{\varepsilon}^*}.$$

Traditionnellement, l'algorithme de Remez est arrêté à la j-ième étape si la qualité $\eta^{[j]}$ dépasse à un paramètre $\overline{\eta}$, $\eta^{[j]} \leq \overline{\eta}$. Ce paramètre est pris en argument de l'algorithme.

Par la suite, on appellera polynôme de Remez $p = p^{[j]}$ le polynôme de degré n approchant f dans le domaine dom à une qualité d'au moins $\overline{\eta}$.

Évidemment, le polynôme minimax p^* et donc son erreur maximale $\widehat{\varepsilon}^*$ sont inconnus au cours de l'algorithme – l'algorithme est fait pour l'approcher en y convergent. Les valeurs de la suite $\left(\eta^{[j]}\right)_j = \left(\frac{\widehat{\varepsilon}^{[j]}}{\widehat{\varepsilon}^*}\right)_j$ ne peuvent pas être calculées explicitement. L'algorithme ne peut donc déterminer son critère d'arrêt $\eta^{[j]} \leq \overline{\eta}$ que par une majoration de $\eta^{[j]}$. Une telle majoration est fournie par un théorème de La Vallée Poussin [23]. Le critère est basé sur une

[5] On considère tout de suite des fonctions d'erreurs relatives. Le cas des erreurs absolues est analogue.

comparaison des valeurs des extrema de la fonction d'erreur $\varepsilon^{[j]}$. Son calcul est donc aussi cher que le calcul d'une norme infini sur la fonction d'erreur.

En admettant quelques hypothèses, explicitées plus loin et souvent remplies en pratique, Veidinger [126] montre que la convergence de l'algorithme est quadratique. Sous des hypothèses moins fortes, cette fois-ci toujours remplies en pratique, une variante de l'algorithme garde une convergence linéaire [112].

Cœur de l'algorithme de Remez Le théorème de Tchebychev [23] relie l'optimalité d'un polynôme minimax de degré n au fait que l'erreur $\varepsilon(x) = \frac{p(x)}{f(x)}$ du polynôme $p(x)$ par rapport à la fonction $f(x)$ admet $n+2$ extrema globaux $\varepsilon(x_k)$ dans l'intervalle dom, qu'ils sont tous de même hauteur $\widetilde{\varepsilon} = |\varepsilon(x_k)|$ et que leurs signes alternent, $\widetilde{\varepsilon} = (-1)^k \cdot \varepsilon(x_k)$.

L'algorithme de Remez est basé sur ce théorème. Il essaie de trouver un polynôme interpolateur dont l'erreur satisfait le théorème. Pour cela, l'algorithme calcule à chaque étape un choix de points d'interpolation, interpole la fonction en ces points et obtient un polynôme dont un nouveau choix de points peut être déduit.

À chaque étape[6], l'algorithme calcule donc à partir de $n+1$ points x_k une valeur $\widetilde{\varepsilon}$ tels que la fonction d'erreur

$$\varepsilon(x) = \frac{\sum_{i=0}^{n} p_i x^i}{f(x)} - 1$$

prenne la valeur $(-1)^k \cdot \widetilde{\varepsilon}$ en ces points x_k. Il s'agit donc du problème d'interpolation suivant :

$$\varepsilon\left(x_k\right) = \frac{\sum_{i=0}^{n} p_i x_k^i}{f\left(x_k\right)} - 1 = (-1)^k \cdot \widetilde{\varepsilon}$$

qui s'écrit également :

$$\sum_{i=0}^{n} p_i \cdot \frac{x_k^i}{f(x_k)} - (-1)^k \cdot \widetilde{\varepsilon} = 1.$$

Ceci donne en notation matricielle [112, 101] :

$$\begin{pmatrix} \frac{x_0^0}{f(x_0)} & \frac{x_0^1}{f(x_0)} & \cdots & \frac{x_0^n}{f(x_0)} & (-1)^0 \\ \frac{x_1^0}{f(x_1)} & \frac{x_1^1}{f(x_1)} & \cdots & \frac{x_1^n}{f(x_1)} & (-1)^1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{x_{n+1}^0}{f(x_{n+1})} & \frac{x_{n+1}^1}{f(x_{n+1})} & \cdots & \frac{x_{n+1}^n}{f(x_{n+1})} & (-1)^{n+1} \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \\ -\widetilde{\varepsilon} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

Si ce système n'est pas singulier, il existe donc une solution qui est unique car la matrice est carrée. La non-singularité du système est conditionnée par un propriété appelée condition de Haar, discutée plus loin [23, 112].

L'algorithme résout donc le système pour le choix de points $x_k^{[j]}$ de l'étape j et obtient le polynôme $p^{[j]}$ dont l'erreur est $\varepsilon^{[j]}(x)$. Comme il s'agit d'une simple interpolation en $n+2$

[6] On devrait donc indexer les valeurs encore par l'étape j, c'est-à-dire écrire des horreurs comme $\widetilde{\varepsilon}^{[j]}$ ou $\sum_{i=0}^{n} p_i^{[j]} {x_k^{[j]}}^i$. On se permet le petit abus de langage de laisser ces indices sur j.

points, les vrais extrema de $\varepsilon^{[j]}(x)$ vont être proches des valeurs voulues $(-1)^k \cdot \widetilde{\varepsilon}$ mais ne vont pas satisfaire cette contrainte exactement. L'algorithme de Remez calcule alors les $n+2$ points x'_k où la fonction d'erreur $\varepsilon^{[j]}(x)$ atteint ses vrais extrema. Ces valeurs x'_k sont ensuite pris comme les nouveaux points d'interpolation $x_k^{[j+1]}$.

Le processus est amorcé sur les points x_k où le polynôme de Tchebychev $T_{n+1}(x)$ atteint ses extrema. L'algorithme de Remez est donc un raffinement itératif des polynômes interpolateurs dans les points de Tchebychev [101].

Qualité des polynômes de Remez L'algorithme de Remez prend un paramètre $\overline{\eta}$ en entrée. La qualité η d'un polynôme de Remez peut donc être aussi bonne que l'on veut. En conséquence, il est clair que la qualité γ en degré nécessaire pour satisfaire une erreur cible $\overline{\varepsilon}$ donnée peut, en pratique, toujours être rendue optimale : $\gamma = 1$.

Coût pratique de l'algorithme de Remez Pour analyser le coût de l'algorithme de Remez, il convient d'abord de s'intéresser au nombre J d'itérations nécessaires pour atteindre une qualité $\overline{\eta} = 1 + 2^{-\tau}$. En pratique, l'amorçage de l'algorithme se fait sur les points de Tchebychev. Comme on a vu à la section précédente, le premier polynôme $p^{[1]}$, un polynôme interpolateur dans les points de Tchebychev, fournit donc déjà une qualité $\eta^{[1]} = 1 + 2^{\mathcal{O}(1)}$. La convergence de l'algorithme étant quadratique, le nombre d'itérations est $J = \mathcal{O}(\log \tau)$.

Le coût d'une itération est donné par le coût des trois tâches de chaque étape. L'algorithme calcule d'abord la matrice d'interpolation. Pour cela, principalement $n+2$ évaluations de la fonction f en les points x_k sont nécessaires. Le coût du calcul des puissances x_k^i ainsi que des divisions $\frac{x_k^i}{f(x_k)}$ est négligeable devant les évaluations de fonction. L'algorithme résout ensuite le système linéaire par un algorithme dont le coût est borné par $\mathcal{O}(n^3)$ multiplications et divisions. En pratique, même ce coût reste négligeable devant les évaluations de fonction. Finalement, l'algorithme cherche les extrema de la fonction d'erreur en approchant les zéros de sa dérivée. Par le théorème de Tchebychev, en pratique, l'algorithme peut s'attendre à trouver n zéros de la dérivée de l'erreur. Avec un découpage de l'intervalle en $c \cdot n$, $c \in \mathbb{N}$ petit (5), sous-intervalles, il est possible d'encadrer ces zéros suffisamment bien pour qu'une itération de Newton trouve ensuite assez précisément ces n zéros. Ces itération de Newton convergent quadratiquement et demandent $\mathcal{O}(\log \tau)$ évaluation.

En somme toute, la complexité de l'algorithme de Remez est de $\mathcal{O}(n \cdot \log^2(\log(\overline{\eta} - 1)))$.

En pratique, l'algorithme de Remez peut être implanté pour une performance très élevée. Pour le calcul de polynômes dans le cadre donné, avec des degrés inférieurs à 50 et une qualité de $\overline{\eta} = 1 + 2^{-5}$, l'implantation de Remez dans l'outil Sollya peut fournir des polynômes en quelques secondes.

Conservation de symétries Comme on a déjà vu, le polynôme minimax p^* approchant une fonction f paire ou impaire est pair ou impair sur un intervalle symétrique $[-a; a]$. La série de polynômes $p^{[j]}$ calculés par l'algorithme de Remez converge vers ce polynôme minimax p^*. Par un argument de continuité, on se convainc donc qu'un polynôme de Remez p est quasiment pair ou impair pour une fonction paire ou impaire. Ici, on appelle quasiment pair un polynôme dont les coefficients impairs sont nuls ou très petites par rapport aux valeurs $p(x)$ du polynôme pour x dans le domaine.

Cette propriété que les coefficients impairs d'un polynôme de Remez p de qualité $\overline{\eta}$ pour une fonction f paire dans l'intervalle $dom = [-1;1]$ restent petits peut être montrée d'une façon que nous esquissons dans la suite. La preuve a une certaine importance car, à la section 6.3.2, nous allons baser un algorithme sur elle.

Soit n un degré pair, soit p^* le polynôme minimax pour f, n et dom. Soit

$$\varepsilon^*(x) = \frac{p^*(x)}{f(x)} - 1 \quad \text{et} \quad \varepsilon(x) = \frac{p(x)}{f(x)} - 1$$

les erreurs associées. On sait que le polynôme minimax p est pair. Il s'ensuit que la fonction d'erreur ε^* est paire. Par le théorème de Tchebychev, on sait qu'il existe au moins $n+2$ points où l'erreur ε^* atteint un maximum global $\widehat{\varepsilon}^* = \|\varepsilon^*\|_\infty^{dom}$. Comme la fonction d'erreur ε^* est paire et $dom = [-1;1]$ est symétrique, il est possible d'en choisir $n/2$ paires symétriques $(x_i; -x_i)$, $i \in [0, n/2 - 1] \cap \mathbb{N}$. On choisit un extremum ξ supplémentaire pour avoir $2 \cdot n/2 + 1 = n + 1$ points d'interpolation.

Notons $\overrightarrow{p}^*$ le vecteur des coefficients de p^* et $\overrightarrow{p}$ le vecteur des coefficients de p. Notons $\overrightarrow{\delta} = \overrightarrow{p} - \overrightarrow{p}^*$ le vecteur des erreurs absolues sur les coefficients du polynôme de Remez par rapport au polynôme minimax. Posons ensuite la matrice A d'interpolation suivante :

$$A = \begin{pmatrix} \frac{x_0^0}{f(x_0)} & \frac{x_0^1}{f(x_0)} & \cdots & \frac{x_0^n}{f(x_0)} \\ \frac{(-x_0)^0}{f(-x_0)} & \frac{(-x_0)^1}{f(-x_0)} & \cdots & \frac{(-x_0)^n}{f(-x_0)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{x_{n/2/-1}^0}{f(x_{n/2-1})} & \frac{x_{n/2-1}^1}{f(x_{n/2-1})} & \cdots & \frac{x_{n/2-1}^n}{f(x_{n/2-1})} \\ \frac{\left(-x_{n/2-1}\right)^0}{f(-x_{n/2-1})} & \frac{\left(-x_{n/2-1}\right)^1}{f(-x_{n/2-1})} & \cdots & \frac{\left(-x_{n/2-1}\right)^n}{f(-x_{n/2-1})} \\ \frac{\xi^0}{f(\xi)} & \frac{\xi^1}{f(x_0)} & \cdots & \frac{\xi^n}{f(\xi)} \end{pmatrix}$$

Alors on obtient par évaluation de p^* en ces points d'extrema :

$$A\overrightarrow{p}^* = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} + \widehat{\varepsilon}^* \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \\ \vdots \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} + \widehat{\varepsilon}^* \overrightarrow{s}.$$

Il est facile de se convaincre que pour un polynôme de Remez de qualité suffisamment bonne, le signe de l'erreur ε du polynôme de Remez au points des extrema du minimax est égal au signe de l'erreur ε^* du minimax – la preuve est plus compliquée. On obtient donc pour les évaluations de p aux points d'extrema du minimax choisis :

$$A\overrightarrow{p} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} + \widehat{\varepsilon}^* \begin{pmatrix} \eta_0 & & \\ & \ddots & \\ & & \eta_n \end{pmatrix} \overrightarrow{s}$$

où les valeurs $\eta_i \leq \eta$ indiquent la qualité du polynôme de Remez au point considéré. En prenant $A\overrightarrow{p} = A\overrightarrow{p}^* + A\overrightarrow{\delta}$ et $\beta_i = \eta_i - 1$, on déduit donc :

$$A\overrightarrow{\delta} = \widehat{\varepsilon}^* \begin{pmatrix} \beta_0 & & \\ & \ddots & \\ & & \beta_n \end{pmatrix} \overrightarrow{s}$$

Soustrayons alors les lignes impaires de l'équation des lignes paires. La fonction f étant paire, on obtient :

$$\widetilde{A} = \begin{pmatrix} \frac{x_0^0}{f(x_0)} & \frac{x_0^1}{f(x_0)} & \cdots & \frac{x_0^n}{f(x_0)} \\ 0 & -2\frac{x_0^1}{f(x_0)} & \cdots & 0 \\ \frac{x_1^0}{f(x_1)} & \frac{x_1^1}{f(x_1)} & \cdots & \frac{x_1^n}{f(x_1)} \\ 0 & -2\frac{x_1^1}{f(x_1)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{x_{n/2-1}^0}{f(x_{n/2-1})} & \frac{x_{n/2-1}^1}{f(x_{n/2-1})} & \cdots & \frac{x_{n/2-1}^n}{f(x_{n/2-1})} \\ 0 & -2\frac{x_{n/2-1}^1}{f(x_{n/2-1})} & \cdots & 0 \\ \frac{\xi^0}{f(\xi)} & \frac{\xi^1}{f(x_0)} & \cdots & \frac{\xi^n}{f(\xi)} \end{pmatrix}$$

et donc

$$\widetilde{A}\overrightarrow{\delta} = \widehat{\varepsilon}^* \begin{pmatrix} \beta_0 & 0 & & \\ -\beta_0 & \beta_1 & & \\ & \ddots & \ddots & \\ & & -\beta_{n-1} & \beta_n \end{pmatrix} \overrightarrow{s}$$

Considérons maintenant la sous-matrice $\widetilde{A}'$ de la matrice $\widetilde{A}$ définie en ne prenant que les lignes et colonnes paires de $\widetilde{A}$. Définissons de la même façon A' sous-matrice de A et $\overrightarrow{\delta}'$ sous-vecteur de $\overrightarrow{\delta}$ ne contenant que les éléments d'indice pair de $\overrightarrow{\delta}$. On obtient :

$$\widetilde{A}' = -2 \begin{pmatrix} \frac{x_0^1}{f(x_0)} & \frac{x_0^3}{f(x_0)} & \cdots & \frac{x_0^{n-1}}{f(x_0)} \\ \frac{x_1^1}{f(x_1)} & \frac{x_1^3}{f(x_1)} & \cdots & \frac{x_1^{n-1}}{f(x_1)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{x_{n/2-1}^1}{f(x_{n/2-1})} & \frac{x_{n/2-1}^3}{f(x_{n/2-1})} & \cdots & \frac{x_{n/2-1}^{n-1}}{f(x_{n/2-1})} \end{pmatrix} = -2A'$$

Le vecteur $\overrightarrow{s}$ est de la forme $s_{2i} = s_{2i+1} \in \{-1; 1\}$, on peut donc prendre le vecteur $\overrightarrow{s}'$ définie par $s'_i = s_{2i}$. Puisque les entrées sur les colonnes impaires et lignes paires de la matrice $\widetilde{A}$ sont nulles et que $\overrightarrow{s}$ est de cette forme particulière, on obtient :

$$\widetilde{A}'\overrightarrow{\delta}' = \widehat{\varepsilon}^* \begin{pmatrix} -\beta_0 & \beta_1 & & \\ & -\beta_2 & \beta_3 & \\ & & \ddots & \ddots \\ & & -\beta_{n-2} & \beta_{n-1} \end{pmatrix} \overrightarrow{s}'$$

En prenant $\beta'_i = -2 \cdot s_i \cdot (\beta_{2i+1} - \beta_i)$ et la matrice de type Vandermonde

$$V = \begin{pmatrix} x_0^1 & x_0^3 & \dots & x_0^{n-1} \\ x_1^1 & x_1^3 & \dots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n/2-1}^1 & x_{n/2-1}^3 & \dots & x_{n/2-1}^{n-1} \end{pmatrix},$$

on peut réécrire le système comme

$$V \overrightarrow{\delta}' = \widehat{\varepsilon}^* \begin{pmatrix} \beta'_0 \cdot f(x_0) \\ \beta'_1 \cdot f(x_1) \\ \vdots \\ \beta'_{n/2-1} \cdot f(x_{n/2-1}) \end{pmatrix}.$$

Il est alors facile de voir que les coefficients impairs du polynôme de Remez – éléments du vecteur $\overrightarrow{\delta}'$ – sont les coefficients d'un polynôme interpolant la fonction f pondérée par de valeurs très petites $\widehat{\varepsilon}^*$ et β'_i. Cette interpolation se fait dans les points où le polynôme minimax p^* atteint ses extrema. En pratique, ces points sont proches des points de Tchebychev. En outre, les dérivées successives de f décroissent en pratique. Le phénomène de Runge [95] ne se produit donc pas en pratique. Le polynôme δ' reste donc petit en norme et comme le domaine dom ne contient que des valeurs plus petites que 1, les coefficients restent petits.

Récapitulons donc l'algorithme de Remez classique calcule des polynômes de Remez d'une qualité $\overline{\eta} = 1 + \overline{\beta}$ ajustable. Pour des fonctions paires ou impaires, les polynômes de Remez ne sont pas paires ou impaires en général mais leurs coefficients impaires respectivement paires seront très petits. Leur petitesse sera d'autant plus remarquable que la qualité du polynôme sera bonne.

Extensions Comme le phénomène des coefficients impairs ou pairs petits mais non-nuls pour les fonctions paires ou impaires n'est en quelque sorte dû qu'au caractère numérique l'algorithme de Remez, on peut se demander si un calcul direct des coefficients d'un polynôme pair ne serait pas possible.

Un remplacement simple des coefficients non-nuls petits par zéro brise en pratique la propriété de bonne approximation d'un polynôme de Remez. Sa fonction d'erreur n'oscille plus de façon à satisfaire le théorème de Tchebychev. Elle n'atteint plus suffisamment de fois des extrema ou ses extrema ne sont pas globaux. La qualité du polynôme en souffre donc de façon non négligeable.

La littérature classique [93, 101] propose une recette pour pouvoir exploiter une implantation classique de l'algorithme de Remez dans le but de calculer un tel polynôme de Remez pair ou impair pour une fonction paire ou impaire : l'approche passe principalement par un changement de variable $X = x^2$ et un calcul d'un polynôme en X de degré moitié pour la fonction $g(x) = f(\sqrt{X})$ dans le cas d'une fonction paire ou $g(x) = \frac{f(\sqrt{X})}{\sqrt{X}}$ dans le cas d'une fonction impaire. Seul la moitie positive du domaine $dom = [-a; a]$ est considérée. Le domaine devient donc $I = [0; -\sqrt{a}]$. Le polynôme $p(X) = p(x^2)$ (ou $x \cdot p(X) = x \cdot p(x^2)$) est alors un polynôme dont les coefficients impairs (respectivement pairs) sont nuls. La fonction, le

polynôme et le domaine dom étant symétriques, le polynôme obtenu est proche du minimax avec une qualité $\overline{\eta}$.

La symétrie d'une fonction se traduit par un développement de Taylor dans le point de symétrie qui contient des coefficients nuls. Puisque le développement de Taylor peut en pratique fournir une bonne approximation, le polynôme de Remez suit la structure du polynôme de Taylor. La preuve de cette propriété est une adaptation de la preuve précédente que les coefficients impairs d'un polynôme de Remez pour une fonction paire restent petits.

Quand on essaie donc d'appliquer la recette connue à des fonction dont les symétries ou particularités dans le développement de Taylor sont plus complexes, on observe les phénomènes suivants :

- La recette s'adapte facilement à des fonctions dont la symétrie est d'ordre supérieur, comme $\cos x^2$. Il suffit de faire un changement de variable $X = x^{2^k}$, où k est l'ordre de la symétrie.
- Des fonctions comme $\sin x + \cos x^2$, somme de fonctions symétriques de différent ordre, ont des développements de Taylor avec des coefficients nuls pour les degrés satisfaisant une certaine congruence. En l'occurrence pour la fonction d'exemple $\sin x + \cos x^2$, les coefficients de degré $2 + 4k$ sont nuls. Il devient très difficile d'adapter la recette du changement de variables à de telles structures. On pourrait penser à l'appliquer aux fonctions sommandes, $\sin x$ et $\cos x^2$ dans l'exemple, une par une et d'additionner les polynômes de Remez. Cette technique produit un polynôme ayant la structure voulue mais rien n'assure son caractère de minimisation de l'erreur maximale, c'est-à-dire sa qualité.
- Certaines fonctions comme $e^{\sin x + \cos x^2}$ ont un seul coefficient nul dans leur développement de Taylor. En l'occurrence, seul le coefficient de degré 3 est nul. Il semble impossible de trouver un changement de variable tel que l'algorithme de Remez classique puisse être utilisé pour calculer un polynôme ayant un seul coefficient nul. Peut-être le lecteur peut-il en trouver un ?

Bases de monômes complètes et incomplètes Il s'avère en fait que la recette du changement de variable n'a été inventée que pour la raison que les mises en œuvre de l'algorithme de Remez communément disponibles dans les outils comme Maple ou Mathematica n'implantent qu'un cas particulier de l'algorithme de Remez. Elles ne traitent que le cas d'approximations par des polynômes

$$p(x) = \sum_{k=0}^{n} p_k \psi_k(x)$$

dont les fonctions de base ψ_k forment la base complète des monômes $\psi_k(x) = x^k$. En réalité, l'algorithme de Remez peut travailler sur un ensemble de fonctions de base ψ_k quelconque. Ceci est déjà prévu par la formulation initiale du problème donnée par Remez [112].

Il est en particulier possible de considérer des bases monomiales $\psi_k(x) = x^{i_k}$ avec $i_k \in \ell$. Ici ℓ est une liste de degré de monômes à considérer. Pour approcher une fonction paire, par exemple $f = \cos$, par un polynôme pair de degré n il suffit donc d'appliquer l'algorithme de Remez sur f, le domaine dom et la liste $\ell = \{0, 2, 4, \ldots, n\}$. Pour une fonction avec une symétrie d'ordre supérieur ou un développement de Taylor particulier, on choisit facilement la liste ℓ appropriée.

Techniquement, cette extension de l'algorithme présenté ci-dessus demande d'abord une modification du système d'interpolation qui devient

$$\begin{pmatrix} \frac{x_0^{i_0}}{f(x_0)} & \frac{x_0^{i_1}}{f(x_0)} & \cdots & \frac{x_0^{i_n}}{f(x_0)} & (-1)^0 \\ \frac{x_1^{i_0}}{f(x_1)} & \frac{x_1^{i_1}}{f(x_1)} & \cdots & \frac{x_1^{i_n}}{f(x_1)} & (-1)^1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{x_{n+1}^{i_0}}{f(x_{n+1})} & \frac{x_{n+1}^{i_1}}{f(x_{n+1})} & \cdots & \frac{x_{n+1}^{i_n}}{f(x_{n+1})} & (-1)^{n+1} \end{pmatrix} \cdot \begin{pmatrix} p_{i_0} \\ p_{i_1} \\ \vdots \\ p_{i_n} \\ -\widetilde{\varepsilon} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

Ensuite, un problème théorique doit être résolu. Il est relié à une propriété appelée condition de Haar. Cette condition établit si l'ensemble des fonctions ψ_i forme, sur le domaine dom, une base d'un espace de dimension aussi grande qu'il y a de coefficients inconnus. Elle s'exprime classiquement [23, 112] par le caractère non-nul du déterminant $\det V$ de la matrice de Vandermonde V associée aux fonctions ψ_i.

$$V = \begin{pmatrix} \frac{x_0^{i_0}}{f(x_0)} & \frac{x_0^{i_1}}{f(x_0)} & \cdots & \frac{x_0^{i_n}}{f(x_0)} \\ \frac{x_1^{i_0}}{f(x_1)} & \frac{x_1^{i_1}}{f(x_1)} & \cdots & \frac{x_1^{i_n}}{f(x_1)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{x_{n+1}^{i_0}}{f(x_{n+1})} & \frac{x_{n+1}^{i_1}}{f(x_{n+1})} & \cdots & \frac{x_{n+1}^{i_n}}{f(x_{n+1})} \end{pmatrix}$$

Pour que la condition de Haar soit vérifié, $\det V$ doit être non-nul pour tout choix de points x_i dans le domaine, $x_0, \ldots, x_n \in dom$.

La vérification de la condition de Haar est une des hypothèses du théorème de Tchebychev. Ce théorème est à la base du bon fonctionnement de l'algorithme de Remez qui essaie de le satisfaire en raffinant itérativement les polynômes $p^{[j]}$ qu'il calcule. Sans condition de Haar, il n'est plus clair que les extrema globaux de la fonction erreur du polynôme minimax satisfassent la condition d'alternance des signes [119, 23, 112].

La matrice V est également une sous-matrice du système résolu à chaque itération de l'algorithme de Remez. La condition de Haar assure alors aussi que le système ne sera jamais singulier aux différentes étapes de l'algorithme.

L'implantation de l'algorithme de Remez dans l'outil Sollya est capable de gérer le cas d'une liste de degré de monômes, le cas de l'approximation avec minimisation de l'erreur relative et non absolue et le cas d'une fonction f qui s'annule dans le domaine dom. Si le système ainsi défini satisfait la condition de Haar, l'algorithme est très efficace en pratique et a toujours fourni un résultat de bonne qualité. Récemment des modifications ont été apportées[7] à cette implantation pour pouvoir traiter des problèmes sans condition de Haar. En pratique, cette implantation, basée sur les travaux de Stiefel [119], donne souvent des résultats exploitables de bonne qualité. L'efficacité de l'algorithme baisse considérablement dans ces cas d'absence de condition de Haar ; la convergence semble être linéaire. Théoriquement, aucune garantie sur la convergence ni sur la qualité du polynôme finale ne peut être donnée pour l'instant.

[7] L'implantation de l'algorithme de Remez dans Sollya est l'œuvre de S. Chevillard.

Estimer le degré d'un polynôme Remarquons que l'algorithme de Remez peut être facilement modifié et combiné avec une recherche par dichotomie afin de résoudre le problème calculatoire suivant. Un générateur automatique d'approximations polynomiales doit estimer à un moment le degré n nécessaire pour satisfaire une erreur cible $\overline{\varepsilon}$ donnée. Cette estimation ne doit pas être très précise car elle peut ensuite être raffinée en calculant effectivement un polynôme d'approximation à une bonne qualité η et en ajustant le degré n si son erreur maximale $\widehat{\varepsilon}$ est trop grande ou beaucoup trop petite.

Un tel algorithme d'intuition d'un degré est intégré à l'outil Sollya. Il est basé sur la première étape de l'algorithme de Remez calculant une polynôme interpolateur dans les points de Tchebychev et une erreur estimé $\widetilde{\varepsilon}$. Une dichotomie raffine ensuite un intervalle de degré possibles pour satisfaire une erreur cible $\overline{\varepsilon}$. Dans la suite, nous ferons référence à cet algorithme sous son nom Sollya `guessdegree`[8].

6.3.2 Détermination de polynômes réels à évitement de cancellations

Symétries et polynômes de Remez L'étude des différentes techniques d'approximation polynomiale à la section précédente a montré que l'utilisation de l'algorithme de Remez nous permet de disposer de polynômes approximateurs de qualité aussi bonne que voulue à un coût pratique relativement bas. Pour des fonctions paires ou impaires, ces polynômes de Remez ne sont pas forcément pairs ou impairs mais ont des coefficients impairs ou pairs petits. Pour permettre une évaluation rapide du polynôme d'approximation en économisant la moitié des chargements mémoire et la moitié des opérations pour une fonction symétrique, il est nécessaire de trouver un moyen de détecter la présence d'une fonction symétrique dans le générateur automatique de fonctions et de calculer un polynôme d'approximation symétrique.

Nous présentons à présent une variante d'implantation de l'algorithme de Remez qui permet la détermination de bons polynômes d'approximation dans une base de monômes quelconque. Le calcul d'un polynôme de Remez symétrique pour une fonction symétrique est alors possible. En pratique, ces techniques se transposent à des fonctions ayant des symétries d'ordre supérieur ou juste quelques coefficients nuls dans leur développement de Taylor.

Cancellations dans l'évaluation des polynômes d'approximation Les polynômes approximateurs de fonctions servent à remplacer celles-ci dans le processus d'évaluation. Dans le processus automatique de génération d'implantations, ils sont donc calculés pour être fournis au générateur de séquence d'évaluations polynomiales. Un tel générateur a été décrit à la section 6.2. Force est de constater que l'approximation polynomiale ne peut pas être complètement découplée de la génération de séquences d'évaluation. À la section 6.2.2, afin d'assurer que l'évaluation du polynôme puisse bien se faire avec la technique proposée, l'hypothèse suivante a dû être faite : soit

$$p(x) = \sum_{i=0}^{n} p_i \cdot x^i$$

[8]Cette commande a été développé conjointement avec S. Chevillard

le polynôme à évaluer par schéma de Horner avec le sous-polynômes $p_j(x)$ de l'étape j

$$p_j(x) = p_j + x \cdot q_j(x) = p_j + x \cdot \sum_{i=j+1}^{n} p_i \cdot x^i.$$

Soit

$$\beta_j(x) = \frac{q_j(x)}{p_j + x \cdot q_j(x)} = 1 - \frac{1}{1 + \frac{x \cdot q_j(x)}{p_j}} = 1 - \frac{1}{1 + \alpha_j(x)}$$

le rapport entre la valeur du polynôme de l'étape $j+1$ et la valeur du polynôme de l'étape j. Soit

$$\overline{\beta}_j = \|\beta_j\|_\infty^{dom}$$

le maximum de ce rapport. Pour que l'évaluation de Horner avec adaptation des précisions multi-double se fasse bien, ces facteurs d'amplification d'erreur $\overline{\beta}_j$, dépendants du polynôme p, doivent vérifier que $\overline{\beta}_j \leq 1$.

Dans l'analyse des erreurs d'arrondi dans une évaluation de polynôme sous forme de Horner (cf. section 6.2.2), c'était le facteur d'amplification $\beta_j(x)$ qui convenait. Pour ce qui suit, il est pourtant mieux de considérer le rapport $\alpha_j(x)$ entre les opérandes p_j et $x \cdot q_j(x)$ de l'addition à l'étape j. Une quantité s'exprime par l'autre ; on a

$$\alpha_j(x) = \frac{x \cdot q(x)}{p_j} = \frac{1}{1 - \beta_j(x)} - 1.$$

Pareillement, la condition sur $\overline{\beta}_j$ s'exprime par une condition sur $\alpha_j(x)$:

$$|\beta_j(x)| \leq 1 \Rightarrow \alpha_j(x) \geq -1/2$$

En d'autres mots, la condition exprime le fait que l'opérande $x \cdot q_j(x)$ de l'addition à l'étape j du schéma de Horner ne devient pas plus petite que l'opposé de la moitié de l'autre opérande p_j :

$$\alpha_j(x) = \frac{x \cdot q_j(x)}{p_j} \geq -1/2$$
$$\Rightarrow \quad -1/2 \cdot p_j \leq x \cdot q(x)$$

L'addition flottante ne soustrait jamais plus que la moitié de p_j ; les exposants de p_j et du résultats ne diffèrent donc au plus de 1.

La condition sur les $\overline{\beta}_j$ n'est donc rien d'autre qu'une condition d'absence de cancellation lors des additions flottantes dans l'évaluation de Horner. Pour que l'évaluation puisse bien se faire par ce schéma en arithmétique multi-double, le processus d'approximation polynomiale doit donc générer des polynôme qui ne provoqueront pas de cancellations.

D'une pierre deux coups – non-cancellation et symétries L'approximation polynomiale doit alors se faire sous deux prémisses :

- pour une fonction f donnée ayant une symétrie, un polynôme symétrique doit être calculé et,
- pour toute fonction, symétrique ou non, le polynôme d'approximation ne doit pas engendrer de cancellations.

Nous allons proposer un algorithme de calcul d'approximants polynomiaux qui répond aux deux problématiques à la fois.

Notre algorithme sera basé sur deux observations simples sur les coefficients d'un polynôme de Remez dans la base de monômes complète :

– Comme déjà vu, pour des fonctions impaires ou paires, le polynôme de Remez a des coefficients pairs ou impaires très petits. En considérant juste le rapport entre des coefficients consécutifs $\frac{p_{j+1}}{p_j}$, on peut contourner les effets d'ordre de grandeur dans la petitesse des coefficients. Si la fonction est paire et que j est impair, le rapport $\frac{p_{j+1}}{p_j}$ sera très grand. Typiquement, pour une qualité $\eta = 1 + \beta$ du polynôme de Remez, le rapport sera de l'ordre de $\mathcal{O}(1/\beta)$. En effet, comme on a

$$|p_{j+1}| \gg |p_j|$$

et que l'on peut rendre le rapport $\frac{p_{j+1}}{p_j}$ quasiment aussi grand que l'on veut en influant sur la qualité η du polynôme de Remez, on peut même partir de l'hypothèse que

$$|4 \cdot x \cdot p_{j+1}| > |p_j|$$

pour $x \in dom$ avec les domaines autour de zéro habituels. Avec l'hypothèse que les coefficients du polynôme d'approximation décroissent avec leur degré, vérifiée en pratique, on en déduit que

$$|2 \cdot x \cdot (p_{j+1} + x \cdot q_{j+1}(x))| > |p_j| .$$

En conséquence,

$$\alpha_j(x) = \frac{x \cdot q_j(x)}{p_j} = \frac{x \cdot (p_{j+1} + x \cdot q_{j+1}(x))}{p_j}$$

admettra un extremum global de valeur absolue relativement grand et au moins plus grand que $\frac{1}{2}$. En pratique, la norme $\|\alpha_j\|_\infty^{dom}$ sera donc bien plus grande que $1 > 1/2$ pour les coefficients p_j impairs ou pairs des polynômes de Remez pour des fonctions paires ou impaires.
Le test si un coefficient dans un polynôme de Remez est donc une valeur petite qui, dans le polynôme minimax, serait nulle, se réduit donc à un calcul de norme infini.

– Tel qu'il est calculé dans la base de monômes complète par l'algorithme de Remez, un polynôme peut déjà être approprié à une évaluation sous schéma de Horner. Le test qui établit qu'un polynôme peut être évalué sans cancellations ou non est simple. Il suffit de calculer le minimum

$$\underline{\alpha}_j = \min \{\alpha_j(x) | x \in dom\}$$

pour chaque degré de monôme j dans le polynôme de Remez. Si $\underline{\alpha}_j < -1/2$, il y a une cancellation lors de la j-ième addition dans le schéma de Horner. En effet, pour $\underline{\alpha}_j < -1/2$ on a

$$|x \cdot q_{j+1}(x)| \geq 1/2 \cdot p_i$$

ce qui implique une cancellation tant que la valeur absolue de $|x \cdot q_{j+1}(x)|$ ne dépasse pas $2 \cdot p_i$. Ce dépassement n'est pas observé en pratique. Principalement, la condition donnée est une condition sur les exposants relatifs de $x \cdot q_{j+1}(x)$ et de p_i. D'ailleurs, la condition a déjà été nécessaire pour le calcul des précision, explicité à la section 6.2.2.
Le calcul du minimum $\underline{\alpha}_j$ est aussi difficile que le calcul d'une norme infini. La norme infini $\|\alpha\|_\infty^{dom}$ peut être déduite de $\underline{\alpha}_j$ et $\overline{\alpha}_j = \max \{\alpha_j(x) | x \in dom\}$.

Il nous est alors possible de calculer un polynôme de Remez dans la base complète et de tester à la fois si polynôme contient des coefficients petits à la place de coefficients nuls dans le polynôme minimax pour une fonction symétrique et si le polynôme peut être évalué sous forme de Horner. Nous obtenons dans tous les cas une base de monômes B à garder pour la raison que les coefficients associés ne sont ni des faux zéros ni opérandes d'additions qui cancellent.

À partir de la base de monômes B à garder, il est alors possible de recalculer un polynôme de Remez approchant la fonction donnée. Ce polynôme aura donc des coefficients nuls là où il avait auparavant des cancellations ou des valeurs petites sans information pertinente. Cela veut dire que cet algorithme d'approximation par polynômes évite

- les cancellations en supprimant complètement l'opération qui cancellerait et
- des opérations superflues pour l'évaluation de fonctions symétriques en permettant le précalcul de puissances appropriées de la variable libre du polynôme.

Évidemment, en ôtant des monômes de la base complète, nous donnons moins d'espace d'optimisation à l'algorithme de Remez. Le polynôme de Remez recalculé dans la base B incomplète peut donc engendrer une erreur d'approximation plus grande que le polynôme de Remez dans la base complète. En particulier, il n'est pas sûr que le polynôme dans la base incomplète satisfasse une erreur cible $\overline{\varepsilon}$ donnée si celui dans la base complète la satisfait. Dans le cas où le polynôme sans cancellations et symétrique provoque une erreur trop grande, il est alors nécessaire d'itérer sur le processus. En pratique, avec des domaines dom de taille petite et contenant l'origine, on peut supposer qu'une augmentation du degré des polynômes fait baisser l'erreur d'approximation.

L'algorithme complet se lit donc :

Entrées : une fonction f, un domaine dom, une erreur cible $\overline{\varepsilon} \in \mathbb{R}^+$, une limite $l \in \mathbb{N}$
Sorties : un polynôme p, n'engendrant pas de cancellations sous évaluation de Horner, tel que $\|p/f - 1\|_\infty^{dom} \leq \overline{\varepsilon}$, $\perp$ si aucun tel polynôme ne peut être trouvé

Algorithme `approcheFonction` (f,dom,l) :
début
  $n \leftarrow$ `guessdegree`$(f,dom,\overline{\varepsilon}) - 1$; /* Calculer une estimation initiale du degré */
  répéter /* Itérer jusqu'à trouver le degré n nécessaire */
    $n \leftarrow n + 1$;
    $p^* \leftarrow$ `remez`$(f, dom, \{x^0, \ldots, x^n\})$; /* Remez en base complète */
    $\varepsilon \leftarrow \|p^*/f - 1\|_\infty^{dom}$; /* Calculer la borne d'erreur */
  jusqu'à $\varepsilon \leq \overline{\varepsilon}$;
  $k \leftarrow 1$; /* Compte des itérations */
  répéter /* Itérer jusqu'à trouver un polynôme non-cancellant */
    $B \leftarrow \{x^n\}$; /* B contient la base des additions non-cancellantes */
    nonCancellant $\leftarrow$ vrai ; $q_n \leftarrow p_n^*$; $k \leftarrow k + 1$;
    pour $i \leftarrow n - 1$ **à** 0 **faire** /* Simuler statiquement les étapes de Horner */
      $\underline{\alpha'} \leftarrow \inf\{x \cdot q_{i+1}(x) | \, x \in dom\}$; $\overline{\alpha'} \leftarrow \sup\{x \cdot q_{i+1}(x) | \, x \in dom\}$;
      $\alpha' \leftarrow \left(|\underline{\alpha'}|, |\overline{\alpha'}|\right)$;
      si ($\alpha' \leq 1/2\,|p_i^*|$) *ou* ($\underline{\alpha'}, \overline{\alpha'}, p_i^*$ *ont le même signe*) **alors** /* Cancellation ? */
        $B \leftarrow B \cup \{x^i\}$; /* Pas de cancellation, rajouter le monôme */
        $q_i \leftarrow p_i^* + x \cdot q_{i+1}$; /* Étape de Horner statiquement simulé */
      sinon
        nonCancellant $\leftarrow$ faux ; /* Cancellation */
        $q_i \leftarrow x \cdot q_{i+1}$; /* Étape de Horner statiquement simulé */
      fin
    fin
    si nonCancellant **alors renvoyer** p^*; /* Polynôme est non-cancellant */
    else
      $p \leftarrow$ `remez`(f, dom, B); /* Remez en base incomplète B */
      $\varepsilon \leftarrow \|p/f - 1\|_\infty^{dom}$; /* Calculer la borne d'erreur associée */
      si $\varepsilon \leq \overline{\varepsilon}$ **alors** $p^* \leftarrow p$; /* L'erreur du polynôme est bonne ? */
      sinon
        $n \leftarrow n + 1$; /* Augmenter le degré n de 1 */
        $p^* \leftarrow$ `remez`$(f, dom, \{x^0, \ldots, x^n\})$; /* Remez en base complète */
      fin
    end
  jusqu'à $k > l$;
  renvoyer $\perp$; /* Aucun polynôme n'a été trouvé en l itérations */
fin

Algorithme 12 : Algorithme pour le calcul de polynômes non-cancellants

Fonctions boîtes noires L'algorithme 12 proposé a été développé pour répondre à deux buts, qui sont l'approximation polynomiale n'engendrant pas de cancellations et l'optimisa-

tion de fonctions symétriques. En quelque sorte, il est une sur-couche numérique de l'algorithme de Remez pour palier à certains défauts de cet algorithme dans la pratique (cf. section 6.3.1). Puisque l'algorithme de Remez peut fonctionner avec une fonction à approcher donnée par une boîte noire et qu'aucune technique de manipulation formelle n'est nécessaire sur la fonction, nous avons alors un moyen de générer automatiquement une approximation polynomiale, adaptée à l'évaluation et optimisée pour ce qui est des symétries, même pour des fonctions définies par un code externe. Nous considérons que ceci est un point important car il nous permet de traiter plus de fonctions automatiquement que des simples fonctions de base ou leurs composées.

Symétries d'ordre supérieur et développements de Taylor particuliers Évidemment, tout ce qui a été dit sur la gestion de fonctions symétriques se transpose à des fonctions de symétrie d'ordre supérieur ou asymétriques mais ayant un ou plusieurs coefficients de leur développement de Taylor nuls. En pratique, notre algorithme force exactement ces coefficients dans le polynôme de Remez à zéro. Comme le lien entre le polynôme de Taylor et le polynôme de Remez est faible, certains problèmes peuvent être observés lors de ce processus heuristique.

Problématiques de l'approche proposée Par exemple, notre algorithme échoue dans la tentative de trouver un polynôme non-cancellant pour des fonctions dont une dérivée s'annule dans le domaine dom mais que ce zéro n'est pas en l'origine. Ce phénomène est rare pour des fonctions considérés dans le cadre d'implantation de bibliothèques mathématiques comme CRLibm. Il est pourtant possible de construire la fonction $f(x) = \cos(\pi + 1/16 + x)$ dans le domaine $dom = [-1/4; 1/4]$ pour l'observer sur l'algorithme mis en œuvre.

Exemples pratiques Les exemples pratiques suivants illustrent bien le fonctionnement et les limitation de l'algorithme proposé.

Une fonction symétrique Commençons par l'exemple de la fonction $\sin x$, symétrique dans le domaine $dom = [-\pi/64; \pi/64]$. Choisissons comme erreur cible $\overline{\varepsilon} = 2^{-61}$, comme dans des implantations de $\sin$ pour la double précision. Notre algorithme sélectionne un polynôme impair dans la base $\{x^1, x^3, x^5, x^7, x^9\}$. Ayant les coefficients suivants, il engendre une erreur d'approximation maximale de $2^{-69.3}$:

$$\begin{array}{lcllcl} p_1 & = & 1 & p_7 & = & -1830034014729925 \cdot 2^{-63} \\ p_3 & = & -6004799503160661 \cdot 2^{-55} & p_9 & = & 6500423238003495 \cdot 2^{-71} \\ p_5 & = & 2401919801258541 \cdot 2^{-58} & & & \end{array}$$

Un développement de Taylor particulier Considérons maintenant la fonction $f(x) = e^{\sin x - \cos x^2}$ dans le domaine $dom = \left[-2^{-5}; 2^{-8}\right]$ avec une erreur cible de 2^{-90}. Ni la fonction ni le domaine sont symétrique. En revanche, la fonction présente un coefficient nul pour le monôme de degré 3 dans son développement de Taylor.

Notre algorithme choisit la base monomiale $\{x^0, x^1, x^2, x^4, x^5, x^6, x^7, x^8, x^9\}$. Il omet alors

aussi le monôme x^3. Les coefficients du polynôme final s'écrivent :

$$
\begin{aligned}
p_0 &= 11938370416962674342846939687834 3 \cdot 2^{-108} \\
p_1 &= 29845926042406685857117349204375 \cdot 2^{-106} \\
p_2 &= 11938370416962674342843662138536 3 \cdot 2^{-109} \\
p_4 &= 4970345142530923 \cdot 2^{-55} \\
p_5 &= 358969371405011 \cdot 2^{-51} \\
p_6 &= 6516674741954513 \cdot 2^{-56} \\
p_7 &= 589077943038783 \cdot 2^{-57} \\
p_8 &= 5559725200690211 \cdot 2^{-59} \\
p_9 &= 5320394595779079 \cdot 2^{-58}
\end{aligned}
$$

Avec ce polynôme, l'erreur d'approximation $\|p/f - 1\|_\infty^I$ est bornée par $2^{-90.4}$. L'évaluation peut se faire sans cancellations. Nous avons généré la séquence d'évaluation avec nos algorithmes. La preuve Gappa pour la borne d'erreur d'arrondi associée démontre également qu'il n'y a pas de cancellations possibles.

Dépendance de la base de monômes choisie en fonction de l'erreur cible Intéressons-nous maintenant à l'évolution de la base monomiale choisie par notre algorithme en fonction de l'erreur cible $\bar{\varepsilon}$. À titre d'exemple, prenons la fonction $f(x) = e^{\cos x^2 + 1}$, paire au degré 2 dans le domaine $dom = \left[-2^{-8}; 2^{-5}\right]$. Le tableau suivant illustre cette évolution :

erreur cible $\bar{\varepsilon}$	base monomiale
2^{-40}	$\{x^0, x^4\}$
2^{-50}	$\{x^0, x^4, x^8\}$
2^{-60}	$\{x^0, x^4, x^8\}$
2^{-70}	$\{x^0, x^4, x^8, x^{12}\}$
2^{-80}	$\{x^0, x^4, x^8, x^{12}\}$
2^{-90}	$\{x^0, x^4, x^8, x^{12}\}$
2^{-100}	$\{x^0, x^4, x^8, x^{12}, x^{13}, x^{14}, x^{15}\}$
2^{-110}	$\{x^0, x^4, x^8, x^{12}, x^{16}\}$
2^{-120}	$\{x^0, x^4, x^8, x^{12}, x^{16}, x^{17}, x^{18}\}$

Il est intéressant d'observer que pour certaines erreurs cibles comme pour $\bar{\varepsilon} = 2^{-100}$, notre algorithme choisit une base asymétrique, c'est-à-dire une base dont les degrés des monômes non-nuls ne sont pas tous de même reste pour une division avec un petit entier. En l'occurrence, il s'agit de la base $\{x^0, x^4, x^8, x^{12}, x^{13}, x^{14}, x^{15}\}$, pour laquelle on a par exemple $12 \bmod 4 = 0$ et $15 \bmod 4 = 3$. Pour écrire le polynôme dans cette base, plus de coefficients sont nécessaires que pour la base choisie à une erreur cible correspondant à une précision plus grande : en effet pour écrire la base pour $\bar{\varepsilon} = 2^{-100}$, 7 coefficients doivent être stockés, alors que pour la base pour $\bar{\varepsilon} = 2^{-110}$, seulement 5 coefficients sont nécessaires. Ce comportement peut être vu comme un défaut de notre algorithme. En pratique, il n'en est pas un : l'effet affecte seulement les coefficients de monômes de haut degré. L'évaluation peut donc déjà commencer pendant le précalcul de la puissance, en l'occurrence x^4, se fait.

L'exemple d'une fonction boîte noire L'exemple suivant montre que notre algorithme s'applique sans problèmes à des fonctions définies par des boîtes noires. Nous avons implanté la fonction $\operatorname{argerf} = \operatorname{erf}^{-1}$ par une itération de Newton-Raphson sur la fonction $\operatorname{erf} x = \frac{2}{\pi} \cdot \int_0^x e^{-t^2} dt$. Bien sûr cette technique d'implantation est très inefficace. La fonction

argerf est impaire mais l'algorithme d'approximation polynomiale ne détient pas cette information formelle.

Pourtant pour le domaine $dom = [-1/4; 1/4]$ et l'erreur cible $\overline{\varepsilon} = 2^{-60}$ par exemple, notre algorithme trouve bien la base monomiale impaire $\{x^1, x^3, x^5, x^7, x^9, x^{11}, x^{13}, x^{15}, x^{17}, x^{19}\}$. Le polynôme a les coefficients suivants :

$$
\begin{array}{lcllcl}
p_1 &=& 7189927001527084853557783390719 7 \cdot 2^{-106} & p_{11} &=& 7455281238343373 \cdot 2^{-57} \\
p_3 &=& 3764636974640733041107088597691 3 \cdot 2^{-107} & p_{13} &=& 3086390951797773 \cdot 2^{-56} \\
p_5 &=& 2297847774298601 \cdot 2^{-54} & p_{15} &=& 5269462590206135 \cdot 2^{-57} \\
p_7 &=& 3118369096730189 \cdot 2^{-55} & p_{17} &=& 8758767795225423 \cdot 2^{-58} \\
p_9 &=& 2340416807028733 \cdot 2^{-55} & p_{19} &=& 5369190506948897 \cdot 2^{-57}
\end{array}
$$

L'erreur d'approximation engendré par ce polynôme est borné par $2^{-62.9}$.

Illustration des limitations Dans certains cas, notre algorithme échoue dans l'essai de trouver un polynôme approximateur non-cancellant sous évaluation de Horner. Dans la plupart de cas, cet échec est dû au fait que la fonction s'approche mal dans le domaine donné, c'est-à-dire que la dynamique de ses dérivées successives dans le domaine, donc des coefficients des développements de Taylor dans des points du domaine, est trop grande.

La fonction $f(x) = \log 1 + x$ dans le domaine très large $dom = [-1/2; 1/2]$ en est un bel exemple. Le tableau ci-dessous montre, pour différentes erreurs cibles, la base de monômes choisie par notre algorithme quand il marche. Les irrégularités des bases ne sont pas dues à une quelconque symétrie de la fonction mais purement à l'évitement de cancellations. L'algorithme ne trouve pas de polynôme pour des erreurs cibles donnant plus de précision que 50 bits. Cela dit, il n'est même pas clair si de telles approximations polynomiales existent.

erreur cible	base monomiale
2^{-20}	$\{x^1, \ldots, x^8, x^{11}, x^{12}\}$
2^{-30}	$\{x^1, \ldots, x^{12}, x^{15}, x^{16}\}$
2^{-40}	$\{x^1, \ldots, x^{15}, x^{18}, x^{19}, x^{20}, x^{22}, x^{23}\}$
2^{-45}	$\{x^1, \ldots, x^{17}, x^{20}, x^{21}, x^{22}, x^{24}, x^{25}\}$
2^{-50}	$\{x^1, \ldots, x^{20}, x^{23}, \ldots, x^{28}\}$
2^{-55}	échec

6.3.3 Optimisation de polynômes pour l'arithmétique multi-double

Nous disposons alors d'un algorithme d'approximation polynomiale qui, pour une fonction f, un domaine dom et une erreur cible $\overline{\varepsilon}$ donnés, peut calculer un polynôme d'approximation $p^\star$ dont l'évaluation sous schéma de Horner ne provoquera pas de cancellations. Tout de même, ces polynômes $p^\star$ ne peuvent toujours pas être utilisés directement pour une implantation de f sur machine : leur coefficients sont réels ou pour au moins stockés sur des flottants de très grande précision. Dans le cadre donné, seule la génération de séquences d'évaluation pour des polynômes à coefficients flottants multi-double nous est possible. Il est donc nécessaire de passer de $p^\star$, à coefficients réels, à un polynôme d'approximation

$p(x) = \sum_{i=0}^{n} p_i \cdot x^i$ à coefficients p_i flottants multi-double, c'est-à-dire $p_i \in \mathbb{F}_{53}$, $p_i \in \mathbb{F}_{53} + \mathbb{F}_{53}$ ou $p_i \in \mathbb{F}_{53} + \mathbb{F}_{53} + \mathbb{F}_{53}$.

Arrondi des coefficients réels La littérature classique sur l'implantation de fonctions mathématiques [122, 65, 93, 26, 101] voit dans le problème de passer du polynôme d'approximation $p^\star$ à coefficients réels à un polynôme d'approximation p à coefficients flottants surtout un problème d'arrondi. Les coefficients $p_i^\star$ sont arrondis pour donner les coefficients $p_i = \circ(p_i^\star)$ d'un polynôme p. Si les erreurs de ces arrondis ne sont pas trop grandes, le polynôme p reste un bon approximant de la fonction f. Si une borne sur l'erreur d'approximation $\varepsilon(x) = {}^{p(x)}/_{f(x)} - 1$ est nécessaire, celle-ci peut être obtenue par un calcul de norme infini sur ε ou éventuellement par une technique donnée par Krämer [65].

Ces techniques connues s'appliquent certainement si la précision disponible sur les formats flottants pour stocker les coefficients est plus grande que la précision à obtenir par l'approximation de la fonction f par le polynôme p à coefficients flottants [26]. Dans les cas où la précision des formats flottants disponibles est trop petite, l'erreur d'arrondi des coefficients brise la propriété de bonne approximation de la fonction par le polynôme. Par exemple, on observe en pratique que les oscillations de la courbe d'erreur d'approximation $\varepsilon^\star = {}^{p^\star}/_{f} - 1$ disparaissent : la courbe d'erreur $\varepsilon = {}^{p}/_{f} - 1$ n'atteint souvent que deux extrema. Les conditions du théorème de Tchebychev [23] ne sont donc pas remplies ; l'optimalité du polynôme ne peut même pas être espérée.

La figure 6.1 illustre ce phénomène d'arrondi. Elle trace $\varepsilon^\star$, erreur petite oscillant autour de zéro, et ε, erreur plus grande. L'exemple est basé sur la fonction $f = \operatorname{asin}$ dans $dom = [-1/4; 1/4]$ à une erreur cible équivalente à 60 bits, les coefficients du polynôme de Remez impair sont arrondis à des flottants double précision.

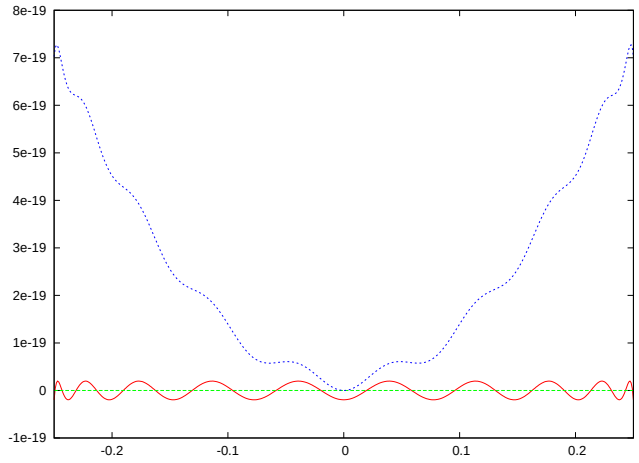


FIG. 6.1 – Simple arrondi des coefficients réels

Dans le cadre de l'arithmétique multi-double, qui est le nôtre, il est bien sûr toujours possible d'augmenter la précision des coefficients en rajoutant des flottants double aux expansions utilisées. La précision finira par être plus grande que la précision cible et donc suffisante. Tout de même, cette approche n'est pas satisfaisante, car intrinsèquement sous-optimale.

État de l'art : une approche à base de réduction de réseaux euclidiens En fait, quand une simple procédure d'arrondi est utilisée pour passer du polynôme $p^\star$ à coefficients réel au polynôme p à coefficients flottants, les arrondis des coefficients $p_i^\star$ se font indépendamment les uns des autres. Les erreurs qu'ils engendrent ne peuvent donc pas se compenser.

La technique pour le passage de $p^\star$ à p proposée par Brisebarre et Chevillard [17], permet explicitement une telle compensation. Leur algorithme, basé sur la réduction de réseaux euclidiens, permet trouver, dans la limite des approximations faites par l'algorithme de réduction LLL [17], le polynôme p à coefficients flottants le plus proche de $p^\star$ pour la norme de l'erreur maximale $\|p/f - 1\|_\infty$. Les résultats obtenus avec cette techniques sont si bons [17] que nous choisissons de l'intégrer à nos algorithmes. Figure 6.2 illustre le bon comportement de la technique sur le même exemple que celui utilisé pour la figure 6.1 ci-dessus. Quasiment aucune différence entre l'erreur du polynôme $p^\star$ et celle de p n'est visible.

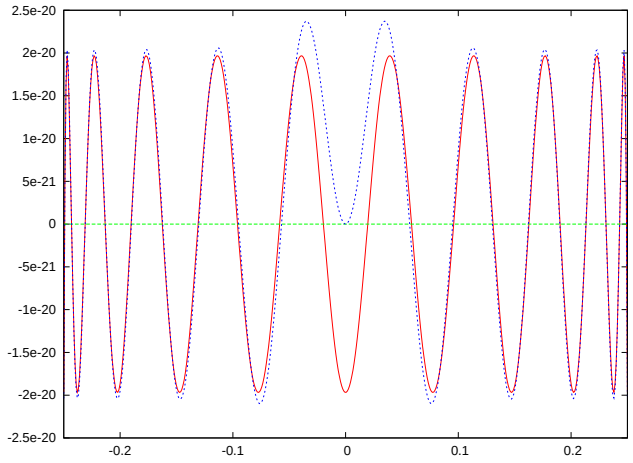


FIG. 6.2 – Application de l'algorithme `fpminimax`

Techniquement, l'algorithme par Brisebarre et Chevillard, que nous appellerons dans la suite `fpminimax` pour *floating-point minimax*, prend comme entrée les informations suivantes :

- un polynôme d'approximation à coefficients réels $p^\star$,
- une liste de précisions $\ell = [k_0, k_1, \ldots, k_n]$, $k_i \in \mathbb{Z}$, pour le format flottant à k_i bits à utiliser sur le i-ième coefficient du polynôme p à calculer, et
- une liste Z de zéros $z_i \in dom$ de la fonction d'erreur $\varepsilon^\star = p^\star/f - 1$.

L'algorithme `fpminimax` renvoie un polynôme $p(x) = \sum_{i=0}^{n} p_i \cdot x^i$ qui vérifie que le i-ième coefficient s'écrit en flottant sur k_i bits, c'est-à-dire $\circ_{k_i}(p_i) = p_i$. On observe que l'erreur $\varepsilon = p/f - 1$ est petite et souvent du même ordre de grandeur que celle du polynôme $p^\star$ à coefficients réels. Pourtant, l'algorithme `fpminimax` ne garantit rien sur l'erreur maximale $\widehat{\varepsilon} = \|\varepsilon\|_\infty^{dom}$. Il essaie juste d'interpoler f dans les points z_i, zéros de la fonction d'erreur et points d'interpolation de f par $p^\star$, compte tenu des contraintes de précision.

Bien qu'intéressante pour l'optimisation, l'utilisation de l'algorithme `fpminimax` dans notre générateur automatique d'implantations n'est donc pas directe :

- La liste ℓ de formats flottants à utiliser doit être calculée à partir de la fonction f, l'erreur cible $\overline{\varepsilon}$ pour être fournie à l'algorithme `fpminimax`.

– Le polynôme p obtenu en résultat de l'algorithme doit être soumis à une vérification de la borne d'erreur. Cela veut dire que la norme infini $\widehat{\varepsilon} = \|p/f - 1\|_\infty^{dom}$ doit être calculée et comparée à l'erreur cible $\overline{\varepsilon}$.
Si l'erreur cible ne peut pas être satisfaite par le polynôme p, la liste des formats doit être adaptée. En pratique, la nécessité d'une telle adaptation arrive souvent.

Parcours automatique des combinaisons multi-double Pour donner un résultat optimal pour les chargements mémoire dans la séquence finale d'évaluation, il est clair que la liste ℓ doit contenir les formats flottants les moins précis possibles : dans notre cadre des multi-double, plus de précision veut dire plus de chargements. Pourtant, il est en pratique impossible de déterminer la liste optimale du premier coup. Pour cela, il faudrait pouvoir prédire exactement le comportement de `fpminimax`.

Nous proposons alors l'algorithme suivant pour exploiter la routine `fpminimax` :

– Calcul d'une liste $\underline{\ell}$ de précisions dites *bornes de précision inférieures*. Un calcul montre qu'il n'y a pas de polynôme à coefficients flottants avec moins de précision que $\underline{\ell}$ satisfaisant l'erreur cible.
Les précisions dans la liste sont exprimées sous la forme de multi-doubles. Cela veut dire que la liste n'indique pas une précision en bits, par exemple 107, mais le nombres de flottants double précision dans une expansion multi-double, dans l'exemple un double-double.
– Calcul d'une liste $\overline{\ell}$ de précisions dites *bornes de précision supérieures*. On montre qu'il y a un polynôme à coefficients flottants avec au moins la précision donnée par $\overline{\ell}$.
Cette liste indique également les précisions sous forme de multi-doubles.
– Parcours de l'espace entre $\underline{\ell}$ et $\overline{\ell}$, commençant à $\underline{\ell}$, pour trouver un polynôme p à un nombre de flottants pour ses coefficients optimisé.
Pour chaque combinaison ℓ de multi-doubles par coefficient, $\underline{\ell} \preceq \ell \preceq \overline{\ell}$, un polynôme p est calculé avec l'algorithme `fpminimax`. Son erreur maximale $\widehat{\varepsilon}$ est calculée par une procédure de norme infini et comparée à l'erreur cible $\overline{\varepsilon}$. Si elle la satisfait, le polynôme p est retenu comme solution du problème du passage du polynôme $p^\star$ à coefficients réels à un polynôme à coefficients machine multi-double.
À chaque itération de ce processus au moins un flottant double précision est rajouté à la liste des formats ℓ par rapport à l'itération d'avant. En plus, le parcours commence par la borne de précision inférieure. La solution p est donc optimale par rapport au nombre total de flottants doubles pour écrire les coefficients du polynôme.
L'ordre du parcours a une influence sur l'efficacité de l'algorithme de génération automatique d'implantations. Dans certains cas, un ordre d'énumération peut être calculé pour optimiser cette efficacité.
Techniquement, l'énumération de tous les cas entre $\underline{\ell}$ et $\overline{\ell}$ est une énumération de tous les entiers l entre $\underline{\ell}$, vu comme un nombre écrit en une base multiple avec des chiffres double (D), double-double (DD) et triple-double (TD), et $\overline{\ell}$, interprété de la même façon. La conversion de base inverse traduit l'entier l en une liste ℓ.
Illustrons ce point à l'aide d'un exemple : supposons que l'on ait $\underline{\ell} = [D, DD, D, D, D]$ et $\overline{\ell} = [TD, DD, DD, D, D]$. En passant en écriture avec des chiffres de 0 à 2, on obtient $\underline{\ell} = [0, 1, 0, 0, 0]$ et $\overline{\ell} = [2, 1, 1, 0, 0]$. La différence composante par composante de ces vecteurs (listes), nous donne une base multiple, en l'occurrence $b = [2, 0, 1, 0, 0]$. Il suffit alors de décaler la borne de précision supérieure $\overline{\ell}$ de la borne de précision inférieure

$\underline{\ell}$ pour obtenir un nombre en cette borne multiple b jusqu'auquel il faut compter en commençant par 0. En occurrence, on obtient $\overline{\ell} - \underline{\ell} = [2, 0, 1, 0, 0] = 00102_b = 5$. En réinterprétant alors les entiers $0, 1, 2, \ldots, 5$ par le processus inverse, on obtient alors d'abord en base multiple les vecteurs $00000_b, 00001_b, 00002_b, 00100_b, 00101_b, 00102_b$ et ensuite les combinaisons (redécalées) ℓ, $\underline{\ell} \preceq \ell \preceq \overline{\ell}$, voulues, dont la suite commence par $[D, DD, D, D, D], [DD, DD, D, D, D], [TD, DD, D, D, D], [D, DD, DD, D, D] \ldots$.

L'algorithme de parcours de l'espace entre $\underline{\ell}$ et $\overline{\ell}$ est donc simple et ne mérite plus d'explications détaillées. Intéressons-nous alors au problème de calcul des bornes de précision supérieures et inférieures.

Calcul automatique de bornes de précision supérieures La liste de bornes de précision supérieures $\overline{\ell}$ donne pour chaque monôme x^i du polynôme $p^\star$ une précision d_i tel qu'il existe un polynôme p qui satisfait l'erreur cible $\overline{\varepsilon}$ et dont le coefficient p_i s'écrit sur d_i flottants double précision. Le calcul de cette liste se réduit donc à la recherche d'une solution non optimisée et en pratique sous-optimale du problème de passer du polynôme $p^\star$ à coefficients réels à un polynôme p à coefficients flottants.

Il est important de connaître la liste $\overline{\ell}$ de bornes de précision supérieures. D'une part, elle assure la terminaison de notre algorithme. D'autre part, le fait de la connaître permet de faire certaines optimisations, comme un parcours non-linéaire de l'espace entre $\underline{\ell}$ et $\overline{\ell}$.

Comme la solution peut être non-optimisée, la technique de l'arrondi séparé de tous les coefficients peut être utilisée. Pour déterminer les précision des arrondis, on peut considérer les coefficients réels comme exactement stockés dans une séquence d'évaluation et chargés pour l'évaluation avec un arrondi. Une modification simple de l'approche proposée à la section 6.2.2 pour l'adaptation des précisions nécessaires pour les opérations dans une évaluation de Horner, permet d'intégrer cette nouvelle source d'erreur d'arrondi. Notons

$$\varepsilon_R(x) = \frac{\circ_{k_i}(p_i^\star) - p_i^\star}{p_i^\star}$$

l'erreur relative de l'arrondi du i-ième coefficient $p_i^\star$ à une précision de k_i bits. L'équation 6.1, à la section 6.2.2, donne l'erreur totale de l'évaluation de Horner à l'étape i en fonction de l'erreur sur l'addition, ε_A, sur la multiplication, ε_M, et l'erreur du sous-polynôme entrant, ε_Q. Il est facile d'y rajouter l'erreur d'arrondi ε_R du coefficient du polynôme. L'équation 6.1 devient :

$$\begin{aligned}
\varepsilon(x) &= \varepsilon_A(x) + (1-\beta(x)) \cdot \varepsilon_R(x) + \beta(x) \cdot \varepsilon_T(x) \\
&+ (1-\beta(x)) \cdot \varepsilon_A(x) \cdot \varepsilon_R(x) + \beta(x) \cdot \varepsilon_A(x) \cdot \varepsilon_T(x) + \beta(x) \cdot (1-\beta(x)) \cdot \varepsilon_T(x) \cdot \varepsilon_R(x) \\
&+ \beta(x) \cdot (1-\beta(x)) \cdot \varepsilon_A(x) \cdot \varepsilon_T(x) \cdot \varepsilon_R(x) \qquad (6.2)
\end{aligned}$$

On remarque donc que l'erreur d'arrondi de l'addition $\varepsilon_A(x)$ de l'étape i du schéma de Horner et l'erreur d'arrondi du i-ième coefficient $p_i^\star$ se comportent pareillement par rapport à l'erreur totale $\varepsilon(x)$. En effet, la valeur $1 - \beta(x)$ reste avec $|\beta(x)| \leq \overline{\beta} \leq 1$ clairement plus petite que 2. La borne pour l'erreur d'addition $\overline{\varepsilon}_A$ donnée à la section 6.2.2 peut donc modifiée pour y rajouter l'erreur de l'arrondi du coefficient. L'algorithme 6 d'adaptation des précisions (cf. page 116) peut donc, dans une version modifiée, fournir une liste $\overline{\ell}$ des précisions multi-double pour laquelle l'arrondi des coefficients et des opérations d'évaluation n'affecte le polynôme que de l'erreur cible $\overline{\varepsilon}$.

Calcul automatique de bornes de précision inférieures Calculer la liste $\underline{\ell}$ des bornes de précisions inférieures est une tâche moins évidente. Il s'agit d'établir le fait qu'il n'existe pas de polynôme d'approximation satisfaisant l'erreur cible $\overline{\varepsilon}$ dont le coefficient p_i peut s'écrire sur k_i bits respectivement d_i flottants double précision. Actuellement, nous utilisons un algorithme particulier pour répondre à cette question. Cet algorithme, présenté ci-dessous, ne marche pas toujours en pratique. Pourtant, il ne se trompe pas, c'est-à-dire soit il renvoie une telle liste correcte soit signale son échec. Le travail sur cet algorithme est toujours en cours.

L'algorithme repose sur l'adaptation d'une technique, connue comme l'approche des polytopes, présentée par Brisebarre et al. [19]. Dans cette approche[9], toutes les solutions d'approximation polynomiale $p(x) = \sum_{i=0}^{n} p_i \cdot x^i$ d'une fonction f à une erreur cible $\overline{\varepsilon}$ sont représentées par un polytope de points formés par les vecteurs $\overrightarrow{p}$ des coefficients p_i des polynômes p. Ces polytopes de solutions font partie d'un espace vectoriel n-dimensionnel sur les réels.

On utilise l'intersection de cet espace avec la grille flottante dans chacune des dimensions. Les polynômes à coefficients flottants solutions du problème d'approximation sous erreur cible $\overline{\varepsilon}$ sont formés par les points sur la grille flottante contenu dans le polytope. En conséquence, il faut choisir la grille flottante, c'est-à-dire fixer une précision, de telle sorte qu'au moins un point de cette grille appartienne au polytope. Résoudre ce problème particulier est difficile et nécessite l'utilisation de programmation linéaire [19]. Il devient plus simple quand une sur-approximation du polytope est utilisée, ce qui est fait pour l'algorithme que nous utilisons.

Une telle sur-approximation de l'espace des solutions est obtenue par un encadrement du polytope par un parallélépipède rectangle n-dimensionnel suivant les axes. Bref, chaque coefficient p_i est borné par un intervalle $P_i = \left[\underline{p}_i; \overline{p}_i\right]$. Quand les tailles de ces intervalles sont petites, typiquement quand les exposants de $\underline{p}_i$ et de $\overline{p}_i$ ne diffèrent qu'au plus un, une borne pour la précision minimale k_i' nécessaire pour le coefficient p_i peut être déduite : elle doit être telle qu'il y a au moins un flottant entre $\underline{p}_i$ et $\overline{p}_i$. En dessous de cette précision, l'intersection de l'encadrement du polytope avec la grille flottante est vide. Il est ainsi possible de calculer la liste $\underline{\ell}$ de bornes de précision inférieures.

Exemples pratiques Intéressons-nous maintenant à des exemples pour le comportement de l'algorithme complet passant les coefficients du polynôme d'approximation dans une forme flottante. Pour cela, nous indiquons pour quelques fonctions f, domaines dom et erreur cible $\overline{\varepsilon}$, les informations suivantes :

- le degré d du polynôme
- la combinaison ℓ de flottants double, double-double et triple-double (D,DD,TD) finale,
- le nombre moyen μ de double par coefficient qui en résulte,
- les bornes $\underline{\ell}$ et $\overline{\ell}$ calculées au cours de l'algorithme et
- le nombre d'itérations c nécessaires sur la procédure `fpminimax` pour trouver la solution retenue.

La liste $\overline{\ell}$ des bornes de précision supérieures, comparée à la liste de formats retenue ℓ, donne idée de l'efficacité de notre approche d'optimisation de coefficients flottants. En effet, sans cette approche, $\overline{\ell}$ est la combinaisons qui aurait dû être choisie.

[9] Il vaut la peine d'annoter qu'E. Remez, dans son ouvrage [112], donne déjà des représentations et graphiques équivalentes pour le problème d'approximation qu'il classifie de *discret* et qu'il résout par sa méthode *graphique*.

Voici les résultats de trois exemples :

1. Fonction exponentielle, faible précision :

$$\begin{aligned} f &= \exp \\ dom &= \left[-2^{-7}; 2^{-7}\right] \\ \overline{\varepsilon} &= 2^{-60} \\ d &= 6 \\ \ell &= [D, \ldots, D] \\ \mu &= 1 \\ \underline{\ell} &= [D, \ldots, D] \\ \overline{\ell} &= [DD, DD, D, \ldots, D] \\ c &= 1 \end{aligned}$$

2. Fonction exponentielle, grande précision :

$$\begin{aligned} f &= \exp \\ dom &= \left[-2^{-15}; 2^{-15}\right] \\ \overline{\varepsilon} &= 2^{-120} \\ d &= 7 \\ \ell &= [DD, DD, DD, DD, D, D, D] \\ \mu &= 1.6 \\ \underline{\ell} &= [DD, DD, DD, DD, D, D, D] \\ \overline{\ell} &= [TD, TD, DD, DD, DD, D, D] \\ c &= 1 \end{aligned}$$

3. Fonction logarithme, grande précision :

$$\begin{aligned} f(x) &= \log_2(1+x) \\ dom &= [-1/4; 1/4] \\ \overline{\varepsilon} &= 2^{-120} \\ d &= 40 \\ \ell &= [TD, TD, DD \times 12, D, \ldots, D] \\ \mu &= 1.4 \\ \underline{\ell} &= [TD, TD, DD \times 13, D, \ldots, D] \\ \overline{\ell} &= [TD \times 12, DD, \ldots, DD] \\ c &= 2 \end{aligned}$$

Il convient de remarquer également que le nombre moyen de flottants double par coefficient nécessaire pour atteindre des précisions cibles jusqu'à 120 bits pour des fonctions usuelles ne dépasse en pratique pas 1.7 avec l'approche présentée ici. La figure 6.5 donnée à la section 6.5.2 indiquera plus de détails.

6.4 Vers une gestion automatisée de la réduction d'argument

Les algorithmes et techniques proposés pour un générateur automatique d'implantations de fonction mathématiques ne sont adaptées vraiment qu'à des domaines $dom = [a; b]$ de diamètre petit, typiquement $b - a < 1$. En plus, pour que l'adaptation des précision des étapes d'une évaluation de Horner se fasse correctement (cf. section 6.2.2), le domaine est censé contenir zéro, $0 \in dom$.

Pour une automatisation complète de l'implantation de fonctions mathématique dans des domaines quelconques, il faut alors résoudre le problème de générer une séquence d'opérations pour un réduction d'argument. Cette séquence calcule une fonction définie par une expression simple projetant les arguments du code dans un grand intervalle dom sur un domaine $\widetilde{dom}$ de diamètre petit.

Pour la recherche d'une réduction d'argument appropriée, des propriétés algébrique par rapport à l'arithmétique flottante sous-jacente doivent être analysées. L'espace de recherche est très grand pour certaines fonctions. Pour d'autres, surtout des fonctions composées, aucune réduction d'argument, utilisant éventuellement des tables, n'est envisageable.

Il dépasserait alors le cadre de cette thèse que de vouloir résoudre le problème de la génération automatique de codes de réduction d'argument. Avec l'expérience de l'implantation manuelle de fonctions mathématiques, il me semble que ce sujet peut même remplir une thèse complète.

Dans la suite, nous allons alors considérer juste une réduction d'argument par décalage du domaine et de la fonction : $\widetilde{f}(x) = f(x + t)$ et $\widetilde{dom} = [a - t; b - t]$. Cette réduction simple nous permet ensuite de traiter l'implantation automatique de fonctions définies sur un domaine dom ne contenant pas zéro. Le domaine devra juste être de diamètre inférieur à 1.

De première vue, générer un tel décalage n'est pas difficile. Tout simplement, on pourrait prendre comme valeur de décalage t le milieu $t = \frac{a+b}{2}$ de l'intervalle $dom = [a; b]$. De faire ainsi peut conduire à des implantations sous-optimales pour deux raisons :

1. La fonction f donnée pourrait avoir une symétrie autour d'un point $t' \in dom$. Si le décalage ne se fait donc pas par t', les optimisations apportées à l'approximation par des polynômes pair ou impairs et à l'évaluation de tels polynômes avec un précalcul des puissances nécessaires ne servent à rien. On voudra donc détecter automatiquement de telles symétries. La prochaine section 6.4.1 abordera cette question.

2. La séquence d'instructions flottantes qui implante la réduction d'argument consiste en une soustraction de la valeur de décalage t à l'argument x du code final pour la fonction f. Cette valeur de décalage t doit donc être stockée sur un flottant que la séquence charge sur registre. Pour qu'aucun arrondi n'ait lieu et la réduction soit exacte, elle doit donc être un flottant. Pour que le chargement soit rapide, des valeurs multi-double sont à exclure.

   En revanche, quand il est assuré que t est flottant et qu'alors la séquence de réduction d'argument est exacte, la valeur précise de t n'importe que très peu. La valeur du décalage peut facilement varier de quelques pourcents autour du milieu t' du domaine dom. Le domaine réduit sera légèrement asymétrique ce qui ne pose pas de problème à l'approximation polynomiale. Ce qui est clair, c'est que la valeur du décalage t' a une influence sur les coefficients du polynôme d'approximation. Pour certaines valeurs t, ceux-ci s'arrondiront à des valeurs flottantes générant une erreur moindre que pour d'autres valeurs de t. Il y a donc ici un moyen de réduire davantage les tailles multi-

doubles des coefficients des polynômes, optimisées déjà par le processus décrit à la section 6.3.3. Nous proposons donc une technique d'optimisation des décalages à la section 6.4.2.

6.4.1 L'utilisation de symétries

Pour pouvoir profiter des optimisations faites pour les fonctions symétriques, on cherche alors un algorithme qui, étant donné une fonction f et un domaine $dom = [a; b]$ puisse calculer un point $t \in dom$ tel que

$$\forall x \in dom, f(x+t) = f(-x+t) \quad \text{pour une symétrie axiale autour de } t$$

ou bien

$$\forall x \in dom, f(x+t) = -f(-x+t) \quad \text{pour une symétrie centrale en } f(t)$$

ou, s'il n'existe pas de tel t, l'indique.

De premier abord, il peut sembler que cet algorithme doit établir une certaine propriété algébrique de la fonction f et qu'il doit donc faire appel à des techniques de calcul formel. Ceci interdirait également de considérer des fonctions f définies par des boîtes noires. Mais comme l'information de symétrie ne nous sert qu'à mieux exploiter des optimisations pour des polynômes d'approximations pairs ou impairs, une certaine erreur peut entacher la valeur calculée t. L'algorithme de calcul d'un polynôme approximateur, en l'occurrence l'algorithme de Remez, pourra compenser cette inexactitude. Il convient alors de considérer l'erreur relative de la fonction f par rapport au symétrique d'elle-même :

$$\varepsilon_S(x) = \frac{f(x+t)}{f(-x+1)} - 1 \quad \text{pour une symétrie axiale}$$

ou bien

$$\varepsilon_S(x) = \frac{f(x+t)}{-f(-x+1)} - 1 \quad \text{pour une symétrie centrale.}$$

L'algorithme peut donc exhiber un point t qu'il soupçonne d'être un point de symétrie. Ensuite, le test si c'en est un est simple : il suffit de calculer la norme infini $\|\varepsilon_S(x)\|_\infty^{[a-t;b-t]}$ et de comparer sa valeur à une constante $\overline{\varepsilon}_S$.

Cette constante ε_S peut être l'erreur cible $\overline{\varepsilon}$ donnée en argument au générateur automatique d'implantations. Alors, la quasi-symétrie de la fonction est constatée avec tant de certitude – c'est-à-dire une erreur si petite – qu'elle peut être prise comme une vraie symétrie lors de l'algorithme. Par exemple, le domaine $[a-t; b-t]$ peut n'être considéré que sur sa partie positive pour approcher $f(x+t)$ ou $\operatorname{sgn}(x+t) \cdot f(x+t)$; il serait même possible de faire un deuxième décalage pour le centrer en 0. Par contre, comme nos algorithmes d'approximation polynomiale et de génération de séquences d'évaluation gèrent bien les quasi-symétries tout en compensant leurs erreurs ε_S, il est aussi possible de fixer $\overline{\varepsilon}_S$ à une valeur plus grande que l'erreur cible $\overline{\varepsilon}$. Des investigations plus poussées éclaireront ce point à l'avenir.

On peut donc tester si t ou $f(t)$ sont un axe ou un centre de symétrie. La question de trouver un tel t par calcul se pose toujours. Il s'avère qu'une méthode probabiliste s'applique. Si la fonction f est effectivement symétrique autour de t, toute valeur $\xi \in dom$ (sauf $\xi = t$) a une valeur symétrique $2t - \xi$ différente de ξ. Il est alors possible de choisir une valeur

$\xi \in dom$ quelconque, par exemple $\xi = \frac{3a+b}{4}$, et de chercher une valeur $t \in dom$, $t \neq \xi$, pour laquelle $f(\xi)$ a un symétrique. La probabilité que l'axe de symétrie soit le ξ choisi est nulle. Au cas où il suffit d'itérer. On définit donc

$$g(t) = f(\xi) - f(2t - \xi) \quad \text{pour une symétrie axiale}$$

ou

$$g(t) = f(\xi) + f(2t - \xi) \quad \text{pour une symétrie centrale}$$

et on cherche les zéros $t_i \neq \xi$ de $g(t)$. Pour ne jamais évaluer la fonction f à l'extérieur du domaine dom donné, où elle pourrait ne pas être définie, on restreint cette recherche à l'intervalle $\left[\frac{a+\xi}{2}; \frac{b+\xi}{2}\right]$. Cette restriction n'a pas d'influence en pratique ; de toute façon, il est mieux de ne pas décentrer trop le domaine réduit, donc de garder t au milieu de dom. La figure 6.3 illustre la recherche d'une symétrie axiale.

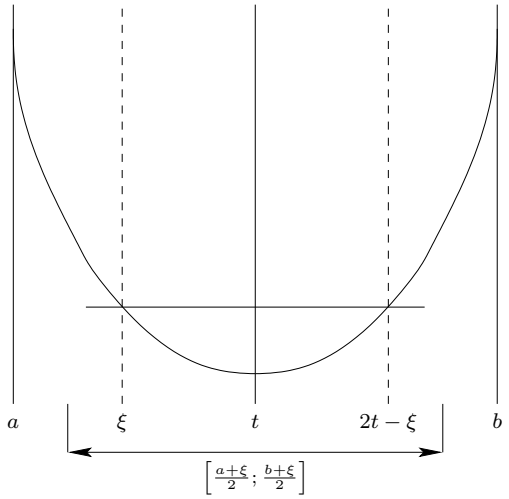


FIG. 6.3 – Recherche d'un axe de symétrie

La recherche de zéros sur la fonction g donne donc une liste d'axes de symétrie t_i possibles. Clairement, il existe des fonctions f qui ne sont pas symétrique mais qui engendrent une liste non vide de tels zéros. Il faut donc procéder un test de symétrie pour toutes les valeurs t_i trouvées.

En pratique, les résultats de cet algorithme, facilement implanté en Sollya, sont convaincants. Le tableau suivant donne les résultats sur exemples. Il indique la fonction, le domaine dom, son axe de symétrie t', le point de décalage t retenu et les degrés de la base de monômes choisie pour satisfaire une erreur cible $\overline{\varepsilon}$.

Fonction	dom	axe t'	axe trouvé t	$\overline{\varepsilon}$	monômes
$\sin(\pi \cdot (x-4))$	$[3.99; 4.01]$	4	4	2^{-60}	$1, 3, 5, 7$
$e^{(x-1/3)^2}$	$[0.33; 0.34]$	1/3	$6004799503160661 \cdot 2^{-54}$	2^{-60}	$0, 2, 4, 6$
$\cos(\pi x) + 10$	$[4.93; 5.12]$	5	5	2^{-54}	$0, 2, \ldots, 10$
$\mathrm{erf}^2(x - \mathrm{e}) + 2$	$[2.7; 2.8]$	e	$6121026514868073 \cdot 2^{-51}$	2^{-57}	$0, 2, \ldots, 10, 11$
$\cos x + \frac{1}{1000}\sin 2x + 10$	$[15.70; 15.72]$	-	$4420835649453769 \cdot 2^{-48}$	2^{-32}	$0, 2$

Il est intéressant de remarquer que l'algorithme réussit même de trouver un axe de symétrie t pour la fonction $\cos x + \frac{1}{1000}\sin 2x + 10$. Cette fonction n'est pas symétrique dans le sens mathématique du terme. Elle apparaît pourtant symétrique sous la résolution des 32 bits de précision demandés – c'est cette pseudo-symétrie que détecte l'algorithme.

Remarquons que la technique de détection de symétries proposé, passant par un calcul probabiliste d'un axe et un test numérique à base de normes infini, peut s'étendre à d'autres problèmes de réduction d'argument. La périodicité d'une fonction peut être testée ainsi. Les réductions d'argument additives et multiplicatives $f(a+b) = f(a) \cdot f(b)$ ou $f(a \cdot b) = f(a) + f(b)$ demandent d'être soigneux mais peuvent être détectés également avec cette technique. Il est clair que la recherche de zéros doit alors se faire avec une précision garantie et assez élevée.

6.4.2 L'optimisation des décalages

Dans le cas où, pour une fonction f, aucune symétrie ou périodicité et aucun zéro ne peut être détecté dans un domaine $dom = [a;b] \not\ni 0$, un décalage quelconque $t \in dom$ peut être fait. Le domaine réduit sera $\widetilde{dom} = [a-t; b-t]$; il n'est pas trop asymétrique en pratique si t reste proche du milieu de dom. Évidemment, la fonction sur le domaine réduit s'écrit $\widetilde{f}(x) = f(x+t)$. Remarquons que la valeur de t doit s'écrire sur un flottant pour avoir une réduction de domaine exacte. En pratique, les valeurs de décalage seront stockés sur des flottants double précision.

Il est facile de voir que si

$$p(x) = \sum_{i=0}^{n} p_i \cdot x^i$$

est un polynôme d'approximation de f sur dom à une erreur maximale

$$\widehat{\varepsilon} = \left\| \frac{p}{f} - 1 \right\|_{\infty}^{dom},$$

alors

$$\widetilde{p}(x) = p(x+t) = \sum_{i=0}^{n} \widetilde{p}_i(t) \cdot x^i$$

est un polynôme d'approximation de $\widetilde{f}$ sur $\widetilde{dom}$ ayant la même erreur maximale $\widehat{\varepsilon}$. Les coefficients $\widetilde{p}_i$ du polynôme $\widetilde{p}$ sont des polynômes dans le décalage t :

$$p_i(t) = \sum_{s=0}^{n-i} \binom{i+s}{s} p_{i+s} \cdot t^s$$

Comme décrit à la section 6.3.3, les coefficients du polynôme d'approximation à coefficients réels doivent être mis sur des flottants multi-double. L'algorithme proposé énumère et teste toutes les combinaisons de coefficients sur des flottants double, double-double et triple-double entre une borne inférieure $\underline{\ell}$ et une borne supérieure $\overline{\ell}$. La borne inférieure est calculée telle qu'il n'y a aucun polynôme à coefficients flottants avec des précisions plus petites que la borne qui satisfait une erreur cible donnée. Puisque notre cadre limite le choix des précisions disponibles aux formats multi-double, de grandes erreurs de surestimation

de la précision peuvent être observées. Par exemple, pour un coefficient $\widetilde{p}_i$ donné, le processus de calcul des bornes inférieures indique par exemple que le coefficient doit au moins être mis sur un flottant à 54 bits. Il est alors impossible de trouver un polynôme, satisfaisant l'erreur cible, où ce coefficient $\widetilde{p}_i$ est stocké sur un flottant double précision à 53 bits. Un double-double est mis ce qui provoque une surestimation de $107 - 54 = 53$ bits. Dans le cas général, rien ne peut être fait contre cette source de non-optimalité due à la discrétisation des précisions en formats multi-double.

Or, dans le cadre où un décalage du domaine de définition de la fonction est nécessaire, l'optimisation suivante peut être faite pour le coefficient $\widetilde{p}_i(t)$. Un algorithme de recherche exhaustive peut chercher une valeur de décalage flottante $t \in \mathbb{F}_k$ telle que l'arrondi de $\widetilde{p}_i(t)$ se fasse avec une erreur $|\circ_{k'}(\widetilde{p}_i(t)) - \widetilde{p}_i(t)|$ beaucoup moindre que la borne d'erreur d'arrondi pour le format. En d'autres mots, l'algorithme de recherche une valeur t flottante à précision k telle que $p_i(t)$ soit proche d'un flottant à précision k'. En effet, une valeur $\widetilde{p}_i(t)$ très proche d'un flottant à k' bits s'écrit avec un certain nombre de zéros (ou uns) après le k'-ième bit :

$$\begin{array}{rcllll} \widetilde{p}_i(t) & = & 1.0\ldots1 & 0000000 & 1\ldots \\ \circ_{k'}\left(\widetilde{p}_i(t)\right) & = & 1.0\ldots1 & & \\ \circ_{k'+\kappa}\left(\widetilde{p}_i(t)\right) & = & 1.0\ldots1 & 0000 & \\ \delta & = & & & 1\ldots \end{array}$$

Les arrondis de la valeur réelle de $\widetilde{p}_i(t)$ en un format à k' bits ou un format légèrement plus grand à $k' + \kappa$ bits provoquent donc la même erreur. Un polynôme ainsi décalé peut donc avoir une représentation multi-double plus courte et optimisée. La recherche du t approprié peut se faire en énumérant tous les flottants consécutifs $t, t^+, t^{++}, t^{+++}\ldots$ et en testant

$$|\circ_{k'}\left(\widetilde{p}_i(t)\right) - \widetilde{p}_i(t)| \leq \overline{\varepsilon}_G$$

où $\overline{\varepsilon}_G$ est l'erreur cible à obtenir sur le coefficient $\widetilde{p}_i(t)$.

À ce stade, remarquons que l'approche de chercher un antécédent flottant t d'une fonction tel que l'image soit proche d'un flottant à k' bits et, ainsi, fournisse plus de précision que k' bits n'est pas neuve. Pour ce qui est des réduction d'argument par tables, elle est connue sur le nom de la méthode de Gal [52, 117]. À notre connaissance, elle n'a pourtant jamais été appliquée à l'optimisation du bon arrondi des polynômes d'approximation. L'algorithme de calcul des antécédents flottants proposé par Stehlé et Zimmermann [117] s'applique à notre cas au moins tant que seul un coefficient $\widetilde{p}_i(t)$ doit être optimisé. Vu qu'il s'agit dans notre cas non de fonctions transcendantes mais de polynômes $\widetilde{p}_i$, leur technique peut éventuellement encore être améliorée en termes de complexité.

En effet, en pratique, on n'observe une surestimation de la précision par des formats multi-double pas seulement sur un seul coefficient $\widetilde{p}_i(t)$. La discrétisation des formats peut provoquer une sous-optimalité sur un ensemble de coefficients $\widetilde{p}_{i_1}(t)$ à $\widetilde{p}_{i_n}(t)$. L'algorithme doit donc trouver une valeur t que simultanément tous les $\widetilde{p}_{i_r}(t)$ s'arrondissent avec une erreur très petite. Comme de telles valeurs sont rares [52], l'optimisation doit se restreindre à un nombre n de coefficients simultanément considérés petit.

Cet algorithme d'optimisation des décalages pour un meilleur arrondi des coefficients a été implanté en Sollya et intégré au générateur automatique d'implantations de fonctions mathématique. De bons résultats, rendant superflues les optimisations sur les coefficients flottants des polynômes approximateurs à base de l'algorithme LLL [17] (cf. section 6.3.3),

ont été obtenus pour l'implantation de la fonction asin dans la bibliothèque CRLibm. Donner de plus amples détails sur l'efficacité de la méthode dépasserait le cadre de ce document.

6.5 Intégration logicielle du générateur automatique et résultats

6.5.1 Intégration logicielle

Dans les sections qui précèdent, les différentes briques de base pour la construction d'un générateur automatique de codes évaluateurs de fonctions mathématiques ont été présentées. La combinaison de ces algorithmes de base donne l'algorithme 13 complet ci-dessous.

Entrées : une fonction $f : \mathbb{R} \to \mathbb{R}$, un domaine $dom = [a, b]\,, b - a < 1$, et une erreur cible $\overline{\varepsilon} \in \mathbb{R}^+$.
Sorties : un code F, écrit en langage C, pour évaluer f tel que

$$\forall x \in dom, \left| \frac{F(x) - f(x)}{f(x)} \right| \leq \overline{\varepsilon}$$

et une preuve Gappa pour la borne d'erreur d'arrondi.

Algorithme `implanteFonction` (f,dom,n) :
début
  si $0 \notin dom$ **alors**
    chercher un point de symétrie $t \in dom$; /* cf. section 6.4.1 */
    optimiser le décalage $f(x) \leftarrow f(x - t)$ autour de t ; /* cf. section 6.4.2 */
  fin
  calculer un polynôme $p^* \in \mathbb{R}_n[x]$ non-cancellant pour $1/3 \cdot \overline{\varepsilon}$; /* cf. section 6.3.2 */
  passer le polynôme à des coefficients flottants, $p \in \mathbb{F}[x]$; /* cf. section 6.3.3 */
  implanter le polynôme p avec une erreur cible $1/3 \cdot \overline{\varepsilon}$; /* cf. section 6.2 */
  exécuter la preuve Gappa sur l'outil Gappa
fin

Algorithme 13 : L'algorithme `implanteFonction`

Cet algorithme a été implanté en pratique, principalement à l'aide de l'outil Sollya. Certains algorithmes de base, comme le générateur de séquences de d'évaluation de polynômes, la norme infini nécessaire à beaucoup de calculs et l'algorithme de Remez sont directement intégrés à l'outil. D'autres algorithmes, comme la détermination de polynômes à évitement de cancellations, sont implantés en langage Sollya. Seuls les algorithmes pour l'optimisation de polynômes pour l'arithmétique multi-double restent extérieurs à l'outil Sollya. Leur interfaçage se fait soit par un chargement d'un module binaire externe en Sollya soit par la génération de fichiers de transfert et l'appel d'un exécutable externe par une commande Sollya. Le tableau suivant donne plus de détails :

Algorithme de base	Technique d'implantation
évaluation de fonction composées	langage C à l'intérieur de Sollya
norme infini	langage C à l'intérieur de Sollya
`guessdegree`	langage C à l'intérieur de Sollya
algorithme de Remez	langage C à l'intérieur de Sollya
détection de symétries	langage Sollya
optimisations de décalages de fonction	langage Sollya avec une commande spéciale écrite en C intégrée à Sollya
recherche de polynômes à évitement de cancellations	langage Sollya
calcul de minorations de précisions pour coefficients flottants d'un polynôme	langage C++ à travers un module binaire dynamiquement chargé
approximation polynomiale à coefficients flottants	langage Pari/GP, interfacé par des fichiers de transfert
Algorithme de base	Technique d'implantation
optimisation de polynômes à coefficients multi-double	langage Sollya
générateur de séquence d'évaluation de polynômes	langage C à l'intérieur de Sollya
algorithme de haut niveau dirigeant les différentes briques de base	langage Sollya

Actuellement, cette implantation d'un générateur automatique d'implantations de fonctions dans un petit domaine consiste en plus que 6200 lignes C pour le générateur de séquences d'évaluations, 1200 lignes Sollya, 420 lignes Pari/GP[10] et quelques dizaines de lignes de glu d'interfaçage en script shell.

6.5.2 Résultats de l'étude sur l'automatisation

Le générateur automatique d'implantation de fonctions a été conçu dans le but pratique de faciliter le développement de fonctions mathématiques dans CRLibm. Comme son efficacité et la qualité du code produit s'avèrent hautes, il permet également de répondre à certaines questions d'optimisations nécessitant le parcours d'un espace de recherche relativement grand. Les deux sections suivantes essaient d'illustrer ces deux points.

Quelques fonctions dans CRLibm

Avec la bibliothèque mathématique correctement arrondie CRLibm, nous proposons actuellement des implantations pour 20 fonctions en double précision. Parmi toutes, 14 ont été développées durant cette thèse. Presque la moitié des implantations, c'est-à-dire 9 fonctions, contiennent des codes automatiquement générés avec les algorithmes présentés ici. Seuls les codes de réduction d'argument y ont été écrits à la main.

[10] dues à S. Chevillard

Au vu du grand nombre de codes automatiquement générés dans CRLibm, la question de la qualité de ces codes se pose. La qualité de la bibliothèque ne serait-elle pas encore meilleure si on était resté à un mode de développement manuel ? Pour ce qui est la qualité d'une bibliothèque par rapport au nombre de fonctions supportées, la réponse est non : avec un coût de 2 à 3 mois de développement manuel pour les premières fonctions dans CRLibm, il aurait été difficile de proposer autant de fonctions maintenant. Quant à la qualité de la bibliothèque en termes de performances de ses fonctions, il convient de considérer le tableau suivant. Il donne le temps d'évaluation de fonctions de CRLibm en moyenne et au pire cas. Les temps sont indiqués relatifs à ceux des fonctions correctement arrondies dans la bibliothèque libm proposé par Debian Linux 2.6.22 pour AMD64. Cette bibliothèque consiste principalement en codes développés par Ziv pour IBM [130]. En prenant les temps de cette libm comme étalon, il est ensuite possible de comparer la qualité des codes manuellement développés à celle des codes automatique. Ce choix d'étalon est bien sûr discutable. Il faudrait bien sûr comparer toute une libm manuelle à un libm automatique. Malheureusement, le coût de développement que cette comparaison implique ne reste pas en relation avec son intérêt. Les mesures ont été effectuées sur un AMD Bi-Opteron 244 cadencé à 1.8 GHz sur un système 64 bit ; CRLibm a été compilé avec gcc 4.2.2-3.

Fonction	développement automatique/ manuel	rapport à la libm en moyenne	rapport à la libm au pire cas de temps de mesure
`pow`	automatique	82 %	1 %
`log`	automatique	92 %	1 %
`asin`	automatique	220 %	1 %
`acos`	automatique	236 %	1 %
`exp`	manuel	123 %	1 %
`sin`	manuel	124 %	1 %
`cos`	manuel	118 %	1 %
`tan`	manuel	119 %	1 %
`atan`	manuel	186 %	5 %

On remarque donc que les résultats sont mitigés. Les implantations CRLibm manuellement développées restent toujours à peu près 20 % plus lent que leur correspondant dans la libm par défaut. Cette différence s'explique principalement par leur conception moins gourmande en mémoire. En revanche, les implantations intégrant des approximations et évaluations polynomiales automatiquement générées sont soit jusqu'à 20 % plus rapides que celles de la libm soit jusqu'à 2.4 fois plus lents. Remarquons que l'impact mémoire des deux implantations, asin et acos de la libm est jusqu'à 3 fois plus grand que celui de CRLibm.

Résultats du parcours d'un grand espace de recherche

L'analyse précédente montre aussi qu'il est très difficile de trouver un bon compromis entre temps d'approximation polynomiale, temps de réduction d'argument, mémoire et

d'autres aspects. On peut donc vouloir parcourir un grand espaces d'implantations réelles d'approximations et évaluations polynomiales.

Avec le générateur automatique présenté, de tels recherches pratiques sont possibles. Nous avons implanté quelques fonctions, notamment e^x et $\log(1+x)$, pour des arguments réduits x dans des domaines $dom = [-a; a]$ de tailles $2a$ différentes. Toutes les tailles $2a = 2^k$ pour $k \in [\![-14; -2]\!]$ ont été considérées. Nous avons également fait varier la précision cible $\overline{\varepsilon}$ des implantations entre 20 et 120 bits par pas de 1 bit. Pour chaque configuration, le générateur a produit un code d'approximation et d'évaluation polynomiale qui a été compilé et chronométré. Une preuve formelle en langage Gappa a été produite et vérifiée pour la borne d'erreur d'arrondi de toutes ces quelque 1200 implantations par fonction.

Par fonction, le parcours de l'espace de recherche décrit a été accompli en à peu près 18 heures de calcul sur un AMD Bi-Optéron 244 cadencé à 1.8 GHz avec Linux 2.6.22 AMD64. La version de l'outil Sollya a été la version 650 du SVN Sollya, compilée par gcc 4.2.2-3 avec MPFR version 2.3.1 et MPFI version 1.3.4. La vérification s'est faite avec Gappa version CVS du 28 septembre 2007. Les temps de calcul des implantations sont données en unités de mesures arbitraires. Ces mesures ont été faites sur le même système que le système de génération du code. Nous donnons tous ces détails car on observe des différences non-négligeables de performances de Sollya sur les systèmes 32 bits et 64 bits.

Dans la dépouille des données générées, intéressons-nous d'abord à la dépendance du temps d'évaluation de la fonction $\log(1+x)$ des paramètres taille de l'intervalle et précision cible. Figure 6.4 trace la courbe de cette dépendance. L'erreur cible est indiquée sous forme de précision cible en bits ce qui est équivalent à une échelle logarithmique. La taille de l'intervalle est aussi reportée sur une échelle logarithmique. Notons que ce choix est discutable car l'impact mémoire des réductions d'argument classiques pour ces fonctions est linéaire dans la taille du domaine réduit [122, 43].

On observe donc une augmentation très importante du temps d'évaluation de la fonction entre le cas le plus simple, $\overline{\varepsilon} = 2^{-20}$, $dom = \left[-2^{-15}; 2^{-15}\right]$, et le cas le plus difficile, $\overline{\varepsilon} = 2^{-120}$, $dom = \left[-2^{-3}; 2^{-3}\right]$: le deuxième cas est 65 fois plus cher. L'augmentation du temps de calcul dépend plus de la précision cible que de la largeur (logarithmique) du domaine. La figure 6.4 montre également très bien les paliers entre la double précision et la double-double ainsi qu'entre la double-double et la triple-double. Remarquons aussi que dans la partie concernant la double précision, la courbe est monotone croissante. En revanche, dans les partie de la double-double ou triple-double, cette monotonie n'est plus parfaite à cause des différents effets de discrétisation. La dépendance du temps d'évaluation de l'approximation polynomiale de la fonction e^x des paramètres choisis ressemble à celle de la fonction $\log(1+)$. Nous n'en donnons pas donc la courbe.

Les courbes de temps d'évaluation ne font pas apparaître de minimum local. C'est en combinaison avec un chronométrage des réductions d'argument engendrant les diverses tailles de domaines que l'on peut trouver un minimum local. Ces analyses restent à faire, car notre générateur d'implantation actuel ne supporte pas encore les réduction d'argument par tables. Avec l'expérience acquise en la matière, je soupçonne tout de même qu'un minimum local ne peut être observé que si une pondération par l'impact mémoire est faite. En pratique, le temps d'exécution d'une réduction d'argument classique ne dépend quasiment pas de la taille du domaine réduit : le coût principal est dans le nombre de lectures dans les tables, qui est borné par 1 ou 2.

Les analyses du comportement des implantations en fonction de la taille du domaine et la précision cible ne s'arrêtent pas au temps d'évaluation. Pour évaluer le coût de l'utilisa-

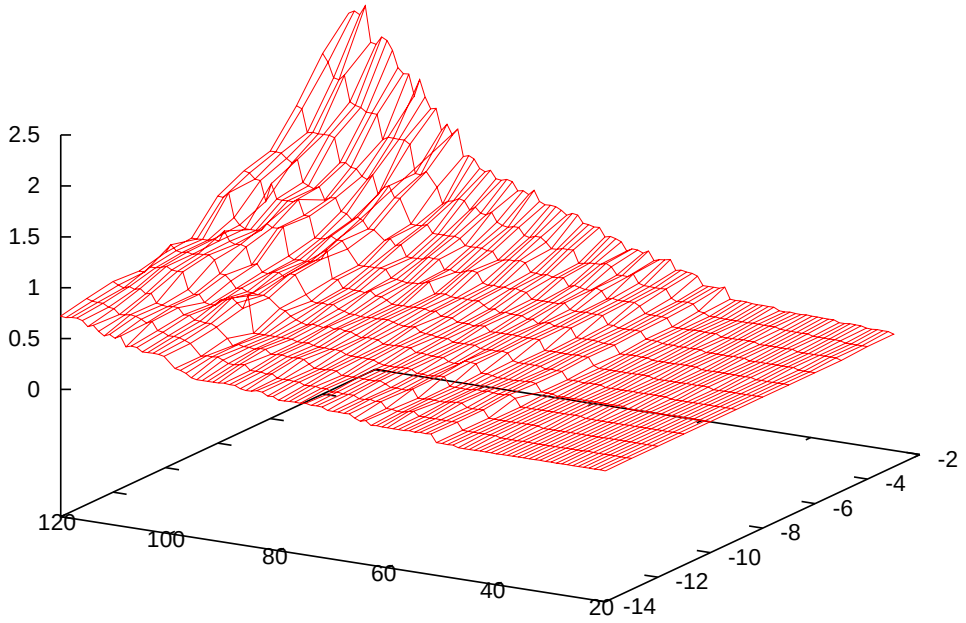


FIG. 6.4 – Temps d'évaluation mesuré – fonction $\log(1+x)$

tion de l'arithmétique multi-double, on peut par exemple s'intéresser au nombre de flottants double précision par coefficient des polynômes d'approximation utilisés en pratique. La figure 6.5 montre cette dépendance.

Le nombre moyen de flottants double précision devient donc – évidemment – plus grand quand la précision augmente. En revanche, il ne dépend presque pas de la taille du domaine d'implantation. Il est remarquable que ce nombre moyen reste borné par à peu près 1.7. Ceci semble peu puisqu'il s'agit tout de même de générer des approximation sur jusqu'à trois doubles. Le lecteur aura certainement remarqué les effets de discrétisation importants ayant lieu lors de l'optimisation des multi-doubles, décrite à la section 6.3.3 et illustrée par la figure 6.5.

La valeur de 1.7 flottants double précision par coefficient est d'autant plus remarquablement petite que le facteur entre le temps d'évaluation de l'implantation dans cas le plus simple et le temps de celle dans le cas le plus difficile est de 65. Il convient alors d'analyser les coûts respectifs moyens des étapes de Horner dans les diverses implantations. La figure 6.6 illustre ces coûts. Elle indique le temps d'évaluation d'une fonction divisé par le degré du polynôme utilisé sur une échelle logarithmique. On remarque clairement trois paliers pour la double, double-double et la triple-double. Comme une étape de Horner est à peu près 4 fois plus chère dans la partie de la triple-double qu'une étape en double précision, le facteur 65 s'explique alors facilement.

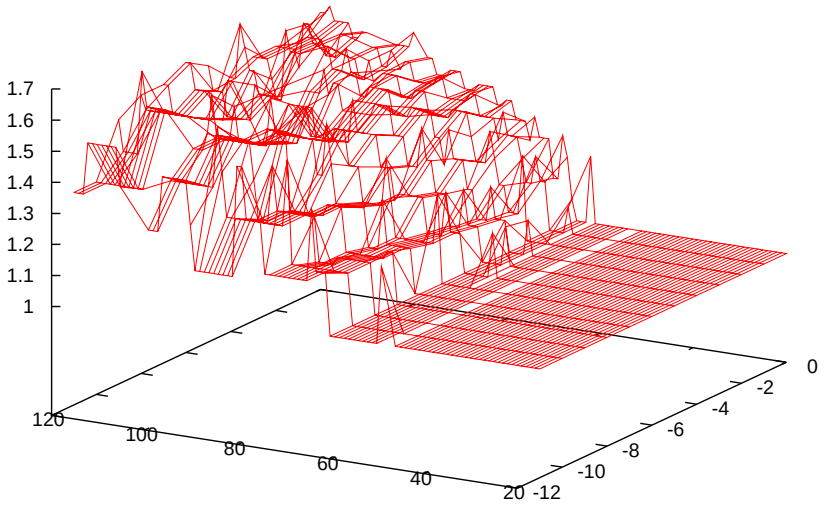


FIG. 6.5 – Nombre moyen de doubles par coefficient – fonction $\exp$

6.6 Quand les boucles deviennent des polynômes...

L'intérêt principale de la conception d'un générateur automatique d'implantations de fonctions mathématiques a été de simplifier le travail des spécialistes humains. Par l'élargissement de l'espace de recherche qui peut être parcouru à la recherche du code optimisé pour certaines caractéristiques des machines actuelles, la génération automatique permet également la production de code de meilleur qualité car plus performant. Tout en faisant appel à certaines techniques du calcul formel, l'algorithme d'implantation automatique présenté ici utilise principalement des approches du calcul numérique.

Prenons par exemple la fonction $\mathrm{argerf} = \mathrm{erf}^{-1}$, inverse de la fonction intégrale de la distribution d'erreur

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Or que la fonction $\mathrm{erf}(x)$ est souvent disponible dans des logiciels comme MPFR ou Sollya, la fonction argerf ne l'est pas. On pourrait bien sûr l'implanter en analysant sa série entière mais il y a aussi la possibilité de la calculer par une recherche zéro de la fonction $y \to \mathrm{erf}(y) - x$. Cette recherche peut se faire par une boucle d'itération de Newton. Évidemment, une telle implantation de la fonction argerf n'est ni intelligente ni efficace. Il s'agit d'un code brut à optimiser par transformation de code.

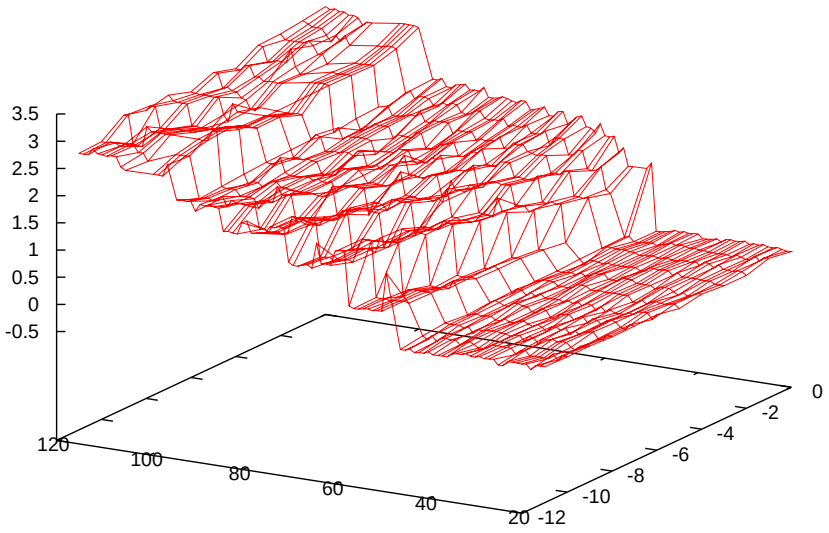


FIG. 6.6 – Temps moyen d'une étape de Horner – échelle logarithmique – fonction exp

Traditionnellement, la transformation optimisante de codes, flottants ou non, se base sur une analyse statique d'un code donné sous forme d'un source. L'algorithme d'optimisation passe d'abord par l'interprétation abstraite du programme donné [27]. Ensuite, une fois que des bornes pour les différentes valeurs dans un programme sont connues, certaines branches du programme inaccessibles peuvent éventuellement être éliminées, des recalculs d'expressions évitées ou des expressions constantes remplacées par leur valeur [2]. Ces optimisations sont locales et gardent exactement la sémantique du programme donné. Comme l'interprétation abstraite produit souvent des surestimations importantes des domaines des valeurs, en particulier quand il s'agit d'un programme flottant d'une taille non négligeable [94], ces optimisations peuvent rapidement atteindre leurs limites.

Pouvoir disposer de techniques d'optimisation exactes qui maintiennent la sémantique binaire d'un programme est très important. Comme l'arrondi correct [35], ces techniques permettent une compatibilité binaire de programmes même à travers des différents systèmes matériels.

Tout de même, les caractéristiques d'un programme numérique ne se limitent pas qu'à la sémantique au niveau binaire. Dans certains contextes, il est sensé de considérer deux programmes flottants comme égaux s'ils calculent le même résultat à une marge d'erreur numérique près. Il semble difficile d'adapter les techniques traditionnelles d'optimisation de programmes à une telle notion d'égalité.

Les algorithmes pour la génération automatique d'implantations de fonctions, présentés

ici, interagissent majoritairement avec la fonction à implanter comme s'il s'agissait d'une boîte noire. Seul un échantillonnage de la fonction et de ses deux premières dérivées a lieu. Il est alors possible de considérer la transformation optimisante suivante : d'abord le code à optimiser est compilé en une boîte noire adaptée. Ensuite un code optimisé est régénéré par le générateur proposé. Ce processus est effectué sur la prémisse d'une marge d'erreur qui devient l'erreur cible en entrée du générateur. La figure 6.7 illustre la technique.

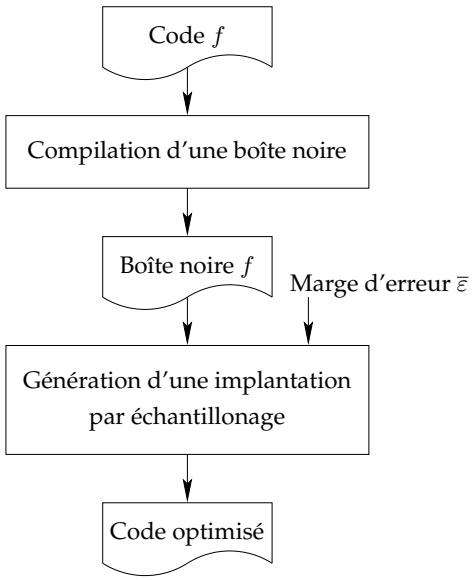


FIG. 6.7 – Transformation de codes numérique

Ce qui semble très prometteur avec cette technique de transformation de codes, c'est la complète opacité de la boîte noire par rapport à des constructions sémantiques dans le code de départ qui sont difficiles à appréhender dans l'analyse. Par exemple, le code de départ peut contenir une boucle intégrant une équation différentielle. Dans le code optimisé, cette boucle sera remplacée par une construction complètement différente, c'est-à-dire un polynôme. Si la fonction intégrée par l'équation différentielle était paire dans l'exemple, l'évaluation du polynôme se ferait même dans le carré x^2 de la variable d'entrée x de la fonction.

Bien sûr, l'approche présentée se trouve encore au stade de l'ébauche. Certaines problématiques doivent être résolues. D'une part, il manque un support en algorithmes d'approximation, de calcul de normes infini et de preuve pour les fonctions multivariées ; les techniques de génération automatique ne s'appliquent actuellement qu'à des fonctions univariées. Pourtant, beaucoup de fonctions bi- et multivariées se trouveront dans les codes numériques à optimiser. Ensuite, les fonctions en entrée du générateur automatique sont actuellement supposées continues et deux fois dérivables. Pour être complètement automatique, un transformateur numérique de codes aura besoin soit d'une analyse statique assurant cette propriété, soit de techniques d'approximation améliorées qui n'auront pas besoin de telles hypothèses. De la même façon, le problème du calcul de la première et seconde dérivée de la fonction du code à optimiser reste à résoudre. Il est bien sûr possible d'y appliquer la technique des différences divisées (cf. section 6.3.1) en pratique. Cela implique en revanche de pouvoir disposer d'un code très précis en entrée. Typiquement, avec cette différentiation numérique, on ne s'attend qu'à une qualité de précision simple pour la se-

conde dérivée d'une fonction si la fonction elle-même est calculée en précision quadruple. La différentiation automatique [56] pourrait être une autre solution. Finalement, le générateur automatique est actuellement limité à des approximations polynomiales sans réductions d'argument par tabulation. Par cela, la limitation affecte également la largeur du domaine de définition. Il se peut alors qu'actuellement, le code produit pour de larges domaines où la fonction du code initial varie beaucoup, ne soit pas meilleur que le code d'entrée. Seules des recherches futures plus approfondies cernant la génération automatique de réductions d'argument peuvent y apporter des réponses.

Certaines expérimentations autour de cette technique de transformation de code ont déjà été faites. Par exemple, il a été très facile d'implanter la fonction $\mathrm{argerf} = \mathrm{erf}^{-1}$ comme déjà) mentionné par une itération de Newton sur erf. Ce code brut, extrêmement inefficace d'ailleurs, a ensuite été lié dynamiquement à Sollya pour servir de fonction d'entrée au générateur d'implantation. Ce générateur a produit des codes simples, basés sur des polynômes de degré petit. Il a bien détecté la présence d'une fonction impaire et généré un code avec précalcul de x^2. Il est clair que d'autres expérimentations restent à faire à l'avenir.

CHAPITRE 7

Conclusion

On ne peut tirer de conclusion que de ce que l'on ne comprend pas.

Peter Høeg, *Smilla et l'amour de la neige*

Dans cette thèse, nous avons cherché à élargir le domaine accessible à l'arrondi correct de fonctions mathématiques. Nous nous sommes dirigés d'un travail sur des fonctions bivariées comme x^n et x^y vers une approche d'implantation automatique qui ne s'applique pas seulement à l'arrondi correct mais au domaine plus large des fonctions mathématiques en virgule flottante. Souvent, une question en cachait une autre, ce qui donnait d'une part une suite logique à nos travaux et d'autre part nous laisse encore beaucoup de choses à découvrir à l'avenir. Venons donc aux contributions et perspectives de cette thèse.

Faire des choses plus compliquées – vers l'arrondi correct des fonctions puissance

Après avoir analysé de différentes techniques pour l'arrondi correct efficace des fonctions univariées comme la fonction $\log$ [43], nous nous sommes particulièrement intéressés aux fonctions puissance x^n et x^y.

La fonction de puissance entière x^n peut être vue comme une famille de fonctions univariées. Une recherche de pire cas est donc possible pour des petites valeurs de n. Nous avons donc proposé des approches pour l'arrondi correct de cette fonction x^n [74]. Cette étude, portée sur le fleuron des processeurs superscalaires adaptés au calcul numérique, l'Itanium, a montré en particulier que les techniques d'approximation polynomiale et de tabulation étaient la meilleure solution pour une performance en moyenne élevée. Les recherches de pire cas pour x^n sont encore en cours. Il conviendra donc de reconsidérer l'implantation de cette fonction en toute généralité dans un avenir au moyen terme.

Pour la fonction puissance bivariée en deux arguments flottants x^y, la recherche de pire cas est actuellement impossible. Une implantation correctement arrondie doit donc faire recours au processus itératif de Ziv. Pour garantir la terminaison de ce processus sur des instances aussi simples que 9^{17}, il est nécessaire de filtrer les arguments (x, y) pour lesquels x^y tombe juste sur un flottant ou un milieu de flottants consécutifs. Ce filtre était jusque là très gourmand en opérations de grande latence : boucles, extractions de racines carrées répétées etc. Nous avons proposé une technique qui réduit ce filtre à juste 9 comparaisons avec des constantes précalculées [83]. Notre approche est innovatrice car elle utilise une information

partielle sur les pires cas de la fonction x^y, non directement pour l'arrondi correct, mais pour la détection de cas de frontière d'arrondi. Le gain en vitesse correspondant est de 39% en moyenne et une chute du surcoût de la détection de 50% à 9%. La technique s'applique évidemment à toutes les autres fonctions pour lesquelles une information complète de pire cas est disponible. Pour les fonctions où juste une information partielle peut être calculée, il est nécessaire de déterminer un petit sous-ensemble d'arguments contenant tous les cas de frontière possible. Des recherches plus poussées montreront si cela est possible par exemple pour la fonction $\mathrm{atan2pi}(x, y) = {}^1\!/\!_\pi \cdot \mathrm{atan}\, {}^y\!/\!_x$, définie comme opération recommandée par la norme révisée IEEE 754-2008 [67].

Tous nos travaux d'implantation de fonctions correctement arrondies se basent sur les arithmétiques double-double et triple-double qui nous fournissent la précision nécessaire pour l'arrondi correct des fonctions usuelles. Nous avons implanté ces arithmétiques à l'intérieur de la bibliothèque CRLibm. Nous avons analysé son comportement concernant la performance, qui est à peu près 10 fois meilleure que celle des techniques précédentes [81]. Nous avons également introduit une approche d'adaptation de la précision au minimum qui est complètement sûre. Elle vient avec une base de théorèmes de précision et de chevauchements maximaux [81].

Être sûr de faire les choses bien – la certification des implantations

Les cas où l'arrondi d'une fonction transcendante est difficile sont rares. Une implantation relativement précise et assez soignée peut donc bien fournir l'arrondi correct (quasiment) toujours. Tout de même, avant que le tampon « IEEE 754-2008 compliant correctly rounded implementation » ne puisse être apposé, l'implantation doit être certifiée. Cela veut dire qu'une preuve (formelle si possible) doit établir le fait que l'erreur totale du code numérique par rapport à la fonction mathématique mise en œuvre est sûrement inférieure à la borne dictée par son pire cas. Établir une telle preuve est un vrai défi qui ne peut pas être résolu sans assistance de l'ordinateur. Nous avons essayé de le relever. L'erreur totale étant composée de deux sources d'erreurs différentes, nous avons poussé l'étude dans deux directions.

D'une part, nous avons présenté un algorithme sûr, basé sur l'arithmétique d'intervalle, pour un calcul certifié de normes infini [24]. Une majoration sûre de la norme infini de la fonction d'erreur d'approximation exprime une borne sur la deuxième source d'erreur dans une implantation de fonction mathématique. Notre algorithme résout quelques problèmes spécifiques intervenant lors de l'analyse d'erreurs relatives, définies par des fractions. En l'occurrence, l'algorithme est capable de prolonger la fonction d'erreur en un faux pôle (flottant) au lieu de calculer une majoration par l'infini. Évidemment, le prolongement ne se fait que s'il est mathématiquement valide et que l'algorithme arrive à prouver cette condition automatiquement. Notre algorithme essaie de trouver également une solution au problème de grande cancellation dans les fonctions d'erreurs, définies par la différence entre la fonction à implanter et le polynôme d'approximation, calculée exprès pour bien se rapprocher de la fonction. La solution proposée jusque là consiste en une utilisation du théorème de Taylor et de l'arithmétique multiprécision. Les résultats obtenus satisfont les besoins de la bibliothèque CRLibm bien que les temps de calcul soient encore très hauts. Nous avons commencé à pousser plus en détail cette étude pour trouver des algorithmes encore meilleurs ; les publications comme [70] laissent déjà apparaître de nouvelles possibilités. La différentiation automatique [56] semble également être un élément de réponse. Nous continuerons à

travailler sur ce sujet.

D'autre part, nous avons proposé une méthodologie presque automatique, qui standardise et simplifie l'exploitation de l'outil Gappa pour le calcul et la preuve de bornes sur la première source d'erreurs, l'erreur d'arrondi [41, 42]. Il s'avère qu'à travers l'outil Gappa qui est un moteur de preuve formelle de haut niveau, la preuve formelle d'une implantation devient aussi simple, voire plus simple, qu'une preuve sur papier bien soignée. En plus, nous avons pu automatiser la rédaction d'une preuve Gappa pour un code généré en même temps. Ainsi, on assiste à une vraie démocratisation des méthodes exactes et prouvées pour les implantations de fonctions mathématiques, seraient-elles correctement arrondies ou non. À l'avenir, il sera nécessaire d'élargir ces techniques automatiques à des domaines de calcul numérique plus larges.

Ne faire les choses qu'une seule fois – l'outil Sollya

Le concepteur d'une implantation d'une fonction mathématique doit accomplir une multitude de tâches : la dynamique de la fonction doit être analysée, un polynôme d'approximation doit être calculé, des valeurs de tables doivent être formatées pour une précision arithmétique donnée etc. Passer constamment d'un logiciel à un autre pour faire une tâche, puis une autre, ralentit le processus et le rend fastidieux et moins sûr. Connaissant les besoins lors du développement par la pratique, nous avons réalisé l'outil intégré Sollya[1] qui essaie d'y répondre. Sollya fournit un langage de script complet, basé sur une philosophie de calcul exact proche du calcul réel par arrondi fidèle garanti. Il donne à l'utilisateur la sécurité lors du développement qui nous manquait dans d'autres outils précédemment utilisés, comme Maple. Étant un logiciel libre, il libère également des bibliothèques comme CRLibm d'une dépendance propriétaire. Avec sa petite centaine de fonctionnalités supportées [25] basées sur quelque 75000 lignes de code, l'outil Sollya est déjà utilisé pour d'autres domaines que l'arrondi correct et par d'autres équipes de recherche.

Le développement de Sollya continue. Nous coopérons pour intégrer des techniques d'optimisation d'approximations polynomiales à coefficients flottants [17] à l'intérieur de l'outil et avons comme projet de fournir des algorithmes de différentiation automatique de termes d'expressions. Ce développement continu expose aussi quelques problématiques liées à l'outil, comme par exemple le fait que Sollya ne supporte, pour l'instant, que des fonctions univariées. L'avenir nous montrera comment elles pourront être résolues. Peut-être qu'une reconception de l'outil s'imposera.

L'outil Sollya nous a servi de base robuste pour un travail d'automatisation de l'implantation de fonctions mathématiques.

Faire faire les choses – l'automatisation de l'implantation de fonctions

L'implantation de fonctions mathématiques a été un travail manuel et fastidieux. L'expérience de la bibliothèque CRLibm montre qu'un spécialiste a besoin d'un à trois mois pour concevoir manuellement une implantation correctement arrondie. Avec à peu près 35 fonctions typiquement supportées dans une bibliothèque mathématique libm, cela entraîne un coût important de développement et de maintien. En plus, le travail étant fastidieux, les spécialistes humains peuvent rechigner à essayer plusieurs implantations dans un but d'optimisation.

[1] `http://sollya.gforge.inria.fr`

Nous nous sommes donc intéressés à l'automatisation du processus d'implantation d'une fonction mathématique, donnée préférentiellement juste sous forme de boîte noire. Avec cet objectif, nous nous sommes engagés dans la génération automatiques de séquences d'opération flottantes évaluatrices de polynômes. Nous avons analysé les différentes contributions d'erreur d'arrondi dans une évaluation de Horner dans le but d'adapter automatiquement la précision de calcul au strict minimum. Notre implantation, intégrée à l'outil Sollya, est capable de générer une séquence d'évaluation de Horner avec utilisation d'arithmétique multi-double pour des polynômes d'approximation typiques. Si le polynôme contient des coefficients à zéro, des puissances de la variable du polynôme sont automatiquement précalculées. La certification des séquences produites est assurée par une génération automatique de preuves Gappa associées.

Nous nous sommes ensuite intéressés à l'automatisation du processus de calcul d'un polynôme d'approximation respectant les contraintes de l'environnement flottant. Après avoir analysé différentes techniques, comme l'approximation selon Taylor et Tchebychev, nous avons proposé un algorithme numérique à base de l'algorithme de Remez pour générer des approximants optimisés [82]. L'algorithme assure à la fois que les polynômes produits ne provoquent pas de cancellations lors de l'évaluation sous schéma de Horner, et qu'ils ne contiennent pas de valeurs numériques très petites au lieu de vrais zéros mathématiques. La technique proposée, avec une analyse esquissée dans cette thèse, remet certaines recettes, données dans les livres de référence [93, 101], sur des fondations plus stables tout en les élargissant : on n'approche plus une fonction paire par un polynôme paire parce que c'est écrit dans un livre, mais on essaie de comprendre pourquoi et quand c'est bien de le faire.

Ces approches d'approximation polynomiale donnent déjà des résultats très satisfaisants qui atteignent la même qualité qu'une implantation conçue à la main. Elles sont pourtant basées sur un certain nombre d'heuristiques dont il reste à construire des fondations mathématiques plus solides.

L'approximation par polynômes et l'évaluation de ceux-ci sont les principales briques de base pour une implantation de fonction mathématique. La réduction d'argument, éventuellement à base de tables, en est une autre. Nous avons commencé à travailler sur son automatisation mais le but final est encore loin à l'horizon. Une étude plus approfondie s'impose à moyen terme, d'autant plus qu'elle ne paraît en aucun cas facile.

Avec notre mise en œuvre prototypée d'un générateur automatique d'implantations de fonctions, il est possible de parcourir rapidement de grands espaces de recherche afin d'optimiser une implantation finale. En l'occurrence, le temps nécessaire pour implanter une fonction dans un petit domaine a chuté d'un mois au cas manuel à quelques secondes ou minutes. Il nous a été ainsi possible de produire en une nuit 1200 implantations de fonctions comme $\log 1 + x$ ou asin, toutes (formellement) certifiées par Gappa, de les compiler et de mesurer leur performance afin de les comparer entre elles.

Ce qui reste à faire – des projets à l'horizon

À l'avenir, nous devons chercher à atteindre une intégration plus complète de ces techniques automatiques. On s'intéressera à la question de générer automatiquement de fond en comble une bibliothèque libm, peut-être même bien une CRLibm. De telles techniques de méta-programmation de libm rejoignent également tout à fait ce qui est en train de se développer dans d'autres domaines, en l'occurrence plus proches encore du matériel [127, 37, 39].

Ainsi, les travaux présentés ici s'insèrent bien dans des projets comme EVA-Flo[2] et rejoignent des projets comme FloPoCo[3]. Nous avons également l'intime conviction que nos techniques automatiques, vues sous l'angle de la réutilisation de code, intéresseront des partenaires industriels, qui doivent fournir, en plus de produits matériels, une multitude de codes logiciels de type libm pour différents systèmes et compilateurs.

Il reste également à étudier comment nos techniques de génération automatique de code peuvent s'intégrer dans des compilateurs optimisants. L'idée est de générer statiquement un code optimisé pour des fonctions composées au lieu de les composer dynamiquement au moment de l'exécution. Nos approches actuelles doivent connaître un domaine de définition pour les fonctions. Comment un tel domaine peut-il être estimé par une analyse sémantique du code ? Ou bien, y a-t-il d'autres techniques pour contourner ce problème ? Finalement, je suis d'avis que les réponses reposeront aussi sur la disposition des utilisateurs de passer à de nouveaux langages numériques qui auront une sémantique plus mathématique à un niveau qui fait abstraction des phénomènes flottants, cachés dans les compilateurs. C'est notre capacité de concevoir de tels nouveaux langages qui fera ou pas la réussite de ces techniques.

On peut croire que l'idée derrière l'arrondi correct et celle de la génération automatique de codes pour fonctions composées se contredisent. L'impression actuelle va bien sûr dans ce sens. L'arrondi correct assure la portabilité bit-à-bit parce que toute fonction de base fait le même arrondi. Un code généré à la compilation implantant une fonction composée fait un autre arrondi. Mais là, pour une fois, on peut tout à fait avoir le beurre et l'argent du beurre : il suffit de définir une sémantique d'arrondi correct sur toute la fonction composée et s'assurer de générer le code correctement arrondi. Au fait, Sollya a déjà une telle sémantique quand on lui demande d'évaluer une fonction mathématique composée. Ainsi tous les systèmes produiront le même affichage tout en calculant rapidement avec des fonctions composées optimisées. Bien sûr, la génération de codes correctement arrondis à la volée dans une compilation est un projet au long terme...

Avec nos travaux sur l'arrondi correct, nous avons essayé d'apporter la preuve que la science est à la hauteur de ce que représente la standardisation de fonctions mathématiques usuelles correctement arrondies dans une norme internationale. La norme IEEE 754-2008 a été votée avec l'arrondi correct des fonctions, définies comme des opérations recommandées. Après ce vote, le travail n'est pas fini, il commence : il s'agit maintenant de fournir un environnement qui implante cette norme révisée, non seulement avec des fonctions mathématiques correctement arrondies, mais aussi avec toutes ses autres nouveautés. On y cite l'arithmétique flottante décimale, le support généralisé bien que souvent en logiciel de l'opération FMA, le passage amorcé vers la précision binary128[4], les nouvelles opérations de réduction de sommes et de produits scalaires etc. Oh combien de pain sur la planche...

[2] `http://www.ens-lyon.fr/LIP/Arenaire/EVA-Flo/EVA-Flo.html`

[3] `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

[4] appelée *précision quad* précédemment

Bibliographie

[1] M. ABRAMOWITZ et I. A. STEGUN : *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, Inc., 9ème édition, 1972.

[2] A. V. AHO, M. S. LAM, R. SETHI et J. D. ULLMAN : *Compilers : Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, août 2006.

[3] G. ALEFELD et D. CLAUDIO : The basic properties of interval arithmetic, its software realizations and some applications. *Computers and Structures*, 67:3–8, 1998.

[4] J. ALEX : *Zur Entstehung des Computers - von Alfred Tarski zu Konrad Zuse*. VDI-Verlag, Düsseldorf, 2007.

[5] AMERICAN NATIONAL STANDARDS INSTITUTE AND INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS (ANSI/IEEE) : IEEE Standard for Binary Floating-point Arithmetic (IEEE754-1985), 1985.

[6] R. AUSBROOKS, S. BUSWELL, D. CARLISLE *et al.* : *Mathematical Markup Language (MathML) Version 2.0 (Second Edition)*, 2003. Disponible sur `http://www.w3.org/TR/2003/REC-MathML2-20031021/mathml-p.pdf`.

[7] M. BERZ et G. HOFFSTÄTTER : Computation and application of Taylor polynomials with interval remainder bounds. *Reliable Computing*, 4(1):83–97, 1998.

[8] S. BOLDO : *Preuves formelles en arithmétiques à virgule flottante*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, novembre 2004.

[9] S. BOLDO et M. DAUMAS : A mechanically validated technique for extending the available precision. *Dans Proceedings of the 35th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, 2001. IEEE Computer Society Press.

[10] S. BOLDO et M. DAUMAS : A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 37(1-4):45–60, 2004.

[11] S. BOLDO et G. MELQUIOND : Emulation of a FMA and correctly-rounded sums : proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4): 462–471, 2008.

[12] J. P. BOYD : *Chebyshev and Fourier Spectral Methods*. Springer-Verlag Berlin/Heidelberg, 2ème édition, 2001.

[13] R. P. BRENT : *Algorithms for minimization without derivatives*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

[14] R. P. BRENT : Fast multiple-precision evaluation of elementary functions. *Journal of the ACM*, 23(2):242–251, 1976.

[15] H. BRIGGS : *Logarithmorum Chilias prima*. London, 8vo, 1617.

[16] H. BRIGGS : *Arithmetica Logarithmica sive Logarithmorum chiliades triginta, pro numeris naturali serie crescentibus ab unitate ad 20000 et a 90000 ad 100000*. Londini, Excudebat Guliemus Iones, 1624.

[17] N. BRISEBARRE et S. CHEVILLARD : Efficient polynomial L^∞-approximations. *Dans* P. KORNERUP et J.-M. MULLER, éditeurs : *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 169–176, Montpellier, France, juin 2007. IEEE Computer Society Press, Los Alamitos, CA.

[18] N. BRISEBARRE et J.-M. MULLER : Correct rounding of algebraic functions. *RAIRO, Theoretical Informatics and Applications*, 41:71–83, 2007.

[19] N. BRISEBARRE, J.-M. MULLER et A. TISSERAND : Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2):236–256, 2006.

[20] J. BÜRGI : *Aritmetische und Geometrische Progreß-Tabulen, sambt gründlichem unterricht wie solche nützlich in allerley Rechnungen zugebrauchen und verstanden werden sol*. Prag, Paul Sessen, der löblichen Universitet Buchdruckern, 1620.

[21] F. CHÁVES et M. DAUMAS : A library of Taylor models for PVS automatic proof checker. *CoRR*, abs/cs/0602005, 2006.

[22] F. J. CHÁVES ALONSO : *Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, septembre 2007.

[23] E. W. CHENEY : *Introduction to Approximation Theory*. McGraw-Hill, New York, 1966.

[24] S. CHEVILLARD et Ch. LAUTER : A certified infinite norm for the implementation of elementary functions. *Dans* A. MATHUR, W. E. WONG et M. F. LAU, éditeurs : *Proceedings of the Seventh International Conference on Quality Software*, pages 153–160, Portland, OR, 2007. IEEE Computer Society Press, Los Alamitos, CA.

[25] S. CHEVILLARD, Ch. LAUTER, N. JOURDAN et M. JOLDES : *Users' manual for the Sollya tool, Release 1.1*, 2008. Disponible sur `http://gforge.inria.fr/frs/download.php/7055/sollya.pdf`.

[26] M. CORNEA, J. HARRISON et P. T. P TANG : *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[27] P. COUSOT : Interprétation abstraite. *Technique et science informatique*, 19(1-2-3):155–164, janvier 2000.

[28] M. DAUMAS et G. MELQUIOND : Generating formally certified bounds on values and round-off errors. *Dans* V. BRATTKA, C. FROUGNY et N. MÜLLER, éditeurs : *Proceedings of the 6th Conference on Real Numbers and Computers*, pages 55–70, Schloß Dagstuhl, Germany, novembre 2004.

[29] M. DAUMAS, G. MELQUIOND et C. MUÑOZ : Guaranteed proofs using interval arithmetic. *Dans* P. MONTUSCHI et E. SCHWARZ, éditeurs : *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, MA, USA, juin 2005. IEEE Computer Society Press, Los Alamitos, CA.

[30] M. DAUMAS, L. RIDEAU et L. THÉRY : A generic library of floating-point numbers and its application to exact computing. *Dans* R. J. BOULTON et P. B. JACKSON, éditeurs : *Theorem Proving in Higher Order Logics : 14th International Conference, TPHOLs 2001*, volume 2152 de *LNCS*, pages 169–184. Springer-Verlag, 2001.

[31] D. DEFOUR : *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, septembre 2003.

[32] D. DEFOUR, F. de DINECHIN et Ch. LAUTER : Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Rapport technique 2004-10, LIP, École Normale Supérieure de Lyon, mars 2004. Disponible sur `http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-10.pdf`.

[33] D. DEFOUR, F. de DINECHIN, Ch. LAUTER, C. DARAMY, N. GAST *et al.* : CR-Libm, a library of correctly rounded elementary functions in double-precision. `http://lipforge.ens-lyon.fr/www/crlibm/`.

[34] D. DEFOUR et B. GOSSENS : Implémentation de l'opérateur ADD2. Rapport de recherche RR2004-03, Laboratoire LP2A, Équipe de recherche DALI, Université de Perpignan, Perpignan, France, décembre 2004.

[35] D. DEFOUR, G. HANROT, V. LEFÈVRE, J.-M. MULLER, N. REVOL et P. ZIMMERMANN : Proposal for a standardization of mathematical function implementation in floating-point arithmetic. *Numerical Algorithms*, 37(1–4):367–375, décembre 2004.

[36] T. J. DEKKER : A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[37] J. DETREY : *Arithmétiques réelles sur FPGA : virgule fixe, virgule flottante et système logarithmique*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, janvier 2007.

[38] V. S. DIMITROV, G. A. JULLIEN et W. C. MILLER : Complexity and fast algorithms for multiexponentiations. *IEEE Transactions on Computers*, 49(2):141–147, 2000.

[39] F. de DINECHIN : Matériel et logiciel pour l'évaluation des fonctions numériques, 2007. Thèse d'habilitation, Université Claude Bernard Lyon 1.

[40] F. de DINECHIN, A. ERSHOV et N. GAST : Towards the post-ultimate libm. *Dans* P. MONTUSCHI et E. SCHWARZ, éditeurs : *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, Cape Cod, MA, USA, juin 2005. IEEE Computer Society Press, Los Alamitos, CA.

[41] F. de DINECHIN, Ch. LAUTER et G. MELQUIOND : Assisted verification of elementary functions using Gappa. *Dans* P. LANGLOIS et S. RUMP, éditeurs : *Proceedings of the 21st Annual ACM Symposium on Applied Computing - MCMS Track*, volume 2, pages 1318–1322, Dijon, France, avril 2006. ACM, Inc.

[42] F. de DINECHIN, Ch. LAUTER et G. MELQUIOND : Certifying floating-point implementations using Gappa. Rapport technique arXiv :0801.0523, LIP, CNRS/ENS Lyon/INRIA/Université de Lyon, Lyon, France, janvier 2008.

[43] F. de DINECHIN, Ch. LAUTER et J.-M. MULLER : Fast and correctly rounded logarithms in double-precision. *RAIRO, Theoretical Informatics and Applications*, 41:85–102, 2007.

[44] F. de DINECHIN, C. LOIRAT et J.-M. MULLER : A proven correctly rounded logarithm in double-precision. *Dans* V. BRATTKA, C. FROUGNY et N. MÜLLER, éditeurs : *Proceedings of the 6th Conference on Real Numbers and Computers*, Schloß Dagstuhl, Germany, novembre 2004.

[45] F. de DINECHIN et S. MAIDANOV : Software techniques for perfect elementary functions in floating-point interval arithmetic. *Dans Proceedings of the 7th conference on Real Numbers and Computers*, juillet 2006.

[46] C. B. DUNHAM : Feasibility of "perfect" function evaluation. *SIGNUM Newsl.*, 25(4): 25–26, 1990.

[47] A. FELDSTEIN et R. GOODMAN : Convergence estimates for the distribution of trailing digits. *Journal of the ACM*, 23(2):287–297, avril 1976.

[48] C. FINOT-MOREAU : *Preuves et algorithmes utilisant l'arithmétique flottante normalisée IEEE*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, juillet 2001.

[49] L. FOUSSE : *Intégration numérique avec erreur bornée en précision arbitraire*. Thèse de doctorat, Université Henri Poincaré Nancy 1, Nancy, France, décembre 2006.

[50] L. FOUSSE, G. HANROT, V. LEFÈVRE, P. PÉLISSIER et P. ZIMMERMANN : MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), juin 2007.

[51] W. FRASER : A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable. *Journal of the ACM*, 12(3):295–314, 1965.

[52] S. GAL : Computing elementary functions : A new approach for achieving high accuracy and good performance. *Dans Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.

[53] G. H. GOLUB et L. B. SMITH : Algorithm 414 : Chebyshev approximation of continuous functions by a Chebyshev system of functions. *Communications of the ACM*, 14(11): 737–746, 1971.

[54] D. M. GORDON : A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[55] S. GRAILLAT, P. LANGLOIS et N. LOUVET : Algorithms for accurate, validated and fast computations with polynomials. *Japan Journal of Industrial and Applied Mathematics*, (Special issue on Verified Numerical Computation), À paraître.

[56] A. GRIEWANK : *Evaluating derivatives : Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, PA, USA, 2000.

[57] G. HANROT, V. LEFÈVRE, D. STEHLÉ et P. ZIMMERMANN : Worst cases of a periodic function for large arguments. *Dans* P. KORNERUP et J.-M. MULLER, éditeurs : *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 133–140, Montpellier, France, juin 2007. IEEE Computer Society Press, Los Alamitos, CA.

[58] E. HANSEN et G. W. WALSTER : *Global Optimization Using Interval Analysis, Second Edition, Revised and Expanded*. Marcel Dekker, Inc., New York, Basel, 2004.

[59] G. H. HARDY et E. M. WRIGHT : *An introduction to the theory of numbers*. Oxford University Press, 1979.

[60] J. HARRISON : Floating point verification in HOL light : the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[61] J. HARRISON : Formal verification of floating point trigonometric functions. *Dans Formal Methods in Computer-Aided Design : Third International Conference FMCAD 2000*, volume 1954 de *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.

[62] J. HARRISON : Formal verification of square root algorithms. *Formal Methods in Systems Design*, 22:143–153, 2003.

[63] Y. HIDA, X. S. LI et D. H. BAILEY : Algorithms for quad-double precision floating-point arithmetic. *Dans* N. BURGESS et L. CIMINIERA, éditeurs : *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, juin 2001. IEEE Computer Society Press, Los Alamitos, CA.

[64] N. J. HIGHAM : *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, USA, 2ème édition, 2002.

[65] W. HOFSCHUSTER et W. KRÄMER : FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format. Rapport technique 98/7, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, 1998.

[66] W. HOFSCHUSTER, W. KRÄMER, M. LERCH, G. TISCHLER et J. W. von GUDENBERG : filib++ a fast interval library supporting containment computations. *ACM Transactions on Mathematical Software*, 2005. À paraître.

[67] IEEE COMPUTER SOCIETY : IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754™-2008 (Revision of IEEE Std 754-2008)*, août 2008. Sponsored by the Microprocessor Standards Committee, IEEE, 3 Park Avenue, New York, NY, USA.

[68] ISO/IEC : *International Standard ISO/IEC 9899 :1999(E). Programming languages – C*. 1999.

[69] C.-P. JEANNEROD, H. KNOCHEL, C. MONAT et G. REVY : Faster floating-point square root for integer processors. *Dans Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES'07)*, Lisbon, Portugal, juillet 2007.

[70] M. JOLDES : Computation of various infinite norms, in one or several variables, designed for the development of mathematical libraries. Mémoire de D.E.A., École Normale Supérieure de Lyon, 2008.

[71] W. KAHAN : The Table Maker's Dilemma and other Quandraries. *Dans Mathematical Software II, Informal Proceedings of a conference, dans : ACM Transactions on Mathematical Software, Papers from Mathematical Software II*, pages 25–34, West Lafayette, Indiana, USA, 1974.

[72] W. KAHAN : Minimizing q*m-n. Texte accessible en version électronique sous `http://http.cs.berkeley.edu/~wkahan/`. Discussion au début du fichier `nearpi.c`, 1983.

[73] D. KNUTH : *The Art of Computer Programming, vol.2 : Seminumerical Algorithms*. Addison Wesley, 3ème édition, 1997.

[74] P. KORNERUP, Ch. LAUTER, V. LEFÈVRE, N. LOUVET et J.-M. MULLER : Computing correctly rounded integer powers in floating-point arithmetic. Rapport technique

RR 2008-15, SDU, Odense, Denmark et LIP, CNRS/ENS Lyon/INRIA/Université de Lyon, Lyon, France, mai 2008. À paraître dans ACM Transactions on Mathematical Software.

[75] P. KORNERUP, V. LEFÈVRE et J.-M. MULLER : Computing integer powers in floating-point arithmetic. Rapport de recherche RR2007-23, Laboratoire de l'Informatique du Parallélisme, Lyon, France, mai 2007.

[76] W. KRÄMER : Sichere und genaue Abschätzung des Approximationsfehlers bei rationalen Approximationen. Rapport technique, Institut für angewandte Mathematik, Universität Karlsruhe, 1996.

[77] V. KREINOVICH et S. RUMP : Towards optimal use of multi-precision arithmetic : A remark. *Reliable Computing*, 12(5):365–369, 2006.

[78] P. LANGLOIS et N. LOUVET : More instruction level parallelism explains the actual efficiency of compensated algorithms. Rapport de recherche hal-00165020, DALI Research Team, HAL-CCSD, juillet 2007.

[79] P.-J. LAURENT : *Approximation et optimisation*. Hermann, Paris, 1972.

[80] Ch. LAUTER : A correctly rounded implementation of the exponential function on the Intel Itanium architecture. Rapport technique RR-5024, INRIA, novembre 2003. Disponible sur `http://www.inria.fr/rrrt/rr-5024.html`.

[81] Ch. LAUTER : Basic building blocks for a Triple-Double intermediate format. Rapport technique RR-5702, INRIA, septembre 2005.

[82] Ch. LAUTER et F. de DINECHIN : Optimizing polynomials for floating-point implementation. *Dans* J. D. BRUGUERA et M. DAUMAS, éditeurs : *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 7–16, Santiago de Compostela, Espagne, juillet 2008.

[83] Ch. LAUTER et V. LEFÈVRE : An efficient rounding boundary test for pow(x,y) in double precision. Technical Report RR-2007-36, Laboratoire de l'Informatique du Parallélisme, 2007. À paraître dans IEEE Transactions on Computers.

[84] V. LEFÈVRE : *Moyens arithmétiques pour un calcul fiable*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, 2000.

[85] V. LEFÈVRE : New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. *Dans* P. MONTUSCHI et E. SCHWARZ, éditeurs : *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 68–75, Cape Cod, MA, USA, juin 2005. IEEE Computer Society Press, Los Alamitos, CA.

[86] V. LEFÈVRE et J.-M. MULLER : Worst cases for correct rounding of the elementary functions in double precision. *Dans* N. BURGESS et L. CIMINIERA, éditeurs : *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 111–118, Vail, Colorado, juin 2001. IEEE Computer Society Press, Los Alamitos, CA.

[87] V. LEFÈVRE et J.-M. MULLER : Worst cases for correct rounding of the elementary functions in double precision, 2004. Disponible sur `http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm`.

[88] V. LEFÈVRE, J.-M. MULLER et A. TISSERAND : Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, novembre 1998.

[89] C. E. LEISERSON, H. PROKOP et K. H. RANDALL : Using de Bruijn sequences to index a 1 in a computer word. Disponible par `ftp://theory.lcs.mit.edu/pub/cilk/debruijn.ps.gz`.

[90] R.-C. LI : Near optimality of Chebyshev interpolation for elementary function computations. *IEEE Transactions on Computers*, 53(6):678–687, 2004.

[91] R.-C. LI, P. MARKSTEIN, J. P. OKADA et J. W. THOMAS : The libm library and floating-point arithmetic for HP-UX on Itanium. Rapport technique, Hewlett-Packard company, avril 2001.

[92] N. LOUVET : *Algorithmes compensés en arithmétique flottante : précision, validation, performances*. Thèse de doctorat, Université de Perpignan Via Domitia, Perpignan, France, novembre 2008.

[93] P. MARKSTEIN : *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN : 0130183482.

[94] M. MARTEL : An overview of semantics for the validation of numerical programs. *Dans VMCAI*, pages 59–77, 2005.

[95] J. C. MASON et D. HANDSCOMB : *Chebyshev polynomials*. Chapman & Hall/CRC, Boca Raton, London, New York, Washington, D.C., 2003.

[96] G. MELQUIOND : *Arithmétique d'intervalles et certification de programmes*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, novembre 2006.

[97] G. MELQUIOND et S. PION : Formal certification of arithmetic filters for geometric predicates. *Dans Proceedings of the 15th IMACS World Congress on Computational and Applied Mathematics*, 2005.

[98] Sun MICROSYSTEMS : libmcr, a reference correctly-rounded library of basic double-precision transcendental elementary functions. Disponible sur `http://www.sun.com/download/products.xml?id=41797765`.

[99] B. MÖLLER : Algorithms for multi-exponentiation. *Dans SAC '01 : Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, pages 165–180, London, UK, 2001. Springer-Verlag.

[100] O. MØLLER : Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[101] J.-M. MULLER : *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 2ème édition, 2006.

[102] F. D. MURNAGHAN et J. W. WRENCH : David Taylor Model Basin. Rapport technique Report No. 1175, Md., 1960.

[103] J. L. of M. NAPIER : *Mirifici logarithmorum canonis descriptio, ejusque usus, in utraque trigonometria*. Edinburgi, Ex officinae A. Hart, 1614.

[104] M. NEHER : ACETAF : A software package for computing validated bounds for Taylor coefficients of analytic functions. *ACM Transactions on Mathematical Software*, 2003.

[105] A. NEUMAIER : *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.

[106] T. OGITA, S. M. RUMP et S. OISHI : Accurate sum and dot product. *SIAM Journal on Scientific Computing (SISC)*, 2005.

[107] M. J. D. POWELL : On the maximum errors of polynomial approximations defined by interpolation and by least squares criteria. *The Computer Journal*, 9(4):404–407, février 1967.

[108] D. M. PRIEST : Algorithms for arbitrary precision floating point arithmetic. *Dans* P. KORNERUP et D. W. MATULA, éditeurs : *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, juin 1991. IEEE Computer Society Press, Los Alamitos, CA.

[109] C. H. REINSCH : Principles and preferences for computer arithmetic. *ACM Signum Newsletter*, 14(1):12–27, mars 1979.

[110] E. REMEZ : Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *Comptes Rendus de l'Académie des Sciences, Paris*, 198, 1934.

[111] G. REVY : Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Mémoire de D.E.A., École Normale Supérieure de Lyon, 2006.

[112] Е. Я. Ремез (E. YA. REMEZ) : Основы численных методов чебышевского приближения. Академия Наук Украинской ССР, Институт математики, Наукова Думка, Kiev, 1969.

[113] J.R. SHEWCHUK : Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Dans Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.

[114] R. A. SMITH : A continued-fraction analysis of trigonometric argument reduction. *IEEE Transactions on Computers*, 44(11):1348–1351, novembre 1995.

[115] D. STEHLÉ : *Algorithmique de la réduction de réseaux et application à la recherche de pires cas pour l'arrondi de fonctions mathématiques*. Thèse de doctorat, École Doctorale IEAM Lorraine, Université Henri Poincaré – Nancy 1, novembre 2005.

[116] D. STEHLÉ, V. LEFÈVRE et P. ZIMMERMANN : Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers*, 54(3):340–346, mars 2005.

[117] D. STEHLÉ et P. ZIMMERMANN : Gal's accurate tables method revisited. *Dans* P. MONTUSCHI et E. SCHWARZ, éditeurs : *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 257–264, Cape Cod, MA, USA, juin 2005. IEEE Computer Society Press, Los Alamitos, CA.

[118] P. H. STERBENZ : *Floating point computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[119] E. STIEFEL : Methods–old and new–for solving the Tchebycheff approximation problem. *Journal of the SIAM : Series B, Numerical Analysis*, 1:164–176, 1964.

[120] S. STORY et P. T. P. TANG : New algorithms for improved transcendental functions on IA-64. *Dans Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia*, pages 4–11. IEEE Computer Society Press, avril 1999.

[121] P. T. P. TANG : Table lookup algorithms for elementary functions and their error analysis. *Dans* P. KORNERUP et D. W. MATULA, éditeurs : *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, juin 1991. IEEE Computer Society Press, Los Alamitos, CA.

[122] P. T. P. TANG : Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 18(2):211–222, juin 1992.

[123] P. L. TCHEBYCHEV : Théorie des mécanismes connus sous le nom de parallélogrammes. *Mémoires des Savants étrangers présentés à l'Academie de Saint-Pétersbourg*, 7:539–586, 1854.

[124] MPFR TEAM : The MPFR library : Algorithms and proofs. Fichier `algorithms.tex`, révision 4629 (July 4, 2007), Disponible dans le SVN sur `http://www.mpfr.org/gforge.html`.

[125] E. G. THURBER : Efficient generation of minimal length addition chains. *SIAM Journal of Computation*, 28(4):1247–1263, 1999.

[126] L. VEIDINGER : On the numerical determination of the best approximations in the Chebyshev sense. *Numerische Mathematik*, 2:99–105, 1960.

[127] N. VEYRAT-CHARVILLON : *Opérateurs arithmétiques matériels pour des applications spécifiques*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, juin 2007.

[128] K. WEIHRAUCH : *Computable Analysis, An Introduction*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag Berlin/Heidelberg, 2000.

[129] W. F. WONG et E. GOTO : Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3): 278–294, mars 1994.

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

[131] R. ZUMKELLER : Formal global optimisation with Taylor models. *Dans IJCAR*, pages 408–422, 2006.

Résumé :

Cette thèse élargit l'espace de recherche accessible en pratique pour des implantations de fonctions mathématiques correctement arrondies en virgule flottante. Elle passe d'abord de l'arrondi correct de fonctions univariées comme $\log$ à des familles de fonctions univariées x^n, puis à la fonction bivariée x^y. Une approche innovatrice pour la détection de cas de frontière d'arrondi de x^y à l'aide d'une information partielle sur les pires cas de cette fonction est proposée. Cette innovation provoque un gain en vitesse conséquent.

Ensuite, cette thèse propose des approches automatiques pour certifier une implantation de fonction correctement arrondie. Pour la certifications des bornes d'erreur d'approximation, un nouvel algorithme pour le calcul de normes infini certifiées est présenté et mis en pratique. Puis les erreurs d'arrondi dans une implantation de fonction mathématique sont analysées par des techniques développées pour l'outil de preuve formelle Gappa.

Finalement, des algorithmes sont développés qui permettent l'automatisation de l'implantation de fonctions. Une première mise en œuvre dans l'outil Sollya permet de générer et certifier, sans aucune intervention humaine, le code pour évaluer une fonction mathématique. À l'aide d'un tel outil automatique, de larges espaces de recherches peuvent être parcouru afin d'optimiser une implantation. Au futur, une intégration de ces techniques dans un compilateur est envisageable.

Mots-clefs :

fonctions mathématiques et élémentaires, fonction power, arrondi correct, libm, virgule flottante, certification, norme infini, norme sup, Sollya, synthèse de code numérique

---

Abstract:

This work extends the practically accessible research space for correctly rounded floating-point implementations of mathematical functions. Interest goes first from univariate functions like $\log$ to classes of univariate functions like x^n, then to the bivariate function x^y. An innovative approach for detecting the rounding boundary cases of x^y using partial worst-case information is proposed. The novel techniques yields to a consequent speed-up.

Then automatic approaches are contributed that allow for certifying a correctly rounded implementation of a function. For the certification of bounds on approximation errors, a novel algorithm for computing certified infinity norms is presented and implemented. Concerning round-off error in an implementation of a mathematical function, techniques are developed for analysing them using the Gappa tool.

Finally, algorithms are developed that permit to automate the implementation process for mathematical functions. A first realisation in the Sollya tool allows for generating and certifying, without any human intervention, the code for evaluating a function. With such an automatic tool, large solution spaces can be searched in order to find an optimised implementation. In the future, an integration of the techniques in a compiler could be investigated.

Keywords:

Mathematical and elementary functions, Power function, Correct rounding, libm, Floating-point code, Certification, Infinity norm, Supremum norm, Sollya, Numerical code synthesis