Amélioration de la fiabilité numérique de codes de calcul industriels

Romain Picot

THÈSE DE DOCTORAT
DE SORBONNE UNIVERSITÉ

Spécialité : Informatique

réalisée

au Laboratoire d'Informatique de Paris 6

présentée par

Romain PICOT

pour obtenir le grade de :

DOCTEUR DE SORBONNE UNIVERSITÉ

Sujet de la thèse :

Amélioration de la fiabilité numérique de codes de calcul industriels

Table des matières

Introduction

1 Pourquoi simuler ?

Dans un contexte industriel, il est important de pouvoir simuler les phénomènes physiques. En effet, certaines expériences ont des coûts élevés ou ne sont pas réalisables, et il est intéressant de les simuler. Il est compliqué par exemple de construire un bâtiment à la taille réelle et d'effectuer ensuite des expérimentations sur celui-ci dans le but de vérifier sa conception ou la façon dont sa structure va évoluer durant ses dizaines années d'utilisation. Des outils permettent la simulation numérique en mécanique des structures et en particulier Code_aster [29] développé par EDF. Ce programme permet de simuler des bâtiments avant leur construction et d'ajouter des contraintes sur l'environnement et ainsi de tester les effets du vent, de séismes, de chocs, de l'usure naturelle, ... [30].

Le temps est un facteur limitant les expériences réelles. Prenons le cas de CIGÉO : Centre Industriel de stockage GÉOlogique [24]. Il s'agit d'un projet français d'enfouissement des déchets radioactifs en couche géologique profonde. Le but est d'enterrer des déchets radioactifs ne pouvant être ni recyclés, ni stockés à long terme en surface pour des raisons de sécurité et de radioprotection. Un recours à des simulations numériques est nécessaire pour savoir si le projet est sûr ou non sur plusieurs milliers d'années [1].

La sécurité des expérimentateurs est également très importante. Un contrôle non destructif[1] par radiographie, par rayons X ou par rayons gamma, amène l'exposition de l'opérateur à ces rayons. Il faut donc pouvoir limiter le nombre d'expérimentations. Cela nécessite donc de prévoir le nombre et la position des différents contrôles. Cette préparation effectuée par simulation numérique limite l'exposition aux rayons [116].

Les simulations numériques permettent également de prendre des décisions en testant plusieurs scénarios en avance et en écartant certains au fur et à mesure. Par exemple, apogene est un outil interne à EDF dont l'objectif est de planifier la production électrique en optimisant le coût, sous les contraintes de respect de l'équilibre production–consommation. Apogene s'appuie pour ce faire sur CPLEX [28], un solveur de programmation linéaire permettant de résoudre des problèmes capable de gérer des problèmes comportant 2^{30} paramètres et ayant 2^{30} contraintes.

Les simulations numériques sont donc des outils puissants et nécessaires pour une entreprise telle qu'EDF. Les développements logiciels en interne concernent notamment la simulation des structures avec Code_aster [29], la simulation en mécanique des fluides avec code Saturne [25] ou Telemac [66] pour les calculs en surface libre, la simulation en

1. Méthode permettant d'examiner l'état d'une structure sans la dégrader.

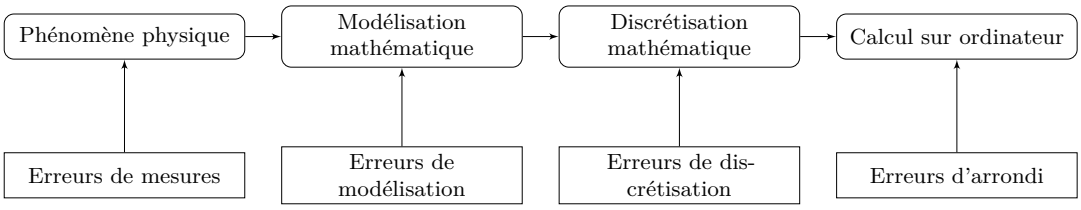


FIGURE 1 – Les étapes de la simulation numérique et les erreurs associées

neutronique avec COCAGNE [16] ou encore une plateforme de pré et post traitement pour les simulations numérique SALOME [123]. Ces outils ne sont qu'une partie de ceux développés à EDF, mais montrent l'étendue des compétences nécessaires au bon fonctionnement des moyens de production d'électricité et à leur modélisation.

2 Les problématiques de la simulation numérique

Pour simuler un problème physique, plusieurs étapes sont à effectuer mais elles peuvent être sources d'erreurs, voir figure 1. Ces erreurs peuvent provenir de mesures sur le phénomène à étudier, sur sa modélisation [2], de la discrétisation mathématique [3] et enfin de l'arithmétique de l'ordinateur. Pour limiter au maximum les erreurs, des étapes de V&V [4] doivent être effectuées. La validation est la comparaison des résultats par rapport à des mesures quand la vérification est l'analyse du logiciel en fonction des spécifications de celui-ci, et donc la comparaison entre les résultats du programme et ceux provenant des modèles mathématiques. Dans la pratique, ces termes sont régulièrement utilisés l'un à la place de l'autre malgré la nuance entre les deux.

Toutes les erreurs s'accumulent et ont un impact sur le résultat de la simulation numérique. Nous nous intéresserons ici plus particulièrement au dernier type d'erreur : les erreurs d'arrondi, dues à l'arithmétique à virgule flottante.

Les erreurs d'arrondi peuvent ainsi amener de nombreux problèmes. Nous pouvons citer ici le cas du missile Patriot qui n'a pas réussi à intercepter un missile Scud durant la première guerre du Golfe en 1991. Une multiplication par 0.1 était utilisée pour mesurer le temps. Les registres du missile Patriot étaient limités à 24 chiffres binaires. Or $(0.1)_{10}$ n'est pas représentable en binaire de manière exacte et doit donc être arrondi. La perte de précision est de 9.5×10^{-8} [7] à chaque calcul. Au bout d'une centaine d'heures de calcul, toutes ces pertes de précision ont créé un décalage de 0.34 seconde. Le temps étant utilisé pour déterminer la position d'un missile à intercepter, l'erreur de 0.34 seconde a empêché son interception [7].

Ces problèmes peuvent impacter fortement le résultat d'un code de simulation et amener à des conclusions erronées.

EDF doit garantir l'état de ses moyens de production, et cela durant toute leur durée de vie. La validation numérique doit être un élément important de la conception et de l'amélioration des logiciels de calcul. EDF travaille depuis plusieurs années sur la fiabilité numérique des codes. En particulier en 2010, un travail sur la validation numérique avec

2. Cela peut être le choix d'une modélisation non adaptée au phénomène étudié
3. Un critère d'arrêt trop faible pour un algorithme itératif est source d'erreur par exemple
4. vérification et validation

l'équipe PEQUAN du LIP6 [5] a débuté. La bibliothèque CADNA [39, 76], qui implémente l'Arithmétique Stochastique Discrète [138] et qui sera décrite en section I.4.3, a été utilisée pour étudier la qualité numérique des codes industriels développés à EDF. Une fois son utilisation validée dans le contexte d'EDF [103], un partenariat a été mis en place pour améliorer la qualité numérique des codes internes mais aussi permettre l'ajout de fonctionnalités au sein de CADNA.

Dans le cadre de sa thèse CIFRE EDF-LIP6 [101], Montan a modifié la bibliothèque CADNA afin qu'elle puisse contrôler la fiabilité numérique de codes parallèles utilisant la bibliothèque de communication MPI [103]. Cette collaboration a également amené au développement d'une bibliothèque d'algèbre linéaire [102] permettant le calcul de produits matriciels basés sur les BLAS [6] (Basic Linear Algebra Subprograms) avec l'Arithmétique Stochastique Discrète [138] de manière efficace.

Ces travaux ont mis en avant des améliorations possibles et en particulier la nécessité d'avoir un outil de post-traitement graphique permettant d'obtenir les lignes de code responsables des instabilités, leur localisation dans les fonctions et les types d'instabilités numériques associés. Cela a amené à la conception et au développement de CADNAgrind [119], fondé sur l'interface graphique Kcachegrind [140].

En outre, un outil de validation numérique sans recompiler est en développement à EDF : verrou [42]. Il est ainsi possible de valider un code à partir de son exécutable grâce à l'utilisation de Valgrind [108]. Verrou sera détaillé dans la section I.2.5.

EDF travaille également sur la reproductibilité des codes de simulation numérique. Des travaux permettent de valider un code ayant différents résultats à l'exécution [10]. Cela est particulièrement utile lorsque des options de compilation ou des architectures matérielles différentes sont utilisées. Néanmoins, cela ne règle pas le problème des codes parallèles qui peuvent souffrir d'une modification de l'ordre des opérations. Il faut donc déterminer les causes de la non-reproductibilité pour obtenir un programme pouvant être utilisé. Durant sa thèse [101], Montan a montré que l'estimation des erreurs d'arrondi et leur correction permettent d'améliorer la reproductibilité d'un code de simulation numérique tel que Telemac2D [65]. La solution proposée est d'utiliser des algorithmes compensés [32, 85, 107, 120]. Ceux-ci améliorent la reproductibilité de Telemac-2D dans un contexte parallèle, sans dégrader les performances lors d'exécutions *memory bound* [101, chapitre 6]. Une question s'est alors posée : s'agit-il de la meilleure méthode pour améliorer la reproductibilité d'un code industriel ?

Pour répondre à cette interrogation, un partenariat avec l'équipe DALI de l'université de Perpignan sur la reproductibilité numérique des codes a été mis en place. Plusieurs options ont été étudiées pour la sommation parallèle utilisée dans TOMAWAC [111], une des bibliothèques de Telemac : algorithmes dédiés reproductibles [34], algorithmes compensés [114] ou encore transformation des opérations flottantes en opérations entières reproductibles [134]. Les algorithmes compensés ont permis d'obtenir les meilleurs résultats et ont été appliqués sur un cas d'étude de Telemac-2D en améliorant sa reproductibilité numérique [112]. Les travaux sur ce sujet sont à poursuivre sur des cas tests pouvant être mal conditionnés, il pourrait devenir nécessaire de développer d'autres méthodes que celles des algorithmes compensés [110].

5. Laboratoire d'Informatique de Paris 6

3 Contributions de la thèse

Grâce à des bibliothèques de validation numérique telles que CADNA [39, 76], nous savons identifier les parties numériquement instables d'un code. Celles-ci doivent être corrigées. Pour ce faire, nous pouvons utiliser des algorithmes qui prennent en compte les erreurs d'arrondi et les corrigent : les algorithmes compensés qui sont fondés sur les transformations exactes [32, 85, 107, 120]. Celles-ci ont été démontrées dans un cadre théorique particulier. La méthode CESTAC nécessite des hypothèses incompatibles avec celui-ci. Nous avons donc étudié le comportement des algorithmes compensés avec la méthode CESTAC et une implémentation logicielle : la bibliothèque CADNA, chapitre II [54, 55].

À l'inverse, le programme que nous utilisons peut fournir une précision supérieure à celle que nous souhaitons obtenir. Il pourrait être plus judicieux de se limiter à la précision des résultats souhaitée dans le but d'améliorer les performances du code (temps d'exécution et utilisation mémoire). Nous avons donc développé un outil nommé PROMISE [6] permettant de modifier automatiquement les types des variables flottantes tout en validant le programme modifié, chapitre III [56, 57].

Certains problèmes sont naturellement difficile à résoudre en raison de leur instabilités ou de leur mauvais conditionnement [7]. Pour de tels problèmes, il est long d'améliorer grâce à des algorithmes la précision des résultats obtenus. Nous pouvons citer par exemple les problèmes chaotiques tels que l'attracteur de Hénon [71] ou la recherche de racines multiples d'un polynôme et de leurs multiplicités [50, 53, 82, 135]. Pour être en mesure de valider les résultats plus rapidement, nous avons donc développé une extension à la bibliothèque CADNA dans le but de pouvoir utiliser une précision supérieure. Enfin, nous avons introduit la possibilité d'utiliser PROMISE, notre outil pour optimiser le type des variables flottantes, avec trois précisions à la place de deux et ainsi limiter le coût d'utilisation d'une précision supérieure, chapitre IV.

6. PRecision OptiMISE
7. La notion de conditionnement sera rappelée dans la section I.2.1.a.

Chapitre I

Arithmétique à virgule flottante et validation numérique

I.1 Arithmétique à virgule flottante

I.1.1 Représentation d'un nombre

Un système de numération se compose d'un ensemble de symboles permettant de représenter des nombres. Le nombre de symboles différents permet en général de définir la base. Dans un système décimal, soit de base 10, nous utilisons les chiffres 0,1, ..., 9. Sur ordinateur, le système binaire est le plus utilisé, soit une base 2, et les nombres sont représentés grâce à une série de 0 et de 1 aussi appelés bits.

Différentes représentations des nombres existent sur ordinateur. Nous pouvons tout d'abord citer la représentation des nombres entiers. Avec une base b, si nous représentons un nombre non signé grâce à n chiffres, alors ils peuvent varier de 0 à $b^n - 1$. Si X est un nombre entier non signé il sera représenté par la série $a_{n-1}a_{n-2}\cdots a_1a_0$ avec $X = \sum_{i=0}^{n-1} a_i b^i$ et $a_i \in \{0, \cdots, b-1\}$. Les nombres entiers signés sont en général codés en base 2 en utilisant le complément à deux. L'opposé est obtenu en inversant chaque bit et en additionnant 1. Dans ce cas, un entier sera compris entre $-b^{n-1}$ et $b^{n-1} - 1$. La représentation de X sera $a_{n-1}a_{n-2}\cdots a_1a_0$ avec $a_i \in \{0, \cdots, b-1\}$ tel que $X = -a_{n-1}b^{n-1} + \sum_{i=0}^{n-2} a_i b^i$.

La représentation d'un nombre réel sur ordinateur est plus compliquée. En effet, il n'est pas possible de stocker un nombre infini de chiffres après la virgule. Deux représentations principales existent :

— l'arithmétique à virgule fixe, utilisé en particulier dans les systèmes embarqués ;
— l'arithmétique à virgule flottante, majoritaire dans les ordinateurs.

Dans le cas de l'arithmétique à virgule fixe, un nombre est représenté avec un nombre fixe de chiffres avant et après la virgule. Avec une base b, un nombre X composé de m chiffres pour la partie entière et f chiffres pour la partie fractionnaire s'écrira sous la forme $a_{m-1}\cdots a_0.a_{-1}\cdots a_{-f}$, avec $X = \sum_{i=-f}^{m-1} a_i b^i$ et $a_i \in \{0, \cdots, b-1\}$. Si $b = 2$, un nombre à virgule fixe non signé sera compris entre 0 et $2^m - 2^{-f}$ et un nombre signé sera majoritairement représenté par le complément à deux et sera compris entre -2^{m-1} et $2^{m-1} - 2^{-f}$.

Dans le cas d'une base b, un nombre X à virgule flottante est représenté par :
— son bit de signe ϵ_X égal à 0 si $\epsilon_X = 1$ et 1 si $\epsilon_X = -1$;
— son exposant E_X, un entier à k chiffres ;
— sa mantisse M_X, codée sur p chiffres.

Ainsi nous avons $X = \epsilon_X M_X b^{E_X}$ avec $M_X = \sum_{i=0}^{p-1} a_i b^{-i}$ et $a_i \in \{0, \cdots, b-1\}$. La mantisse M_X peut être écrite sous la forme $M_X = a_0.a_1 \cdots a_{p-1}$. Les nombres à virgule flottante sont en général codés de manière à avoir unicité de la représentation. Nous avons dans ce cas $a_0 \neq 0$, $M_X \in [1, b[$ et zéro possède une représentation spéciale. Il y a alors unicité de la représentation (il n'existe qu'une et unique écriture d'un nombre) et les comparaisons sont simplifiées (les signes, exposants et mantisses de deux nombres normalisés peuvent être testés séparément).

I.1.2 La norme IEEE 754

Avant toute standardisation des nombres flottants, les fabricants d'ordinateurs décidaient de la représentation de ces nombres. Deux machines différentes pouvaient ainsi fournir deux résultats distincts en raison de cette différence. En 1985, la première version de la norme IEEE 754 est publiée [72]. Elle définit deux types en binaire :
— *simple* précision codé sur 32 bits : 1 de signe, 8 d'exposant et 23 de mantisse explicites et 1 bit implicite [1] ;
— *double* précision codé sur 64 bits : 1 de signe, 11 d'exposant et 52 de mantisse explicites et 1 bit implicite.

Une seconde version de la norme IEEE 754 est publiée en 2008 [73]. En plus du renommage des types précédents (`binary32` et `binary64`), elle définit les types décimaux ainsi que deux nouveaux types binaires :
— `binary16` ou *half* précision codé sur 16 bits : 1 de signe, 5 d'exposant et 10 de mantisse explicites et 1 bit implicite ;
— `binary128` ou *quadruple* précision codé sur 128 bits : 1 de signe, 15 d'exposant et 112 de mantisse explicites et 1 bit implicite.

La normalisation de la mantisse implique que celle-ci doit commencer par un bit égal à 1. Ce bit est défini implicitement. Ainsi, les précisions réelles des mantisses sont de 11 bits en *half*, 24 bits en *simple*, 53 bits en *double* et 113 bits en *quadruple* précision.

L'exposant E est un entier signé de k bits. E_{min} (resp. E_{max}) représente la valeur minimale (resp. maximale) de l'exposant. Il est stocké sous la forme d'un exposant biaisé E_Δ tel que $E_\Delta = E + \Delta$, avec Δ le biais. Le tableau I.1 résume les valeurs extrêmes et le biais pour les types binaires de la norme.

Des valeurs particulières sont définies par la norme. Par exemple le 0 est codé en mémoire en mettant tous les bits de la mantisse et de l'exposant à 0. Le bit de signe permet de faire une distinction entre -0 et 0. Les infinis sont codés en affectant à 1 tous les bits de l'exposant et à 0 ceux de la mantisse.

NaN [2] est une autre valeur spéciale. Il s'agit du résultat d'une opération invalide ou d'une opération dont le résultat n'est pas défini, par exemple $0/0$, $\sqrt{-1}$ ou $0 \times \infty$. Dans ce cas, les bits de l'exposant sont à 1 et la mantisse possède au moins un bit non nul.

1. La définition du bit implicite est donnée dans le paragraphe suivant.
2. Not a Number

précision	taille	biais	exposant non biaisé		exposant biaisé	
	k	Δ	E_{min}	E_{max}	$E_{min}+\Delta$	$E_{max}+\Delta$
half	5	15	-14	15	1	30
simple	8	127	-126	127	1	254
double	11	1023	-1022	1023	1	2046
quadruple	15	16383	-16382	16383	1	32766

Table I.1 – Codage des exposants pour les précisions *half*, *simple*, *double* et *quadruple*

Les nombres dénormalisés sont des nombres dont l'exposant biaisé est égal à 0 et dont au moins un bit de la mantisse est non nul. Dans ce cas le bit implicite est égal à 0. Ainsi les nombres dénormalisés sont compris entre $2^{E_{min}+1-p}$ et $2^{E_{min}}$ avec p le nombre de bits de la mantisse.

Cette représentation des nombres dans un format binaire *fini* implique que des nombres réels ne sont pas représentables. Nous notons $\mathbb{F}$ l'ensemble des nombres flottants représentables. Cet ensemble est borné par des valeurs X_{min} et X_{max}. Soit x un nombre réel non représentable dans l'intervalle $[X_{min}, X_{max}]$, alors il existe $X^- \in \mathbb{F}$ et $X^+ \in \mathbb{F}$ tels que $X^- < x < X^+$ et $]X^-, X^+[\cap \mathbb{F} = \emptyset$. La norme IEEE 754 prévoit plusieurs modes d'arrondi pour x :

- arrondi vers plus l'infini, x est représenté par X^+ ;
- arrondi vers moins l'infini, x est représenté par X^- ;
- arrondi vers zéro, si x est négatif, alors il est représenté par X^+, si x est positif, alors il est représenté par X^- ;
- arrondi au plus près *ties to even*, x est représenté par le flottant le plus proche. Si x est à égale distance de X^- et X^+, il est représenté par le flottant dont la mantisse se termine par 0 en base 2 ;
- arrondi au plus près *ties away from zero*, x est représenté par le flottant le plus proche. Si x est à égale distance de X^- et X^+, il est arrondi à X^- (resp. X^+) si x est négatif (resp. positif).

Soit X le résultat d'un arrondi cité précédemment de x. Par définition, un *overflow* apparaît si $|X| > \max\{|Y| : Y \in \mathbb{F}\}$ et un *underflow* apparaît si $0 < |X| < \min\{|Y| : 0 \neq Y \in \mathbb{F}\}$.

Lors de l'exécution d'un programme en arithmétique à virgule flottante, les exceptions suivantes peuvent être levées :

- *underflow* : lorsque le résultat est trop petit pour être représenté dans le format normalisé des nombres flottants ;
- *overflow* : lorsque le résultat est trop élevé pour être représenté dans le format normalisé des nombres flottants ;
- *inexact* : lorsque le résultat de l'opération flottante n'est pas exact et doit être arrondi ;
- *invalid* : lorsque l'opération n'est pas valide (par exemple une multiplication entre zéro et l'infini) ;
- *divide-by-zero* : lors d'une division par zéro.

I.1.3 Les erreurs d'arrondi

I.1.3.a Nombre de chiffres significatifs exacts

Il est possible de définir la précision d'un résultat de calcul grâce à son nombre de chiffres significatifs exacts. Soit R le résultat calculé et r le résultat exact, alors le nombre de chiffres décimaux exacts de R noté $C_{R,r}$ est le nombre de chiffres en commun avec r :

$$C_{R,r} = \log_{10} \left| \frac{R+r}{2(R-r)} \right|. \tag{I.1.1}$$

De manière similaire, le nombre de bits significatifs exacts de R est :

$$B_{R,r} = \log_{2} \left| \frac{R+r}{2(R-r)} \right|. \tag{I.1.2}$$

Ainsi si $C_{R,r} = 3$, alors l'erreur relative entre R et r est de l'ordre de 10^{-3}. R et r ont ainsi trois chiffres décimaux en commun. Il est à noter que dans certains cas le nombre de chiffres significatifs exacts peut surprendre si nous prenons en compte la représentation décimale de R et r. Par exemple, si $R = 2,4599976$ et $r = 2,4600012$, nous voyons une différence à partir du troisième chiffre. Néanmoins, cette différence n'est pas aussi importante. En effet nous avons $C_{R,r} \approx 5,8$, la différence est donc significative à partir du sixième chiffre.

I.1.3.b Erreur d'arrondi sur une opération

Soit X la représentation du nombre réel x respectant la norme IEEE 754, alors X peut être écrite comme $X = fl(x)$, X étant l'arrondi de x dans le système flottant. Si nous reprenons les notations de la section I.1.1, nous avons alors :

$$X = \epsilon_X M_X 2^{E_X} \tag{I.1.3}$$

et

$$X = x - \epsilon_X 2^{E_X - p} \alpha_X \tag{I.1.4}$$

avec α_X la normalisation de l'erreur d'arrondi :
— en arrondi au plus près, $\alpha_X \in [-0,5, 0,5[$;
— en arrondi vers zéro, $\alpha_X \in [0, 1[$;
— en arrondi vers plus ou moins l'infini, $\alpha_X \in [-1, 1[$.

La distance ϵ entre la valeur 1 et le nombre flottant immédiatement supérieur est appelée la *précision machine*. Nous savons alors que $\epsilon = 2^{1-p}$ avec p la taille de la mantisse en prenant en compte le bit implicite. L'erreur relative sur X est telle que :

$$X = x(1+\delta) + \eta \text{ avec } |\delta| \leq \mathbf{u} \tag{I.1.5}$$

avec $|\eta| \leq 2^{e_{min}}/2$ en arrondi au plus près et $|\eta| \leq 2^{e_{min}}$ sinon, $2^{e_{min}}$ étant le plus petit nombre dénormalisé positif. Si nous ne tenons pas compte des *underflows*, nous avons la relation :

$$X = x(1+\delta) \text{ avec } |\delta| \leq \mathbf{u} \tag{I.1.6}$$

avec $\mathbf{u} = \epsilon/2$ en arrondi au plus près et $\mathbf{u} = \epsilon$ pour tout autre mode d'arrondi.
Soit X_1 (resp. X_2) un nombre flottant représentant le réel x_1 (resp. x_2)

$$X_i = x_i - \epsilon_i 2^{E_i - p}\alpha_i \text{ pour } i = 1, 2. \tag{I.1.7}$$

Les erreurs d'arrondi sur les opérations arithmétiques ayant pour opérandes X_1 et X_2 sont données dans la suite. Nous noterons $\oplus, \ominus, \otimes, \oslash$ les opérateurs arithmétiques sur ordinateur. Soit E_3, ϵ_3 et α_3 respectivement l'exposant, le signe et l'erreur d'arrondi sur le résultat calculé :

$$X_1 \oplus X_2 = x_1 + x_2 - \epsilon_1 2^{E_1 - p}\alpha_1 - \epsilon_2 2^{E_2 - p}\alpha_2 - \epsilon_3 2^{E_3 - p}\alpha_3. \tag{I.1.8}$$

$$X_1 \ominus X_2 = x_1 - x_2 - \epsilon_1 2^{E_1 - p}\alpha_1 + \epsilon_2 2^{E_2 - p}\alpha_2 - \epsilon_3 2^{E_3 - p}\alpha_3. \tag{I.1.9}$$

$$X_1 \otimes X_2 = x_1 x_2 - \epsilon_1 2^{E_1 - p}\alpha_1 x_2 - \epsilon_2 2^{E_2 - p}\alpha_2 x_1 + \epsilon_1\epsilon_2 2^{E_1 + E_2 - 2p}\alpha_1\alpha_2 - \epsilon_3 2^{E_3 - p}\alpha_3. \tag{I.1.10}$$

En négligeant le terme du second ordre en 2^{-p} de l'équation (I.1.10) nous obtenons alors :

$$X_1 \otimes X_2 = x_1 x_2 - \epsilon_1 2^{E_1 - p}\alpha_1 x_2 - \epsilon_2 2^{E_2 - p}\alpha_2 x_1 - \epsilon_3 2^{E_3 - p}\alpha_3. \tag{I.1.11}$$

En négligeant les termes d'ordre plus grand que 2^{-2p} nous avons également :

$$X_1 \oslash X_2 = \frac{x_1}{x_2} - \epsilon_1 2^{E_1 - p}\frac{\alpha_1}{x_2} + \epsilon_2 2^{E_2 - p}\alpha_2\frac{x_1}{x_2^2} - \epsilon_3 2^{E_3 - p}\alpha_3. \tag{I.1.12}$$

Lors d'une addition d'éléments du même signe nous avons $E_3 = \max(E_1, E_2) + \delta$ avec $\delta = 1$ ou $\delta = 0$. L'erreur relative de l'opération est au maximum de $2^{E_3 - p}$ donc de l'ordre de 2^{-p}.

Dans le cas de la multiplication $E_3 = E_1 + E_2 + \delta$, avec $\delta = -1$ ou $\delta = 0$. Pour la division nous avons $E_3 = E_1 - E_2 + \delta$, avec $\delta = 0$ ou $\delta = 1$.

Dans le cas d'une soustraction d'opérandes du même signe, $E_3 = \max(E_1, E_2) - k$. Si X_1 et X_2 sont proches, alors k peut être important. Dans ce cas l'erreur relative est de l'ordre de 2^{-p+k}. En d'autres termes durant cette opération k bits significatifs ont été perdus. Si k est important, il s'agit d'une élimination catastrophique (ou *cancellation*).

I.1.3.c Propagation des erreurs d'arrondi

Un programme de simulation numérique est une suite d'opérations arithmétiques et d'entrées/sorties. Le résultat R d'un programme après n opérations arithmétiques est modélisable au premier ordre en 2^{-p} par [17] :

$$R \approx r + \sum_{i=1}^{n} g_i(d) 2^{E_i - p}\epsilon_i\alpha_i \tag{I.1.13}$$

avec r le résultat exact, E_i les exposants des résultats intermédiaires, p le nombre de bits de la mantisse en comptant le bit implicite, α_i la perte de précision en raison de l'arrondi. On suppose qu'il s'agit de variables aléatoires uniformément distribuées sur $[-1;1]$, ϵ_i les signes des résultats intermédiaires et $g_i(d)$ des coefficients dépendant des données et du code [18].

Grâce à l'équation (I.1.2), nous pouvons approcher $B_{R,r}$:

$$B_{R,r} \approx -\log_2 \left| \frac{R-r}{r} \right| = p - \log_2 \left| \sum_{i=1}^{n} g_i(d) \frac{\alpha_i}{r} \right|. \tag{I.1.14}$$

Le dernier terme de l'équation (I.1.14) représente la perte de précision dans le calcul de R. Il est indépendant de p. En supposant que le modèle au premier ordre de l'équation (I.1.13) est valide, la perte de précision d'un calcul est donc indépendante de la précision utilisée. Nous considérons ici que le chemin d'exécution est le même entre le calcul de R et de r. En particulier les tests utilisent les mêmes branchements et les boucles le même nombre d'itérations.

I.1.4 Impact des erreurs d'arrondi sur les programmes de simulation numérique

L'arithmétique en virgule flottante peut ainsi créer une perte de précision des résultats difficile à estimer. De plus, les propriétés algébriques ne sont plus les mêmes. Ainsi l'addition et la multiplication flottantes restent commutatives mais ne sont pas associatives. En fonction de l'ordre des opérations, différents résultats peuvent être générés. Dans le cas de la norme IEEE 754 en *simple* précision en arrondi au plus près

$$(-2^{30} \oplus 2^{30}) \oplus 1 = 1 \tag{I.1.15}$$

mais

$$-2^{30} \oplus (2^{30} \oplus 1) = 0. \tag{I.1.16}$$

Dans ce dernier cas, une *absorption* a lieu. Cela arrive lorsque deux nombres d'ordres de grandeur très différents sont additionnés : le plus petit ne peut être représenté et est donc "perdu".

En outre, la multiplication n'est plus distributive par rapport à l'addition. Soient A, B et C des nombres flottants, $A \otimes (B \oplus C)$ peut être différent de $(A \otimes B) \oplus (A \otimes C)$. En *simple* précision si $A = fl(3,3333333)$, $B = fl(12345679)$ et $C = fl(1,2345678)$, les résultats sont respectivement 41152264 et 41152268.

Un résultat informatique peut parfois être différent du résultat exact jusqu'au niveau du signe. Prenons le cas du polynôme à deux variables $p(x,y)$ proposé par Rump [125] :

$$p(x,y) := 1335/4 * y^6 + x^2 * (11 * x^2 * y^2 - y^6 + (-121) * y^4 - 2) + 11/2 * y^8 + x/(2 * y).$$

Sur un processeur Intel Core 2 Quad CPU Q9550 et gcc 6.3.1 sans option de compilation, lorsque $x = 77617$ et $y = 33096$ nous obtenons :

— 6,338253e29 en *simple* précision ;
— 1,17260394005317 en *double* précision ;

— 1,172603940053178631858834904520 18 en *quadruple* précision.

Nous pouvons penser que le résultat en précision supérieure est plus précis dans ce cas. En particulier, les premiers chiffres de la *quadruple* précision sont en commun avec la *double*. Néanmoins, aucun de ces résultats n'est valide. En utilisant une précision infinie, les premiers chiffres du résultat calculé sont $-0,827396059946821$.

I.2 Détection des erreurs d'arrondi

I.2.1 Analyse directe/analyse inverse

I.2.1.a Définitions

Soit un problème mathématique $\mathcal{P}$

$$\mathcal{P} : \text{ pour } y \text{ donné, trouver } x \text{ tel que } F(x) = y$$

avec F une fonction continue entre deux espaces vectoriels.

Définition I.2.1. $\mathcal{P}$ est bien posé dans le sens de Hadamard si $x = F^{-1}(y)$ existe, est unique et si F^{-1} est continue au voisinage de y.

Si cela n'est pas vérifié, alors le problème est mal posé. Si nous calculons dans ce cas une solution soit nous obtenons un résultat :

— sans fondement car aucune solution n'existe ou elle n'est pas unique ;

— faux numériquement car la fonction inverse n'est pas continue au voisinage de la solution.

Nous nous intéresserons à des problèmes bien posés dans la suite de cette section.

Pour mesurer la difficulté de résolution d'un problème $\mathcal{P}$ bien posé nous allons introduire la notion de conditionnement. Il s'agit d'une valeur qui permet d'indiquer la sensibilité d'une solution à la perturbation des données. Comme nous supposons que $\mathcal{P}$ est bien posé, nous pouvons dire que $x = G(y)$ avec $G = F^{-1}$.

Nous notons $\mathcal{E}$ (resp. $\mathcal{S}$) l'espace d'entrée (resp. de sortie) des données du problème. Nous notons $||.||_{\mathcal{E}}$ et $||.||_{\mathcal{S}}$ les normes sur ces espaces. Soient $\epsilon > 0$ nous définissons $\mathcal{P}(\epsilon) \subset \mathcal{E}$ la boule tel que $\mathcal{P}(\epsilon) = \{\Delta_y; ||\Delta_y||_{\mathcal{E}} \leq \epsilon\}$ avec l'ensemble de perturbations Δ_y des données d'entrées y, alors le problème perturbé associé au problème $\mathcal{P}$ est défini par

$$\text{Trouver le } \Delta_x \in \mathcal{S} \text{ tel que } F(x + \Delta_x) = y + \Delta_y \text{ pour une valeur donnée } \Delta_y \in \mathcal{P}(\epsilon).$$

En supposant que les valeurs de x et y sont non nulles, Le conditionnement d'un problème $\mathcal{P}$ pour les données d'entrées y est défini par

$$\text{cond}(\mathcal{P}, y) := \lim_{\epsilon \to 0} \sup_{\Delta_y \in \mathcal{P}(\epsilon), \Delta_y \neq 0} \left\{ \frac{||\Delta_x||_{\mathcal{S}}}{||\Delta_y||_{\mathcal{E}}} \right\}.$$

Un problème sera mal conditionné si la valeur du conditionnement est importante. Sinon il est considéré comme bien conditionné. En effet un conditionnement élevé implique qu'une petite perturbation en entrée fournira généralement un résultat bien

différent.Nous allons maintenant montrer en exemple le cas d'une somme de n termes $y_i, 1 \leq i \leq n$.

Soit une somme de termes $x = \sum_{i=1}^{n} y_i$, nous supposons que $\sum_{i=1}^{n} y_i \neq 0$. L'ensemble $\mathcal{E}$ est doté de la norme $||\Delta_y||_{\mathcal{E}} = \max_{1 \leq i \leq n} \frac{|\Delta_{y_i}|}{|y_i|}$ et l'ensemble $\mathcal{S}$ de la norme $||\Delta_x||_{\mathcal{S}} = \frac{|\Delta_x|}{|x|}$. Comme $|\Delta_x| = |\sum_{i=1}^{n} \Delta_{y_i}| \leq ||\Delta_y||_{\mathcal{E}} \sum_{i=1}^{n} |y_i|$ nous avons

$$\frac{||\Delta_x||_{\mathcal{S}}}{||\Delta_y||_{\mathcal{E}}} \leq \frac{\sum_{i=1}^{n} |y_i|}{|\sum_{i=1}^{n} y_i|}.$$

Cette borne est atteinte dans le cas où $||\Delta_y||_{\mathcal{E}} = \frac{|\Delta_y|}{|y_i|}$ donc [68] :

$$\text{cond}\left(\sum_{i=1}^{n} y_i\right) = \frac{\sum_{i=1}^{n} |y_i|}{|\sum_{i=1}^{n} y_i|}. \tag{I.2.1}$$

Nous pouvons remarquer que d'après la définition du conditionnement, celui-ci est indépendant de l'algorithme utilisé pour résoudre le problème.

I.2.1.b Stabilité d'un algorithme

Pour résoudre un problème, nous utilisons un algorithme qui est composé d'un ensemble d'opérations et de tests. Cela consiste à définir et utiliser la fonction G précédente. Néanmoins, en raison des erreurs d'arrondi, nous ne calculons pas $x = G(y)$ mais plutôt $\widehat{x} = \widehat{G}(y)$.

Le but de l'analyse directe est d'étudier l'exécution de l'algorithme $\widehat{G}$ sur les données d'entrées y. Il y a une estimation de la propagation des erreurs d'arrondi, ce qui permet d'estimer une borne sur la différence entre le résultat exact x et le résultat calculé $\widehat{x}$. Cette différence est aussi appelée l'erreur directe.

Il est difficile de mesurer l'ensemble des erreurs d'arrondi pour toutes les variables intermédiaires d'un code. L'analyse inverse permet de dépasser ce problème en travaillant directement avec la fonction G. Le problème résolu est étudié pour savoir s'il est "proche" du problème initial. Nous cherchons donc la valeur Δ_y telle que $\widehat{x} = G(y + \Delta_y)$. La valeur Δ_y est aussi appelée l'erreur inverse.

L'erreur inverse associée à $\widehat{x} = \widehat{G}(y)$ est le nombre $\eta(\widehat{x})$ défini, lorsqu'il existe, par la relation

$$\eta(\widehat{x}) = \min_{\Delta_y \in \mathcal{E}} \{||\Delta_y||_{\mathcal{E}} : \widehat{x} = G(y + \Delta_y)\}. \tag{I.2.2}$$

Si $\eta(\widehat{x})$ n'existe pas, il est affecté à $+\infty$. Un algorithme est inverse stable pour un problème $\mathcal{P}$ si la solution calculée $\widehat{x}$ a une erreur inverse $\eta(\widehat{x})$ faible, c'est-à-dire de l'ordre de $\mathbf{u}$.

I.2.1.c Précision de la solution

La précision de la solution calculée dépend du conditionnement et de la stabilité de l'algorithme utilisé. Le conditionnement représente les perturbations introduites sur les données et la stabilité est liée aux erreurs introduites par l'algorithme. Au premier ordre nous avons l'approximation suivante :

$$\text{erreur directe} \lesssim \text{conditionnement} \times \text{erreur inverse.} \tag{I.2.3}$$

Dans le cas d'un algorithme inverse stable, nous avons donc :

$$\text{erreur directe} \lesssim \text{conditionnement} \times \mathbf{u}. \tag{I.2.4}$$

Pour la sommation de n nombres p_i, dont le résultat calculé en arrondi au plus près est $\widehat{s}$ et le résultat exact s, nous avons :

$$\frac{|\widehat{s}-s|}{|s|} \leq \gamma_{n-1}(\mathbf{u}) \times \text{cond}\left(\sum_{i=1}^{n} p_i\right) \tag{I.2.5}$$

avec cond défini dans l'équation I.2.1, $n\mathbf{u} < 1$ et γ_n défini par :

$$\gamma_n(\mathbf{u}) = \frac{n\mathbf{u}}{1-n\mathbf{u}} \text{ pour } n \in \mathbb{N}. \tag{I.2.6}$$

Comme $\gamma_{n-1}(\mathbf{u}) \approx (n-1)\mathbf{u}$, l'erreur provient majoritairement de $n\mathbf{u}\,\text{cond}\,(\sum_{i=1}^{n} p_i)$. Nous sommes proches de l'approximation pour un algorithme inverse stable, équation I.2.4.

Cette relation sera rappelée dans le chapitre II qui traite des algorithmes compensés en arrondi dirigé.

I.2.2 L'arithmétique d'intervalles

L'arithmétique d'intervalles [74, 104] consiste à effectuer les calculs non pas sur des nombres réels, mais sur des intervalles. Soient $X = [\underline{x}, \overline{x}]$ et $Y = [\underline{y}, \overline{y}]$, alors nous avons les résultats suivants :

$$X + Y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]. \tag{I.2.7}$$

$$X - Y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]. \tag{I.2.8}$$

$$X \times Y = [\min(\underline{x}\times\underline{y}, \overline{x}\times\overline{y}, \underline{x}\times\overline{y}, \overline{x}\times\underline{y}), \max(\underline{x}\times\underline{y}, \overline{x}\times\overline{y}, \underline{x}\times\overline{y}, \overline{x}\times\underline{y})]. \tag{I.2.9}$$

$$X^2 = [\min(\underline{x}^2, \overline{x}^2), \max(\underline{x}^2, \overline{x}^2)] \text{ si } 0 \notin [\underline{x}, \overline{x}], \tag{I.2.10}$$

$$= [0, \max(\underline{x}^2, \overline{x}^2)] \text{ sinon.} \tag{I.2.11}$$

$$1/Y = [\min(1/\underline{y}, 1/\overline{y}), \max(1/\underline{y}, 1/\overline{y})] \text{ si } 0 \notin [\underline{y}, \overline{y}]. \tag{I.2.12}$$

$$X/Y = [\underline{x}, \overline{x}] \times (1/[\underline{y}, \overline{y}]) \text{ si } 0 \notin [\underline{y}, \overline{y}]. \tag{I.2.13}$$

Plus généralement, pour une fonction mathématique f, l'intervalle résultat est un intervalle obtenu en appliquant la fonction f à tous les éléments de l'intervalle d'entrée X et en prenant l'enveloppe convexe du résultat :

$$\forall x \in X, f(x) \in f(X). \tag{I.2.14}$$

Ainsi nous avons par exemple $\exp[\underline{x}, \overline{x}] = [\exp(\underline{x}), \exp(\overline{x})]$.

Le principe de l'arithmétique d'intervalles sur ordinateur est donc de représenter un nombre non par une approximation, mais par un intervalle le contenant. Dans le but de calculer les bornes de ces intervalles, nous pouvons utiliser les arrondis dirigés vers plus ou moins l'infini. Soient $X = [\underline{x}, \overline{x}]$ et $Y = [\underline{y}, \overline{y}]$ nous aurons par exemple pour l'addition :

$$X + Y = [\nabla(\underline{x} + \underline{y}), \Delta(\overline{x} + \overline{y})] \tag{I.2.15}$$

avec ∇ (resp. Δ) l'arrondi vers moins (resp. plus) l'infini.

L'arithmétique d'intervalles à tendance à surestimer la taille de l'intervalle résultat. Nous pouvons citer le cas de l'opération $X - X$ avec $X = [0, 1]$. Dans ce cas le résultat sera $[-1, 1]$ et non $[0, 0]$ qui serait le résultat exact.

Nous pouvons citer comme bibliothèque d'arithmétique d'intervalles C-XSC [70], Boost Interval Arithmetic Library [11] ou INTLAB [126].

I.2.3 Expansions de Taylor

Nous présenterons ici les expansions de Taylor puis un outil les utilisant et enfin une amélioration de cette méthode : les expansions de Taylor symboliques.

Les expansions de Taylor sont une autre possibilité pour estimer la propagation des erreurs d'arrondi [93]. Pour simplifier les explications nous nous plaçons dans un cas en une dimension, la méthode étant équivalente en dimension n.

Soit $[a, b]$, avec a et b des nombres réels, l'intervalle d'entrée d'un programme qui fournira alors un résultat flottant. Alors, il peut être assimilé à une fonction $f : [a, b] \to \mathbb{F}$. Supposons que $\exists c \in [a, b]$ tel que f soit n fois dérivable au voisinage de c, avec $n \geq 1$. Alors $\forall x \in [a, b]$:

$$\begin{aligned} f(x) &= \sum_{i=0}^{n} \frac{f^{(i)}(c)}{i!}(x - c)^i + o((x - c)^n) \\ &= P_f(x - c) + o((x - c)^n) \end{aligned}$$

avec P_f un polynôme de degré n. Ainsi un programme est assimilable à un polynôme. La différence entre la fonction f et le polynôme P_f est une fonction continue et bornée sur $[a, b]$. Nous notons I_r un intervalle contenant les bornes minimales et maximales de cette fonction, nous l'appellerons l'intervalle du reste. Nous avons alors $\forall x \in [a, b], \exists c \in [a, b]$ tel que f soit n fois dérivable au voisinage de c :

$$f(x) \in P_f(x - c) + I_r. \tag{I.2.16}$$

Définition I.2.2. Le couple (P_f, I_r) est une expansion de Taylor de f si l'équation I.2.16 est vérifiée.

Une arithmétique sur les expansions de Taylor peut être définie. Par exemple, la somme des fonctions f et g ayant pour expansions de Taylor respectivement (P_f, I_f) et (P_g, I_g) a pour résultat $(P_f + P_g, I_f + I_g)$.

Le calcul de l'intervalle résultat est effectué à l'aide de l'arithmétique d'intervalles. Il est ainsi possible de borner les erreurs d'arrondi.

COSY INFINITY

COSY INFINITY a été développé pour calculer les faisceaux de particules physiques et en particulier dans les accélérateurs de particules [5]. Pour la résolution des équations différentielles et des équations aux dérivées partielles, les expansions de Taylor sont utilisées dans le but d'obtenir des résultats précis. COSY utilise son propre langage de programmation, le COSYScript qui est un langage orienté objet polymorphe dérivé du PASCAL. Dans le but d'utiliser les expansions de Taylor définies dans COSY INFINITY dans d'autres programmes, des interfaces en C, C++, Fortran77 et Fortran90 existent [94].

Nous pouvons noter que dans COSY INFINITY, la multiplication d'un scalaire avec une expansion de Taylor et la somme et la multiplication de deux expansions de Taylor retournent un intervalle du reste contenant toutes les erreurs d'arrondi ou de troncatures commises [121].

Expansions de Taylor symboliques

Les expansions de Taylor déterminent l'intervalle du reste grâce à l'arithmétique d'intervalles. Comme nous l'avons vu dans la section I.2.2, les intervalles résultats sont régulièrement surestimés. Pour éviter cela, Solovyev et al. proposent d'utiliser des expansions de Taylor symboliques [130]. Plutôt que de calculer le couple (P_f, I_r) des expansions de Taylor grâce à l'arithmétique à virgule flottante, il est obtenu grâce au calcul symbolique, voir section I.3.2.a. Il est ainsi possible de réduire la taille de I_r et donc d'améliorer l'estimation de l'erreur commise sur le résultat.

Cette approche a amené au développement de l'outil FPTaylor qui implémente les expansions de Taylor symboliques. Pour calculer les bornes de l'intervalle du reste à la fin de l'exécution, il peut utiliser sa propre méthode de recherche de maximum et de minimum fondée sur un algorithme de séparation et évaluation ou il peut utiliser des outils externes tels que Maxima [97] pour simplifier les expressions symboliques, ou les optimiseurs Gelpia [47] et Z3 [31] pour la recherche du maximum global.

I.2.4 Analyse statique

L'analyse statique d'un code permet d'obtenir des informations sur un programme sans avoir à l'exécuter. Il est par exemple possible de détecter des erreurs de programmation ou d'obtenir la valeur maximum et minimum du résultat en fonction d'un ensemble d'entrés. Nous pouvons donc savoir à l'avance si une division par zéro peut être présente ou si nous allons chercher à calculer la racine carrée d'un nombre négatif en raison d'une erreur dans le code.

Fluctuat [49] étudie les codes écrits en C ou en Ada 95 et nous permet d'avoir les informations suivantes :

— bornes d'erreurs sur les variables internes du programme ;
— propagation des erreurs d'arrondi ;
— preuves fonctionnelles (erreur de méthode, erreur d'implémentation de certaines expressions) ;
— génération des pires cas ;
— analyse de sensibilité des entrées.

Les analyses statiques sont utilisées en particulier dans les systèmes critiques en aéronautique, dans le nucléaire ou l'aérospatiale. Cette méthode nécessite une version modifiée de l'arithmétique d'intervalle pour réduire la surestimation des erreurs d'arrondi, et pouvoir détecter les lignes créant des instabilités dans le code. À notre connaissance, cette approche n'a pas permis jusqu'à présent la validation numérique de code de simulation de taille importante.

I.2.5 Approche probabiliste

L'approche probabiliste consiste à utiliser plusieurs exécutions d'un même programme pour estimer la précision des résultats. Il existe deux grandes approches pour effectuer les calculs :

— synchrone : les opérations sont effectuées le nombre de fois défini avant de passer à la suivante ;
— asynchrone : les exécutions du programme sont effectuées les unes après les autres.

Nous parlerons tout d'abord de la méthode CESTAC, puis de l'Arithmétique de Monte-Carlo et enfin nous décrirons trois outils probabilistes que sont la MCALIB, verificarlo et verrou.

Méthode CESTAC

La méthode CESTAC (Contrôle et Estimation STochastique des Arrondis de Calculs) [139] exécute un même programme N fois en utilisant un mode d'arrondi aléatoire : l'arrondi est vers plus ou moins l'infini avec une probabilité de 0,5. Grâce aux différents résultats obtenus, il est possible d'estimer le nombre de chiffres significatifs exacts du résultat. Ainsi, les chiffres en commun entre les différents résultats peuvent être considérés comme exacts. Les autres sont impactés par les erreurs d'arrondi. Dans la pratique, trois exécutions permettent d'estimer le nombre de chiffres significatifs exacts du résultat.

Cette approche sera discutée plus longuement dans la section I.4.

Arithmétique de Monte-Carlo

L'arithmétique de Monte-Carlo [3] [117] introduit des perturbations au niveau des opérations soit sur les opérandes en entrée, sur le résultat ou sur les deux pour étudier la propagation des erreurs d'arrondi. Soit $\bullet$ l'opération flottante représentant l'opération réelle $\diamond$ avec $\diamond \in \{+, -, \times, /\}$. Alors il est possible d'étudier l'erreur commise sur l'opération réelle $\diamond$:

3. Parfois abrégé *MCA* pour *Monte-Carlo Arithmetic*

1. $x \bullet y = (x \diamond y) + \epsilon$ qui met en avant l'erreur directe ;
2. $x \bullet y = (x + \epsilon_x) \diamond (y + \epsilon_y)$ qui met en avant l'erreur inverse ;
3. $x \bullet y = ((x + \epsilon_x) \diamond (y + \epsilon_y)) + \epsilon$ qui met en avant l'erreur directe et l'erreur inverse.

Chacune des variables d'erreurs ϵ peut être modélisée par une variable aléatoire. Ainsi une expression du type $x + \epsilon$ peut être remplacée par la fonction *inexact* :

$$inexact(x) = x + 2^{e_x - t}\xi \tag{I.2.17}$$

avec e_x l'exposant du nombre flottant x, t la précision virtuelle de $inexact(x)$ qui peut être différente de la précision réelle p et ξ une variable aléatoire distribuée sur $[-\frac{1}{2}, \frac{1}{2}]$. La fonction d'arrondi est notée par la suite $round()$. Une opération flottante $x \bullet y$ avec $\bullet \in \{+, -, \times, /\}$ sera transformée en une des opérations de l'arithmétique de Monte-Carlo suivantes :

1. $x \bullet y = round(inexact(x \diamond y))$ qui permet le calcul de l'erreur directe (appelé ici *output precision bounding* ou *Random Rounding* dans le cas de verificarlo (voir page 14)) ;
2. $x \bullet y = round(inexact(x) \diamond inexact(y))$ qui permet le calcul de l'erreur inverse (appelé ici *input precision bounding* ou *Precision Bounding* dans le cas de verificarlo) ;
3. $x \bullet y = round(inexact(inexact(x) \diamond inexact(y)))$ qui permet le calcul de l'erreur directe et l'erreur inverse (appelé ici *input and output precision bounding* ou *full MCA*).

Dans le cas où $t = p$, Parker propose l'utilisation de trois arrondis différents pour définir la distribution des erreurs [117, Section 6.4] :

— arrondi au plus près : tous les arrondis se font au plus près ;
— arrondi aléatoire vers plus ou moins l'infini avec une probabilité de 0,5, cet arrondi est celui qui est utilisé dans la méthode CESTAC ;
— arrondi aléatoire vers plus ou moins l'infini avec une probabilité proportionnelle à la distance entre le résultat exact et les arrondis. Dans les faits, le résultat exact n'est pas connu et un calcul en précision supérieure est utilisé. La façon dont cela est mis en place dépend de l'outil. Dans le cas de verificarlo, outil que nous présentons par la suite, il a été choisi d'effectuer les calculs en *quadruple* précision.

L'arithmétique de Monte-Carlo nécessite un grand nombre d'exécutions du programme. Il n'y a pas de consensus sur le nombre préconisé, plus le nombre est important plus l'estimation sera précise. Le nombre de chiffre décimaux corrects est alors estimé par :

$$C_{MCA} \approx -\log_{10}\left(\frac{\sigma}{\mu}\right) \tag{I.2.18}$$

avec μ la moyenne et σ l'écart-type des résultats.

MCALIB :

Une implémentation de l'arithmétique de Monte-Carlo est la bibliothèque MCALIB [46]. Les développeurs de cet outil recommandent d'effectuer au moins 50 exécutions. Cette valeur provient de tests expérimentaux. Ce nombre important d'exécutions limite les performances globales de l'outil.

Pour pouvoir utiliser l'arrondi aléatoire proportionnel, MCALIB fait appel à la bibliothèque MPFR [45] ce qui limite d'autant plus les performances.

Verificarlo : Verificarlo [35] est un outil historiquement fondé sur une version modifiée de MCALIB et sur LLVM [4] [88]. La bibliothèque MPFR est ici utilisée pour calculer les résultats avec une précision supérieure. Ce résultat est ensuite arrondi en fonction de la précision virtuelle.

Les défauts de MCALIB s'appliquent également à verificarlo : un nombre d'exécutions supérieur à 50, valeur reprise des tests de MCALIB, et l'utilisation de MPFR qui dégrade d'autant plus les performances. Ainsi verificarlo avec 128 échantillons est 285 fois plus lent que l'ancienne version de CADNA, voir section I.4.3, qui ne permettait pas d'optimisation à la compilation sur le cas de la sommation compensée de Kahan pour un vecteur de 1000000 éléments [35, tableau 1].

Maintenant, verificarlo dispose d'un backend [5] fondée sur MPFR, un sur la *quadruple* précision pour simuler les résultats en précision supérieure [6] et enfin un utilisant des transformations exactes tout comme verrou [7].

Verificarlo utilise la chaîne de compilation de LLVM en utilisant en particulier le compilateur clang et ses dérivés pour les autres langages de programmation. Cela permet de déterminer les options de compilation pouvant amener une instabilité du code, ainsi que d'utiliser des bibliothèques externes dont les sources sont disponibles. Clang n'est pas actuellement un compilateur de référence mais un travail important de la communauté de ses développeurs est en cours pour arriver au niveau de compilateurs comme GCC [48] et ICC [75].

Les exécutions asynchrones rendent la recherche d'instabilités numériques extrêmement difficile. Verificarlo permet de déterminer des instabilités dans les fonctions. En effet, l'utilisateur peut choisir les fonctions instrumentées ou non et localiser ainsi des instabilités. Ainsi, le résultat de l'outil est une estimation de la précision des résultats d'un code.

Verrou :

Verrou [42] se situe entre la méthode CESTAC et l'arithmétique de Monte-Carlo asynchrone. Cet outil se base sur le *framework* de valgrind [108] pour instrumenter les binaires. Néanmoins, valgrind ne permet pas le changement du mode d'arrondi qui est fixé au plus près. Pour émuler un changement de mode d'arrondi, les transformations exactes sont utilisées, voir section I.3.1. Cela a permis l'implémentation des arrondis

4. Il s'agit d'une infrastructure de compilation permettant la modification d'un code. Cela permet d'introduire dans le code des opérations supplémentaires.

5. backend : dans le contexte de verificarlo, il s'agit des instructions intermédiaires introduites lors d'une opération pour modifier son résultat et tester la stabilité numérique du code. Le backend choisi peut avoir un impact sur la performance ou sur la qualité du résultat numérique fourni.

6. Les deux backends disposent du calcul de l'erreur directe (ou *RR*), de l'erreur inverse (ou *PB*) et du calcul *full MCA*

7. *cf. section suivante*

vers plus ou moins l'infini. Dans un premier temps, l'outil ne disposait que de l'arrondi aléatoire de la méthode CESTAC. Grâce à l'utilisation des transformations exactes, la distance entre le résultat de l'opération et les arrondis vers plus ou moins l'infini est connue et donc le mode d'arrondi aléatoire avec une probabilité proportionelle a été implémenté.

Valgrind fonctionne sur le binaire d'un programme. Il est donc possible d'instrumenter des codes dont les sources ne sont pas disponibles, par exemple certaines bibliothèques externes. De plus, aucune modification du programme n'est effectuée, l'implémentation des arrondis aléatoires étant faite en interne dans l'outil.

Les développeurs de verrou limitent le nombre d'exécutions à celui de la méthode CESTAC, donc trois, dans un premier temps. Après avoir corrigé un certain nombre d'instabilités, il peut être intéressant d'augmenter le nombre d'exécutions pour améliorer l'intervalle de confiance du résultat.

La recherche des opérations instables est difficile en raison des exécutions asynchrones et l'utilisation d'un algorithme spécialisé est nécessaire. Des travaux en cours utilisent l'algorithme de Delta-Debug [141] et une étude de la couverture de code pour identifier les branchements instables [43].

I.3 Méthodes d'amélioration de la précision des calculs

Une fois les erreurs d'arrondi identifiées, il existe des méthodes pour essayer de les supprimer ou du moins de les limiter. Nous allons voir ici le cas général qui seront développé dans les chapitres suivants pour les transformations exactes dans le cas des algorithmes compensés (cf. chapitre II) et avec les expansions de taille fixe (cf. partie IV.3.1).

I.3.1 Transformations exactes

Les transformations exactes [8] sont des algorithmes permettant de calculer l'erreur commise lors des opérations de base. Pour l'opération $\mathrm{o} \in \{+, \times\}$, nous avons alors la propriété $\forall (a,b) \in \mathbb{F}^2, \exists (x,y) \in \mathbb{F}^2$ tel que $x = fl(a \mathrm{\ o\ } b)$ et $a \mathrm{\ o\ } b = x + y$. La division dispose d'une transformation exacte particulière dans le sens où nous calculerons l'erreur produite sous forme de reste : si nous avons $fl(a/b) = x$, alors nous calculons $y \in \mathbb{F}$ tel que $a = x \times b + y$ [52].

Ces algorithmes sont particulièrement utiles dans le cadre des algorithmes compensés, cf. chapitre II, ainsi que pour l'utilisation des *double-double*, cf. partie IV.3.1. Cela permet d'améliorer la précision des calculs au sein des programmes. Tout d'abord nous nous intéressons aux algorithmes de transformations exactes de l'addition/soustraction et ensuite à ceux de la multiplication. Dans tous les cas, nous supposerons qu'aucun *overflow* ne survient.

Ces algorithmes utilisent les propriétés des nombres flottants et en particulier le lemme de Sterbenz [133] :

Lemme I.3.1 (Sterbenz)**.** *Dans le cas de l'arithmétique à virgule flottante, avec la présence de nombres dénormalisés, si x et y sont des nombres flottants finis tel que*

8. *Error-Free Transformation en anglais*

$y/2 \leq x \leq 2y$, alors $x - y$ est exactement représentable.

I.3.1.a Transformation exacte de l'addition

L'algorithme I.1 (`FastTwoSum`) permet de calculer le résultat d'une somme et l'erreur commise sur celle-ci en utilisant uniquement trois opérations flottantes et un branchement[9]. Il a été introduit par Dekker en 1971 [32] lorsque le calcul d'une opération était aussi coûteux qu'un branchement.

Algorithme I.1 `FastTwoSum` : Transformation exacte de la somme de deux flottants en arrondi au plus près

fonction $[c, d] =$ `FastTwoSum`(a, b)

1: **si** $|b| > |a|$ **alors**
2: échanger a et b
3: **fin si**
4: $c \leftarrow a + b$
5: $z \leftarrow c - a$
6: $d \leftarrow b - z$

L'algorithme I.2 (`TwoSum`) introduit par Knuth en 1969 [85] est maintenant le plus utilisé. Il nécessite six opérations flottantes pour calculer l'erreur commise lors de l'addition de deux nombres flottants.

Algorithme I.2 `TwoSum` : Transformation exacte de la somme de deux flottants en arrondi au plus près

fonction $[c, d] =$ `TwoSum`(a, b)

1: $c \leftarrow a + b$
2: $a_1 \leftarrow c - b$
3: $b_1 \leftarrow c - a_1$
4: $\delta_a \leftarrow a - a_1$
5: $\delta_b \leftarrow b - b_1$
6: $d \leftarrow \delta_a + \delta_b$

Dans les deux cas précédents, l'arrondi au plus près *doit* être utilisé. Utiliser un arrondi dirigé ne permet pas nécessairement d'obtenir la relation d'égalité $a + b = c + d$, avec a et b les nombres flottants en entrée, c le résultat flottant de $a + b$ et d l'erreur flottante calculée. Dans le cas où nous souhaitons utiliser un arrondi dirigé, il existe l'algorithme de sommation de Priest [120, p.14-15] : `TwoSumPriest` (algorithme I.3). Celui-ci est défini pour tous les modes d'arrondis dirigés et nécessite deux tests et sept opérations flottantes. Il est donc plus coûteux que `FastTwoSum`, qui nécessite un test et trois opérations flottantes, et que `TwoSum`, qui, lui, nécessite six opérations flottantes.

`TwoSumPriest` permet donc de vérifier $a + b = c + d$ avec tous les modes d'arrondi et donc les arrondis dirigés en particulier. Néanmoins, nous n'avons pas nécessairement $fl(a + b) = c$.

9. Il est possible de supprimer le test dans certains cas, par exemple lorsque nous connaissons à l'avance l'ordre de grandeur et les signes des valeurs.

Algorithme I.3 `TwoSumPriest` : Transformation exacte de la somme de deux flottants

fonction $[c, d] =$ `TwoSumPriest`(a, b)

si $|b| > |a|$ **alors**
  échanger a et b
fin si
$c \leftarrow a + b$
$e \leftarrow c - a$
$g \leftarrow c - e$
$h \leftarrow g - a$
$f \leftarrow b - h$
$d \leftarrow f - e$
si $d + e \neq f$ **alors**
  $c \leftarrow a$
  $d \leftarrow b$
fin si

I.3.1.b Transformation exacte de la multiplication

Nous supposerons dans cette section qu'aucun *underflow* ne survient.

Pour pouvoir calculer l'erreur commise lors d'un produit sans utiliser de `FMA` [10], il est nécessaire d'avoir une fonction séparant un nombre flottant x en deux nombres flottants ayant des mantisses de même longueur, x_h et x_l, tels que :

$$x = x_h + x_l.$$

Pour cela, nous pouvons utiliser l'algorithme de Veltkamp [32] (algorithme I.4). β est la base utilisée pour définir les nombres flottants. s est le nombre de bits dans la mantisse divisé par deux et arrondi à l'entier supérieur. La valeur de C dans l'algorithme I.4 dépend donc du type utilisé. Dans le cas de variables en double précision binaire, $\beta = 2$ et $s = \lceil 53/2 \rceil$ donc $C = 134217729$.

Algorithme I.4 Algorithme de séparation de Veltkamp

Fonction $[x_h, x_l] = split(x)$

Nécessite: $C \leftarrow \beta^s + 1$
$\gamma \leftarrow C \times x$
$\delta \leftarrow x - \gamma$
$x_h \leftarrow \gamma + \delta$
$x_l \leftarrow x - x_h$

Pour pouvoir calculer la transformation exacte de la multiplication, nous pouvons utiliser la fonction `TwoProd` [32] (algorithme I.5). Elle dépend de la fonction `split` définie précédemment, algorithme I.4.

Malheureusement, `TwoProd` nécessite dix-sept opérations flottantes [11]. Si les `FMA` sont

10. Fused-Multiply Add : une multiplication et une addition effectuées lors d'un même cycle processeur et n'utilisant qu'un arrondi

11. sept multiplications et dix additions/soustractions

Algorithme I.5 `TwoProd` : Transformation exacte du produit de deux flottants

Fonction $[r1, r2] =$ `TwoProd`(a, b)

$(x_h, x_l) \leftarrow Split(x)$
$(y_h, y_l) \leftarrow Split(y)$
$r_1 \leftarrow x \times y$
$t_1 \leftarrow -r_1 + x_h \times y_h$
$t_2 \leftarrow t_1 + x_h \times y_l$
$t_3 \leftarrow t_2 + x_l \times y_h$
$r_2 \leftarrow t_3 + x_l \times y_l$

disponibles sur le processeur, la fonction `TwoProdFMA` [107, p. 152] est plus performante. Comme nous pouvons le constater dans l'algorithme I.6, elle ne nécessite que deux opérations pour obtenir l'erreur sur la multiplication. De plus, l'arrondi au plus près n'est pas nécessaire lors de cette transformation exacte. Ainsi quel que soit l'arrondi, l'erreur commise sur x sera exactement représentable.

Algorithme I.6 `TwoProdFMA` : Transformation exacte du produit de deux flottants près avec utilisation d'une opération `FMA`

fonction $[x, y] =$ `TwoProdFMA`(a, b)

1: $x \leftarrow a \times b$
2: $y \leftarrow$ `FMA`$(a, b, -x)$

I.3.2 Amélioration de la précision au niveau logiciel

Pour corriger des instabilités numériques dans un code, il existe d'autres options que l'utilisation d'algorithmes compensés. Il est en effet possible d'utiliser des bibliothèques logicielles permettant d'améliorer la précision des calculs. Nous parlerons tout d'abord de l'approche par calcul symbolique, puis la possibilité d'utiliser des nombres flottants de taille arbitraire et enfin les expansions qui peuvent être de taille arbitraire ou fixe.

I.3.2.a Calcul symbolique

Le but du calcul symbolique est de manipuler des expressions mathématiques plutôt que les résultats numériques. Le résultat d'une opération sera donc mathématiquement vrai. Si nous prenons la fonction $f(x) = x^2 + \cos(x)$ alors le résultat pour $x = 2$ sera $f(2) = 4 + \cos(2)$. Cette expression sera réutilisée dans les calculs suivants. Pour obtenir de meilleures précisions, les nombres non entiers sont définis sous forme de fractions rationnelles. L'utilisateur peut lorsqu'il le souhaite évaluer la précision des expressions et ainsi obtenir un résultat facilement lisible.

Un certain nombre de logiciels peuvent effectuer ce genre de calculs. Nous pouvons citer Matlab avec la "Symbolic Math Toolbox" [96], ou le paquet "symbolic" de GNU Octave [38]. Il est également possible d'utiliser le calcul symbolique avec différents langages de programmation tel que C++ [64] ou encore Python [98].

Le calcul symbolique reste peu utilisé en industrie. En effet, la contrepartie au gain en précision est un temps de calcul beaucoup plus important et une utilisation

plus importante de la mémoire. La méthode du pivot de Gauss peut amener à la manipulation de coefficients rationnels de tailles importantes.

I.3.2.b Calcul flottant en précision arbitraire

La précision arbitraire consiste à sortir des formats de la norme IEEE 754 tout en gardant un signe, un exposant et une mantisse. Il existe par exemple les bibliothèques ARPREC [4], et MPFR [45] fondée sur GMP [61].

La bibliothèque ARPREC permet de calculer des résultats avec une précision jusqu'à 1000 chiffres décimaux. Il est possible de calculer des opérations arithmétiques classiques, des fonctions transcendantales ou combinatoires, des sommes de séries ...

Avec MPFR, un nombre de bits de mantisse est choisi par l'utilisateur. La précision des résultats peut donc être améliorée par rapport à la norme IEEE 754 lorsque le nombre de bits est supérieur à 113, le nombre de bits maximum d'une mantisse dans la norme. MPFR permet d'utiliser les opérations arithmétiques ainsi que les fonctions élémentaires, par exemple $\log, \exp, \cos, \sin, \cdots$ avec les modes d'arrondi définis dans la norme IEEE 754.

I.3.2.c Expansions de taille arbitraire

Le but des expansions arbitraires est d'utiliser plusieurs nombres pour en représenter un. Ainsi un nombre x est représenté par une expansion telle que $x = \sum_{i=1}^{m} x_i$ avec $x_i \in \mathbb{F}$ avec un non-recouvrement des x_i. Priest [120] et Shewchuk [129] proposent de ne pas fixer le nombre m de flottants dans l'expansion. Dans ce cas lorsqu'une multiplication entre $x = \sum_{i=1}^{m} x_i$ et $y = \sum_{i=1}^{n} y_i$ est effectuée, le résultat peut être de $n \times m$ termes. La différence majeure entre les deux approches est la définition du non recouvrement. Priest considère qu'il y a non recouvrement de x et y ($|x| \leq |y|$) lorsque le bit le plus significatif de x est inférieur au bit le moins significatif de y. Pour Shewchuck, il faut que ce soit le bit non nul le plus significatif de x qui soit inférieur au bit non nul le moins significatif de y. Rump et al. proposent d'utiliser ce format pour améliorer la précision lors d'additions [127].

I.3.2.d Expansions de taille fixe

Bailey propose une approche similaire à celle de Priest et Shewchuck, mais limite le nombre d'éléments dans l'expansion [67]. Il se limite ainsi à deux (*double-double*) ou quatre (*quad-double*) nombres en *double* précision. Joldes et al. ont précisé les bornes d'erreurs sur les résultats des opérations lorsque deux nombres sont utilisés [83]. Ils ont également proposé de nouveaux algorithmes pour calculer plus précisément les opérations élémentaires.

Les *double-double* seront étudiés de manière plus détaillée dans la section IV.3.1.

I.3.2.e Arithmétique d'intervalles en précision arbitraire

MPFI [12] [122] est une bibliothèque d'arithmétique d'intervalles, *cf.* section I.2.2, écrite en C qui permet l'utilisation de la précision arbitraire. Nous avons ainsi les

12. Multiple Precision Floating-point Interval

avantages de l'arithmétique d'intervalles, en particulier un résultat garanti, ainsi que ceux de la précision arbitraire, et donc un résultat plus précis. MPFI se base sur la bibliothèque MPFR. Cela permet d'avoir une bibliothèque portable et qui respecte la norme IEEE 754.

I.4 Arithmétique Stochastique Discrète et implémentation

I.4.1 Méthode CESTAC

I.4.1.a Principe

Comme vu dans la section I.2.5, la méthode CESTAC est une méthode probabiliste pour étudier l'impact des erreurs d'arrondi.

Pour ce faire, la méthode CESTAC utilise le mode d'arrondi aléatoire. Celui-ci consiste à choisir aléatoirement l'arrondi vers plus (resp. moins) l'infini avec une probabilité de 0,5. Ces perturbations impliquent que le résultat R au premier ordre en 2^{-p} d'un programme est modélisable par [17] :

$$R \approx r + \sum_{i=1}^{n} g_i(d) 2^{E_i - p} \epsilon_i (\alpha_i - h_i) \tag{I.4.1}$$

avec n le nombre d'opérations, r le résultat exact, E_i les exposants des résultats intermédiaires, p le nombre de bits de la mantisse en comptant le bit implicite, ϵ_i les signes des résultats intermédiaires, α_i la perte de précision en raison de l'arrondi, h_i les perturbations aléatoires et $g_i(d)$ des coefficients dépendant des données et du code.

Comme nous l'avons vu dans la section I.2.5, nous avons besoin de plusieurs exécutions pour estimer la qualité numérique du programme. Notons N le nombre d'exécutions. Nous avons alors N résultats R_i avec $1 \leq i \leq N$ où les $(\alpha_i - h_i)$ sont des variables aléatoires que nous supposons indépendantes distribuées uniformément sur $[-1, 1]$. Nous notons $\overline{R}$ la moyenne des R_i. Nous avons donc :

— l'espérance mathématique de la variable R est le résultat mathématique r ;

— la distribution de R est quasi-gaussienne.

Un intervalle de confiance pour l'espérance d'une gaussienne à partir d'une probabilité est fourni par le test de Student. Le nombre de chiffres significatifs exacts de $\overline{R}$ sous une probabilité β est :

$$C_{\overline{R}} = \log_{10} \left(\frac{\sqrt{N} |\overline{R}|}{\sigma \tau_\beta} \right) \tag{I.4.2}$$

avec τ_β la valeur du test de Student pour une distribution à $N-1$ degrés de liberté et une probabilité de $1-\beta$, $\sigma^2 = \frac{1}{N-1} \sum_{i=1}^{n} (R_i - \overline{R})^2$.

I.4.1.b Validité de la méthode

La méthode est valide si deux hypothèses sont respectées [17] :

1. les erreurs d'arrondi $(\alpha_i - h_i)$ sont indépendantes centrées et uniformément réparties ;

2. l'approximation au premier ordre en 2^{-p} dans la modélisation du résultat informatique R est valide.

L'hypothèse 1 est en pratique rarement valide. Les variables aléatoires α_i ne sont pas rigoureusement centrées. Le test de Student fournira une estimation biaisée du résultat exact r. Cette erreur est de l'ordre de quelques σ, l'écart-type de R [22]. Ainsi, nous pouvons considérer que le résultat est valide si nous le tenons pour exact à un chiffre décimal près [89, tableau 3.1].

L'hypothèse 2 n'est plus vérifiée lors :

1. d'une division par un nombre non significatif ;
2. d'une multiplication de deux nombres non significatifs.

Il faut donc pouvoir détecter ces cas durant l'exécution d'un code.

Lorsque $\beta = 0,05$ et $N = 3$, la probabilité de surestimer la précision d'un chiffre est de 0,00054 et la probabilité de la sous-estimer d'un chiffre est de 0,29 [19, 137].

I.4.2 Arithmétique Stochastique Discrète

La détection des instabilités numériques peut être effectuée grâce à une implémentation synchrone de la méthode CESTAC. Un nombre X obtenu par la méthode CESTAC synchrone avec $N = 3$ est défini par le triplet (X_1, X_2, X_3). L'opération $\mathrm{o} \in \{+, -, \times, /\}$ entre deux nombres X et Y fournis par la méthode CESTAC est définie par X o $Y = (X_1 \diamond Y_1, X_2 \diamond Y_2, X_3 \diamond Y_3)$ où $\diamond$ est l'opération flottante correspondante en arrondi aléatoire.

Nous commencerons par la définition du zéro informatique [136] :

Définition I.4.1. Un triplet X fourni par la méthode CESTAC est un zéro informatique noté @.0 si et seulement si $C_{\overline{X}} \leq 0$ ou $\forall i, X_i = 0$.

Nous avons alors les relations stochastiques suivantes :

Définition I.4.2. X est stochastiquement égal à Y si et seulement si $X - Y = @.0$

Définition I.4.3. X est stochastiquement strictement supérieur à Y si et seulement si $\overline{X} > \overline{Y}$ et $X - Y \neq @.0$

Définition I.4.4. X est stochastiquement supérieur ou égal à Y si et seulement si $\overline{X} \geq \overline{Y}$ ou $X - Y = @.0$

L'Arithmétique Stochastique Discrète [13] est définie comme l'association de la méthode CESTAC synchrone, du zéro informatique et des relations stochastiques.

I.4.3 La bibliothèque CADNA

CADNA (Control of Accuracy and Debugging for Numerical Applications) est une bibliothèque de validation numérique qui permet d'estimer les erreurs d'arrondi

13. ASD ou DSA pour Discrete Stochastic Arithmetic

dans un programme de simulation numérique grâce à l'implémentation de l'Arithmétique Stochastique Discrète. Plusieurs versions sont disponibles entre autres la version C/C++ [39, 87] et Fortran [76] [14].

Grâce à cette bibliothèque il est possible :

— de mesurer l'effet de la propagation des erreurs d'arrondi ;
— de détecter les instabilités numériques.

Pour cela CADNA remplace les types flottants classiques par les types stochastiques. Ceux-ci sont composés de trois nombres flottants et d'un entier. Ce dernier permet de garder en mémoire la précision d'un nombre qui n'est pas modifié. Les trois nombres flottants représentent le triplet (X_1, X_2, X_3) et sont appelés x, y et z. Les opérateurs arithmétiques, les relations d'ordre, les fonctions mathématiques sont redéfinis pour être utilisés avec les types stochastiques. Des fonctions particulières permettent de contrôler le nombre de chiffres significatifs exacts des résultats. De plus il est possible de ne détecter que certaines instabilités numériques. Les principaux modes de détection des instabilités sont :

— aucune instabilité : nous utilisons l'ASD sans nous préoccuper des instabilités dans le code ;
— auto-validation : nous demandons de vérifier qu'aucune division par un nombre non significatif ou multiplication entre deux nombres non significatifs n'ait lieu. Dans le cas contraire, un message sera affiché à la fin de l'exécution du programme ;
— toutes les instabilités : nous recherchons toutes les instabilités au sein du code.

Des modifications du code sont à prévoir pour pouvoir utiliser CADNA. En particulier, il est nécessaire d'indiquer le début et la fin de l'instrumentation du code, modifier éventuellement les entrées et les sorties ainsi que remplacer les types flottants par les types stochastiques.

La bibliothèque CADNA reçoit régulièrement des améliorations. Il est ainsi possible d'utiliser CADNA sur carte graphique grâce à CUDA [15] [77, 79] et également de contrôler des codes utilisant OpenMP [40] et MPI [100, 101]. Une étude a également été réalisée sur la parallélisation des trois exécutions de CADNA [78].

Il est à noter qu'il existe une version multiprécision sous le nom de SAM [58] [16] ainsi qu'une version pour l'arithmétique à virgule fixe [20].

CADNA a été utilisée dans des contextes académiques et industriels pour des domaines d'application variés. Nous pouvons citer par exemple l'astrophysique [80], la physique atomique [128], la science du climat [12, 78], l'hydraulique [101], la géophysique [79] ainsi que la neutronique [57].

14. disponible au téléchargement à http://cadna.lip6.fr/
15. Compute Unified Device Architecture : technologie développée par Nvidia pour programmer sur carte graphique.
16. http://www-pequan.lip6.fr/~jezequel/SAM/

Outil	instrumentation	synchronisme	recherche d'instabilités	bibliothèques externes
CADNA	code source	synchrone	oui	nécessite des modifications pour introduire CADNA
verificarlo	représentation intermédiaire de LLVM	asynchrone	manuellement avec la désinstrumentation de fonction	Si les sources sont disponibles ou compilées avec clang
verrou	binaire	asynchrone	oui nécessite l'utilisation de l'algorithme Delta Debug ou la recompilation avec la couverture de code [33]	oui

Table I.2 – Résumé des fonctionnalités des différents outils probabilistes

I.5 Comparaison d'outils probabilistes de validation numérique

I.5.1 Récapitulatif des fonctionnalités

Le tableau I.2 résume pour trois outils probabilistes l'étape à laquelle se passe l'instrumentation, si l'outil est synchrone, si une recherche des instabilités numériques est possible et ce qu'il est nécessaire de faire pour instrumenter des bibliothèques externes.

Au niveau des langages supportés par les outils, verificarlo est disponible en C/C++ et fortran (avec l'utilisation de dragonegg [36]). CADNA permet la validation numérique de codes en C/C++, fortran, avec OpenMP [40], MPI [100, 101] et CUDA [17] [77, 79]. Un démonstrateur pour codes en Python existe également. Verrou est, à ce niveau, l'outil fonctionnant avec le plus de langages différents. En effet, il fonctionne avec le binaire directement, grâce à valgrind [108]. Ainsi l'instrumentation peut se faire sur un interpréteur tel que Python.

Une autre différence majeure provient du mode synchrone ou asynchrone des outils. En effet, CADNA peut permettre un contrôle dynamique des résultats et des tests. Nous verrons cela plus particulièrement dans la section I.5.2.b.

I.5.2 Étude de la qualité numérique des résultats fournis

Nous souhaitons étudier la qualité numérique des résultats estimée par les différents outils. Pour cela, nous avons choisi de prolonger les résultats obtenus dans [37]. Nous

17. Compute Unified Device Architecture : technologie développée par Nvidia pour programmer sur carte graphique.

avons instrumenté les exemples fournis avec CADNA pour les différents outils probabilistes avec leurs différents modes d'exécution possibles. À cela, nous avons ajouté l'étude de la qualité numérique d'une somme et d'un produit scalaire. Tous les codes utilisent la *double* précision et sont exécutés 100 fois par verrou et verificarlo sauf indication contraire. De même la précision virtuelle de verificarlo est fixée à 53 sauf cas particulier.

Les tests ont été effectués sur une station de travail avec un processeur Intel Xeon E3-1240 cadencé à 3.4 GHz et disposant de 16Go de RAM. Le compilateur utilisé est GCC 4.9.2 avec l'option d'optimisation *O3*, la version de llvm est la 3.5.

Nous testons les outils de validation suivants :

— CADNA version 2.0.0 [18] ;
— verrou version 1.1 [19] avec le mode aléatoire et proportionnel, nous n'utilisons pas l'algorithme de recherche d'instabilités de verrou ici ;
— verificarlo version 0.2.0 [20] avec le calcul *full MCA*, *Random Rounding* (RR) et *Precision Bounding* (PB), voir page 13.

Pour chacun des outils, nous observons le résultat obtenu : dans le cas de CADNA nous reportons la valeur avec le nombre de chiffres significatifs exacts estimé et dans le cas de verrou et verificarlo nous affichons la moyenne du résultat sur cent exécutions, l'écart-type et le nombre de chiffres significatifs exacts estimé. Nous avons choisi de ne pas intégrer une exécution en arithmétique IEEE dans les cas de verrou et verificarlo. Même si cette procédure est recommandée par les développeurs [21], nous ne l'avons pas introduite pour comparer des comportements similaires entre CADNA, verrou et verificarlo. Cette exécution peut permettre de se rendre compte de divergence de résultat important entre le résultat IEEE et les résultats perturbés par l'outil de validation.

I.5.2.a Polynôme de Rump

L'exemple 1 de CADNA est le calcul du polynôme à deux variables $p(x, y)$ proposé par Rump [125] :

$$p(x,y) := 1335/4 * y^6 + x^2 * (11 * x^2 * y^2 - y^6 + (-121) * y^4 - 2) + 11/2 * y^8 + x/(2 * y).$$

Le code se composant d'une seule ligne de calcul, l'utilisation de l'option de compilation *O3* optimisera trop le code qui renvoie ainsi tout le temps les mêmes valeurs avec verrou et verificarlo. Avec ces deux outils, nous utilisons donc l'option *O0* dans ce cas. Nous savons que le résultat exact arrondi à 15 chiffres significatifs est $-0,827396059946821$.

18. http://cadna.lip6.fr/
19. https://github.com/edf-hpc/verrou
20. https://github.com/verificarlo/verificarlo
21. https ://github.com/edf-hpc/verrou/tree/ecole-precis

Résultat exact	-0,827396059946821		
Résultat IEEE	1,17260394005317		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA	@.0		0
verrou mode aléatoire	-6,965490562232726e+20	2,215583e+21	0
verrou mode proportionnel	-2,302153660398952e+21	1,730080e+21	0
verificarlo *full MCA*	8,070648666173981e+20	3,751863e+21	0
verificarlo RR	-1,416709944860894e+20	1,920267e+21	0
verificarlo PB	-1,042625193483441e+20	3,092641e+21	0

Nous pouvons voir que le résultat fourni par CADNA est un zéro informatique : CADNA estime que le résultat n'a aucun chiffre significatif exact ce qui est cohérent avec la comparaison avec le résultat exact. Dans le cas de verrou et verificarlo, le résultat est une moyenne et un écart-type qui est de l'ordre de grandeur de la moyenne. Le résultat est donc sans chiffre significatif exact également.

Comme nous l'avons dit précédemment, lorsque nous utilisons l'option d'optimisation *O3*, alors le résultat est toujours le même avec verrou et verificarlo : $1,17260394005317$. Il s'agit de la valeur obtenue en *double* précision IEEE mais qui ne dispose en fait d'aucun chiffre significatif en commun avec le résultat exact. Les optimisations du compilateur ont donc un impact non négligeable sur la qualité du résultat et comparer les résultats obtenus avec plusieurs d'entre elles peut être important.

I.5.2.b Calcul des racines d'un polynôme du second degré

Dans cet exemple, nous souhaitons calculer la solution d'une équation du second degré en utilisant le code 1. Il s'agit d'un code développé pour fonctionner avec CADNA en priorité et auquel nous avons retiré CADNA pour avoir le code IEEE. En effet dans un code non pensé comme devant fonctionner avec CADNA ou étant une implémentation naïve d'un algorithme mathématique, nous n'aurions pas une comparaison de d à 0 mais à un epsilon dépendant de l'ordre de grandeur de b. Nous avons ici exécuté les outils verrou et verificarlo 20 fois chacun. Avec verificarlo nous avons modifié la valeur de la précision virtuelle à 23 (calcul en *simple* précision).

Le résultat exact est la racine double : 3,5. Le calcul avec l'utilisation de la *simple* précision avec l'arrondi au plus près nous donne pour résultat deux racines complexes conjuguées [22] : z1 = 3,5 + i×9,765625e-04 et z2 = 3,5 - i×9,765625e-04. CADNA détecte une instabilité numérique et considère que le discriminant est un zéro informatique (@.0) et indique la présence d'une racine double (3,499999). Cela fonctionne bien car l'algorithme est construit pour fournir comme résultat une racine double dans le cas où le discriminant s'apparente à du bruit numérique.

L'utilisation de verificarlo [23] ne permet pas d'obtenir ce résultat. En effet sur 20 exécutions, nous avons obtenus 8 valeurs positives et 12 négatives pour le discriminant avec pour valeur absolue la plus faible 4,646524e-08. La moyenne des discriminants est -1,29766e-07 et l'écart type 1,39702e-05. Dans le cas de verrou [24], nous obtenons lors de dix exécutions un discriminant nul. Dans les autres cas il est positif 4 fois et négatif 6

22. Le discriminant est non nul dans ce cas
23. Nous sommes ici en *full MCA* avec une précision de 23 bits
24. Nous exécutons la version aléatoire de verrou

Code 1 Code de calcul des racines d'un polynôme du second degré

```
#include <stdio.h>
#include <math.h>
#include <cadna.h>

using namespace std;
int main()
{

  cadna_init(-1);

  float_st a = 0.3;
  float_st b = -2.1;
  float_st c = 3.675;
  float_st d, x1,x2;
  //       CASE: A = 0
  if (a==0)
    if (b==0.) {
      if (c==0.) printf("Every complex value is solution.\n");
      else printf("There is no solution.\n");
    }
    else {
      x1 = - c/b;
      printf("The equation is degenerated.\n");
      printf("There is one real solution %s\n",strp(x1));
    }
  else {
    //      CASE: A /= 0
    b = b/a;
    c = c/a;
    d = b*b - 4.0*c;
    printf("d = %s\n",strp(d));
    //    DISCRIMINANT = 0
    if (d==0.) {
      x1 = -b*0.5;
      printf("Discriminant is zero.\n");
      printf("The double solution is %s\n",strp(x1));
    }
    else {
      //        DISCRIMINANT > 0
      if (d>0.) {
          x1 = ( - b - sqrtf(d))*0.5;
          x2 = ( - b + sqrtf(d))*0.5;
          printf("There are two real solutions.\n");
          printf("x1 = %s x2 = %s\n",strp(x1),strp(x2));
      }
      else {
        //        DISCRIMINANT < 0
        x1 = - b*0.5;
        x2 = sqrtf(-d)*0.5;
        printf("There are two complex solutions.\n");
        printf("z1 = %s  +  i * %s\n",strp(x1),strp(x2));
        printf("z2 = %s  +  i * %s\n",strp(x1), strp(-x2));
      }
    }
  }

  cadna_end();
}
```

fois avec une moyenne de -9,53674e-07 et un écart type de 3,59372e-06. Nous obtenons alors comme résultat de l'équation soit :

— une racine double ;
— deux racines réelles ;
— deux racines complexes conjuguées.

Il est impossible de comparer ces résultats. En effet, cela implique un choix arbitraire dans la sélection des valeurs à prendre en compte.

Il y a dans ce cas une instabilité numérique évidente. Verrou et verificarlo ne permettent pas d'extraire cette information sur une seule exécution et continuent l'exécution du code. Lors des comparaisons des différents résultats, nous pouvons nous rendre compte de la présence d'une instabilité en raison de la différence dans les résultats obtenus. Le résultat obtenu avec le code en *double* précision IEEE et le résultat CADNA ne sont pas équivalents. L'utilisation de l'Arithmétique Stochastique Discrète permet de valider un algorithme légèrement différent tout en indiquant l'instabilité de branchement.

Nous pouvons voir ici un intérêt à CADNA dans un cadre différent de la validation numérique. En effet, en utilisant les informations apportées par les trois exécutions synchrones et avec un code adapté, il est possible d'utiliser CADNA pour faire des choix dynamiquement en fonction de l'amplitude des incertitudes numériques.

I.5.2.c Calcul du déterminant de la matrice de Hilbert

Nous cherchons ici à calculer le déterminant d'une matrice de Hilbert de taille 11. Les premiers chiffres de sa valeur exacte sont $3,0190953344493e^{-65}$.

Résultat exact	$3,0190953344493e^{-65}$		
Résultat IEEE	$3,026439382718219e^{-65}$		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA	0,302E-064		3
verrou mode aléatoire	3,018498367401331e-65	4,955968e-68	2,78
verrou mode proportionnel	3,018837468996046e-65	4,272274e-68	2,84
verificarlo *full MCA*	3,019364968483173e-65	7,805986e-68	2,58
verificarlo RR	3,018127844876275e-65	7,244655e-68	2,61
verificarlo PB	3,020758429487229e-65	5,168696e-68	2,76

Pour tous les outils de validation, nous pouvons voir que le résultat obtenu est en accord avec le résultat exact lorsque l'on tient compte du nombre de chiffres significatifs exacts estimé.

I.5.2.d Suite récurrente d'ordre 2

Nous calculons ici la suite récurrente d'ordre 2 proposée par J.-M Muller [106] :

$$U_{n+1} = 111 - \frac{1130}{U_n} + \frac{3000}{U_n U_{n-1}}$$

avec $U_0 = 5,5$ et $U_1 = \frac{61}{11}$. 30 itérations sont effectuées. Mathématiquement cette suite tend vers 6.

Résultat exact	6		
Résultat IEEE	100		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA	100		15
verrou mode aléatoire	100	0	15
verrou mode proportionnel	100	0	15
verificarlo *full MCA*	100	0	15
verificarlo RR	100	0	15
verificarlo PB	100	0	15

Nous pouvons voir que dans tous les cas le résultat tend vers 100 qui est un autre point fixe de la suite. Il se trouve qu'en raison d'instabilités numériques, le résultat obtenu est faux. CADNA possède ici l'avantage d'être synchrone et donc de pouvoir exhiber ces instabilités. Comme CADNA détecte des divisions instables et une multiplication instable, un message avertit l'utilisateur que les résultats fournis ne sont pas fiables. Pour les autres outils, il faudrait étudier les résultats intermédiaires et se rendre compte du problème de perte de précision durant les différentes itérations.

I.5.2.e Calcul d'une racine d'un polynôme par la méthode de Newton

Nous cherchons ici à calculer une racine du polynôme

$$P(x) = 1,47x^3 + 1,19x^2 - 1,83x + 0,45$$

grâce à la méthode de Newton. Nous utilisons pour valeur initiale $0,5$ et comme critère d'arrêt 100 itérations ou lorsque $|x_k - x_{k-1}| < 10^{-12}$ avec x_k les éléments successifs obtenus par la méthode de Newton. Les premiers chiffres de la racine exacte $(\frac{3}{7})$ sont : $0,428571428571428571$.

Résultat exact	0,428571428571428571		
Résultat IEEE	0,4285714252078272		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA	0,4285714E+000		7
verrou mode aléatoire	4,285714301958174e-01	5,268356e-09	7,91
verrou mode proportionnel	4,285714309376100e-01	1,053671e-08	7,60
verificarlo *full MCA*	4,285714292337627e-01	4,848627e-09	7,94
verificarlo RR	4,285714286651280e-01	1,053671e-08	7,60
verificarlo PB	4,285714306211604e-01	5,268356e-09	7,91

Les résultats obtenus sont corrects et ici nous disposons à chaque fois de 7 chiffres significatifs exacts. Nous pouvons tout de même noter que le code utilise la fonction `pow` de la bibliothèque mathématique et qu'il faut "désinstrumenter" cette fonction avec verrou. Il s'agit de la procédure standard définie dans la documentation de l'outil. CADNA et verificarlo ne nécessitent pas cette étape car elle est déjà effectuée au sein même de l'outil.

I.5.2.f Résolution d'un système linéaire

Nous résolvons ici un système linéaire par la méthode de Gauss en *simple* précision. Le résultat exact est $(1, 1, 10^{-8}, 1)$. Le résultat obtenu en utilisant l'arithmétique IEEE est $(62, 61988, -8, 953979, 0, 000000, 0, 9999999)$. Pour que les résultats avec verificarlo soient cohérents, nous avons modifié la valeur de la précision virtuelle à 23.

Nous pouvons tout d'abord voir que CADNA fournit une bonne estimation du résultat exact sur les quatre éléments du vecteur ce qui n'est ni le cas de verrou ni de verificarlo. Néanmoins, nous savons que le résultat obtenu en *simple* précision est entaché d'erreur. Le résultat fourni par verrou et verificarlo met en avant la perte de précision. Nous voyons donc clairement la présence d'une instabilité numérique.

Dans ce cas, CADNA permet l'obtention du résultat exact, au nombre de chiffres significatifs exacts prêts, ce qui peut servir de résultat de référence. La différence des résultats s'explique dans le cas présent, lors de la réduction d'une des colonnes l'élément non-nul a pour valeur 4864 avec le calcul en norme IEEE mais est en réalité un zéro informatique. CADNA permet donc de ne pas choisir cette valeur comme pivot ce qui évite une erreur sur le résultat final.

Résultat exact	1 1 1e-8 1		
Résultat IEEE	6,261988e+01 -8,953979 0,000000 9,999999e-01		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA	0,100E+001 0,1000E+001 0,100000E-007 0,100000E+001		3 4 6 6
verrou mode aléatoire	3,277869e+01 -4,133483e+00 4,842877e-09 9,999999e-01	1,869834e+02 3,020501e+01 3,034454e-08 1,168577e-07	0 0 0 6,93
verrou mode proportionnel	7,868864e+01 -1,154970e+01 -2,607703e-09 9,999999e-01	1,844864e+02 2,980165e+01 2,993937e-08 9,002570e-08	0 0 0 7,04
verificarlo *full MCA*	-8,269260e+01 1,451957e+01 2,358205e-08 9,999999e-01	5,472286e+02 8,839848e+01 8,880700e-08 3,394718e-07	0 0 0 6,46
verificarlo RR	-8,770661e+00 2,578338e+00 1,158565e-08 9,999998e-01	2,513969e+02 4,061027e+01 4,079794e-08 1,251165e-07	0 0 0 6,90
verificarlo PB	7,435404e+01 -1,084949e+01 -1,904280e-09 9,999998e-01	5,097475e+02 8,234384e+01 8,272443e-08 1,870080e-07	0 0 0 6,72

I.5.2.g Lorsque CADNA se trompe

Nous avons ici un cas construit de manière à avoir une fois sur quatre une erreur avec CADNA. Il s'agit de calculer $((z-x)+y)+((z-y)+x-2)$ avec $x = 6,83561e^5$, $y = 6,83560e^5$ et $z = 1,00000000007$. Le résultat exact est $1,4e^{-10}$ et le résultat obtenu en *double* précision est $2,32830643653870e^{-10}$. Le code étant trop optimisé en *O3*, nous devons utiliser l'option *O0* avec verrou et verificarlo pour obtenir des résultats exploitables.

Résultat exact	1,4e-10		
Résultat IEEE	2,32830643653870e-10		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA (1 fois sur 4)	0,116415321826935E-009		15
CADNA (3 fois sur 4)	@.0		0
verrou mode aléatoire	3,527384251356124e-10	8,306378e-11	0,62
verrou mode proportionnel	3,317836672067642e-10	7,431456e-11	0,64
verificarlo *full MCA*	1,405571166883002e-10	1,067385e-10	0,11
verificarlo RR	1,257285253686291e-10	8,661801e-11	0,16
verificarlo PB	1,401308202725531e-10	9,750979e-11	0,15

Nous pouvons voir que CADNA indique la présence de 15 chiffres significatifs exacts dans 1 cas sur 4 et d'un zéro informatique le reste du temps. Verrou et verificarlo en indiquent 0 même lorsque les premiers chiffres correspondent au résultat exact. L'erreur dans le résultat de CADNA 1 fois sur 4 provient d'un enchaînement d'arrondi causant une absorption en raison des différences d'ordre de grandeur entre x, y et z.

I.5.2.h Somme des éléments d'un vecteur

Nous calculons ici une somme naïve de 200 éléments choisis aléatoirement de manière à obtenir entre 5 et 6 chiffres significatifs exacts en norme IEEE [25] dont le résultat exact arrondi à quinze chiffres est $3,1263728551008141e^{-1}$.

Résultat exact	3,1263728551008141e-1		
Résultat IEEE	0,312635789772658		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA	0,31263		5
verrou mode aléatoire	3,126330373605491e-01	1,821377e-06	5,23
verrou mode proportionnel	3,126346418430799e-01	2,079866e-06	5,23
verificarlo *full MCA*	3,126368976653799e-01	2,298845e-06	5,23
verificarlo RR	3,126373131103828e-01	2,027708e-06	5,18
verificarlo PB	3,126372371410722e-01	1,886130e-06	5,21

Tous les outils probabilistes indiquent ici des moyennes similaires, en accord avec le résultat exact et un nombre de chiffres significatifs exacts de 5. Ils sont donc équivalents dans ce cas.

25. cf. section II.9

I.5.2.i Produit scalaire

Nous calculons ici le produit scalaire de deux vecteurs de 100 éléments choisis aléatoirement de manière à obtenir entre 5 et 6 chiffres significatifs exacts en norme IEEE [26]. Les 15 premiers chiffres du résultat exact sont ici $7,0445043553545839e^{-1}$.

Résultat exact	7,0445043553545839e-1		
Résultat IEEE	0,704449873788690		
	Moyenne	Ecart-Type	Nombre de chiffres significatifs exacts
CADNA	0,70445		5
verrou mode aléatoire	7,044505247710550e-01	7,070989e-07	6,86
verrou mode proportionnel	7,044505548209782e-01	7,853208e-07	5,95
verificarlo *full MCA*	7,044503983023637e-01	8,201055e-07	5,93
verificarlo RR	7,044505011153153e-01	7,897615e-07	5,95
verificarlo PB	7,044505169049080e-01	8,033508e-07	5,94

Verrou avec le mode aléatoire indique la présence d'un chiffre significatif supplémentaire. Néanmoins, en comparant au résultat exact, nous pouvons nous rendre compte qu'il s'agit d'une erreur car nous ne disposons au final que de 5 chiffres en commun avec le résultat exact, l'arrondi se faisant à $7,04451$. Dans tous les autres cas, nous disposons d'un résultat similaire mais avec un nombre de chiffres exacts non surestimé.

Conclusion des comparaisons en qualité numérique :

Dans la majorité des cas les résultats sont similaires entre les outils. Ce qui varie se situe sur les difficultés d'implémentation, le nombre d'exécutions que nous pouvons modifier ou non en fonction de l'outil et le synchronisme qui permet de détecter les instabilités numériques et éventuellement d'améliorer la précision des résultats. De plus comme nous l'avons vu, en fonction des options de compilations utilisées et de l'outil de validation, les résultats obtenus peuvent être différents.

Nous avons également vu que l'utilisation de CADNA peut être intéressante dans le cas d'une utilisation en production dans le but d'obtenir des résultats prenant en compte les zéros informatiques obtenus durant l'exécution. Nous pouvons alors utiliser avantageusement les relations d'ordre stochastiques pour tenir compte des zéros informatiques.

I.5.3 Test de performances

Pour effectuer nos tests de performances nous avons utilisé six codes différents :
- addition compute bound ;
- addition memory bound ;
- multiplication compute bound ;
- multiplication memory bound ;
- stencil ;
- gradient conjugué (CG) avec une matrice de taille 7000 ayant 8 valeurs non nulles par colonne et avec 15 itérations.

26. cf. section II.9

Les codes d'addition et de multiplication sont écrits sous la forme de deux boucles imbriquées. L'ordre des boucles permet d'obtenir un programme dont la limitation en performance provient des calculs (compute bound) ou de la mémoire (memory bound). L'algorithme I.7 présente le cas de l'addition compute bound et l'algorithme I.8 montre l'addition memory bound [27]. Nous utiliserons les termes memory bound et compute bound pour nous référer à ces cas même si nous n'avons pas fait le travail de vectorisation et de parallélisme à mémoire partagée ce qui explique la faible différence en temps entre les deux cas. Dans nos cas tests, nous avons N=2^{24} et $K = 128$. Le programme stencil provient des cas tests de `ispc`. Ce code permet de simuler la propagation d'une vague via une simulation 3D et la méthode des différences finies.

Dans chacun des cas, nous ne prenons pas en compte les phases d'initialisations.

Algorithme I.7 Code de l'addition compute bound

```
pour i = 0 à N-1 faire
  pour j = 0 à K-1 faire
    a[i] = b[i] + a[i]
  fin pour
fin pour
```

Algorithme I.8 Code de l'addition memory bound

```
pour j = 0 à K-1 faire
  pour i = 0 à N-1 faire
    a[i] = b[i] + a[i]
  fin pour
fin pour
```

Les tests ont été effectués sur une station de travail avec un processeur Intel Xeon E3-1240 cadencé à 3.4 GHz et disposant de 16Go de RAM. Le compilateur utilisé est GCC 4.9.2 avec l'option d'optimisation *O3*, la version de llvm est la 3.5.

Nous testons les outils de validation suivant :

— CADNA version 2.0.0 [28] avec les trois modes de détections d'instabilités principaux ;

— verrou version 1.1 [29] avec le mode aléatoire et proportionnel, nous n'utilisons pas l'algorithme de recherche d'instabilités de verrou ici ;

— verificarlo version 0.2.0 [30] avec le calcul *full MCA*, *Random Rounding* (RR) et *Precision Bounding* (PB), voir page 13, nous testons ici le backend QUAD et le backend verrou.

Les temps de calcul nécessaires pour les différents outils sont visibles dans le tableau I.3 avec également les temps de calcul de la *double* précision de la norme IEEE 754, nous présentons également les ratios par rapport au temps obtenu avec le type *double* de la norme IEEE . Ce ratio est obtenu en divisant le temps d'exécution de

27. Pour la multiplication il faut prendre ces cas et modifier l'addition en une multiplication
28. http://cadna.lip6.fr/
29. https://github.com/edf-hpc/verrou
30. https://github.com/verificarlo/verificarlo

l'outil par le temps d'exécution des flottants classique de la norme IEEE 754. Nous avons choisi d'effectuer trois exécutions pour verrou et verificarlo dans le but de comparer facilement leurs temps de calcul avec CADNA. Les différentes exécutions de verrou et de verificarlo sont effectuées séquentiellement. Toutefois, il est possible de les exécuter en parallèle. Nous pouvons également noter que CADNA dispose directement du post-traitement alors que les autres outils doivent passer par une étape de post-traitement effectuée par l'utilisateur. Souvent les outils de régression peuvent être utilisés pour cette tâche.

Nous pouvons remarquer que dans la majorité des cas hors IEEE, les cas tests memory bound sont plus rapides que l'équivalent compute bound (6 cas sur 18 ne le sont pas). Tandis que l'utilisation du type *double* de la norme IEEE 754 fait que nous obtenons des temps de calcul plus importants pour les cas tests memory bound. En effet dans le cas de la norme IEEE 754, la mémoire est limitante. Néanmoins, lorsque nous utilisons CADNA nous modifions l'utilisation mémoire et nous introduisons des calculs intermédiaires avec tous les outils. Ces calculs impactent moins les performances lorsque nous utilisons des calculs memory bound car nous sommes alors limités par la mémoire et non par la vitesse du CPU. Lorsque nous sommes compute bound, la limitation du processeur est déjà atteinte et nous rajoutons de nouveaux calculs en plus. Il y a donc un ratio plus important.

Nous pouvons également voir qu'à un nombre d'exécutions égal, les versions de CADNA sont plus rapides, viennent ensuite les versions de verificarlo avec le backend verrou, les versions de verrou et enfin verificarlo avec le backend QUAD.

Nous observons également que les temps de calcul sont similaires dans les deux versions de verrou étudiées (la version proportionnelle étant légèrement plus lente : 10% au maximum avec le stencil). De la même façon, l'utilisation du backend verrou de verificarlo permet d'obtenir des temps de calcul similaires dans les deux modes. Les différences de performances entre verificarlo avec le backend verrou et verrou s'expliquent par l'utilisation de valgrind. En effet, l'instrumentation de verrou se fait sur le binaire et nécessite du temps à l'exécution. Dans le cas de verificarlo, des instructions supplémentaires sont ajoutées mais à la compilation, ce qui nécessite un temps de compilation plus important mais qui permet d'avoir une exécution plus rapide.

Une grande différence existe entre les trois versions du backend QUAD de verificarlo, où la version *full MCA* nécessite jusqu'à 2,3 fois plus de temps que le calcul *Random Rounding*, le calcul *Precision Bound* permet d'obtenir des temps de calcul intermédiaires. En effet la version *full MCA* nécessite de perturber trois valeurs, la version erreur indirecte deux valeurs et une unique pour rechercher l'erreur directe.

	Addition compute bound	Addition memory bound	multiplication compute bound	multiplication memory bound	stencil	CG
IEEE						
	2,23 (×1)	3,62 (×1)	2,85 (×1)	3,574 (×1)	0,792 (×1)	0,192 (×1)
CADNA (trois exécutions synchrones)						
Toutes les instabilités	28 (×12,6)	27 (×7,5)	23 (×8,1)	19 (×5,3)	30 (×38)	6,36 (×33)
Auto-validation	15 (×6,7)	19 (×5,2)	21 (×7,4)	19 (×5,3)	16 (×20)	2,5 (×13)
Aucune instabilité	15 (×6,7)	19 (×5,2)	11 (×3,9)	15 (×4,3)	13 (×17)	2,27 (×12)
verrou (trois exécutions asynchrones)						
Mode aléatoire	60 (×26,9)	51 (×14)	114 (×40)	105 (×29,4)	93 (×117)	25 (×128)
Mode proportionnel	63 (×28,3)	51 (×14,9)	114 (×43,2)	105 (×29,4)	102 (×128)	26 (×135)
verificarlo (trois exécutions asynchrones)						
full MCA backend QUAD	1779 (×797,8)	1788 (×493,9)	1875 (×657,9)	1866 (×522,1)	2115 (×2670)	375 (×1953)
PB backend QUAD	1272 (×570,4)	1344 (×371,3)	1383 (×485,3)	1383 (×387)	1509 (×1905)	273 (×1421)
RR backend QUAD	837 (×375,3)	834 (×230,4)	849 (×287,9)	882 (×245,8)	951 (×1201)	171 (×890)
backend verrou	29 (×13,1)	28 (×7,8)	75 (×26,3)	69 (×19,3)	66 (×83,3)	16 (×81,2)
backend verrou proportionnel	33 (×14,8)	30 (×8,2)	75 (×26,3)	72 (×20,1)	72 (×90,9)	16 (×84,4)

Table I.3 – Temps d'exécution (en secondes) des codes de tests et ratio du temps d'exécution par rapport à la *double* précision de la norme IEEE-754 pour différents outils probabilistes

Chapitre II

Validation numérique d'algorithmes compensés

II.1 Introduction

Une des solutions pour améliorer la précision de certains algorithmes est d'utiliser les algorithmes compensés [21]. Ils utilisent les transformations exactes : pour l'opération $\circ \in \{+, \times\}$, nous avons alors la relation $\forall (a, b) \in \mathbb{F}^2, \exists (x, y) \in \mathbb{F}^2$ tel que $x = fl(a \circ b)$ et $a \circ b = x + y$. Comme nous l'avons rappelé, en arrondi dirigé, certaines transformations ne sont plus exactes. L'Arithmétique Stochastique Discrète, décrite en section I.4, nécessite un mode d'arrondi aléatoire qui utilise les arrondis dirigés. La validation des codes utilisant les algorithmes compensés avec la méthode CESTAC n'est donc *a priori* pas assurée.

Nous allons ici étudier les algorithmes compensés et l'Arithmétique Stochastique Discrète. Pour ce faire, nous donnerons les définitions et les notations utilisées par la suite (section II.2) puis nous étudierons la sommation (section II.3), le produit scalaire (section II.4), l'évaluation polynomiale (section II.5) puis les algorithmes compensés K fois que ce soit la somme (section II.6) ou le produit scalaire (section II.7). Nous terminerons enfin par les expérimentations numériques (section II.9). Nous prendrons en compte les *underflows* uniquement dans le cas de la sommation.

II.2 Définitions et notations

Nous supposerons que l'arithmétique à virgule flottante utilisée respecte la norme IEEE 754 [73]. Nous présumons également qu'aucun *overflow* ne se produit. Nous notons $\mathbb{F}$ l'ensemble des nombres flottants représentables et $\mathbf{u}$ l'erreur d'arrondi relative. Nous avons pour la *simple* précision $\mathbf{u} = 2^{-24}$, et pour la *double* précision $\mathbf{u} = 2^{-53}$.

Nous notons $\mathrm{fl}_*(.)$ le résultat d'une opération flottante [1], où toutes les opérations effectuées au sein des parenthèses le sont en arrondi dirigé (donc vers $-\infty$ ou $+\infty$). Nous avons donc [68] :

1. Cette opération peut être une affectation d'une constante.

$\exists(\epsilon_1, \epsilon_2) \in \mathbb{R}^2$ tels que

$$\mathrm{fl}_*(a \circ b) = (a \circ b)(1+\varepsilon_1) = (a \circ b)/(1+\varepsilon_2) \text{ pour } \circ = \{+,-\} \text{ et } |\varepsilon_\nu| \leq 2\mathbf{u} \quad \text{(II.2.1)}$$

ainsi

$$|a \circ b - \mathrm{fl}_*(a \circ b)| \leq 2\mathbf{u}|a \circ b| \text{ et } |a \circ b - \mathrm{fl}_*(a \circ b)| \leq 2\mathbf{u}|\,\mathrm{fl}_*(a \circ b)| \text{ pour } \circ = \{+,-\}. \quad \text{(II.2.2)}$$

Nous utiliserons les notations classiques pour l'estimation des erreurs. Soit n un entier vérifiant $n\mathbf{u} < 1$, alors la valeur γ_n est définie par [68] :

$$\gamma_n(\mathbf{u}) = \frac{n\mathbf{u}}{1 - n\mathbf{u}}.$$

II.3 Sommation

II.3.1 Sommation classique

L'algorithme de sommation naïve est l'algorithme récursif II.1.

Algorithme II.1 Somme de n nombres flottants $p = \{p_i\}$

fonction `res` = `Somme`(p)

1: $s_1 \leftarrow p_1$
2: **pour** $i = 2$ à n **faire**
3: $\quad s_i \leftarrow s_{i-1} + p_i$
4: **fin pour**
5: `res` $\leftarrow s_n$

L'erreur numérique produite est rappelée dans la proposition II.3.1.

Proposition II.3.1 ([68]). *Soit $p_i \in \mathbb{F}$ avec $1 \leq i \leq n$. Soient $s = \sum p_i$ et $S = \sum |p_i|$. En arrondi au plus près, si $n\mathbf{u} < 1$, alors*

$$|\mathtt{res} - s| \leq \gamma_{n-1}(\mathbf{u})S. \quad \text{(II.3.1)}$$

En utilisant un arrondi dirigé, si $n\mathbf{u} < \frac{1}{2}$, alors

$$|\mathtt{res} - s| \leq \gamma_{n-1}(2\mathbf{u})S. \quad \text{(II.3.2)}$$

Le corollaire II.3.2 présente les équations II.3.1 et II.3.2 en fonction du conditionnement de $\sum p_i$:

$$\mathrm{cond}\left(\sum p_i\right) = \frac{S}{|s|}.$$

Corollaire II.3.2. *En arrondi au plus près, si $n\mathbf{u} < 1$, le résultat* `res` *de l'algorithme II.1 est tel que :*

$$\frac{|\mathtt{res} - s|}{|s|} \leq \gamma_{n-1}(\mathbf{u})\,\mathrm{cond}\left(\sum p_i\right).$$

En utilisant un arrondi dirigé, si $n\mathbf{u} < \frac{1}{2}$*, le résultat* `res` *de l'algorithme II.1 est tel que :*

$$\frac{|\texttt{res} - s|}{|s|} \leq \gamma_{n-1}(2\mathbf{u})\,\mathrm{cond}\left(\sum p_i\right).$$

Or $\gamma_{n-1}(\mathbf{u}) \approx (n-1)\mathbf{u}$, donc l'erreur relative provient majoritairement de $n\mathbf{u} \times \mathrm{cond}\,(\sum p_i)$. Dans certains cas, l'erreur obtenue est trop importante pour que le résultat soit numériquement valide. Ainsi, pour des conditionnements importants (supérieurs à $1/\mathbf{u}$), l'algorithme naïf ne retourne aucun chiffre exact. Par exemple, si nous considérons de la somme des trois éléments $2^{100} + 1 + -2^{100}$, alors le conditionnement est de $2^{101} + 1$ qui est plus grand que $1/\mathbf{u}$ en *double* précision. Le résultat de la somme est 1 mais le calcul par ordinateur nous donne 0. Il existe néanmoins des algorithmes de sommation plus précis que nous allons rappeler dans la suite.

II.3.2 Sommation compensée en arrondi au plus près

Comme nous l'avons vu dans la section I.3.1, il existe différentes transformations exactes [2] pour la sommation de deux nombres flottants en utilisant l'arrondi au plus près : `TwoSum` [85] qui nécessite six opérations flottantes (cf. algorithme I.2) et `FastTwoSum` [32] qui nécessite un test et trois opérations flottantes (cf. algorithme I.1). Ces algorithmes calculent à la fois le résultat c de la somme naïve de a et b et l'erreur d commise lors de cette somme avec $c + d = a + b$. Priest propose un autre algorithme [120, p.14-15], cf. algorithme I.3, permettant de calculer cette somme et l'erreur commise en arrondi dirigé mais à un coût plus élevé avec deux tests et sept opérations flottantes.

L'algorithme II.2 réalise la sommation compensée de n nombres flottants [118]. Cette sommation est correcte grâce à la transformation exacte `FastTwoSum` utilisée pour toutes les sommes intermédiaires. En arrondi au plus près, il est possible de remplacer `FastTwoSum` par une autre transformation exacte (`TwoSum` ou `TwoSumPriest`).

Algorithme II.2 Sommation compensée de n nombres flottants $p = \{p_i\}$ avec `FastTwoSum`

fonction `res` = `FastCompSum`(p)

1: $\pi_1 \leftarrow p_1$
2: $\sigma_1 \leftarrow 0$
3: **pour** $i = 2$ à n **faire**
4: $\quad [\pi_i, q_i] \leftarrow$ `FastTwoSum`(π_{i-1}, p_i)
5: $\quad \sigma_i \leftarrow \sigma_{i-1} + q_i$
6: **fin pour**
7: `res` $\leftarrow \pi_n + \sigma_n$

La figure II.1 est une représentation graphique de l'algorithme II.2 (`FastCompSum`). Nous additionnons tous les p_i $(1 \leq i \leq n)$ ainsi que toutes les erreurs commises sur ces sommes (calcul des q_i $(2 \leq i \leq n)$). Enfin, π_n et q_n sont additionnés pour produire le résultat res.

2. En anglais Error-Free Transformation (EFT)

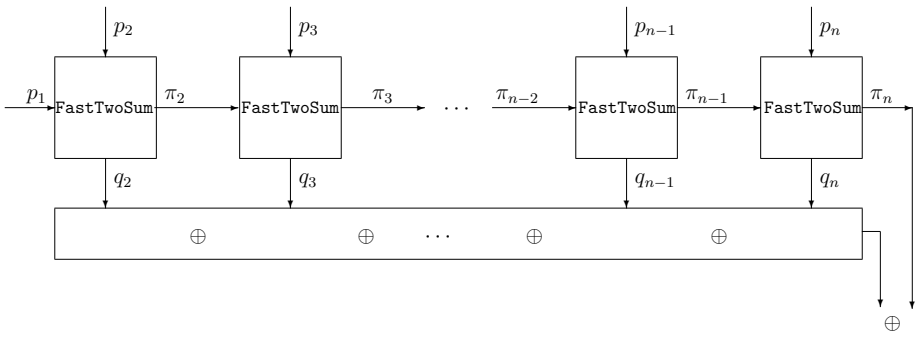


FIGURE II.1 – Représentation graphique de `FastCompSum` (algorithme II.2)

L'erreur sur le résultat `res` de l'algorithme II.2 en arrondi au plus près est étudiée dans [114]. Une borne d'erreur pour l'erreur absolue est rappelée dans la proposition II.3.3 et une borne pour l'erreur relative dans le corollaire II.3.4.

Proposition II.3.3 ([114]). *Supposons que l'algorithme II.2 (*`FastCompSum`*) est appliqué en arrondi au plus près aux flottants $p_i \in \mathbb{F}$, $1 \leq i \leq n$. Soient $s = \sum p_i$ et $S = \sum |p_i|$. Si $n\mathbf{u} < 1$, alors*

$$|\texttt{res} - s| \leq \mathbf{u}|s| + \gamma_{n-1}^2(\mathbf{u})S$$

*et ce, même en cas d'*underflow*.*

Corollaire II.3.4 ([114]). *En arrondi au plus près, si $n\mathbf{u} < 1$, alors le résultat* `res` *de l'algorithme II.2 (*`FastCompSum`*) vérifie*

$$\frac{|\texttt{res} - s|}{|s|} \leq \mathbf{u} + \gamma_{n-1}^2(\mathbf{u}) \operatorname{cond}\left(\sum p_i\right)$$

*et ce, même en présence d'*underflow*.*

De nouveau $\gamma_{n-1}(\mathbf{u}) \approx (n-1)\mathbf{u}$, donc l'erreur relative présentée dans le corollaire II.3.4 est principalement $(n\mathbf{u})^2 \times \operatorname{cond}(\sum p_i) + \mathbf{u}$. Cette borne d'erreur nous indique que la précision du résultat est similaire à celle qui aurait été obtenue en doublant la précision des calculs (terme en $\mathbf{u}^2$). Le premier terme ($\mathbf{u}$) représente l'arrondi à la précision utilisée.

II.3.3 Sommation compensée en arrondi dirigé

L'utilisation de l'arrondi dirigé fait que l'algorithme I.1 (`FastTwoSum`) n'est plus une transformation exacte. L'erreur d calculée par l'algorithme I.1 ne permet pas d'obtenir l'égalité $a + b = c + d$. La proposition II.3.5 indique une borne sur la différence entre d, l'erreur calculée, et e l'erreur réelle avec $a + b = c + e$.

Proposition II.3.5 ([54]). *Soient c et d respectivement la somme des flottants a et b et la correction calculée par l'algorithme I.1 (*`FastTwoSum`*) en arrondi dirigé. Soit e l'erreur réelle sur c : $a + b = c + e$. Alors*

$$|e - d| \leq 2\mathbf{u}|e|.$$

Démonstration. Tout d'abord, nous allons montrer que $z = c - a$ exactement. Pour cela nous étudions deux cas :

1. $a, b \geq 0$:
   Comme $0 \leq b \leq a$,
   $$a \leq a + b \leq 2a$$
   En raison de la monotonie de la fonction fl_*, nous pouvons en déduire que
   $$a \leq \mathrm{fl}_*(a + b) \leq 2a$$
   Alors
   $$a \leq c \leq 2a$$
   C'est pourquoi nous pouvons conclure grâce au lemme de Sterbenz [133] que $z = c - a$.
2. $a \geq 0, b \leq 0$:
   — Si $-b \geq \frac{a}{2}$, alors
   $$a \geq -b \geq \frac{a}{2}$$
   donc $a - (-b)$ est représentable en raison du lemme de Sterbenz [133]. Ainsi $c = a + b$ et donc $z = c - a$.
   — Si $-b < \frac{a}{2}$, alors
   $$0 \geq b > -\frac{a}{2}$$
   ainsi
   $$a \geq a + b > \frac{a}{2}$$
   En raison de la monotonie de la fonction fl_*, nous pouvons en déduire que
   $$a \geq c \geq \frac{a}{2}$$
   C'est pourquoi nous pouvons conclure grâce au lemme de Sterbenz [133] que $z = c - a$.

Les cas $a, b \leq 0$ et $a \leq 0, b \geq 0$, peuvent être traités de la même manière en utilisant $-a$ et $-b$.

Ainsi quels que soient a et b nous avons $z = c - a$.

L'erreur calculée d n'est pas nécessairement représentable lorsqu'un arrondi dirigé est utilisé. Donc $\exists \delta \in \mathbb{R}$ tel que
$$d = b - z + \delta \tag{II.3.3}$$
et
$$|\delta| \leq 2\mathbf{u}|b - c|.$$
Comme $z = c - a$,
$$|\delta| \leq 2\mathbf{u}|a + b - c| \tag{II.3.4}$$
Soit e l'erreur sur l'addition flottante de a et b, alors
$$a + b = c + e \tag{II.3.5}$$

avec

$$|e| \leq 2\mathbf{u}|a+b|. \tag{II.3.6}$$

Ainsi nous pouvons déduire des équations (II.3.4) et (II.3.5) une borne sur $|\delta| = |e-d|$

$$|\delta| \leq 2\mathbf{u}|e|.$$

□

Il a été prouvé par la suite dans [9] que e est l'arrondi fidèle de l'erreur commise lors de l'opération $a+b$.

Le lemme II.3.6 va nous permettre d'établir ultérieurement une borne sur l'erreur absolue de l'algorithme II.2. Il sera utilisé dans d'autres sections de ce chapitre.

Lemme II.3.6. *Supposons que l'algorithme II.2 (*`FastCompSum`*) est exécuté en arrondi dirigé avec les nombres flottants $p_i \in \mathbb{F}$, $1 \leq i \leq n$. Pour $i = 2, ..., n$, nous notons e_i l'erreur flottante sur l'addition de π_{i-1} et p_i : $\pi_{i-1} + p_i = \pi_i + e_i$. Si $n\mathbf{u} < \frac{1}{2}$, alors*

$$\sum_{i=2}^{n} |e_i| \leq \gamma_{n-1}(2\mathbf{u}) \sum_{i=1}^{n} |p_i|. \tag{II.3.7}$$

Démonstration. Soit e_i l'erreur commise sur la somme des nombres flottants p_i et π_{i-1} $(i = 2, ..., n)$:

$$\pi_i + e_i = \pi_{i-1} + p_i$$

Le lemme va être démontré par récurrence.

De l'équation précédente, nous pouvons déduire que si $n = 2$, alors

$$\pi_2 + e_2 = \pi_1 + p_2 \quad \text{et} \quad \pi_1 = p_1$$

C'est pourquoi

$$|e_2| \leq \gamma_1(2\mathbf{u})\,(|p_1| + |p_2|)$$

Supposons maintenant que l'équation (II.3.7) est vraie pour n et qu'un nombre flottant p_{n+1} est ajouté. Alors

$$\pi_{n+1} = \mathrm{fl}_*(\pi_n + p_{n+1})$$

$$\pi_{n+1} = \mathrm{fl}_* \left(\sum_{i=1}^{n+1} p_i \right)$$

Grâce à la proposition II.3.1 [68],

$$|\pi_{n+1}| \leq (1 + \gamma_n(2\mathbf{u})) \sum_{i=1}^{n+1} |p_i| \tag{II.3.8}$$

Soit e_{n+1} l'erreur sur l'addition flottante de π_n et p_{n+1} :

$$\pi_{n+1} + e_{n+1} = \pi_n + p_{n+1}$$

L'équation (II.3.8), nous permet de dire que

$$|e_{n+1}| \leq 2\mathbf{u}|\pi_{n+1}| \leq 2\mathbf{u}\,(1+\gamma_n(2\mathbf{u}))\sum_{i=1}^{n+1}|p_i|$$

Comme l'équation (II.3.7) est supposée vraie pour n

$$\sum_{i=2}^{n+1}|e_i| \leq (\gamma_{n-1}(2\mathbf{u}) + 2\mathbf{u}(1+\gamma_n(2\mathbf{u})))\sum_{i=1}^{n+1}|p_i|$$

Ainsi nous avons grâce à la proposition A.0.1 en annexe

$$\sum_{i=2}^{n+1}|e_i| \leq \gamma_n(2\mathbf{u})\sum_{i=1}^{n+1}|p_i|$$

Donc le lemme (II.3.7) est prouvé par récurrence. □

Une borne sur l'erreur absolue de l'algorithme II.2 (`FastCompSum`) en arrondi dirigé est présentée dans la proposition II.3.7.

Proposition II.3.7 ([54])**.** *Soit l'algorithme II.2 (*`FastCompSum`*) utilisé, en arrondi dirigé, sur les nombres flottants $p_i \in \mathbb{F}$, $1 \leq i \leq n$. Soient $s = \sum p_i$ et $S = \sum |p_i|$. Si $n\mathbf{u} < \frac{1}{2}$, alors, même en présence d'*underflow*,*

$$|\texttt{res} - s| \leq 2\mathbf{u}|s| + 2(1+2\mathbf{u})\gamma_n^2(2\mathbf{u})S.$$

Démonstration. Grâce à la proposition II.3.5, nous savons que

$$|e_i - q_i| \leq 2\mathbf{u}|e_i| \tag{II.3.9}$$

s étant le résultat exact de l'addition des n nombres flottants p_i et π_n étant le résultat flottant de cette addition :

$$s = \sum_{i=1}^{n} p_i = \pi_n + \sum_{i=2}^{n} e_i \tag{II.3.10}$$

L'erreur sur le nombre flottant `res` calculé grâce à l'algorithme II.2 en arrondi dirigé est

$$|\texttt{res} - s| = |\,\mathrm{fl}_*(\pi_n + \sigma_n) - s|$$

C'est pourquoi

$$|\texttt{res} - s| = |(1+\varepsilon)(\pi_n + \sigma_n) - s| \quad \text{avec} \quad |\varepsilon| \leq 2\mathbf{u}$$

et

$$|\texttt{res} - s| = |(1+\varepsilon)(\pi_n + \sigma_n - s) + \varepsilon s|$$

Grâce à l'équation (II.3.10),

$$|\texttt{res} - s| = |(1+\varepsilon)(\sigma_n - \sum_{i=2}^{n} e_i) + \varepsilon s|$$

Ainsi

$$|\texttt{res} - s| \leq (1+2\mathbf{u})|\sigma_n - \sum_{i=2}^{n} e_i| + 2\mathbf{u}|s| \tag{II.3.11}$$

Évaluons une borne d'erreur sur $|\sigma_n - \sum_{i=2}^{n} e_i|$.

$$|\sigma_n - \sum_{i=2}^{n} e_i| \leq |\sigma_n - \sum_{i=2}^{n} q_i| + |\sum_{i=2}^{n} q_i - \sum_{i=2}^{n} e_i|$$

L'erreur commise sur σ_n est obtenue grâce à la proposition II.3.1 [68]

$$|\sigma_n - \sum_{i=2}^{n} q_i| \leq \gamma_{n-2}(2\mathbf{u}) \sum_{i=2}^{n} |q_i| \tag{II.3.12}$$

Grâce à l'équation (II.3.9),

$$|\sum_{i=2}^{n} q_i - \sum_{i=2}^{n} e_i| \leq 2\mathbf{u} \sum_{i=2}^{n} |e_i| \tag{II.3.13}$$

Les équations (II.3.12) et (II.3.13) nous permettent de déduire que

$$|\sigma_n - \sum_{i=2}^{n} e_i| \leq \gamma_{n-2}(2\mathbf{u}) \sum_{i=2}^{n} |q_i| + 2\mathbf{u} \sum_{i=2}^{n} |e_i| \tag{II.3.14}$$

Établissons maintenant une borne sur $\sum_{i=2}^{n} |q_i|$:

$$\sum_{i=2}^{n} |q_i| \leq \sum_{i=2}^{n} |e_i| + \sum_{i=2}^{n} |q_i - e_i|$$

Grâce aux équations (II.3.9) et (II.3.7),

$$\sum_{i=2}^{n} |q_i| \leq \gamma_{n-1}(2\mathbf{u}) \sum_{i=1}^{n} |p_i| + 2\mathbf{u} \sum_{i=2}^{n} |e_i|$$

L'équation (II.3.7) nous permet de dire que

$$\sum_{i=2}^{n} |q_i| \leq \gamma_{n-1}(2\mathbf{u}) \sum_{i=1}^{n} |p_i| + 2\mathbf{u}\gamma_{n-1}(2\mathbf{u}) \sum_{i=1}^{n} |p_i|$$

C'est pourquoi

$$\sum_{i=2}^{n} |q_i| \leq (\gamma_{n-1}(2\mathbf{u}) + 2\mathbf{u}\gamma_{n-1}(2\mathbf{u})) \sum_{i=1}^{n} |p_i|$$

Ainsi, grâce à la proposition A.0.2

$$\sum_{i=2}^{n} |q_i| \leq \gamma_n(2\mathbf{u}) \sum_{i=1}^{n} |p_i| \tag{II.3.15}$$

Le lemme II.3.6, les équations (II.3.14) et (II.3.7) et (II.3.15), nous permettent de dire que

$$|\sigma_n - \sum_{i=2}^{n} e_i| \leq \gamma_{n-2}(2\mathbf{u})\gamma_n(2\mathbf{u}) \sum_{i=1}^{n} |p_i| + 2\mathbf{u}\gamma_{n-1}(2\mathbf{u}) \sum_{i=1}^{n} |p_i|$$

Ainsi

$$|\sigma_n - \sum_{i=2}^{n} e_i| \leq (\gamma_{n-2}(2\mathbf{u})\gamma_n(2\mathbf{u}) + 2\mathbf{u}\gamma_{n-1}(2\mathbf{u})) \sum_{i=1}^{n} |p_i|$$

Nous pouvons donc dire grâce à la proposition A.0.3 que

$$|\sigma_n - \sum_{i=2}^{n} e_i| \leq 2\gamma_n^2(2\mathbf{u}) \sum_{i=1}^{n} |p_i| \tag{II.3.16}$$

Nous pouvons ainsi conclure grâce aux équations (II.3.11) et (II.3.16) que

$$|\texttt{res} - s| \leq 2\mathbf{u}|s| + 2(1 + 2\mathbf{u})\gamma_n^2(2\mathbf{u}) \sum_{i=1}^{n} |p_i|$$

□

À partir de la proposition II.3.7, une borne pour l'erreur relative sur le résultat de l'algorithme II.2 (`FastCompSum`) obtenu en arrondi dirigé est déduite dans le corollaire II.3.8.

Corollaire II.3.8. *En arrondi dirigé, si* $n\mathbf{u} < \frac{1}{2}$*, alors, même en présence d'*underflow*, le résultat* `res` *de l'algorithme II.2 (*`FastCompSum`*) vérifie*

$$\frac{|\texttt{res} - s|}{|s|} \leq 2\mathbf{u} + 2(1 + 2\mathbf{u})\gamma_n^2(2\mathbf{u}) \operatorname{cond}\left(\sum p_i\right).$$

Or comme $\gamma_n(2\mathbf{u}) \approx 2n\mathbf{u}$, la borne sur l'erreur relative présentée dans le corollaire II.3.8 provient essentiellement de $(n\mathbf{u})^2$ fois le conditionnement auquel on ajoute l'arrondi $2\mathbf{u}$ dû à la précision de travail. Comme en arrondi au plus près, la précision du résultat fourni par `FastCompSum` en arrondi dirigé est similaire à celle que nous aurions obtenu en doublant la précision de travail.

L'impact du mode d'arrondi aléatoire sur l'algorithme I.3 (`TwoSumPriest`) va maintenant être étudié. Cet algorithme sera plus particulièrement utile dans la section II.6. L'algorithme de sommation `TwoSumPriest` est décrit ci-après (algorithme II.3).

Le lemme II.3.9 est fourni pour une utilisation ultérieure en section II.6.

Lemme II.3.9. *Soit l'algorithme II.3 (*`PriestCompSum`*) appliqué, en arrondi dirigé, à* n *nombres flottants* $p_i \in \mathbb{F}$, $1 \leq i \leq n$. *Soient* $s = \sum p_i$ *et* $S = \sum |p_i|$. *Si* $n\mathbf{u} < \frac{1}{2}$*, alors*

$$\sum_{i=2}^{n} |q_i| + |\pi_n| \leq |s| + 2\gamma_{n-1}(2\mathbf{u})S. \tag{II.3.17}$$

Algorithme II.3 `PriestCompSum` : Sommation compensée de n nombres flottants $p = \{p_i\}$ utilisant `TwoSumPriest`

fonction `res` = `PriestCompSum`(p)

1: $\pi_1 \leftarrow p_1$
2: $\sigma_1 \leftarrow 0$
3: **pour** $i = 2$ à n **faire**
4: $\quad [\pi_i, q_i] \leftarrow$ `TwoSumPriest`(π_{i-1}, p_i)
5: $\quad \sigma_i \leftarrow \sigma_{i-1} + q_i$
6: **fin pour**
7: `res` $\leftarrow \pi_n + \sigma_n$

Démonstration. Comme

$$\sum_{i=2}^{n} |q_i| + |\pi_n| = \sum_{i=2}^{n} |q_i| + |s - \sum_{i=2}^{n} q_i|,$$

nous avons

$$\sum_{i=2}^{n} |q_i| + |\pi_n| \leq |s| + 2\sum_{i=2}^{n} |q_i|. \tag{II.3.18}$$

Grâce au lemme II.3.6, nous pouvons déduire que

$$\sum_{i=2}^{n} |q_i| \leq \gamma_{n-1}(2\mathbf{u}) \sum_{i=1}^{n} |p_i| = \gamma_{n-1}(2\mathbf{u})S \tag{II.3.19}$$

en effet la différence entre `FastCompSum` et `PriestCompSum` est dans le calcul de l'erreur par un algorithme de transformations exacts différents.

Finalement, l'équation II.3.17 provient des équations II.3.18 et II.3.19. □

Une borne de l'erreur absolue sur le résultat de l'algorithme II.3 (`PriestCompSum`) obtenue en arrondi dirigé est présentée dans la proposition II.3.10.

Proposition II.3.10. *Supposons que l'algorithme* `PriestCompSum` *est appliqué en arrondi dirigé sur les nombres flottants* $p_i \in \mathbb{F}$, $1 \leq i \leq n$. *Soient* $s = \sum p_i$ *et* $S = \sum |p_i|$. *Si* $n\mathbf{u} < \frac{1}{2}$, *alors, même en présence d'*underflow,

$$|\texttt{res} - s| \leq 2\mathbf{u}|s| + \gamma_{n-1}^2(2\mathbf{u})S. \tag{II.3.20}$$

La preuve est similaire à celle fournie en [114] pour la sommation compensée en arrondi au plus près.

Démonstration. Comme $\sigma_n = \mathrm{fl}_*(\sum_{i=2}^{n} q_i)$, nous avons

$$|\sigma_n - \sum_{i=2}^{n} q_i| \leq \gamma_{n-2}(2\mathbf{u}) \sum_{i=2}^{n} |q_i|.$$

Donc, grâce à l'équation II.3.19, nous pouvons déduire que

$$|\sigma_n - \sum_{i=2}^{n} q_i| \leq \gamma_{n-2}(2\mathbf{u})\gamma_{n-1}(2\mathbf{u})S.$$

Comme l'algorithme II.3 est exécuté à l'arrondi dirigé, nous avons :

$$\texttt{res} = \mathrm{fl}_*(\pi_n + \sigma_n) = (1+\varepsilon)(\pi_n + \sigma_n) \quad \text{with} \quad |\varepsilon| \leq 2\mathbf{u},$$

$$|\texttt{res} - s| = |\,\mathrm{fl}_*(\pi_n + \sigma_n) - s|,$$

$$|\texttt{res} - s| = |(1+\varepsilon)(\pi_n + \sigma_n - s) + \varepsilon s|,$$

$$|\texttt{res} - s| = |(1+\varepsilon)(\pi_n + \sum_{i=2}^{n} q_i - s) + (1+\varepsilon)(\sigma_n - \sum_{i=2}^{n} q_i) + \varepsilon s|.$$

Comme

$$s = \sum_{i=1}^{n} p_i = \pi_n + \sum_{i=2}^{n} q_i, \tag{II.3.21}$$

alors

$$|\texttt{res} - s| \leq (1+2\mathbf{u})|\sigma_n - \sum_{i=2}^{n} q_i| + 2\mathbf{u}|s|,$$

$$|\texttt{res} - s| \leq (1+2\mathbf{u})\gamma_{n-2}(2\mathbf{u})\gamma_{n-1}(2\mathbf{u})S + 2\mathbf{u}|s|. \tag{II.3.22}$$

Nous avons

$$(1+2\mathbf{u})\gamma_{n-1}(2\mathbf{u}) < \gamma_n(2\mathbf{u}). \tag{II.3.23}$$

En effet,

$$\gamma_n(2\mathbf{u}) - (1+2\mathbf{u})\gamma_{n-1}(2\mathbf{u}) = 2\mathbf{u}\left(1 + \frac{2n\mathbf{u}}{(1-2(n-1)\mathbf{u})\,(1-2n\mathbf{u})}\right),$$

et donc :

$$\gamma_n(2\mathbf{u}) - (1+2\mathbf{u})\gamma_{n-1}(2\mathbf{u}) > 0.$$

Finalement, l'équation II.3.20 est déduite des équations II.3.22 et II.3.23. □

Nous pouvons déduire de la proposition II.3.10 une borne pour l'erreur relative sur le résultat de l'algorithme II.3 (`PriestCompSum`) obtenue en arrondi dirigé. Elle est présentée dans le corollaire II.3.11.

Corollaire II.3.11. *En arrondi dirigé, si* $n\mathbf{u} < \frac{1}{2}$*, alors, même en présence d'*underflow*, le résultat* `res` *de l'algorithme II.3 (*`PriestCompSum`*) vérifie*

$$\frac{|\texttt{res} - s|}{|s|} \leq 2\mathbf{u} + \gamma_{n-1}^2(2\mathbf{u})\,\mathrm{cond}\left(\sum p_i\right). \tag{II.3.24}$$

Tout comme pour l'algorithme II.2 (`FastCompSum`), nous pouvons déduire du corollaire II.3.11 que la borne d'erreur relative sur le résultat de l'algorithme II.3 (`PriestCompSum`) calculé en arrondi dirigé est essentiellement $(n\mathbf{u})^2$ fois le conditionnement plus l'arrondi $2\mathbf{u}$ dû à la précision de travail.

II.4 Produit scalaire

Nous allons maintenant présenter la précision obtenue à l'aide du produit scalaire classique, mais également décrire un algorithme permettant de calculer un produit scalaire en arrondi au plus près comme si nous doublions la précision de travail. Puis nous analyserons l'impact d'un arrondi dirigé sur ces algorithmes. Nous supposons dans cette partie qu'aucun *underflow* ne survient.

II.4.1 Produit scalaire classique

Le produit scalaire classique est présenté ci-après (algorithme II.4).

Algorithme II.4 Produit scalaire classique de $x = \{x_i\}$ et $y = \{y_i\}$, $1 \leq i \leq n$

fonction $\texttt{res} = \texttt{Dot}(x, y)$

1: $s_1 \leftarrow x_1 y_1$
2: **pour** $i = 2 : n$ **faire**
3: $\quad s_i \leftarrow x_i \cdot y_i + s_{i-1}$
4: **fin pour**
5: $\texttt{res} \leftarrow s_n$

La proposition II.4.1 résume les propriétés de cet algorithme.

Proposition II.4.1. *Soient les nombres flottants $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, soit $\texttt{res} \in \mathbb{F}$ le résultat de l'algorithme II.4* (Dot). *En arrondi au plus près, si $n\mathbf{u} < 1$, nous avons*

$$|\texttt{res} - x^T y| \leq \gamma_n(\mathbf{u})|x^T||y|,$$

En arrondi dirigé, si $n\mathbf{u} < \frac{1}{2}$, nous avons

$$|\texttt{res} - x^T y| \leq \gamma_n(2\mathbf{u})|x^T||y|.$$

Démonstration. La preuve est donnée dans [68, p.63]. □

Les inégalités précédentes peuvent être réécrites à partir du conditionnement défini par

$$\mathrm{cond}(x^T y) = 2\frac{|x|^T|y|}{|x^T y|}.$$

Corollaire II.4.2. *En arrondi au plus près, si $n\mathbf{u} < 1$, le résultat* res *de l'algorithme II.4 vérifie*

$$\frac{|\texttt{res} - x^T y|}{|x^T y|} \leq \frac{1}{2}\gamma_n(\mathbf{u})\,\mathrm{cond}(x^T y).$$

En arrondi dirigé, s $n\mathbf{u} < \frac{1}{2}$, le résultat res *de l'algorithme II.4 vérifie*

$$\frac{|\texttt{res} - x^T y|}{|x^T y|} \leq \frac{1}{2}\gamma_n(2\mathbf{u})\,\mathrm{cond}(x^T y).$$

II.4.2 Produit scalaire compensé en arrondi au plus près

Un algorithme permettant de compenser le produit scalaire a été introduit dans [114]. Il se base sur les transformations exactes `TwoSum` [85] et `TwoProd` [32] pour calculer la somme et le produit de deux nombres flottants ainsi que l'erreur générée par ces calculs en arrondi au plus près. Plutôt que d'utiliser l'algorithme `TwoProd`, il est possible d'utiliser l'algorithme `TwoProdFMA` à condition d'avoir à disposition les FMA, cf. algorithme I.6.

L'algorithme `CompDot` (algorithme II.5) est un produit scalaire compensé utilisant `FastTwoSum` et `TwoProdFMA`. Nous pouvons remarquer qu'en arrondi au plus près, l'erreur commise est la même quelle que soit la transformation exacte utilisée pour la sommation.

Algorithme II.5 Produit scalaire compensé des vecteurs de nombres flottants $x = \{x_i\}$ et $y = \{y_i\}$, $1 \leq i \leq n$

fonction `res`=`CompDot`(x, y)

1: $[p, s] \leftarrow$ `TwoProdFMA`(x_1, y_1)
2: **pour** $i = 2$ à n **faire**
3: $\quad [h, r] \leftarrow$ `TwoProdFMA`(x_i, y_i)
4: $\quad [p, q] \leftarrow$ `FastTwoSum`(p, h)
5: $\quad s \leftarrow s + (q + r)$
6: **fin pour**
7: `res` $\leftarrow p + s$

L'erreur sur le résultat `res` de l'algorithme II.5 en arrondi au plus près a été analysée dans [114]. Une borne sur l'erreur absolue est rappelée dans la proposition II.4.3.

Proposition II.4.3 ([114]). *Soient les nombres flottants $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, et $\texttt{res} \in \mathbb{F}$ le résultat calculé par l'algorithme II.5 (`CompDot`) en arrondi au plus près. Si $n\mathbf{u} < 1$, alors*

$$|\texttt{res} - x^T y| \leq \mathbf{u}|x^T y| + \gamma_n^2(\mathbf{u})|x^T||y|. \tag{II.4.1}$$

Dans le corollaire II.4.4, l'équation II.4.1 est réécrite en fonction du conditionnement du produit scalaire.

Corollaire II.4.4 ([114]). *En arrondi au plus près, si $n\mathbf{u} < 1$, alors, le résultat `res` de l'algorithme II.5 (`CompDot`) vérifie*

$$\frac{|\texttt{res} - x^T y|}{|x^T y|} \leq \mathbf{u} + \frac{1}{2}\gamma_n^2(\mathbf{u})\,\mathrm{cond}\left(x^T y\right). \tag{II.4.2}$$

Ainsi, l'algorithme `CompDot` permet de calculer le produit scalaire quasiment comme si la précision de travail était doublée. Il s'agit du même phénomène que pour la sommation compensée.

II.4.3 Produit scalaire compensé en arrondi dirigé

Nous allons maintenant étudier l'utilisation d'un mode d'arrondi dirigé sur l'algorithme II.5 (`CompDot`). L'algorithme II.6 est une réécriture de cet algorithme pour simplifier l'analyse de l'erreur commise.

Algorithme II.6 Algorithme équivalent à l'algorithme II.5

fonction `res`=`CompDot`(x, y)

1: $[p_1, s_1] \leftarrow$ `TwoProdFMA`(x_1, y_1)
2: **pour** $i = 2$ à n **faire**
3: $\quad[h_i, r_i] \leftarrow$ `TwoProdFMA`(x_i, y_i)
4: $\quad[p_i, q_i] \leftarrow$ `FastTwoSum`(p_{i-1}, h_i)
5: $\quad s_i \leftarrow s_{i-1} + (q_i + r_i)$
6: **fin pour**
7: `res` $\leftarrow p_n + s_n$

Une borne d'erreur absolue sur le résultat `res` de l'algorithme II.5 est donnée dans la proposition II.4.5.

Proposition II.4.5. *Soient les nombres flottants $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, et* `res` $\in \mathbb{F}$ *le résultat calculé par l'algorithme II.5 (*`CompDot`*) en utilisant un mode d'arrondi dirigé. Si $(n+1)\mathbf{u} < \frac{1}{2}$, alors*

$$|\texttt{res} - x^T y| \leq 2\mathbf{u}|x^T y| + 2\gamma_{n+1}^2(2\mathbf{u})|x^T||y|.$$

Démonstration. Grâce à l'algorithme `TwoProdFMA`, nous avons

$$p_1 + s_1 = x_1 y_1, \tag{II.4.3}$$

et pour $i \geq 2$,

$$h_i + r_i = x_i y_i. \tag{II.4.4}$$

Grâce à la proposition II.3.5, nous pouvons déduire que

$$p_i + e_i = p_{i-1} + h_i \quad \text{avec} \quad |q_i - e_i| \leq 2\mathbf{u}|e_i| \tag{II.4.5}$$
et e_i le terme d'erreur réel sur la sommation de `FastTwoSum`

Nous pouvons déduire de l'équation II.4.4 que

$$e_i + r_i = (p_{i-1} + h_i - p_i) + (x_i y_i - h_i) = x_i y_i + p_{i-1} - p_i.$$

Grâce à l'équation II.4.3, nous savons que

$$s_1 + \sum_{i=2}^{n}(e_i + r_i) = (x_1 y_1 - p_1) + \left(\sum_{i=2}^{n} x_i y_i + p_1 - p_n\right) = x^T y - p_n. \tag{II.4.6}$$

L'algorithme `TwoProdFMA` est exécuté avec un mode d'arrondi dirigé pour $i \geq 2$, donc

$$|r_i| \leq 2\mathbf{u}|x_i y_i|.$$

C'est pourquoi nous avons

$$\sum_{i=2}^{n} |r_i| \leq 2\mathbf{u} \sum_{i=2}^{n} |x_i y_i|,$$

et

$$|s_1| + \sum_{i=2}^{n} |r_i| \leq 2\mathbf{u}|x^T||y|. \tag{II.4.7}$$

Du lemme II.3.6, nous déduisons que

$$\sum_{i=2}^{n} |e_i| \leq \gamma_{n-1}(2\mathbf{u}) \left(|p_1| + \sum_{i=2}^{n} |h_i| \right).$$

Ainsi

$$\sum_{i=2}^{n} |e_i| \leq \gamma_{n-1}(2\mathbf{u}) \left(\sum_{i=1}^{n} |\,\mathrm{fl}_*(x_i y_i)| \right),$$

et

$$\sum_{i=2}^{n} |e_i| \leq (1 + 2\mathbf{u})\gamma_{n-1}(2\mathbf{u})|x^T||y|. \tag{II.4.8}$$

Des équations II.3.23 et II.4.8, nous avons

$$\sum_{i=2}^{n} |e_i| \leq \gamma_n(2\mathbf{u})|x^T||y|. \tag{II.4.9}$$

De l'équation II.4.5, nous savons que

$$\sum_{i=2}^{n} |q_i - e_i| \leq 2\mathbf{u} \sum_{i=2}^{n} |e_i|. \tag{II.4.10}$$

Ainsi, de l'équation II.4.9, nous pouvons dire que

$$\sum_{i=2}^{n} |q_i - e_i| \leq 2\mathbf{u}\gamma_n(2\mathbf{u})|x^T||y|. \tag{II.4.11}$$

Nous avons

$$\sum_{i=2}^{n} |q_i| \leq \sum_{i=2}^{n} |e_i| + \sum_{i=2}^{n} |q_i - e_i|\,.$$

C'est pourquoi de l'équation II.4.10 nous pouvons déduire que

$$\sum_{i=2}^{n} |q_i| \leq (1 + 2\mathbf{u}) \sum_{i=2}^{n} |e_i|\,.$$

Des équations II.3.23 et II.4.9, il apparaît que

$$\sum_{i=2}^{n} |q_i| \leq \gamma_{n+1}(2\mathbf{u})|x^T||y|. \tag{II.4.12}$$

Pour un usage ultérieur, nous estimons une borne d'erreur sur l'expression suivante :

$$|s_1 + \sum_{i=2}^{n}(q_i + r_i) - s_n| = \left| s_1 + \sum_{i=2}^{n}(q_i + r_i) - \mathrm{fl}_* \left(s_1 + \sum_{i=2}^{n}(q_i + r_i) \right) \right|.$$

De la proposition II.3.1, nous pouvons dire que

$$|s_1 + \sum_{i=2}^{n}(q_i + r_i) - s_n| \leq \gamma_{n-1}(2\mathbf{u}) \left(|s_1| + \sum_{i=2}^{n} |\mathrm{fl}_*(q_i + r_i)| \right). \tag{II.4.13}$$

Comme nous utilisons un mode d'arrondi dirigé, nous avons

$$\sum_{i=2}^{n} |\mathrm{fl}_*(q_i + r_i)| \leq (1 + 2\mathbf{u}) \sum_{i=2}^{n} |q_i + r_i|.$$

Ainsi de l'équation II.4.13, nous déduisons que

$$|s_1 + \sum_{i=2}^{n}(q_i + r_i) - s_n| \leq (1 + 2\mathbf{u})\gamma_{n-1}(2\mathbf{u}) \left(|s_1| + \sum_{i=2}^{n} |q_i + r_i| \right),$$

et de l'équation II.3.23,

$$|s_1 + \sum_{i=2}^{n}(q_i + r_i) - s_n| \leq \gamma_n(2\mathbf{u}) \left(|s_1| + \sum_{i=2}^{n} |q_i + r_i| \right). \tag{II.4.14}$$

Grâce aux équations II.4.7 et II.4.12, il apparaît que

$$|s_1 + \sum_{i=2}^{n}(q_i + r_i) - s_n| \leq \gamma_n(2\mathbf{u}) \left(2\mathbf{u} + \gamma_{n+1}(2\mathbf{u})\right) |x^T||y|. \tag{II.4.15}$$

Nous pouvons déduire de l'équation II.4.6 que

$$|\left(x^T y - p_n\right) - s_n| = \left| s_1 + \sum_{i=2}^{n}(e_i + r_i) - s_n \right|.$$

Ainsi :

$$|x^T y - p_n - s_n| = \left| s_1 + \sum_{i=2}^{n}(q_i + r_i) - s_n + \sum_{i=2}^{n}(e_i - q_i) \right|,$$

et

$$|x^T y - p_n - s_n| \leq \left| s_1 + \sum_{i=2}^{n}(q_i + r_i) - s_n \right| + \sum_{i=2}^{n} |e_i - q_i|. \tag{II.4.16}$$

C'est pourquoi nous avons, grâce aux équations II.4.11 et II.4.15

$$|x^T y - p_n - s_n| \leq \gamma_n(2\mathbf{u}) \left(4\mathbf{u} + \gamma_{n+1}(2\mathbf{u})\right) |x^T||y|. \tag{II.4.17}$$

Montrons que $\gamma_{n+1}(2\mathbf{u}) \geq 4\mathbf{u}$. Nous avons

$$\gamma_{n+1}(2\mathbf{u}) - 4\mathbf{u} = \frac{2(n+1)\mathbf{u}}{1 - 2(n+1)\mathbf{u}} - 4\mathbf{u},$$

et

$$\gamma_{n+1}(2\mathbf{u}) - 4\mathbf{u} = \frac{2\mathbf{u}}{1 - 2(n+1)\mathbf{u}} \left(n - 1 + 4(n+1)\mathbf{u}\right).$$

donc $(n+1)\mathbf{u} < \frac{1}{2}$, nous pouvons déduire que $\gamma_{n+1}(2\mathbf{u}) - 4\mathbf{u} \geq 0$.

C'est pourquoi de l'équation II.4.17, nous avons

$$|x^T y - p_n - s_n| \leq 2\gamma_n(2\mathbf{u})\gamma_{n+1}(2\mathbf{u})|x^T||y|. \tag{II.4.18}$$

En raison de l'utilisation d'un arrondi dirigé lors de l'exécution de l'algorithme II.6 :

$$|\texttt{res} - x^T y| = |(1+\varepsilon)(p_n + s_n) - x^T y| \quad \text{avec} \quad |\varepsilon| \leq 2\mathbf{u}.$$

C'est pourquoi

$$|\texttt{res} - x^T y| = |\varepsilon x^T y + (1+\varepsilon)(p_n + s_n - x^T y)|,$$

et

$$|\texttt{res} - x^T y| \leq 2\mathbf{u}|x^T y| + (1+2\mathbf{u})|p_n + s_n - x^T y|. \tag{II.4.19}$$

De l'équation II.4.18, il apparaît que

$$|\texttt{res} - x^T y| \leq 2\mathbf{u}|x^T y| + 2(1+2\mathbf{u})\gamma_n(2\mathbf{u})\gamma_{n+1}(2\mathbf{u})|x^T||y|.$$

Finalement de l'équation II.3.23, nous pouvons conclure que

$$|\texttt{res} - x^T y| \leq 2\mathbf{u}|x^T y| + 2\gamma_{n+1}^2(2\mathbf{u})|x^T||y|.$$

□

De la proposition II.4.5, nous en déduisons dans le corollaire II.4.6 une borne d'erreur relative sur le résultat de l'algorithme II.5 (`CompDot`) obtenue avec un mode d'arrondi dirigé.

Corollaire II.4.6. *En arrondi dirigé, si* $(n+1)\mathbf{u} < \frac{1}{2}$*, alors, le résultat* `res` *de l'algorithme II.5 (*`CompDot`*) vérifie*

$$\frac{|\texttt{res} - x^T y|}{|x^T y|} \leq 2\mathbf{u} + \gamma_{n+1}^2(2\mathbf{u})\,\mathrm{cond}\left(x^T y\right).$$

D'après le corollaire II.4.6, la borne d'erreur relative sur le résultat de l'algorithme II.5 (`CompDot`) calculé avec un mode d'arrondi dirigé est principalement $(n\mathbf{u})^2$ fois le conditionnement plus l'arrondi $2\mathbf{u}$ dû à la précision de travail. Nous avons ainsi un résultat de qualité numérique similaire à celle que nous aurions obtenue en doublant la précision de travail.

II.5 Schéma de Horner compensé

Nous allons maintenant présenter les résultats connus sur le schéma de Horner pour l'évaluation polynomiale. Puis nous présenterons l'évaluation polynomiale compensée ainsi que l'erreur en arrondi au plus près. Enfin nous étudierons l'impact du mode d'arrondi dirigé sur l'erreur commise. Nous supposons qu'aucun *underflow* ne survient.

II.5.1 Évaluation polynomiale de Horner

Le schéma de Horner (algorithme II.7) permet d'évaluer un polynôme :

$$p(x) = \sum_{i=0}^{n} a_i x^i.$$

Algorithme II.7 Évaluation polynomiale de Horner

fonction `res` = `Horner`(p, x)

1: $s_n \leftarrow a_n$
2: **pour** $i = n-1$ à 0 **faire**
3: $\quad s_i \leftarrow s_{i+1} \cdot x + a_i$
4: **fin pour**
5: `res` $\leftarrow s_0$

Proposition II.5.1 ([68]). *Pour tous les modes d'arrondi, une borne d'erreur sur le résultat de l'algorithme II.7 est*

$$|p(x) - \texttt{res}| \leq \gamma_{2n}(2\mathbf{u}) \sum_{i=0}^{n} |a_i||x|^i = \gamma_{2n}(2\mathbf{u})\widetilde{p}(|x|)$$

avec $\widetilde{p}(x) = \sum_{i=0}^{n} |a_i| x^i$.

Nous pouvons exprimer ce résultat à partir du conditionnement d'un polynôme défini par :

$$\operatorname{cond}(p, x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|p(x)|} = \frac{\widetilde{p}(|x|)}{|p(x)|}. \tag{II.5.1}$$

Ainsi nous avons :

$$\frac{|p(x) - \texttt{res}|}{|p(x)|} \leq \gamma_{2n}(2\mathbf{u}) \operatorname{cond}(p, x).$$

Si les instructions `FMA` sont présentes alors la ligne 3 de l'algorithme II.7 $s_i \leftarrow s_{i+1} \cdot x + a_i$ peut être réécrite $s_i \leftarrow$ `FMA`(s_{i+1}, x, a_i). Cela améliore la borne d'erreur :

$$|p(x) - \texttt{res}| \leq \gamma_n(2\mathbf{u})\widetilde{p}(|x|).$$

II.5.2 Évaluation polynomiale de Horner compensée

L'algorithme II.8 présente le schéma de Horner compensé grâce aux transformations exactes [59, 60]. En arrondi au plus près, nous pouvons utiliser au choix `TwoSum`, `FastTwoSum` ou `TwoSumPriest` pour l'addition et `TwoProd` ou `TwoProdFMA` pour la multiplication. Nous utiliserons ici `FastTwoSum` et `TwoProdFMA` pour l'algorithme II.8.

Si nous notons p_π et p_σ les polynômes :

$$p_\pi(x) = \sum_{i=0}^{n-1} \pi_i x^i, \qquad p_\sigma(x) = \sum_{i=0}^{n-1} \sigma_i x^i,$$

Algorithme II.8 Évaluation polynomiale de Horner compensée

fonction `res` = `CompHorner`(p, x)

1: $s_n \leftarrow a_n$
2: $r_n \leftarrow 0$
3: **pour** $i = n-1$ à 0 **faire**
4: $\quad [p_i, \pi_i] \leftarrow$ `TwoProdFMA`(s_{i+1}, x)
5: $\quad [s_i, \sigma_i] \leftarrow$ `FastTwoSum`(p_i, a_i)
6: $\quad r_i \leftarrow r_{i+1} \cdot x + (\pi_i + \sigma_i)$
7: **fin pour**
8: `res` $\leftarrow s_0 + r_0$

alors nous avons, grâce aux transformations exactes

$$p(x) = s_0 + p_\pi(x) + p_\sigma(x).$$

Nous avons $s_0 =$ `Horner`(p, x). Ainsi nous pouvons en déduire une nouvelle transformation exacte pour l'évaluation polynomiale :

$$p(x) = \texttt{Horner}(p, x) + p_\pi(x) + p_\sigma(x).$$

Cette compensation consiste à calculer tout d'abord $p_\pi(x) + p_\sigma(x)$, ce qui correspond aux erreurs d'arrondi et à ajouter à cela le résultat de l'évaluation de Horner compensée `Horner`(p, x). Le résultat calculé par l'algorithme II.8 admet une meilleure borne d'erreur que celle obtenue par l'algorithme de Horner classique. La compensation permet de doubler la précision de travail comme nous allons le rappeler dans les théorèmes suivants.

Proposition II.5.2 ([59]). *Soit un polynôme p de degré n ayant des coefficients flottants et un nombre flottant x. En arrondi au plus près, l'erreur directe sur le résultat de l'algorithme* `CompHorner` *est telle que :*

$$|\texttt{CompHorner}(p, x) - p(x)| \leq \mathbf{u}|p(x)| + \gamma_{2n}^2(\mathbf{u})\widetilde{p}(x). \tag{II.5.2}$$

En utilisant le conditionnement tel que défini par l'équation II.5.1, la borne d'erreur de la proposition II.5.2 devient :

$$\frac{|\texttt{CompHorner}(p, x) - p(x)|}{|p(x)|} \leq \mathbf{u} + \gamma_{2n}^2(\mathbf{u})\,\mathrm{cond}(p, x). \tag{II.5.3}$$

Ainsi, la borne d'erreur relative sur le résultat calculé est essentiellement $\gamma_{2n}^2(\mathbf{u})$ fois le conditionnement de l'évaluation polynomiale. Ainsi l'évaluation compensée est aussi précise que si les opérations avaient été effectuées avec une précision doublée.

II.5.3 Évaluation polynomiale de Horner compensée en arrondi dirigé

Nous allons maintenant étudier l'utilisation en arrondi dirigé sur l'algorithme II.8 (`CompHorner`).

Soit τ_i l'erreur d'arrondi sur l'addition flottante de p_i et a_i (τ_i peut ne pas être un nombre flottant) :

$$s_i + \tau_i = p_i + a_i.$$

On en déduit que $s_{i+1} \cdot x = p_i + \pi_i$ et $p_i + a_i = s_i + \tau_i$ avec $|\tau_i - \sigma_i| \leq 2\mathbf{u}\tau_i$. Ainsi, nous avons :

$$s_i = s_{i+1} \cdot x - \pi_i - \tau_i \quad \text{pour } i = 0, \ldots, n-1.$$

Par récurrence nous en déduisons que

$$p(x) = s_0 + p_\pi(x) + p_\tau(x),$$

avec

$$s_0 = \mathrm{fl}_*(p(x)), \quad p_\pi(x) = \sum_{i=0}^{n-1} \pi_i x^i, \quad \text{et} \quad p_\tau(x) = \sum_{i=0}^{n-1} \tau_i x^i. \tag{II.5.4}$$

Pour rappel

$$p_\sigma(x) = \sum_{i=0}^{n-1} \sigma_i x^i. \tag{II.5.5}$$

Par la suite, nous noterons $e(x) = p_\pi(x) + p_\sigma(x)$. Dans ce cas, nous avons $p(x) = \mathrm{fl}(p(x)) + e(x) + (p_\tau - p_\sigma)(x)$ et $\texttt{res} = \mathrm{fl}(p(x) + e(x))$.

Lemme II.5.3. *Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ un polynôme avec $a_i \in \mathbb{F}$, $0 \leq i \leq n$ et $x \in \mathbb{F}$. Soit p_π et p_σ définis par (II.5.4) et (II.5.5). Nous avons alors*

$$\widetilde{p_\pi}(|x|) + \widetilde{p_\sigma}(|x|) \leq \gamma_{2n+1}(2\mathbf{u})\widetilde{p}(|x|),$$

avec $\widetilde{p}(x) = \sum_{i=0}^{n} |a_i| x^i$.

Démonstration. D'après l'équation (II.2.1), nous avons, pour $i = 1, \ldots, n$,

$$|p_{n-i}| = |\,\mathrm{fl}_*(s_{n-i+1} \cdot x)| \leq (1 + 2\mathbf{u})|s_{n-i+1}||x|$$

et

$$|s_{n-i}| = |\,\mathrm{fl}_*(p_{n-i} + a_{n-i})| \leq (1 + 2\mathbf{u})(|p_{n-i}| + |a_{n-i}|).$$

Montrons par récurrence que pour $i = 1, \ldots, n$ que

$$|p_{n-i}| \quad \leq \quad (1 + \gamma_{2i-1}(2\mathbf{u})) \sum_{j=1}^{i} |a_{n-i+j}||x^j| \tag{II.5.6}$$

et

$$|s_{n-i}| \quad \leq \quad (1 + \gamma_{2i}(2\mathbf{u})) \sum_{j=0}^{i} |a_{n-i+j}||x^j|. \tag{II.5.7}$$

Pour $i = 1$, comme $s_n = a_n$, nous avons

$$|p_{n-1}| \leq (1 + 2\mathbf{u})|a_n||x| \leq (1 + \gamma_1(2\mathbf{u}))|a_n||x|$$

et donc l'équation (II.5.6) est vérifiée. De la même façon, comme

$$|s_{n-1}| \leq (1 + 2\mathbf{u})\left((1 + \gamma_1(2\mathbf{u}))|a_n||x| + |a_{n-1}|\right) \leq (1 + \gamma_2(2\mathbf{u}))\left(|a_n||x| + |a_{n-1}|\right)$$

alors l'équation (II.5.7) est également vérifiée. Supposons que les équations (II.5.6) et (II.5.7) sont vraies pour un entier i, $1 \leq i < n$. Alors nous avons

$$|p_{n-(i+1)}| \leq (1+2\mathbf{u})|s_{n-i}||x|.$$

Par hypothèse, nous pouvons déduire que

$$\begin{aligned} |p_{n-(i+1)}| &\leq (1+2\mathbf{u})(1+\gamma_{2i}(2\mathbf{u}))\sum_{j=0}^{i}|a_{n-i+j}||x^{j+1}| \\ &\leq (1+\gamma_{2(i+1)-1}(2\mathbf{u}))\sum_{j=1}^{i+1}|a_{n-(i+1)+j}||x^{j}|. \end{aligned}$$

Par conséquent

$$\begin{aligned} |s_{n-(i+1)}| &\leq (1+2\mathbf{u})(|p_{n-(i+1)}|+|a_{n-(i+1)}|) \\ &\leq (1+2\mathbf{u})(1+\gamma_{2(i+1)-1}(2\mathbf{u}))\left[\sum_{j=1}^{i+1}|a_{n-(i+1)+j}||x^{j}|+|a_{n-(i+1)}|\right] \\ &\leq (1+\gamma_{2(i+1)}(2\mathbf{u}))\sum_{j=0}^{i+1}|a_{n-(i+1)+j}||x^{j}|. \end{aligned}$$

Les équations (II.5.6) et (II.5.7) sont ainsi vérifiées par récurrence. Ainsi quel que soit $i = 1, \ldots, n$, nous avons

$$|p_{n-i}||x^{n-i}| \leq (1+\gamma_{2i-1}(2\mathbf{u}))\widetilde{p}(x)$$

et

$$|s_{n-i}||x^{n-i}| \leq (1+\gamma_{2i}(2\mathbf{u}))\widetilde{p}(x).$$

D'après l'équation (II.2.2), nous avons $|\pi_i| \leq 2\mathbf{u}|p_i|$, $|\tau_i| \leq 2\mathbf{u}|s_i|$ et $|\sigma_i| \leq (1+2\mathbf{u})|\tau_i|$ pour $i = 0, \ldots, n-1$. Ainsi

$$(\widetilde{p_\pi}+\widetilde{p_\sigma})(|x|) = \sum_{i=0}^{n-1}(|\pi_i|+|\sigma_i|)|x^i| \leq 2\mathbf{u}(1+2\mathbf{u})\sum_{i=1}^{n}(|p_{n-i}|+|s_{n-i}|)|x^{n-i}|,$$

et donc

$$\begin{aligned} (\widetilde{p_\pi}+\widetilde{p_\sigma})(|x|) &\leq 2\mathbf{u}(1+2\mathbf{u})\sum_{i=1}^{n}\left(2+\gamma_{2i-1}(2\mathbf{u})+\gamma_{2i}(2\mathbf{u})\right)\widetilde{p}(|x|) \\ &\leq 4n\mathbf{u}(1+2\mathbf{u})\left(1+\gamma_{2n}(2\mathbf{u})\right)\widetilde{p}(|x|). \end{aligned}$$

Comme $4n\mathbf{u}(1+\gamma_{2n}(2\mathbf{u})) = \gamma_{2n}(2\mathbf{u})$, nous pouvons déduire que $(\widetilde{p_\pi}+\widetilde{p_\sigma})(|x|) \leq \gamma_{2n+1}(2\mathbf{u})\widetilde{p}(|x|)$. □

Lemme II.5.4. *Soient $p(x) = \sum_{i=0}^{n} a_i x^i$ un polynôme tel que $a_i \in \mathbb{F}$, $0 \leq i \leq n$, $q(x) = \sum_{i=0}^{n} b_i x^i$ un polynôme tel que $b_i \in \mathbb{F}$, $0 \leq i \leq n$ et $x \in \mathbb{F}$. Alors le résultat de l'évaluation polynomiale flottante de $r(x) = p(x) + q(x)$ par l'algorithme suivant*

1: $r_n \leftarrow \mathrm{fl}_*(a_n + b_n)$
2: **pour** $i = n-1$ à 0 **faire**
3: $\quad r_i \leftarrow \mathrm{fl}_*(r_{i+1} \cdot x + (a_i + b_i))$
4: **fin pour**
5: $\mathtt{res} \leftarrow r_0$

vérifie

$$|\mathtt{res} - r(x)| \leq \gamma_{2n+1}(2\mathbf{u})\widetilde{r}(|x|).$$

Démonstration. Si l'on considère l'algorithme précédent, nous avons $r_n = \mathrm{fl}_*(a_n+b_n) = (a_n + b_n)\langle 1\rangle(2\mathbf{u})$ et pour $i = n-1$ à 0,

$$r_i = \mathrm{fl}_*(r_{i+1} \cdot x + (a_i + b_i)) = r_{i+1}x\langle 2\rangle(2\mathbf{u}) + (a_i + b_i)\langle 2\rangle(2\mathbf{u}).$$

Ainsi par récurrence nous pouvons montrer que

$$r_0 = (a_n + b_n)x^n\langle 2n+1\rangle(2\mathbf{u}) + \sum_{i=0}^{n-1}(a_i + b_i)x^i\langle 2(i+1)\rangle(2\mathbf{u}).$$

De plus, si $\theta_{2n+1}(2\mathbf{u}), \theta_{2n}(2\mathbf{u}), \ldots, \theta_1(2\mathbf{u})$ sont tels que $|\theta_i(2\mathbf{u})| \leq \gamma_i(2\mathbf{u})$, alors

$$r_0 = (a_n + b_n)x^n(1 + \theta_{2n+1})(2\mathbf{u}) + \sum_{i=0}^{n-1}(a_i + b_i)x^i(1 + \theta_{2(i+1)}(2\mathbf{u})).$$

Comme $r_0 = \mathrm{fl}_*(p(x) + q(x))$, nous avons donc

$$\left|\mathtt{res} - \sum_{i=0}^{n}(a_i + b_i)x^i\right| \leq \gamma_{2n+1}(2\mathbf{u})\sum_{i=0}^{n}|a_i + b_i||x^i| \leq \gamma_{2n+1}(2\mathbf{u})(\widetilde{p} + \widetilde{q})(|x|).$$

□

Théorème II.5.5. *Soit un polynôme p de degré n avec des coefficients flottants, et un nombre flottant x. En arrondi dirigé, l'erreur commise par l'algorithme de Horner compensé est telle que*

$$|\mathtt{CompHorner}(p, x) - p(x)| \leq 2\mathbf{u}|p(x)| + 2\gamma_{2n+1}(2\mathbf{u})^2\widetilde{p}(x).$$

Démonstration. D'après l'algorithme II.8, nous avons $p(x) = s_0 + e(x) + (p_\tau - p_\sigma)(x)$. Nous pouvons donc déduire que

$$\begin{aligned}
|\mathtt{res} - p(x)| &= |(1+\varepsilon)(s_0 + \mathrm{fl}_*(e(x))) - p(x)| \\
&= |(1+\varepsilon)(s_0 + \mathrm{fl}_*(e(x)) - p(x) + (p_\tau - p_\sigma)(x)) \\
&\quad + \varepsilon p(x) + (1+\varepsilon)(p_\sigma - p_\tau)(x)| \\
&= |(1+\varepsilon)(s_0 + e(x) + (p_\tau - p_\sigma)(x) - p(x)) + (1+\varepsilon)(\mathrm{fl}_*(e(x)) - e(x)) + \\
&\quad \varepsilon p(x) + (1+\varepsilon)(p_\tau - p_\sigma)(x)| \\
&\leq 2\mathbf{u}|p(x)| + (1+2\mathbf{u})|\,\mathrm{fl}_*(e(x)) - e(x)| + (1+2\mathbf{u})|(p_\tau - p_\sigma)(x)|.
\end{aligned}$$

Grâce au lemme II.5.4, nous avons

$$|\,\mathrm{fl}_*(e(x)) - e(x)| \leq \gamma_{2n-1}(2\mathbf{u})\widetilde{e}(|x|) \leq \gamma_{2n-1}(2\mathbf{u})(\widetilde{p_\pi}(|x|) + \widetilde{p_\sigma}(|x|)).$$

De plus, grâce au lemme II.5.3, nous avons

$$\widetilde{p_\pi}(|x|) + \widetilde{p_\sigma}(|x|) \leq \gamma_{2n+1}(2\mathbf{u})\widetilde{p}(|x|).$$

Comme $|\tau_i - \sigma_i| \leq 2\mathbf{u}\tau_i$, nous avons

$$|(p_\tau - p_\sigma)(x)| \leq 2\mathbf{u}\sum_{i=0}^{n-1} |\tau_i||x|^i \leq 2\mathbf{u}\widetilde{p_\tau}(|x|).$$

De plus, comme $|\tau_i| \leq 2\mathbf{u}|s_i|$, nous pouvons déduire que $\widetilde{p_\tau}(|x|) \leq 2n\mathbf{u}\gamma_{2n}(2\mathbf{u})\widetilde{p}(|x|)$. Par conséquent,

$$|\texttt{res} - p(x)| \leq 2\mathbf{u}|p(x)| + (1+2\mathbf{u})\gamma_{2n-1}(2\mathbf{u})\gamma_{2n+1}(2\mathbf{u})\widetilde{p}(|x|) + 2n\mathbf{u}(1+2\mathbf{u})\gamma_{2n}(2\mathbf{u})\widetilde{p}(|x|).$$

Comme $(1+2\mathbf{u})\gamma_{2n-1}(2\mathbf{u}) \leq \gamma_{2n}(2\mathbf{u})$ et $2n\mathbf{u} \leq \gamma_{2n+1}(2\mathbf{u})$, nous pouvons conclure que

$$|\texttt{res} - p(x)| \leq 2\mathbf{u}|p(x)| + 2\gamma_{2n+1}(2\mathbf{u})^2\widetilde{p}(|x|).$$

□

II.6 Algorithme de sommation compensée K fois

Comme nous l'avons rappelé dans la section II.3, les algorithmes II.2 (`FastCompSum`) et II.3 (`PriestCompSum`) calculent la somme de n nombres flottants en doublant la précision de travail même lorsque nous utilisons un arrondi dirigé. Nous allons maintenant présenter un algorithme permettant de multiplier la précision de travail K fois pour la somme : `SumK` [114]. Cette algorithme, tout comme le `DotK` présenté dans la section II.7, permet de compenser l'erreur sur le calcul de naïf mais aussi sur le calcul des erreurs.

Nous rappellerons les résultats en arrondi au plus près puis nous étudierons les effets d'un arrondi dirigé.

II.6.1 L'algorithme de sommation compensée K fois en arrondi au plus près

L'algorithme `SumK` a été introduit dans [114] et est rappelé ci-après (algorithme II.9). Il permet de multiplier par K la précision de travail pour une sommation. La version originelle de `SumK` utilise `TwoSum` [85]. Cependant, la même erreur est obtenue quel que soit l'algorithme de transformation exacte de sommation utilisé. Nous présentons cet algorithme en utilisant l'algorithme I.3 (`TwoSumPriest`). Celui-ci est coûteux mais reste une transformation exacte quel que soit le mode d'arrondi utilisé. Nous pouvons noter également que si $K = 2$, alors l'algorithme II.9 est identique à l'algorithme II.3 (`PriestCompSum`).

L'erreur sur `res` calculé par l'algorithme II.9 en arrondi au plus près est étudiée dans [114]. Une borne de l'erreur absolue est rappelée dans la proposition II.6.1 et de l'erreur relative dans le corollaire II.6.2.

Algorithme II.9 Sommation de n nombres flottants $p = \{p_i\}$ compensée K fois, $K \geq 3$

fonction `res` = `SumK`(p, K)

1: **pour** $k = 1$ à $K - 1$ **faire**
2: **pour** $i = 2$ à n **faire**
3: $[p_i, p_{i-1}] \leftarrow$ `TwoSumPriest`(p_i, p_{i-1})
4: **fin pour**
5: **fin pour**
6: `res` $\leftarrow \sum_{i=1}^{n} p_i$

Proposition II.6.1 ([114]). *Soient $p_i \in \mathbb{F}$, $1 \leq i \leq n$ des nombres flottants. Supposons que $4n\mathbf{u} \leq 1$. Alors, même en présence d'*underflow*, le résultat* `res` *de l'algorithme II.9 (*`SumK`*) en utilisant l'arrondi au plus près, satisfait pour $K \geq 3$*

$$|\texttt{res} - s| \leq \left(\mathbf{u} + 3\gamma_{n-1}^2(\mathbf{u})\right)|s| + \gamma_{2n-2}^K(\mathbf{u})S$$

avec $s = \sum p_i$ et $S = \sum |p_i|$.

Corollaire II.6.2 ([114]). *Supposons que $4n\mathbf{u} \leq 1$. Le résultat* `res` *de l'algorithme II.9 (*`SumK`*) en utilisant l'arrondi au plus près, même en présence d'*underflow*, vérifie*

$$\frac{|\texttt{res} - s|}{|s|} \leq \mathbf{u} + 3\gamma_{n-1}^2(\mathbf{u}) + \gamma_{2n-2}^K(\mathbf{u})\,\mathrm{cond}\left(\sum p_i\right).$$

Grâce au corollaire II.6.2, comme $\gamma_n(\mathbf{u}) \approx n\mathbf{u}$, la borne d'erreur relative sur le résultat est majoritairement l'erreur relative $\mathbf{u}$ plus un terme ($(\alpha\mathbf{u})^K$ fois le conditionnement pour un facteur modéré α). Ainsi le résultat est a priori K fois plus précis.

II.6.2 Sommation compensée K fois en arrondi dirigé

Nous allons maintenant étudier les effets d'un arrondi dirigé sur l'algorithme II.9 (`SumK`). Nous utiliserons pour cette étude le même raisonnement que celui utilisé dans [114] pour l'arrondi au plus près. Nous notons $p^{(0)}$ le vecteur p initialement et $p^{(k)}$ ce même vecteur après avoir effectué k itérations. Nous notons également $S^{(k)} = \sum_{i=1}^{n} |p_i^{(k)}|$ pour $0 \leq k \leq K - 1$.

Lemme II.6.3. *Avec les notations précédentes, les résultats intermédiaires de l'algorithme II.9 (*`SumK`*) vérifient :*

$$s = \sum_{i=1}^{n} p_i^{(0)} = \sum_{i=1}^{n} p_i^{(k)} \quad \text{pour} \quad 1 \leq k \leq K - 1, \tag{II.6.1}$$

$$|\texttt{res} - s| \leq 2\mathbf{u}|s| + \gamma_{n-1}^2(2\mathbf{u})S^{(K-2)}, \tag{II.6.2}$$

$$S^{(k)} \leq 3|s| + \gamma_{2n-2}^k(2\mathbf{u})S^{(0)} \quad \textit{à condition que} \quad 8(n-1)\mathbf{u} \leq 1 \quad \text{et} \quad 1 \leq k \leq K - 1. \tag{II.6.3}$$

Démonstration. L'équation II.6.1 provient des utilisations successives de l'équation II.3.21.

L'équation II.6.2 se déduit en utilisant $s = \sum_{i=1}^{n} p_i^{(K-2)}$ appliquant à la proposition II.3.10.

Nous devons maintenant prouver l'équation II.6.3. Grâce au lemme II.3.9, nous savons que

$$\sum_{i=1}^{n} |p_i^{(1)}| \leq |s| + 2\gamma_{n-1}(2\mathbf{u})S^{(0)}. \tag{II.6.4}$$

En appliquant successivement l'équation II.6.4 et en utilisant l'équation II.6.1, nous avons

$$S^{(2)} \leq |s| + 2\gamma_{n-1}(2\mathbf{u}) \left(|s| + 2\gamma_{n-1}(2\mathbf{u})S^{(0)} \right),$$

et

$$S^{(k)} \leq |s| + \sum_{i=0}^{\infty} (2\gamma_{n-1}(2\mathbf{u}))^i + (2\gamma_{n-1}(2\mathbf{u}))^k S^{(0)} \quad \text{pour } 1 \leq k \leq K-1.$$

Donc

$$\sum_{i=0}^{\infty} (2\gamma_{n-1}(2\mathbf{u}))^i = \frac{1 - 2(n-1)\mathbf{u}}{1 - 6(n-1)\mathbf{u}}.$$

Si $8(n-1)\mathbf{u} \leq 1$, alors nous obtenons $1 - 2(n-1)\mathbf{u} \leq 3(1 - 6(n-1)\mathbf{u})$ et

$$\frac{1 - 2(n-1)\mathbf{u}}{1 - 6(n-1)\mathbf{u}} \leq 3.$$

C'est pourquoi nous avons

$$S^{(k)} \leq 3|s| + (2\gamma_{n-1}(2\mathbf{u}))^k S^{(0)}.$$

Étant donné que $2\gamma_m(2\mathbf{u}) \leq \gamma_{2m}(2\mathbf{u})$, nous pouvons conclure que

$$S^{(k)} \leq 3|s| + (\gamma_{2n-2}(2\mathbf{u}))^k S^{(0)}.$$

□

Une borne de l'erreur absolue sur le résultat de l'algorithme II.9 (`SumK`) en arrondi dirigé est donnée dans la proposition II.6.4.

Proposition II.6.4. *Soient les nombres flottants $p_i \in \mathbb{F}$, $1 \leq i \leq n$. Supposons que $8n\mathbf{u} \leq 1$. Alors, même en présence d'*underflow*, le résultat* `res` *de l'algorithme II.9 (*`SumK`*) en arrondi dirigé vérifie pour $K \geq 3$*

$$|\texttt{res} - s| \leq \left(2\mathbf{u} + 3\gamma_{n-1}^2(2\mathbf{u}) \right) |s| + \gamma_{2n-2}^K(2\mathbf{u})S$$

avec $s = \sum p_i$ et $S = \sum |p_i|$.

Démonstration. La proposition II.6.4 découle de l'insertion de l'équation II.6.3 du lemme II.6.3 au sein de l'équation II.6.2. □

De la proposition II.6.4, nous déduisons dans le corollaire II.6.5 une borne de l'erreur relative sur le résultat de l'algorithme II.9 (`SumK`) obtenu en arrondi dirigé.

Corollaire II.6.5. *Supposons que* $8n\mathbf{u} \leq 1$. *Le résultat* `res` *de l'algorithme II.9 (*`SumK`*) obtenu en arrondi dirigé et même en présence d'*underflow*, vérifie*

$$\frac{|\texttt{res} - s|}{|s|} \leq 2\mathbf{u} + 3\gamma_{n-1}^2(2\mathbf{u}) + \gamma_{2n-2}^K(2\mathbf{u})\,\mathrm{cond}\left(\sum p_i\right). \tag{II.6.5}$$

D'après le corollaire II.6.5, étant donné que $\gamma_n(2\mathbf{u}) \approx 2n\mathbf{u}$, l'erreur relative sur le résultat en arrondi dirigé est majoritairement l'erreur d'arrondi relative $2\mathbf{u}$ plus $(\alpha\mathbf{u})^K$ fois le conditionnement pour un facteur modéré α. Ainsi tout comme en arrondi au plus près, le dernier terme de l'équation II.6.5 met en évidence le fait que le calcul est effectué comme si nous multiplions par K la précision de travail.

II.7 Produit scalaire compensé K fois

Comme nous l'avons vu dans la section II.4, l'algorithme II.5 (`CompDot`) calcule un produit scalaire en doublant la précision de travail et ce, même en arrondi dirigé. Nous allons maintenant étudier l'algorithme `DotK` [114]. Il permet de calculer un produit scalaire en multipliant par K la précision de travail. Nous supposerons ici qu'aucun *underflow* ne survient.

II.7.1 Produit scalaire compensé K fois en arrondi au plus près

L'algorithme `DotK` introduit dans [114] multiplie par K la précision de travail en utilisant les algorithmes `TwoSum` [85] et `TwoProd` [32]. Tout comme pour l'algorithme `SumK` (cf. section II.6) nous utiliserons ici des transformations exactes qui sont valides pour tous les modes d'arrondis : `TwoProdFMA` et `TwoSumPriest`. De la même façon, si $K = 2$, alors l'algorithme II.10 est similaire à l'algorithme II.5 (`CompDot`).

Algorithme II.10 Produit scalaire compensé K fois, $K \geq 3$

fonction `res` = `DotK`(x, y, K)

1: $[p, r_1] \leftarrow$ `TwoProdFMA`(x_1, y_1)
2: **pour** $i = 2$ à n **faire**
3: $\quad[h, r_i] \leftarrow$ `TwoProdFMA`(x_i, y_i)
4: $\quad[p, r_{n+i-1}] \leftarrow$ `TwoSumPriest`(p, h)
5: **fin pour**
6: $r_{2n} \leftarrow p$
7: `res` $\leftarrow$ `SumK`$(r, K-1)$

L'erreur sur le résultat `res` de l'algorithme II.10 en arrondi au plus près est analysée dans [114]. La borne d'erreur absolue est rappelée dans la proposition II.7.1 et la borne d'erreur relative dans le corollaire II.7.2.

Proposition II.7.1 ([114])**.** *Soient* $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$. *Supposons que* $8n\mathbf{u} \leq 1$. *Alors le résultat* `res` *de l'algorithme II.10 (*`DotK`*) en arrondi au plus près vérifie :*

$$|\texttt{res} - x^T y| \leq \left(\mathbf{u} + 2\gamma_{4n-2}^2(\mathbf{u})\right)|x^T y| + \gamma_{4n-2}^K(\mathbf{u})|x^T||y|.$$

Corollaire II.7.2 ([114]). *Supposons que* $8n\mathbf{u} \leq 1$. *Alors le résultat* `res` *de l'algorithme II.10 (*`DotK`*) en arrondi au plus près vérifie :*

$$\left|\frac{\texttt{res} - x^T y}{x^T y}\right| \leq \mathbf{u} + 2\gamma^2_{4n-2}(\mathbf{u}) + \frac{1}{2}\gamma^K_{4n-2}(\mathbf{u})\,\mathrm{cond}\left(x^T y\right).$$

Nous pouvons déduire du corollaire II.7.2 que l'erreur relative sur le résultat est majoritairement l'erreur d'arrondi $\mathbf{u}$ plus $\alpha(K)\mathbf{u}^K$ fois le conditionnement pour un facteur modéré $\alpha(K)$. Le calcul est donc a priori K fois plus précis.

II.7.2 Produit scalaire compensé K fois en arrondi dirigé

Nous allons maintenant analyser les effets d'un arrondi dirigé sur l'algorithme II.10 (`DotK`). Pour ce faire, nous utiliserons l'algorithme II.11 qui est une formulation équivalente de l'algorithme II.10. Cette nouvelle formulation permet néanmoins de faciliter l'écriture et la compréhension de la preuve.

Algorithme II.11 Algorithme équivalent à l'algorithme II.10

fonction `res` = `DotK`(x, y, K)

1: $[p_1, r_1] \leftarrow$ `TwoProdFMA`(x_1, y_1)
2: **pour** $i = 2$ à n **faire**
3: $\quad[h_i, r_i] \leftarrow$ `TwoProdFMA`(x_i, y_i)
4: $\quad[p_i, r_{n+i-1}] \leftarrow$ `TwoSumPriest`(p_{i-1}, h_i)
5: **fin pour**
6: $r_{2n} \leftarrow p_n$
7: `res` $\leftarrow$ `SumK`$(r, K-1)$

Une borne d'erreur absolue sur le résultat de l'algorithme `DotK` en arrondi dirigé est indiquée en proposition II.7.3.

Proposition II.7.3. *Soient* $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$. *Supposons que* $16n\mathbf{u} \leq 1$. *Alors le résultat* res *de l'algorithme II.10 en arrondi dirigé vérifie*

$$|\texttt{res} - x^T y| \leq \left(2\mathbf{u} + 2\gamma^2_{4n-2}(2\mathbf{u})\right)|x^T y| + \gamma^K_{4n-2}(2\mathbf{u})|x^T||y|.$$

Démonstration. `TwoProdFMA` et `TwoSumPriest` sont des transformations exactes en arrondi dirigé. Nous avons donc

$$s = \sum_{i=1}^{2n} r_i = x^T y. \tag{II.7.1}$$

En effet,

$$r_1 = x_1 y_1 - p_1,$$

et pour $i \geq 2$,

$$\begin{aligned} r_i + r_{n+i-1} &= (x_i y_i - h_i) + (p_{i-1} + h_i - p_i), \\ &= x_i y_i + p_{i-1} - p_i. \end{aligned}$$

Comme $r_{2n} = p_n$, nous avons

$$\begin{aligned}\sum_{i=1}^{2n-1} r_i &= (x_1y_1 - p_1) + \left(\sum_{i=2}^{n} x_iy_i + p_1 - r_{2n}\right), \\ &= x^Ty - r_{2n}. \end{aligned} \tag{II.7.2}$$

D'où l'équation II.7.1.

Pour appliquer la proposition II.6.4, nous devons maintenant estimer $S = \sum_{i=1}^{2n} |r_i|$. Étant donné que l'algorithme II.10 est exécuté en arrondi dirigé :

$$|r_1| \leq 2\mathbf{u}|x_1y_1| \tag{II.7.3}$$

et

$$\sum_{i=2}^{n} |r_i| \leq 2\mathbf{u} \sum_{i=2}^{n} |x_iy_i|. \tag{II.7.4}$$

En appliquant le lemme II.3.6 à l'algorithme `TwoSumPriest`, nous pouvons déduire que

$$\begin{aligned}\sum_{i=2}^{n} |r_{n+i-1}| &\leq \gamma_{n-1}(2\mathbf{u}) \left(|p_1| + \sum_{i=2}^{n} |h_i|\right), \\ &= \gamma_{n-1}(2\mathbf{u}) \sum_{i=1}^{n} |fl_*(x_iy_i)|, \\ &\leq (1+2\mathbf{u})\gamma_{n-1}(2\mathbf{u})|x^T||y|. \end{aligned} \tag{II.7.5}$$

Des équations II.7.3, II.7.4 et II.7.5, nous avons

$$\begin{aligned}\sum_{i=1}^{2n-1} |r_i| &\leq 2\mathbf{u}|x^T||y| + (1+2\mathbf{u})\gamma_{n-1}(2\mathbf{u})|x^T||y|, \\ &\leq \frac{2n\mathbf{u}}{1-2(n-1)\mathbf{u}}|x^T||y|. \end{aligned} \tag{II.7.6}$$

L'équation II.7.2 nous permet de montrer que

$$\begin{aligned}|r_{2n}| &= |x^Ty - \sum_{i=1}^{2n-1} r_i|, \\ &\leq |x^Ty| + \sum_{i=1}^{2n-1} |r_i|. \end{aligned}$$

C'est pourquoi

$$\sum_{i=1}^{2n} |r_i| \leq |x^Ty| + 2\sum_{i=1}^{2n-1} |r_i|. \tag{II.7.7}$$

Grâce à l'équation II.7.6

$$\begin{aligned}2\sum_{i=1}^{2n-1} |r_i| &\leq \frac{(2n)(2\mathbf{u})}{1-(n-1)(2\mathbf{u})}|x^T||y|, \\ &\leq \frac{(2n)(2\mathbf{u})}{1-2n(2\mathbf{u})}|x^T||y|, \\ &\leq \gamma_{2n}(2\mathbf{u})|x^T||y|. \end{aligned} \tag{II.7.8}$$

Les équations II.7.7 et II.7.8, nous permettent de dire que

$$\sum_{i=1}^{2n} |r_i| \leq |x^T y| + \gamma_{2n}(2\mathbf{u})|x^T||y|.$$

Grâce à la proposition II.6.4, avec $2\gamma_m(2\mathbf{u}) \leq \gamma_{2m}(2\mathbf{u})$, et comme r est de taille $2n$

$$\begin{aligned}
|\texttt{res} - x^T y| &\leq \left(2\mathbf{u} + 3\gamma_{2n-1}^2(2\mathbf{u})\right)|x^T y| + \gamma_{4n-2}^{K-1}(2\mathbf{u})\left(|x^T y| + \gamma_{2n}(2\mathbf{u})|x^T||y|\right), \\
&\leq \left(2\mathbf{u} + 3\gamma_{2n-1}^2(2\mathbf{u}) + \gamma_{4n-2}^{K-1}(2\mathbf{u})\right)|x^T y| + \gamma_{2n}(2\mathbf{u})\gamma_{4n-2}^{K-1}(2\mathbf{u})|x^T||y|, \\
&\leq \left(2\mathbf{u} + \frac{3}{4}\gamma_{4n-2}^2(2\mathbf{u}) + \gamma_{4n-2}^{K-1}(2\mathbf{u})\right)|x^T y| + \gamma_{4n-2}^{K}(2\mathbf{u})|x^T||y|.
\end{aligned}$$

Si $8(2n-1)\mathbf{u} \leq 1$, alors nous pouvons conclure que $\gamma_{4n-2}(2\mathbf{u}) \leq 1$ et

$$|\texttt{res} - x^T y| \leq \left(2\mathbf{u} + 2\gamma_{4n-2}^2(2\mathbf{u})\right)|x^T y| + \gamma_{4n-2}^{K}(2\mathbf{u})|x^T||y|.$$

□

Nous pouvons donc déduire une borne d'erreur relative sur le résultat de l'algorithme II.10 (`DotK`) en arrondi dirigé qui est présentée dans le corollaire II.7.4.

Corollaire II.7.4. *Supposons que* $16n\mathbf{u} \leq 1$. *Le résultat* `res` *de l'algorithme II.10 (*`DotK`*) obtenu en arrondi dirigé vérifie*

$$\left|\frac{\texttt{res} - x^T y}{x^T y}\right| \leq 2\mathbf{u} + 2\gamma_{4n-2}^2(2\mathbf{u}) + \frac{1}{2}\gamma_{4n-2}^{K}(2\mathbf{u})\,\mathrm{cond}(x^T y).$$

Ainsi nous pouvons déduire du corollaire II.7.4 que la borne d'erreur relative est majoritairement l'erreur d'arrondi relative $2\mathbf{u}$ plus $\alpha(K)\mathbf{u}^K$ fois le conditionnement pour un facteur modéré $\alpha(K)$. Le calcul est effectué comme avec une précision K fois supérieure à la précision de travail.

II.8 Implémentation

Nous avons vu que théoriquement la méthode CESTAC est compatible avec un certain nombre d'algorithmes compensés. Avec verrou, nous n'avons aucune manipulation à faire pour que cela fonctionne. Nous avons donc décidé de les intégrés dans CADNA pour étudier leurs performances avec une implémentation synchrone de la méthode CESTAC et qu'il y a un réel intérêt à cette implémentation.

L'utilisation de l'Arithmétique Stochastique Discrète fait que durant les branchements il est obligatoire de faire passer les trois exécutions dans la même branche d'un test [3]. Un des champs peut donc utiliser le mauvais bloc d'exécution, nous pouvons alors obtenir un cas où la compensation est mal effectuée et le nombre de chiffres significatifs du résultat ne sera pas amélioré.

Pour cela nous prendrons le cas de `FastTwoSum` et en particulier de son test `si |b| > |a|`. Lorsque nous souhaitons le désynchroniser, nous effectuons les tests

3. dans le `if` ou dans le `then`

indépendamment avec les champs x, y et z des variables stochastiques [4]. Nous présentons la version standard sans désynchronisation dans le code 2, la version désynchronisée est visible dans le code 3.

Code 2 Implémentation synchrone de l'algorithme compensé `FastTwoSum` avec CADNA

```
inline void fastTwoSum( double_st&a,  double_st&b, double_st&c, double_st&d){
  double_st a1;
  double_st b1;
  double_st z;

  if(fabs(b) > fabs(a)){
    b1 = a;
    a1 = b;
  } else {
    a1 = a;
    b1 = b;
  }

  c = a1 + b1;
  z = c - a1;
  d = b1 - z;
}
```

De plus nous avons noté durant nos tests que le nombre de chiffres significatifs exacts de la compensation est nul. En effet, la compensation nécessaire peut être négative pour une exécution et positive pour les autres. Il y aura alors une détection d'instabilité qui est à considérer comme un faux positif. Cela crée deux problèmes :

— faux positif à traiter lors de la recherche d'instabilités ;
— ralentissement en raison des tests effectués.

Nous avons donc décidé de supprimer la détection des instabilités sur la majeure partie des transformations exactes. Ainsi la détection des instabilités ne sera présente que sur l'opération principale [5]. Cela permet d'améliorer les performances du code et de limiter les faux positifs. Dans le code 3 nous avons donc décidé de détecter une éventuelle instabilité sur la ligne $s = a1 + b1$ mais pas pour l'opération $e = (s - a1) - b1$.

II.9 Résultats numériques

II.9.1 Précision estimée par CADNA

Nous allons maintenant étudier les effets de la bibliothèque CADNA sur les algorithmes compensés décrits précédemment. Dans les sections précédentes, nous avons analysé l'erreur générée par les algorithmes compensés en arrondi dirigé. Nous avons montré que l'amélioration de la précision en arrondi au plus près est également obtenue en arrondi dirigé. L'Arithmétique Stochastique Discrète permet d'estimer les erreurs d'arrondi grâce à l'utilisation de l'arrondi aléatoire (le résultat est arrondi vers plus ou moins l'infini avec la même probabilité). Les résultats expérimentaux présentés dans cette section permettent de confirmer une amélioration de la précision des résultats des algorithmes compensés avec l'Arithmétique Stochastique Discrète. Nous montrons

4. Le champ x de la variable a est alors obtenu grâce à a.x ; de même pour y (a.y) et z (a.z).
5. Addition dans le cas de `FastTwoSum` et `TwoSumPriest`, multiplication pour `TwoProdFMA`

Code 3 Implémentation désynchronisée de l'algorithme compensé `FastTwoSum` avec CADNA

```
inline void fastTwoSum( double_st&a,  double_st&b, double_st&c, double_st&d){
  double_st a1;
  double_st b1;
  double_st z;

/* Desynchronisation des champs */
  if(fabs(b.x) > fabs(a.x)){
    b1.x = a.x;
    a1.x = b.x;
  } else {
    a1.x = a.x;
    b1.x = b.x;
  }

  if(fabs(b.y) > fabs(a.y)){
    b1.y = a.y;
    a1.y = b.y;
  } else {
    a1.y = a.y;
    b1.y = b.y;
  }

  if(fabs(b.z) > fabs(a.z)){
    b1.z = a.z;
    a1.z = b.z;
  } else {
    a1.z = a.z;
    b1.z = b.z;
  }

  /* Synchronisation des champs */
  c = a1 + b1;
  z = c - a1;
  d = b1 - z;
}
```

également que l'Arithmétique Stochastique Discrète permet d'estimer le nombre de chiffres significatifs exacts d'un résultat provenant d'un algorithme compensé.

Nous effectuons tous nos calculs en *double* précision, c'est-à-dire en `binary64` de la norme IEEE 754 [73]. Les variables stochastiques utilisées comprennent donc trois nombres flottants `binary64`. Les figures II.2 à II.6 montrent le nombre de chiffres significatifs exacts estimé par CADNA pour les résultats des algorithmes classiques et des algorithmes compensés associés. Dans les figures II.2 à II.4 nous avons également reporté le nombre d de chiffres exacts obtenu à partir de la différence relative entre les résultats fournis par CADNA (notés R_{CADNA}) et les résultats exacts (notés R_{exact}).

$$\begin{aligned} \text{Si } R_{exact} \neq 0, \quad d = & \quad -\log_{10} \left| \frac{R_{CADNA} - R_{exact}}{R_{exact}} \right|, \\ \text{sinon} \quad d = & \quad -\log_{10} \left| R_{CADNA} \right|. \end{aligned}$$

Pour tester la précision des résultats dans le cas de la somme et du produit scalaire, nous générons de manière aléatoire un ensemble de nombres flottants. Nous utilisons pour cela l'algorithme utilisé dans [114]. Celui-ci nécessite le nombre n de flottants dans l'ensemble final et un conditionnement cible (celui-ci ne sera pas forcement atteint). La génération est effectuée et nous obtenons en sortie un ensemble de n valeurs, le conditionnement de l'opération ainsi que le résultat exact.

Tout d'abord, nous pouvons remarquer que le nombre d n'est pas nécessairement un entier, au contraire du nombre de chiffres significatifs exacts fourni par CADNA. Nous constatons dans les figures II.2 à II.4 que CADNA estime correctement le nombre de chiffres significatifs exacts des résultats tout en étant légèrement pessimiste sur celui-ci. CADNA peut également être optimiste comme dans le cas de `CompHorner` (figure II.4) pour un conditionnement de 10^{23}.

Nous pouvons observer que dans les figures II.2 à II.6, si le conditionnement augmente, alors le nombre de chiffres significatifs exacts diminue. De plus, avec un algorithme classique, les résultats n'ont plus aucun chiffre correct pour des conditionnements supérieurs à 10^{16}.

Nous pouvons constater sur les figures II.2 à II.4 que pour des conditionnements inférieurs à 10^{15}, les algorithmes compensés fournissent des résultats avec le nombre maximum de chiffres significatifs exacts (15 en double précision). Pour des conditionnements supérieurs à 10^{15}, la précision diminue et nous n'obtenons plus aucun chiffre significatif exact pour des conditionnements supérieurs à 10^{32}. Ces résultats fournis par CADNA sont en accord avec les propriétés des algorithmes compensés en arrondi dirigé données dans les sections II.3.3, II.4.3 et II.5.3. Ainsi, les algorithmes `FastCompSum`, `CompDot` et `CompHorner` calculent des résultats similaires à ceux que nous aurions obtenus en doublant la précision de travail.

Les figures II.5 et II.6 montrent la précision estimée par CADNA pour les algorithmes II.9 (`SumK`) et II.10 (`DotK`). Nous présentons sur ces graphiques les résultats de précision de `SumK` et `DotK` pour $K = 2$. Néanmoins, pour des raisons de performance, il serait plus judicieux dans ce cas d'utiliser `FastCompSum` et `CompDot`. Nous constatons que lorsque le conditionnement est inférieur à $10^{16(K-1)}$, alors les algorithmes `SumK` et `DotK` fournissent un résultat avec le nombre maximum de chiffres significatifs exacts. Cette précision diminue régulièrement lorsque le conditionnement passe de $10^{16(K-1)}$ à

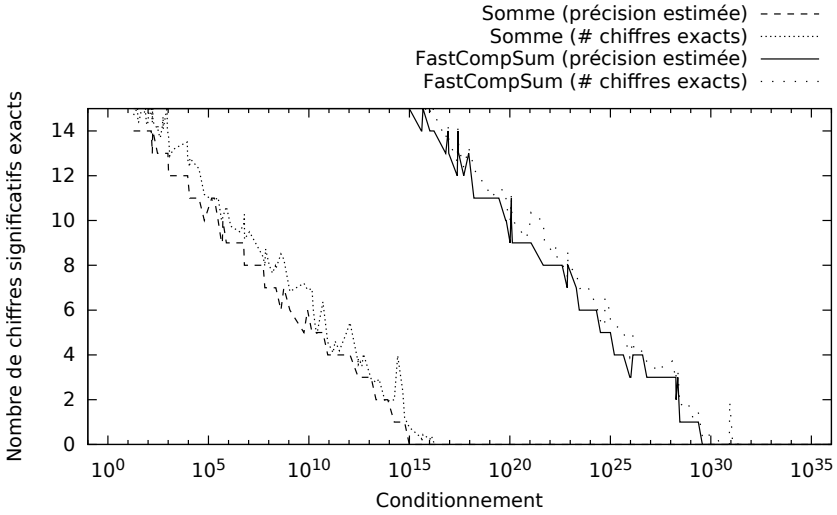


FIGURE II.2 – Précision estimée par CADNA et calculée à partir des résultats exacts pour les algorithmes `Sum` et `FastCompSum` sur la somme de 200 nombres flottants générés aléatoirement avec l'algorithme de [114].

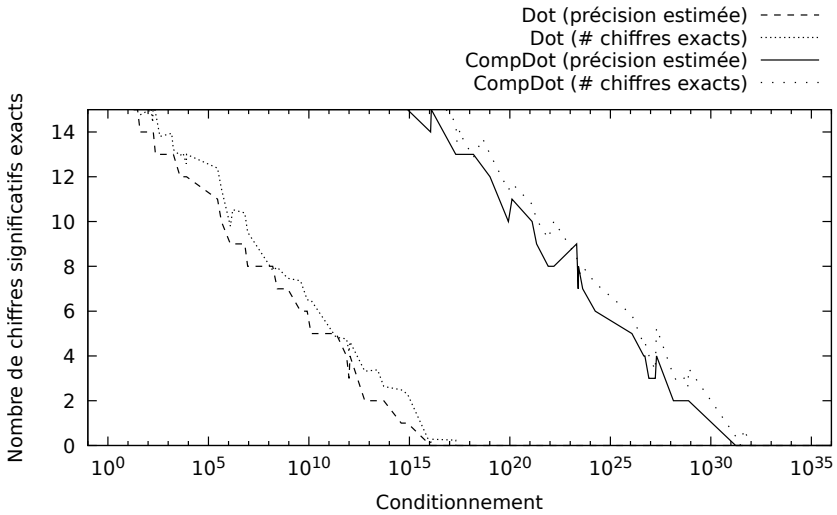


FIGURE II.3 – Précision estimée par CADNA et calculée à partir des résultats exacts pour les algorithmes `Dot` et `CompDot` sur le produit scalaire de deux vecteurs de 100 nombres flottants générés aléatoirement avec l'algorithme de [114].

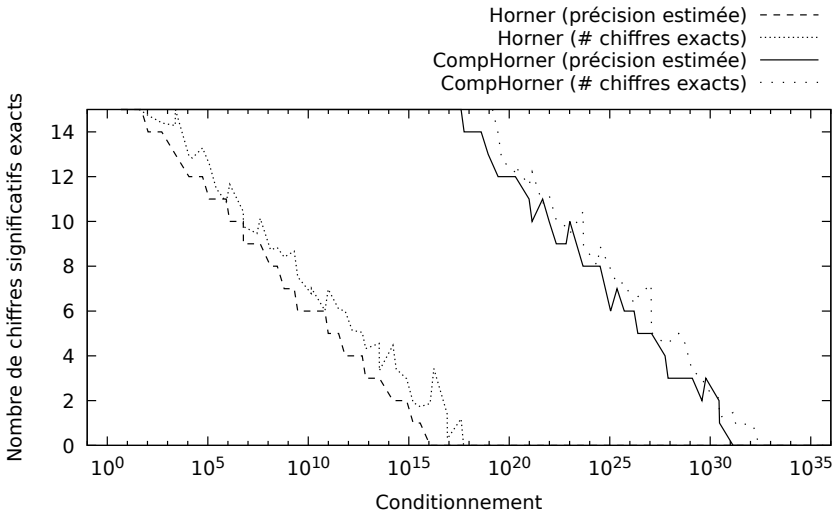


FIGURE II.4 – Précision estimée par CADNA et calculée à partir des résultats exacts avec les algorithmes `Horner` et `CompHorner` utilisés pour calculer le polynôme $(x-1)^n$ pour x proche de 1 et pour différentes valeurs de n.

10^{16K}. Lorsque le conditionnement est de 10^{16K}, nous n'obtenons plus un seul chiffre significatif exact. Nous retrouvons ainsi les propriétés présentées dans les sections II.6.2 et II.7.2 : en arrondi dirigé, les algorithmes `SumK` et `DotK` fournissent des résultats similaires à ceux que nous aurions obtenus en multipliant par K la précision de travail.

Nous avons également effectué d'autres expérimentations. Nous avons ainsi remplacé la transformation exacte `TwoSumPriest` par `FastTwoSum` dans l'algorithme `SumK`, voir figure II.7. Dans ce cas, et peu importe la valeur de K, l'algorithme ne permet pas une compensation K fois et fournit un résultat compensé une unique fois. Il est donc obligatoire d'utiliser la transformation exacte `TwoSumPriest` lorsqu'un arrondi dirigé est utilisé avec `SumK`. De la même façon, `DotK` utilise l'algorithme `SumK` et doit donc également utiliser cette transformation exacte de la somme.

II.9.2 Temps d'exécution

CADNA permet de détecter les instabilités numériques qui surviennent pendant l'exécution du code. L'utilisation de l'Arithmétique Stochastique Discrète nécessite l'auto-validation, c'est-à-dire le contrôle des multiplications et des divisions, comme nous l'avons rappelé dans la section I.4. Une multiplication de deux nombres sans aucun chiffre significatif exact ou une division par un nombre non significatif peut invalider l'estimation de la précision fournie par CADNA. Les algorithmes présentés ici ne nécessitent aucune division. La multiplication est utilisée pour le produit scalaire et l'évaluation polynomiale de Horner. Pour le produit scalaire, les opérandes sont les éléments des vecteurs initiaux. En ce qui concerne l'évaluation d'un polynôme $p(x)$, la valeur x est toujours une des opérandes des multiplications. Il n'y a donc pas de multiplication instable. En revanche, tous les algorithmes présentés peuvent générer des

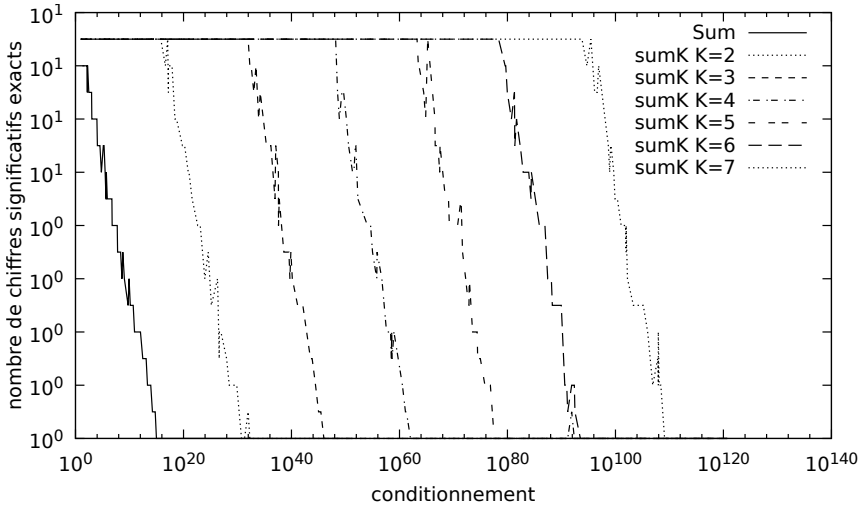


FIGURE II.5 – Précision estimée par CADNA pour les algorithmes `Sum` et `SumK` utilisant l'algorithme `TwoSumPriest` sur la somme de 200 nombres flottants générés aléatoirement avec l'algorithme de [114].

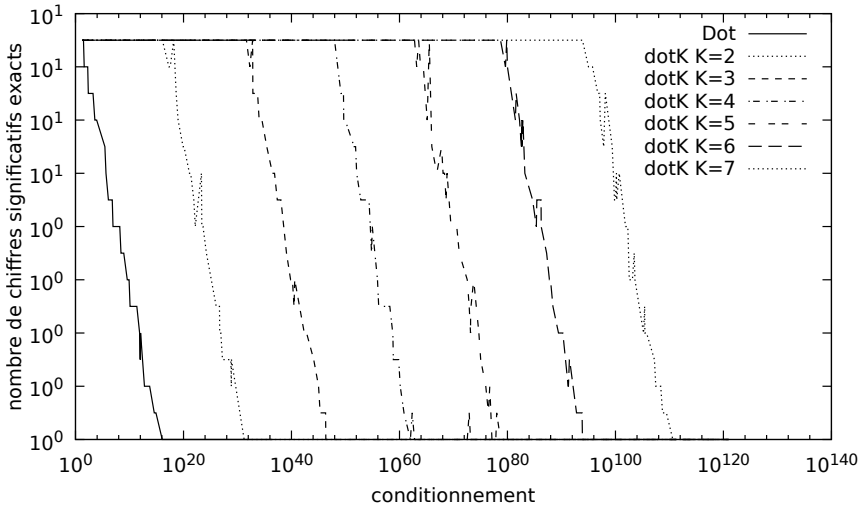


FIGURE II.6 – Précision estimée par CADNA pour les algorithmes `Dot` et `DotK` utilisant l'algorithme `TwoSumPriest` sur le produit scalaire de deux vecteurs de 100 nombres flottants générés aléatoirement avec l'algorithme de [114].

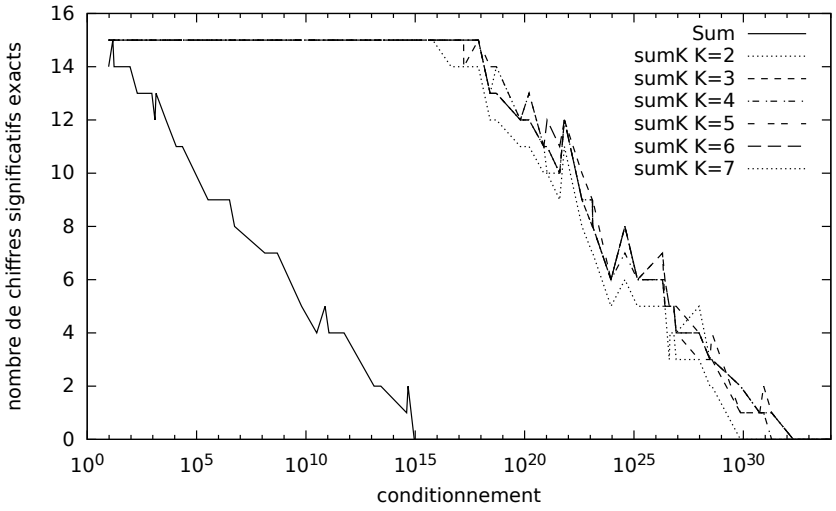


FIGURE II.7 – Précision estimée par CADNA pour les algorithmes `Sum` et `SumK` avec l'algorithme `FastTwoSum` sur la somme de 200 nombres flottants générés aléatoirement avec l'algorithme de [114].

éliminations catastrophiques (ou *cancellations*), c'est-à-dire des pertes de précision en raison de soustractions de deux valeurs proches entachées d'erreur d'arrondi. Le nombre de ces instabilités dépend du conditionnement, de la taille des vecteurs pour la somme et le produit scalaire et du degré du polynôme pour le schéma de Horner. Néanmoins, les cancellations ne peuvent pas invalider l'estimation du nombre de chiffres significatifs exacts fournie par l'Arithmétique Stochastique Discrète.

Les tableaux II.1, II.2 et II.3 présentent les temps d'exécutions obtenus en double précision avec et sans CADNA. La station de travail utilisée est équipée d'un processeur Intel Core i5-4690 CPU (Haswell) à 3,5 GHz et de g++ version 5.3.1. Le processeur dispose de l'instruction `FMA`. Les tests ont été effectués avec trois niveaux de détection d'instabilités :

— aucune détection d'instabilité ;
— auto-validation ;
— détection de toutes les instabilités.

Les algorithmes présentés ici ont des temps d'exécution très proches avec l'auto-validation et sans détection des instabilités. Pour les algorithmes de sommation, ces temps sont même nécessairement identiques. Ainsi, nous ne présentons ici que les temps mesurés avec l'auto-validation ou avec la détection de toutes les instabilités.

Nous pouvons observer grâce aux tableaux II.1, II.2 et II.3 que le coût d'utilisation des algorithmes compensés pour doubler la précision de travail (`FastCompSum`, `CompDot`, `CompHorner`) par rapport aux algorithmes classiques est environ 2 sans CADNA et environ 3 avec CADNA. Ce coût supplémentaire lorsque la bibliothèque CADNA est utilisée provient majoritairement de l'augmentation des mouvements mémoire en raison de l'utilisation des types stochastiques. Les temps d'exécution de `PriestCompSum` et `SumK` lorsque $K = 2$ sont mentionnés dans le tableau II.1. Néanmoins pour des

raisons de performance, `FastCompSum` qui est basé sur `FastTwoSum` est conseillé lorsque nous souhaitons doubler la précision du résultat. En effet, `PriestCompSum` utilise `TwoSumPriest` qui nécessite plus d'opérations flottantes et un branchement conditionnel de plus que `FastTwoSum`. Nous pouvons également remarquer que `SumK` pour $K = 2$ est plus coûteux que `PriestCompSum`. À la fin du calcul `SumK` additionne les erreurs stockées dans un tableau alors que `PriestCompSum` additionne ces mêmes erreurs dès qu'elles sont disponibles.

De la même manière, le tableau II.2 présente les temps d'exécution de `DotK` pour $K = 2$. Comme pour la sommation, il est dans ce cas préférable d'utiliser un autre algorithme pour doubler la précision du résultat : `CompDot`. En effet `DotK` va utiliser l'algorithme `TwoSumPriest` qui est plus coûteux que `FastTwoSum` utilisé par `CompDot`. De plus pour $K = 2$, `DotK` va additionner les erreurs stockées dans un tableau à la fin du calcul, tandis que `CompDot` additionne les erreurs dès que celles-ci sont disponibles.

Nous pouvons observer dans le tableau II.1 que le coût de `SumK` par rapport à la sommation classique augmente régulièrement avec K, et ce quel que soit le niveau de détection des instabilités avec CADNA et même sans utiliser CADNA. Le même résultat est obtenu pour `DotK` par rapport au produit scalaire classique dans le tableau II.2. Dans tous les cas, le coût de CADNA est compris entre 5 et 17 lorsque l'auto-validation est activée. Ce facteur est plus important lors de la détection de toutes les instabilités en raison du coût important de la détection des cancellations.

II.10 Conclusion

Nous avons montré qu'il est possible d'utiliser et de valider des programmes utilisant les algorithmes compensés avec l'Arithmétique Stochastique Discrète. Nous avons étudié la sommation, le produit scalaire et l'évaluation polynomiale. Pour cela, nous avons décrit les effets d'un mode d'arrondi aléatoire sur les transformations exactes de l'addition et de la multiplication. Nous avons également étudié les algorithmes compensés K fois pour la somme et le produit scalaire. Pour cela nous avons dû utiliser la sommation de Priest [120]. Les algorithmes compensés de sommation, de produit scalaire et d'évaluation polynomiale ont ainsi été implémentés dans la bibliothèque CADNA.

Certains algorithmes compensés n'ont pas encore été étudiés avec l'Arithmétique Stochastique Discrète. Nous pouvons citer la compensation du produit de plusieurs nombres flottants et de l'exponentiation entière [51], la compensation de l'évaluation d'une fonction élémentaire symétrique [81] ou la compensation de la méthode de Newton [50, 82]. Ces algorithmes utilisent les transformations exactes déjà décrites. Une autre perspective serait d'étudier avec l'Arithmétique Stochastique Discrète l'algorithme qui permet de compenser au mieux la division (aucune transformation exacte n'existe pour cet opérateur). Cela permettrait d'étudier ensuite des algorithmes tel que la résolution compensée des systèmes triangulaires [91].

algorithme	exécution	temps d'exécution (s)	ratio
`Sum`	sans CADNA	0,0845	1
	CADNA, auto-validation	0,549	6,5
	CADNA, toutes les instabilités	1,62	19,1
`FastCompSum`	sans CADNA	0,161	1
	CADNA, auto-validation	1,76	10,9
	CADNA, toutes les instabilités	4,54	28,1
`PriestCompSum`	sans CADNA	0,379	1
	CADNA, auto-validation	3,65	9,6
	CADNA, toutes les instabilités	5,87	15,5
`SumK`, $K = 2$	sans CADNA	0,761	1
	CADNA, auto-validation	5,12	6,7
	CADNA, toutes les instabilités	7,54	9,9
`SumK`, $K = 3$	sans CADNA	1,13	1
	CADNA, auto-validation	8,44	7,5
	CADNA, toutes les instabilités	11,2	9,9
`SumK`, $K = 4$	sans CADNA	1,51	1
	CADNA, auto-validation	11,9	7,9
	CADNA, toutes les instabilités	14,9	9,9
`SumK`, $K = 5$	sans CADNA	1,87	1
	CADNA, auto-validation	15,2	8,1
	CADNA, toutes les instabilités	18,6	9,8
`SumK`, $K = 6$	sans CADNA	2,27	1
	CADNA, auto-validation	18,6	8,2
	CADNA, toutes les instabilités	22,4	9,8
`SumK`, $K = 7$	sans CADNA	2,64	1
	CADNA, auto-validation	22,0	8,3
	CADNA, toutes les instabilités	26,1	9,9

Table II.1 – Temps d'exécution avec et sans CADNA d'une somme de 10^8 éléments.

algorithme	exécution	temps d'exécution (s)	ratio
`Dot`	sans CADNA	0,0252	1
	CADNA, auto-validation	0,214	8,5
	CADNA, toutes les instabilités	0,540	21,4
`CompDot`	sans CADNA	0,0563	1
	CADNA, auto-validation	0,766	13,6
	CADNA, toutes les instabilités	1,68	29,9
`DotK`, $K = 2$	sans CADNA	0,286	1
	CADNA, auto-validation	1,46	5,1
	CADNA, toutes les instabilités	2,30	8,0
`DotK`, $K = 3$	sans CADNA	0,468	1
	CADNA, auto-validation	3,31	7,1
	CADNA, toutes les instabilités	4,78	10,2
`DotK`, $K = 4$	sans CADNA	0,653	1
	CADNA, auto-validation	4,94	7,6
	CADNA, toutes les instabilités	6,84	10,5
`DotK`, $K = 5$	sans CADNA	0,836	1
	CADNA, auto-validation	6,57	7,8
	CADNA, toutes les instabilités	8,94	10,7
`DotK`, $K = 6$	sans CADNA	1,02	1
	CADNA, auto-validation	8,19	8,0
	CADNA, toutes les instabilités	10,7	10,5
`DotK`, $K = 7$	sans CADNA	1,21	1
	CADNA, auto-validation	9,83	8,2
	CADNA, toutes les instabilités	12,6	10,5

Table II.2 – Temps d'exécution avec et sans CADNA d'un produit scalaire de deux vecteurs de taille $2,5\,10^7$.

algorithme	exécution	temps d'exécution (s)	ratio
`Horner`	sans CADNA	0,042	1
	CADNA, auto-validation	0,451	10,6
	CADNA, toutes les instabilités	1,38	32,4
`CompHorner`	sans CADNA	0,0964	1
	CADNA, auto-validation	1,61	16,7
	CADNA, toutes les instabilités	3,71	38,7

Table II.3 – Temps d'exécution avec et sans CADNA pour l'évaluation de polynômes de degré $5\,10^7$.

Chapitre III

PROMISE

III.1 Introduction

Comme nous l'avons vu précédemment, la norme IEEE 754 [73] définit plusieurs types binaires. Nous rappelons que les types les plus couramment utilisés [1] sont :

— `binary32` : *simple* précision ;
— `binary64` : *double* précision ;
— `binary128` : *quadruple* précision.

Actuellement, la *quadruple* précision est peu utilisée car elle est émulée et non disponible au niveau matériel à l'exception des processeurs `SPARC` V8 [131] et V9 [132] ainsi que des processeurs `POWER9` [115]. Nous ne prendrons en compte dans ce chapitre que les types `binary32` et `binary64`.

Plus un type est précis, plus il prend de place en mémoire. Cela peut amener à des surcoûts dans les transferts de données, dans l'utilisation du cache et ainsi créer un ralentissement. Plutôt que de n'utiliser que la *double* précision comme cela se fait majoritairement, il serait intéressant de pouvoir utiliser au sein des codes de la *simple* précision en vue d'améliorer les performances.

L'utilisation unique de la *double* précision est rapide à mettre en œuvre et ne nécessite pas de réflexion sur la précision des résultats intermédiaires dans le code. Nous pouvons également noter que la recherche d'un sous-ensemble de variables pouvant être défini en *simple* précision est difficile manuellement. Le nombre de sous-ensembles pouvant être testés est exponentiel par rapport au nombre de variables.

Ainsi, la précision mixte présente des avantages mais est difficile à mettre en pratique. Nous souhaitons pouvoir obtenir automatiquement un sous-ensemble de variables pouvant être défini en *simple* précision tout en respectant une précision cible sur le résultat.

Nous rappellerons tout d'abord l'intérêt de la précision mixte, puis nous présenterons un état de l'art sur les outils permettant d'effectuer des optimisations de types. Nous décrirons ensuite l'algorithme de l'outil que nous avons développé : PROMISE [2] dans la section III.4. Nous présenterons les deux versions développées de cet outil (section III.5).

1. Il existe aussi les types `binary16` (*half* précision) et `binary256` (*octuple* précision) ainsi que la définition des types décimaux.
2. PRecision OptiMISEd

Enfin, nous présenterons les évaluations expérimentales sur des cas tests mais également sur un code industriel utilisé pour résoudre l'équation de transport des neutrons.

III.2 De l'intérêt de la précision mixte

De nombreux travaux ont montré l'intérêt de la précision mixte par rapport à la précision *double*. En effet, la précision des algorithmes en calcul flottant peut impacter le temps de calcul, le transfert mémoire et la consommation énergétique [3]. Ainsi, les processeurs permettent grâce à la vectorisation d'effectuer deux fois plus d'opérations en même temps en *simple* précision qu'en *double* précision. Les variables en *simple* précision nécessitent moins de mémoire. En cas de transfert, il y a une utilisation plus faible de la bande passante que ce soit entre la mémoire et les caches ou encore entre la mémoire et une carte graphique.

Les travaux d'optimisation des types numériques portent principalement sur les algorithmes d'algèbre linéaire comme les BLAS [90] [3] et plus particulièrement sur le raffinement itératif que ce soit pour des solveurs directs ou itératifs [15]. Dans ces exemples, le but est d'utiliser la *simple* précision pour améliorer la vitesse de calcul et la *double* précision pour obtenir un résultat plus précis.

L'utilisation de cartes graphiques (GPU) permet d'obtenir de bien meilleures performances en *simple* précision. Par exemple la carte Nvidia Tesla p100 a une puissance de 5,3 TeraFLOPS en *double* précision et 10,6 TeraFLOPS en *simple* précision [113]. Anzt et al. [2] montrent qu'un facteur d'accélération de 1,5 est possible sur CPU sur leur plus gros cas test en utilisant la précision mixte plutôt que la *double* précision. Un facteur d'accélération de 6 est observé entre les version GPU mixte et CPU *double*. La précision mixte permet d'obtenir des accélérations sur CPU, qui peuvent être augmentées grâce à l'utilisation d'une carte graphique. En outre, la puissance électrique nécessaire pour obtenir le résultat est inférieure lorsque la précision mixte est utilisée.

La précision mixte présente donc de nombreux avantages et est un sujet qui reste encore à être exploré. Néanmoins, nous pouvons constater que la plupart des codes utilisant de la précision mixte subissent une optimisation manuelle difficile à répliquer facilement sur des codes différents. Il est intéressant de pouvoir automatiser l'optimisation.

III.3 Obtenir un programme en précision mixte

Certains outils permettent d'obtenir de façon automatique un programme en précision mixte. Des approches différentes existent que ce soit sur les objectifs ou sur la manière de tester les configurations. Nous présentons ici un certain nombre de ces outils.

FPTuner est un outil permettant d'obtenir l'ensemble des variables pouvant être définies en *double* précision et les variables devant être définies en *quadruple* précision [23]. Sur un programme donné, l'utilisateur définit les intervalles d'entrées des valeurs des variables et le seuil d'erreur maximum. L'outil va créer la liste des opérations ainsi que les variables concernées par celle-ci. Il est ainsi possible de calculer une approximation de

3. Basic Linear Algebra Subprograms

l'erreur commise grâce aux expansions de Taylor, via l'outil FPTaylor [130], sur chacune des variables durant l'exécution du programme. La recherche d'un maximum global de variables en *double* précision est effectuée grâce au solveur d'optimisation mathématique Gurobi [62]. Celui-ci résout un problème d'optimisation d'une fonctionnelle quadratique sous contrainte quadratique [4]. L'avantage de cette approche est d'avoir des résultats valides pour des intervalles d'entrées définis par l'utilisateur grâce aux expansions de Taylor symboliques [5]. Néanmoins, l'outil n'est pas capable de gérer de "larges" [6] Les cibles privilégiées sont donc les bibliothèques de calcul scientifique, telles que les bibliothèques de fonctions mathématiques, les BLAS, ..., optimisées avec des résultats validés. Cet objectif est différent du nôtre, nous cherchons en effet à optimiser les types des variables dans un code de simulation numérique industriel.

CRAFT HPC [86] prend en entrée un programme à optimiser sans contrainte particulière sur celui-ci. CRAFT HPC va essayer d'optimiser les types des variables en dégradant leur précision. Le résultat est un programme avec une configuration de variables optimisée. Pour cela, des modifications du binaire sont effectuées en utilisant les bibliothèques Dyninst [14] et XED de la bibliothèque Intel Pin [92]. Les modifications binaires vont concerner les opérateurs arithmétiques. Les opérandes en *double* précision sont modifiées. Cela consiste à simuler une variable en *simple* précision tout en gardant en mémoire la taille d'une variable en *double* précision. Un remplissage mémoire connu est effectué dans le but de garder l'alignement mémoire. Un *double* sur 64 bits sera transformé en *simple* sur 32 bits et les 32 bits restants sont fixés à la valeur hexadécimale 0x7FF4DEAD. Ainsi le *double* est considéré comme un NaN grâce aux quatre premiers éléments hexadécimaux (0x7FF4). Les quatre suivants permettent d'identifier facilement les variables castées lorsqu'un humain lit les résultats hexadécimaux. Les exceptions sont toujours produites ce qui permet d'avoir un contrôle sur les erreurs. Pour obtenir le programme résultat, CRAFT HPC utilise un algorithme de parcours en largeur [7] de l'arbre composé de l'ensemble des variables et de sous-ensembles de celui-ci. Le résultat final est l'ensemble des variables transformées en *simple* précision qui ont permis d'obtenir des résultats respectant la précision demandée. Comme le résultat est une union de variable en *simple* précision, il est possible que la configuration finale ne respecte pas la précision souhaitée par l'utilisateur. Ce dernier pourra néanmoins utiliser les informations fournies par l'outil pour modifier son programme à l'aide d'un éditeur dédié. Néanmoins, aucun facteur d'accélération par rapport à l'exécution en *double* précision ou par rapport à la version mixte n'est présenté. L'objectif de cet outil n'est pas d'obtenir une accélération mais, des informations sur les modifications possibles en vue d'une optimisation.

Martel propose une approche fondée sur une analyse statique du code [95]. L'utilisateur indique pour les variables du programme le nombre de bits significatifs souhaité. À partir de ces valeurs, un ensemble de contraintes est défini grâce à une analyse directe et à une analyse inverse. Il est alors possible d'utiliser un solver SMT [8] tel que

4. *quadratically constrained quadratic program (QCQP)* : la fonction objectif et les contraintes d'optimisation sont des fonctions quadratiques

5. *Symbolic Taylor Expansions* en anglais

6. La majorité des cas tests comportent moins de 30 opérations et le plus grand en comporte 1 023. Il s'agit alors de la somme des éléments d'un vecteur calculée grâce à une réduction.

7. *Breadth-first algorithm* en anglais.

8. solveur Satisfiabilité Modulo des Théories : Le but est de résoudre un problème ayant des

Z3 [31] pour trouver les valeurs minimales en nombre de bits respectant l'ensemble des contraintes. Ces travaux sont récents mais un prototype d'outil existe et a permis de réduire la précision des variables intermédiaires et des résultats sur des exemples tel que le déterminant d'une matrice 3×3, d'une évaluation polynomiale de Horner et d'un contrôleur proportionnel dérivé. L'intérêt de cet outil est de réduire l'utilisation mémoire. Ainsi sur les programmes tests et en supposant que les résultats devaient avoir au minimum 23 bits de précision, alors il est possible de réduire l'occupation mémoire de 83% au maximum par rapport à de l'utilisation de la *double* précision. Tout comme pour CRAFT HPC, aucun facteur d'accélération des codes n'est affiché.

Precimonious [124] est un outil dont l'objectif est d'obtenir un programme plus rapide tout en respectant la précision cible. L'outil va tester le résultat du programme mais également le temps d'exécution de celui-ci. Ainsi le facteur d'accélération fait partie intégrante du test effectué. L'algorithme travaille sur l'ensemble des variables à tester. Pour cela il utilise un algorithme "diviser pour régner". Il s'agit de l'algorithme de delta debug [141, 142]. Celui-ci cherche à maximiser le nombre de variables en *simple* précision. Lorsqu'une configuration respecte la précision cible et permet d'obtenir un facteur d'accélération, alors l'ensemble des variables modifiées en *simple* précision le restera. L'outil va ensuite essayer d'ajouter de nouveaux éléments à l'ensemble des variables en *simple*. La représentation intermédiaire de LLVM [88] est utilisée pour effectuer les modifications au sein du programme. Le résultat est donc un ensemble de variables en *simple* permettant de respecter le critère de précision sur les résultats tout en ayant le facteur d'accélération le plus important par rapport à la *double* précision.

Blame Analysis [109] détermine le plus rapidement possible les variables ne pouvant pas avoir leur précision dégradée. Les opérations du programme testé sont exécutées plusieurs fois avec différentes combinaisons de type pour les opérandes. Les différentes opérations sont effectuées durant la même exécution du programme en utilisant LLVM [88] pour tester toutes les combinaisons des différentes précisions. Par exemple, l'instruction d'addition sera effectuée plusieurs fois avec des opérandes de différentes précisions : *simple*, *double* ou encore *double* tronquée à 8 chiffres décimaux. Toutes les combinaisons possibles entre les différents types à tester sont effectuées. Les résultats sont ensuite comparés au résultat de l'opération avec la précision la plus haute possible. À la fin de l'exécution, si une variable a passé tous ces tests de précision en étant calculée en *simple* précision, alors elle sera considérée comme pouvant être utilisée en *simple*. Il n'y a aucune vérification du temps de calcul. Néanmoins, les auteurs mettent en avant le fait que les informations fournies par cet outil permettent de réduire l'espace de recherche de Precimonious. En effet, le résultat va exclure certains tests de variables en *simple* précision. Cela permet d'améliorer le temps d'exécution de Precimonious. Il serait possible d'utiliser cet outil avec d'autres programmes d'optimisation de types.

Ces outils utilisent une comparaison des résultats au résultat obtenu à la précision la plus élevée. Dans un certain nombre de cas, la référence en précision supérieure peut être fausse. Nous pouvons ici citer le polynôme de Rump [125] dont les résultats en *simple* et *double* précision sont les mêmes mais ceux-ci sont faux, comme vu dans la section I.1.4. Nous avons donc développé un outil appelé PROMISE qui fournit une configuration mixte en validant les résultats grâce à l'Arithmétique Stochastique

contraintes logiques du premier ordre.

Discrète [138] et qui utilise l'algorithme de delta debug [141, 142] pour déterminer les ensembles à tester. Le résultat de PROMISE est un programme modifié avec un maximum de variables en *simple* précision et qui respecte la précision souhaitée par l'utilisateur.

III.4 L'algorithme de PROMISE

III.4.1 Recherche d'une configuration

À partir d'un programme de calcul que nous souhaitons optimiser, PROMISE cherche à obtenir un programme modifié avec un maximum de variables déclarées en précision la plus basse possible tout en respectant une contrainte de précision. Nous pouvons utiliser le nombre de chiffres significatifs exacts fourni par CADNA comme mesure de la précision. Dans le but de simplifier la présentation de l'algorithme, nous nous plaçons dans le cas d'un programme n'ayant initialement que des variables déclarées en *double* précision (`binary64`) [73]. PROMISE modifiera le type de ces variables en *simple* précision (`binary32`). Si le programme dispose de variables en *simple* et *double* précisions il est possible de toutes les transformer en *double* puis d'exécuter l'algorithme d'optimisation de type ou de ne pas tenir compte des variables déjà en *simple* précision.

Pour décrire plus facilement l'algorithme utilisé, nous allons donner des notations et des définitions. Nous noterons C l'ensemble des variables et C^s (resp. C^d) l'ensemble des variables en *simple* (resp. *double*) précision. Nous avons donc $C^d \cap C^s = \emptyset$ et $C^d \cup C^s = C$. Les définitions présentées proviennent des travaux de Zeller sur la recherche automatique d'erreur dans un programme [141, 142]. Elles ont été adaptées pour notre cas : la modification automatique du type des variables.

Définition III.4.1. Une *configuration* est un couple d'ensembles (C^s, C^d) et l'ensemble des configurations possibles est noté $\mathcal{R}$.

Définition III.4.2. Un résultat est *valide* si sa précision vérifie la précision demandée.

Définition III.4.3. La fonction $\texttt{test} : \mathcal{R} \to \{\checkmark, \times\}$ détermine pour une configuration si le résultat est *valide* ($\checkmark$) ou non ($\times$).

Définition III.4.4. Une configuration (C^s, C^d) est *m-maximale* si $\texttt{test}(C^s, C^d) = \checkmark$ et quel que soit $(C^{s\prime}, C^{d\prime})$ tel que $C^s \subset C^{s\prime}$, $C^{d\prime} \subset C^d$, et $C^{s\prime} \cup C^{d\prime} = C^s \cup C^d$

$$\texttt{test}(C^{s\prime}, C^{d\prime}) = \times \text{ and } |C^{s\prime}| - |C^s| \leq m.$$

Définition III.4.5. Une configuration (C^s, C^d) est *pertinente* si elle est 1-maximale selon la définition III.4.4. Donc

$$\forall \delta \in C^d, \texttt{test}(C^s \cup \{\delta\}, C^d \backslash \{\delta\}) = \times.$$

Dans le but de trouver une configuration pertinente, PROMISE utilise une version modifiée de l'algorithme de delta debug [141, 142] basé sur une méthode *diviser pour régner*. Si nous considérons un programme de n variables, rechercher une configuration maximale globale nécessite de tester 2^n configurations possibles. En comparaison,

PROMISE a la même complexité que l'algorithme de delta debug : $O(n^2)$ dans le pire cas et $O(n\log(n))$ en moyenne [141]. L'algorithme de delta debug sert à identifier les erreurs dans les programmes, il a donc été adapté pour permettre d'optimiser la précision des variables. Les modifications effectuées sont des adaptations des tests effectués pour répondre à notre besoin : la recherche d'un maximum local *valide* à la place d'une recherche d'une configuration minimale d'échec.

Deux versions équivalentes de l'algorithme PROMISE sont décrites : tout d'abord la version récursive, puis la version itérative qui est celle qui a été implémentée.

III.4.2 Algorithme récursif

L'algorithme III.1 est la version récursive de l'algorithme utilisé par l'outil PROMISE. Comme nous considérons que toutes les variables sont initialement en *double* précision, l'initialisation se fait par $PROMISE(\emptyset, C, 2)$. L'ensemble C^d des variables en *double* est partitionné en p sous-ensemble C_i^d tels que $\cup_{i=1}^{p} C_i^d = C^d$ avec $|C_i^d| \approx \frac{|C^d|}{p}$, nous créons donc des p partitions. Les variables sont choisies au fur et à mesure pour créer tous les sous-ensembles. Dans la suite, p sera appelé la granularité. PROMISE teste ainsi chacun de ces sous-ensembles grâce à l'algorithme III.1. À chaque récursion, trois branchements sont possibles :

1. S'il existe un sous-ensemble C_i^d avec $\texttt{test}(C^s \cup C_i^d, C^d \backslash C_i^d) = \checkmark$, alors les éléments de C_i^d doivent être définitivement ajoutés à l'ensemble C^s. De plus, la granularité devient $p-1$ avec un minimum de 2. En effet, une granularité de 1 créerait un test redondant. L'exemple le plus simple est le suivant : soit trois sous-ensembles p_1, p_2 et X. La granularité est de 2. Si $\texttt{test}(p_1 \cup p_2, X) = \text{✗}$ et $\texttt{test}(p_1, X \cup p_2) = \checkmark$ alors le test suivant serait de nouveau $\texttt{test}(p_1 \cup p_2, X)$.
2. Si tous les tests donnent un résultat *non précis* et que la granularité p est inférieure à la taille de C^d, alors la granularité est augmentée pour le même couple (C^s, C^d). Le nombre de sous-ensembles ne peut pas dépasser le nombre d'éléments dans C^d, la granularité devient $\min(2p, |C^d|)$.
3. Enfin, si tous les tests retournent un résultat *non précis* et que la granularité est égale à la taille de C^d, alors toutes les possibilités ont déjà été testées. Il est inutile de retester une configuration et l'algorithme retourne C^s l'ensemble des variables pouvant être définies en *simple* précision.

Algorithme III.1 PROMISE : version récursive de l'algorithme

Soit C_i^d une p partition de C^d telle que $\cup_{i=1}^{p} C_i^d = C^d$ avec $|C_i^d| \approx \frac{|C^d|}{p}$
$PROMISE(C^s, C^d, p)=$

$$\begin{cases} \quad \textbf{1} - PROMISE(C^s \cup C_i^d, C^d \backslash C_i^d, \max(p-1, 2)) \\ \text{si } \exists i \in \{1, \cdots, p\} \text{ tel que } \texttt{test}(C^s \cup C_i^d,\ C^d \backslash C_i^d) = \checkmark\,; \\ \quad \textbf{2} - PROMISE(C^s, C^d, \min(2p, |C^d|)) \ \text{ si } p < |C^d|\,; \\ \quad \textbf{3} - \text{retourne } C^s \qquad\qquad\qquad\qquad\qquad \text{Sinon.} \end{cases}$$

steps :	variables v_0	v_1	v_2	v_3	test	PROMISE(C^s, C^d, p)
1					`test`$(\{v_0, v_1\}, \{v_2, v_3\}) =$ ✗	PROMISE($\emptyset$, $\{v_0, v_1, v_2, v_3\}$, 2)
2					`test`$(\{v_2, v_3\}, \{v_0, v_1\}) =$ ✗	
3					`test`$(\{v_0\}, \{v_1, v_2, v_3\}) =$ ✓	PROMISE($\emptyset$, $\{v_0, v_1, v_2, v_3\}$, 4)
4					déjà testée (étape 1)	PROMISE($\{v_0\}$, $\{v_1, v_2, v_3\}$, 3)

FIGURE III.1 – Exemple montrant l'intérêt du cache pour éviter un test redondant. Programme à quatre variables $(v_0, \ldots, v_3)$ avec les variables en *simple* (resp. *double*) précision en gris (resp. blanc). L'étape 4 montre un test évité car déjà effectué.

La fonction `test` introduite par la définition III.4.3 est présentée dans l'algorithme III.2.

Algorithme III.2 Fonction `test` : détermine si une configuration est *valide*

fonction *etat* = `test`(C^s, C^d)

1: **si** (C^s, C^d) a déjà été testé **alors**
2: Retourne le résultat en cache
3: **fin si**
4: Mise à jour du code en fonction de (C^s, C^d)
5: **si** La compilation échoue **alors**
6: Retourne ✗
7: **fin si**
8: Exécution du programme modifié
9: **si** L'exécution échoue **alors**
10: Retourne ✗
11: **sinon si** Le résultat est *valide* **alors**
12: Retourne ✓
13: **fin si**
14: Retourne ✗

Les lignes 1-3 sont présentes pour éviter de tester la même configuration plusieurs fois. Cela peut survenir lorsqu'une configuration (C^d, C^d) échoue mais qu'une configuration $(C^{s\prime}, C^{d\prime})$ avec $C^{s\prime} \subset C^s$ et $C^{s\prime} \cup C^{d\prime} = C^s \cup C^d$ est considérée comme *valide*. La figure III.1 montre un cas à quatre variables $(v_i, i = 0, \ldots, 3)$ où cette situation se produit. La figure III.1 présente des appels successifs aux algorithmes III.1 et III.2. Nous constatons que l'étape 4 teste une configuration déjà testée lors de l'étape 1. Dans ce cas, la fonction `test` retourne le résultat en cache [9].

La compilation est testée (lignes 5-7). En effet la modification des types des variables peut générer une erreur de compilation. Par exemple, une erreur peut survenir lorsqu'une fonction nécessite un tableau de variables en *double* et reçoit en paramètre un tableau en *simple* précision. Ces différences entre prototype et appel de fonction sont générateurs d'erreurs à la compilation.

9. Cela est similaire au principe de mémoïsation où nous ne prenons pas en compte la gestion de l'ordre de la pile [99]. La différence se situe dans la gestion de la pile des résultats. Dans le cas de la mémoïsation, le résultat recherché sera déplacé en haut de la pile pour être obtenu plus rapidement, ce n'est pas le cas dans notre système de cache.

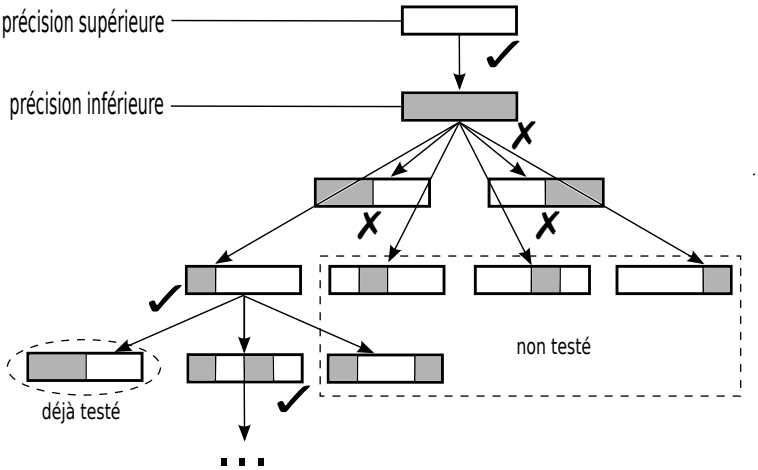


FIGURE III.2 – Création de sous-ensembles par PROMISE. Les sous-ensembles en *simple* (resp. *double*) précision sont représentés en gris (resp. blanc).

Si aucune erreur de compilation ne survient, il faut tenir compte d'une possible erreur à l'exécution (lignes 9-10). Cela peut être par exemple une division par zéro en raison d'une conversion. Par exemple si une variable en *double* précision x égale à $x = 1 + 10^{-14}$ est convertie en *simple* précision, alors sa valeur devient 1 en considérant l'arrondi au plus près. Alors la formule $1/(1 - X)$ produira une division par zéro.

Enfin, le test est vérifié si le résultat est *valide* (lignes 11-12). Les conditions devant être remplies pour qu'un résultat soit considéré comme *valide* sont décrites dans la section III.5.1.

La figure III.2 montre graphiquement l'évolution de l'algorithme III.1. Chaque zone grise ou blanche correspond à au moins une variable. Si la modification d'un sous-ensemble en *simple* précision produit un résultat *valide*, alors les configurations non testées de granularité équivalente dans l'arbre ne sont pas testées. Comme nous l'avons déjà mentionné, le système de cache évite de tester plusieurs fois la même configuration.

III.4.3 Algorithme itératif

L'algorithme III.3 décrit la version itérative de l'algorithme III.1. Il s'agit de la version que nous avons implémentée.

L'ensemble des variables C est un paramètre d'entrée. Comme nous l'avons expliqué précédemment, nous ne considérons ici que les variables en *simple* et *double* précisions et nous supposons que toutes les variables sont en *double* précision à l'initialisation de l'algorithme III.3 (ligne 2). La variable *trouve* sert à déterminer si une configuration *valide* avec de nouvelles variables en *simple* précision vient d'être trouvée ou non.

L'algorithme III.3 est composé de deux boucles imbriquées. La boucle externe est exécutée tant que la granularité p est inférieure à $|C^d|$ et qu'aucun élément n'a été ajouté à C^s lors de la dernière itération (ligne 4). À chacune des itérations de cette boucle, p sous-ensembles sont créés (ligne 5).

La boucle interne est exécutée jusqu'à ce qu'un ensemble C_i^d transformé en *simple*

Algorithme III.3 PROMISE : version itérative de l'algorithme

fonction `PROMISE`(C)

$p \leftarrow 2$
$C^s \leftarrow \emptyset$; $C^d \leftarrow C$
$trouve \leftarrow Faux$
tant que $trouve = Vrai$ ou $p < |C^d|$ **faire**
  partition de C^d telle que $\cup_{i=1}^{p} C_i^d = C^d$ avec $|C_i^d| \approx \frac{|C^d|}{p}$
  $trouve \leftarrow Faux$
  $i \leftarrow 1$
  tant que $i \leq p$ et $trouve \neq Vrai$ **faire**
    si `test`$(C^s \cup C_i^d, C^d \backslash C_i^d) =$ ✓ **alors**
      $C^s \leftarrow C^s \cup C_i^d$; $C^d \leftarrow C^d \backslash C_i^d$
      $trouve \leftarrow Vrai$
    fin si
    $i \leftarrow i + 1$
  fin tant que
  si $trouve = Vrai$ **alors**
    $p \leftarrow \max(p - 1, 2)$
  sinon si $p < |C^d|$ **alors**
    $p \leftarrow \min(2p, |C^d|)$
    $trouve \leftarrow Vrai$
  fin si
fin tant que
retourne C^s

précision donne un résultat *valide* (ligne 9) ou que tous les sous-ensembles précédemment créés aient été testés (ligne 8). Lorsque qu'un résultat est *valide*, C^s et C^d sont modifiés en conséquence (ligne 10).

À la fin de la boucle interne, si une configuration est *valide*, la granularité est modifiée (ligne 16), sinon si la granularité est inférieure à la taille de C^d, elle est doublée avec pour maximum $|C^d|$ (ligne 18). La ligne 19 permet d'exécuter la boucle externe lorsque la granularité est égale à $|C^d|$.

Après cette dernière étape, l'algorithme retourne C^s (ligne 22) si cet ensemble n'a pas été modifié ($trouve = Faux$).

III.4.4 Exemple d'exécution

Nous présentons ici un exemple d'exécution de l'algorithme de PROMISE. La figure III.3 montre cette exécution pour un programme de dix variables : v_0 à v_9 avec le même formalisme que pour la figure III.1 excepté le fait que les étapes en cache ne sont pas représentées. La fonction `test` indique que les résultats sont *valides* lors des étapes 3, 6, 10, 11 et 14. Cela permet d'ajouter définitivement un (des) élément(s) à C^s (cas numéro 1 de l'algorithme III.1. Le cas numéro 2 de l'algorithme III.1) survient lors des étapes 2, 8 et 13, il y a alors diminution de la taille des sous-ensembles qui seront testés en *simple* précision. Nous constatons qu'entre les étapes 14 et 15, l'algorithme devrait tester la configuration n'ayant que $v_2 \in C^d$. Ce test ayant été déjà effectué lors de l'étape 12, le résultat en cache est utilisé. L'ensemble C^s retourné est le dernier ayant eu un résultat *valide* : $\{v_0, v_1, v_3, v_4, v_5, v_7, v_8, v_9\}$ dans notre exemple. Ainsi nous pouvons conclure que dans ce cas les variables v_2 et v_6 doivent être déclarées en *double* précision dans le programme résultant de PROMISE.

III.5 L'outil PROMISE

Nous allons maintenant présenter l'implémentation de l'outil PROMISE. Deux versions de l'outil existent et diffèrent dans la façon de vérifier si un résultat est *valide* ou non. Comme nous avons choisi de travailler sur les sources nous avons utilisé CADNA comme outil de validation lors du développement de PROMISE.

III.5.1 Vérification d'un résultat *valide*

Pour toutes les configurations, PROMISE vérifie si le résultat calculé grâce à cette configuration est *valide*. Une simple comparaison avec un résultat calculé en précision plus élevée en arithmétique à virgule flottante classique n'est pas une bonne solution. En effet, certaines expressions peuvent produire les mêmes résultats, à l'arrondi près, en *simple*, *double* et même en précision étendue comme l'exemple du polynôme de Rump le montre (section I.1.4). C'est pourquoi nous avons choisi d'utiliser l'Arithmétique Stochastique Discrète au sein de PROMISE pour contrôler la qualité numérique des résultats calculés. Nous avons développé deux versions de PROMISE, nous les nommerons dans la suite `entièrement stochastique` et `référence stochastique`.

Dans la version `entièrement stochastique`, toutes les exécutions du programme se font en utilisant l'Arithmétique Stochastique Discrète. Le résultat est considéré

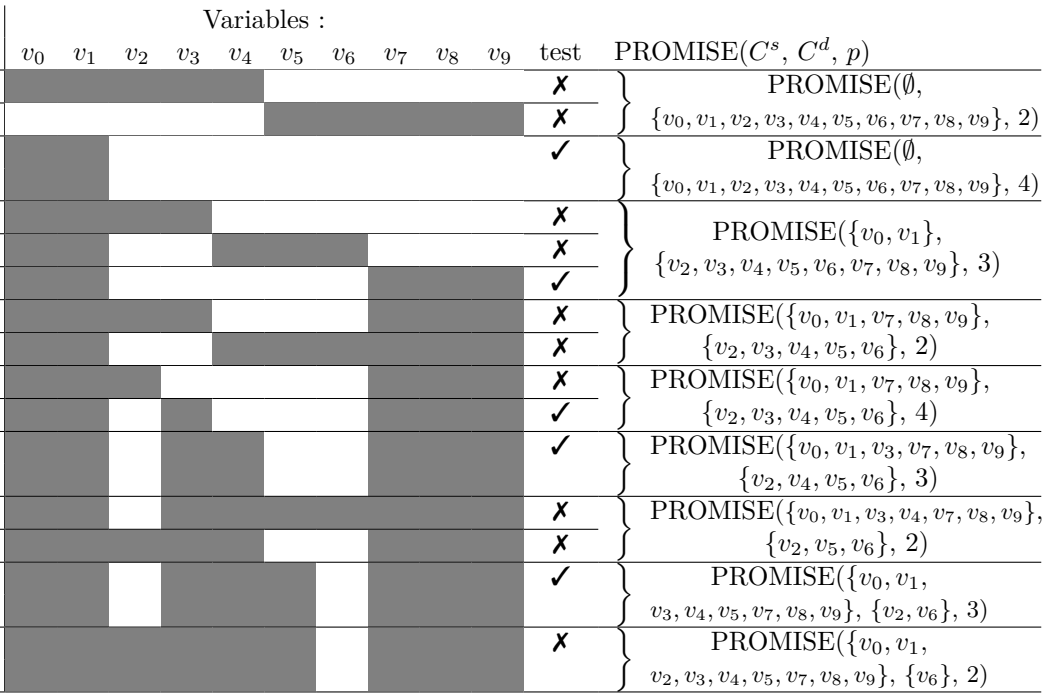


FIGURE III.3 – Exemple d'exécution de l'algorithme récursif de PROMISE avec un programme de $(v_0, \ldots, v_9)$. Les variables en *simple* (resp. *double*) précision sont représentées en gris (resp. blanc)

comme *valide* si deux conditions sont satisfaites. Tout d'abord, le nombre de chiffres significatifs exacts estimé par l'Arithmétique Stochastique Discrète doit au moins être égal au nombre de chiffres demandé par l'utilisateur. De plus, ces chiffres doivent être en commun avec le résultat obtenu dans la précision la plus importante en utilisant l'Arithmétique Stochastique Discrète. Cette condition existe car si de nombreuses absorptions sont présentes, alors les hypothèses de l'Arithmétique Stochastique Discrète peuvent ne pas être respectées et la précision du résultat sera mal déterminée. Cette situation est apparue dans un des programmes testés. Elle est décrite en détail dans [89, p. 29-31].

En raison du coût d'utilisation en temps et en mémoire de l'Arithmétique Stochastique Discrète, une autre version de PROMISE a été développée : la version `référence stochastique`. Une référence est calculée en *double* précision en utilisant l'Arithmétique Stochastique Discrète. Ensuite, pour toutes les configurations testées, le programme est exécuté sans l'Arithmétique Stochastique Discrète, nous utilisons alors l'arithmétique flottante classique. Le résultat obtenu est comparé au résultat de référence, qui est la moyenne des N échantillons de l'Arithmétique Stochastique Discrète.

Il est utile de noter que le résultat peut être un unique scalaire ou un ensemble de valeurs, par exemple un tableau. Les algorithmes III.4 et III.5 décrivent comment la précision de n résultats est testée par chacune des versions de PROMISE.

Algorithme III.4 Vérification de la précision pour la version `entiérement stochastique` de PROMISE

fonction `result_accuracy`(val, ref, req)
val_i $(i = 1, \ldots, n)$: Résultats calculés avec CADNA
ref_i $(i = 1, \ldots, n)$: Résultats de référence calculés avec CADNA
req : Précision demandée

1: **pour** $i = 1, \ldots, n$ **faire**
2: **si** Le nombre de chiffres significatifs exacts de $val_i < req$ **alors**
3: retourne ✗
4: **fin si**
5: **si** $|val_i - ref_i| \geq |ref_i| * 10^{-req}$ **alors**
6: retourne ✗
7: **fin si**
8: **fin pour**
9: retourne ✓

Algorithme III.5 Vérification de la précision pour la version `référence stochastique` de PROMISE

fonction `result_accuracy`(val, ref, req)
val_i $(i = 1, \ldots, n)$: Résultats calculés sans CADNA
ref_i $(i = 1, \ldots, n)$: Résultats de référence avec CADNA
req : Précision demandée

1: **pour** $i = 1, \ldots, n$ **faire**
2: **si** $|val_i - ref_i| \geq |ref_i| * 10^{-req}$ **alors**
3: retourne ✗
4: **fin si**
5: **fin pour**
6: retourne ✓

III.5.2 Implémentation

PROMISE est écrit en Python et utilise une version modifiée de l'algorithme de delta debug implémenté par Zeller [33]. La figure III.4, page 88, résume l'arbre d'exécution des deux versions de PROMISE : `entièrement stochastique` et `référence stochastique`. PROMISE modifie automatiquement un programme en C ou en C++ pour pouvoir utiliser la bibliothèque CADNA. Il s'agit en particulier de modifier les déclarations pour pouvoir utiliser les types stochastiques. Peu importe la version de PROMISE utilisée, le programme est exécuté une première fois avec CADNA en *double* précision pour avoir une référence. PROMISE va alors estimer la précision de ce résultat grâce à CADNA en tenant compte de la précision souhaitée.

Dans le cas où la configuration en *simple* précision uniquement ne retourne pas un résultat *valide*, alors nous allons rechercher une configuration mixte qui le sera, en utilisant l'algorithme III.3. Ces configurations testées seront exécutées avec CADNA si nous utilisons la version `entièrement stochastique` de PROMISE. Avec la version `référence stochastique`, toutes les modifications nécessaires à la bibliothèque CADNA sont supprimées [10] et le programme sera exécuté par la suite en arithmétique à virgule flottante. Lorsque PROMISE termine son exécution, le code résultat peut être fourni avec ou sans les instructions CADNA en fonction du choix de l'utilisateur. Par défaut, PROMISE va optimiser toutes les variables flottantes du programme. Il est toutefois possible de limiter ce comportement et de faire des optimisations uniquement sur les variables choisies par l'utilisateur. Cela permet aux développeurs ayant une connaissance préalable de leur code et des instabilités de celui-ci de limiter les modifications et de diminuer le temps d'exécution de PROMISE.

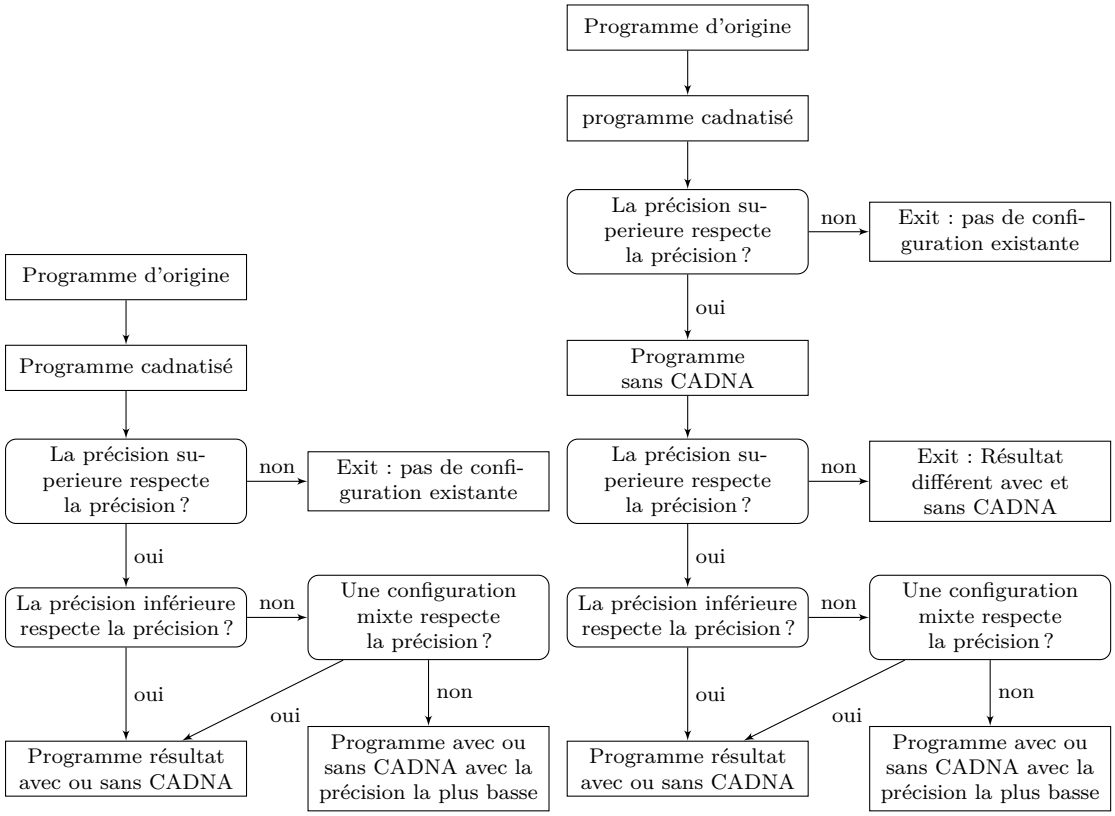
III.6 Évaluation expérimentale

III.6.1 Contexte expérimental

Tous les tests ont été effectués sur une station de travail équipée d'un processeur Intel Xeon CPU E5-2670 à 2.6 GHz avec 128 GB RAM. Tous les codes mentionnés sont écrits en C++ et compilés avec GCC 4.9.2 avec une optimisation en O3. Nous avons utilisé la version C/C++ 2.0.0 de CADNA [39]. Les options `-frounding-math` pour GCC et `-fp-model strict -fp-model no-except` pour ICC vont nous assurer que les optimisations effectuées ne sont pas fait en fonction du type et du mode d'arrondi. Les codes proviennent de différentes sources, le nom entre parenthèses dans la liste sera celui utilisé par la suite.

— Des programmes courts : calcul d'une longueur d'arc (arclength), calcul d'une intégrale par la méthode des rectangles (rectangle), calcul d'une racine carrée par la méthode des Babyloniens (squareRoot), multiplication de matrices (MatMul) ;
— Des programmes de la GNU Scientific Library [26] : transformée de Fourier rapide (FFT), somme des éléments d'une série de Taylor (sum), évaluation et calcul de racines de polynômes (poly) ;

10. Avant tous les tests les codes sont écrits avant d'être compilés et exécutés il n'y a donc pas de perte de temps dans cette suppression.



(a) PROMISE : `entièrement stochastique` (b) PROMISE : `référence stochastique`

FIGURE III.4 – Logigramme de l'exécution de PROMISE

— Des programmes de SNU NPB Suite [27] : méthode du gradient conjugué (CG) avec une matrice de taille 7000 ayant 8 valeurs non nulles par colonne et avec 15 itérations, un solveur penta-diagonal (SP) utilisé pour résoudre l'équation de Navier-Stokes en 3D.

Nous avons tenté d'optimiser le type de toutes les variables définies dans chacun des programmes. Lorsque le résultat est composé de plusieurs valeurs, nous les testons toutes. En particulier dans le cas des matrices ou des vecteurs, tous les éléments doivent passer le test de précision comme nous l'avons mentionné dans la section III.5.1. Dans le but de faire des comparaisons avec Precimonious, un autre outil d'optimisation de type décrit dans la section III.3, nous avons réutilisé certains programmes mentionnés dans [124]. En particulier, le programme "arclength" provient directement de [124], mais avec les variables définies en *double* précision à la place du type *long double*.

Les résultats de ces tests sont décrits dans la section suivante. Nous avons également testé PROMISE sur MICADO, un code industriel plus important et plus complexe à optimiser, voir section III.6.5.

III.6.2 Résultats expérimentaux obtenus avec les différents programmes

Pour tous les programmes cités précédemment, nous testons quatre niveaux de précision, à savoir 4, 6, 8 et 10 chiffres. Les tableaux III.1a et III.1b présentent les résultats obtenus respectivement avec la version `entièrement stochastique` et la version `référence stochastique`. Les résultats reportés sont :

— le nombre de compilations effectuées, et donc de configurations testées (# comp) ;
— le nombre d'exécutions (# exec), une compilation pouvant renvoyer une erreur, nous n'avons pas nécessairement l'égalité entre le nombre de compilations et d'exécutions ;
— le nombre de variables devant rester en *double* précision (# double) ;
— le nombre de variables pouvant être passées en *simple* précision (# float) ;
— le temps d'exécution total de PROMISE ;
— le facteur d'accélération entre la version initiale et la version finale ($\frac{Temps_{initial}}{Temps_{final}}$), les temps sont une moyenne de plusieurs exécutions des programmes.

Pour les neuf programmes et les quatre précisions considérés, les deux versions de PROMISE ont toujours trouvé une nouvelle configuration avec des variables pouvant être passées en *simple* précision. Dans quatorze de ces cas tests, les résultats diffèrent entre les deux versions de PROMISE. En effet, la version `entièrement stochastique` peut amener à une configuration avec plus de variables en *double* précision. Les deux versions n'effectuent pas exactement le même test. Ainsi dans la version `entièrement stochastique` un test supplémentaire est effectué : la précision du résultat estimée par CADNA est comparée à la précision souhaitée. De plus, dans cette version, nous utilisons les tests stochastiques pour comparer le résultat calculé et la référence. Dans le cas où la différence est du bruit numérique, le résultat calculé est considéré comme non *valide*. Nous ne détectons pas ces instabilités avec la version `référence stochastique`, en effet les tests sont effectués avec l'arithmétique flottante classique.

Nous pouvons remarquer que lorsque la même configuration est obtenue par les deux versions de PROMISE, une compilation et une exécution supplémentaires sont effectuées avec la version `référence stochastique`. En effet, dans ce cas nous testons

Programme	# chiffres	# comp - # exec	# double - # float	temps (mm:ss)	facteur d'accélération
arclength	10	20-20	8-1	0:37	1,01
	8	25-25	7-2	0:48	1,00
	6	18-18	3-6	0:32	1,01
	4				
MatMul	10	6-6	2-1	0:05	0.99
	8				
	6				
	4				
rectangle	10	14-14	4-3	0:11	1,00
	8	15-15	3-4	0:10	1,01
	6	6-6	1-6	0:06	1,02
	4				
squareRoot	10	20-20	6-2	0:16	1,00
	8				
	6	2-2	0-8	0:01	1,02
	4				
FFT	10	23-9	3-19	0:18	1,11
	8				
	6	2-2	0-22	0:03	1,10
	4				
poly	10	232-74	53-66	7:07	1,12
	8				
	6	187-28	41-78	5:05	1,11
	4	2-2	0-119	0:05	1,11
sum	10	307-46	54-45	7:56	1,06
	8				
	6	299-38	52-47	7:36	1,04
	4	2-2	0-99	0:05	1,05
CG	10	123-28	24-23	6:54	1,02
	8				
	6	114-22	23-24	6:07	1,03
	4	2-2	0-47	0:15	1,03
SP	10	276-31	64-46	275:24	1,00
	8				
	6	264-18	61-49	209:07	1,01
	4	2-2	0-110	9:59	1,02

(a) version `entièrement stochastique` de PROMISE

Programme	# chiffres	# comp - # exec	# double - # float	temps (mm:ss)	facteur d'accélération
arclength	10	21-21	8-1	0:13	1,01
	8	26-26	7-2	0:15	1,00
	6	16-16	2-7	0:09	0,98
	4				
MatMul	10	7-7	2-1	0:03	0,99
	8				
	6				
	4	3-3	0-3	0:02	1,00
rectangle	10	15-15	4-3	0:06	1,00
	8	16-16	3-4	0:06	1,01
	6	3-3	0-7	0:01	1,01
	4				
squareRoot	10	21-21	6-2	0:07	1,00
	8				
	6	3-3	0-8	0:01	1,02
	4				
FFT	10	24-10	3-19	0:07	1,11
	8				
	6	3-3	0-22	0:01	1,10
	4				
poly	10	233-75	53-66	2:16	1,12
	8	229-71	52-67	2:13	1,11
	6	3-3	0-119	0:04	1,11
	4				
sum	10	297-46	50-49	5:40	1,03
	8				
	6	3-3	0-99	0:05	1,05
	4				
CG	10	115-23	23-24	2:18	1,03
	8				
	6	3-3	0-47	0:11	1,03
	4				
SP	10	265-19	61-49	25:04	1,01
	8				
	6	3-3	0-110	5:31	1,02
	4				

(b) version `référence stochastique` de PROMISE

Table III.1 – Résultats expérimentaux de PROMISE sur plusieurs cas tests.

deux fois la version en *double* précision : avec et sans CADNA.

Le temps d'exécution de PROMISE dépend de la version choisie, du temps d'exécution du programme testé, du nombre de variables à tester et du nombre de compilations/exécutions effectuées. La version `entièrement stochastique` est plus lente que la version `référence stochastique`. Le ratio entre les deux versions est au maximum de 87 ici (programme sum avec 6 chiffres significatifs). Cela provient majoritairement du surcoût de CADNA. De plus, le test étant plus conservatif dans la version `entièrement stochastique`, nous pouvons ne pas considérer une configuration comme *valide* alors que la version `référence stochastique` la considère comme telle. Nous testons alors plus de configurations dans un cas que dans l'autre augmentant ainsi le temps d'exécution global.

SP est le programme le plus lent pour lequel nous trouvons une configuration résultat. En effet, il s'agit d'un code de calcul disposant d'un temps de compilation et d'exécution long. De plus, un nombre important de configurations est testé, jusqu'à 276 (SP avec 8 et 10 chiffres significatifs exacts), mais cela est à mettre en relation avec le nombre total de configurations dans ce cas qui est de 2^{110}. Néanmoins si nous considérons des programmes plus rapides comme arclength, nous obtenons un résultat en moins d'une minute avec neuf variables optimisables. Effectuer ces tests à la main, même sur uniquement deux configurations, nécessiterait plus de temps que d'utiliser PROMISE.

Malgré ces avantages, certains points sont à relever. Nous pouvons par exemple noter que dans certains cas, un grand nombre d'erreurs de compilations surviennent. Dans le cas de SP, seulement 7% des configurations testées sont exécutées. Cela crée un ralentissement en raison du temps des compilations qui génèrent une erreur.

De plus, nous constatons que les facteurs d'accélération obtenus sont assez faibles. Ces programmes sont pour la plupart assez courts. Le passage de *double* à *simple* précision peut impacter en négatif le facteur d'accélération en raison des conversions implicites de types effectuées de manière automatique lors de l'exécution. Néanmoins, les nouvelles configurations proposées peuvent aider les développeurs dans la vectorisation de leurs codes en utilisant au mieux les instructions SIMD du processeur.

III.6.3 Vérification des résultats

Le tableau III.2 présente les résultats des programmes arclength, rectangle et squareRoot avec les configurations obtenues grâce aux deux versions de PROMISE. Les chiffres soulignés sont en commun avec le résultat exact calculé avec Matlab en symbolique puis arrondi à 15 chiffres.

Nous pouvons ainsi constater que ces résultats ont au moins le nombre de chiffres souhaité. Nous pouvons également constater qu'avec les deux versions, le nombre de chiffres significatifs exacts du résultat est le même dans les cas étudiés. Néanmoins la version `entièrement stochastique` est plus conservative sur le nombre de variables restant en *double* précision comme vu précédemment.

Ainsi les deux versions donnent des résultats *valides* au sens de la définition III.4.2.

Programme	# Chiffres	résultat `entièrement stochastique`	résultat `référence stochastique`
arclength	exact	5,79577632241285	5,79577632241285
	10	5,79577632241303	5,79577632241303
	8	5,79577632241303	5,79577632241303
	6	5,79577686259398	5,79577686259398
	4	5,79619549204688	5,79619547341572
rectangle	exact	0,100000000000000	0,100000000000000
	10	0,100000000000002	0,100000000000002
	8	0,100000000000002	0,100000000000002
	6	0,100000001490116	0,100000001490116
	4	0,0999971255660057	0,100003123283386
squareRoot	exact	1,41421356237309	1,41421356237309
	10	1,41421356237309	1,41421356237309
	8	1,41421356237309	1,41421356237309
	6	1,41421353816986	1,41421353816986
	4	1,41421353816986	1,41421353816986

Table III.2 – Comparaison des résultats calculés par le programme optimisé grâce à PROMISE par rapport aux résultats exacts

III.6.4 Comparaison avec Precimonious

Comme nous l'avons mentionné, PROMISE partage certains objectifs et approches avec Precimonious. Il y a néanmoins un certain nombre de différences entre les deux. Tout d'abord, Precimonious se base pour sa référence sur un résultat calculé en précision supérieure en utilisant l'arithmétique flottante, alors que PROMISE utilise l'Arithmétique Stochastique Discrète pour valider les résultats. Ensuite, PROMISE ne va pas prendre en compte le temps d'exécution des programmes modifiés car nous avons souhaité obtenir un maximum de variables en *simple* précision. Precimonious se focalise sur l'obtention du meilleur facteur d'accélération possible. Pour les configurations testées, il vérifie la précision du résultat mais s'assure également qu'un facteur d'accélération est présent. Un résultat précis mais nécessitant plus de temps d'exécution qu'avec les configurations déjà validées ne sera pas retenu. C'est pourquoi la configuration fournie par Precimonious peut dépendre de l'ordinateur et du compilateur utilisés, alors que la solution obtenue avec PROMISE est reproductible [11].

Nous analysons ici les résultats de Precimonious lorsque le temps d'exécution n'est pas pris en compte pour valider une configuration. Nous présentons les résultats obtenus avec deux programmes décrits dans la section III.6.2 (arclength et rectangle). Nous considérons les programmes en *double* précision. Le tableau III.3 montre, pour chacune des précisions souhaitées, le nombre de configurations testées, le nombre de variables de chaque type dans la configuration proposée, le temps d'exécution de Precimonious et le facteur d'accélération obtenu avec le programme optimisé par rapport au programme

11. À la condition que le code testé soit reproductible

Programme	# chiffres	# configurations	# *double*	# *simple*	temps (mm:ss)	facteur d'accélération
arclength	10 8	40	8	1	2:12	1,01
	6	46	7	2	2:32	1,00
	4	60	6	3	3:19	1,02
rectangle	10 8	44	5	2	0:14	1,00
	6	28	3	4	0:09	1,01
	4	2	0	7	0:01	1,01

Table III.3 – Résultats expérimentaux de Precimonious

initial.

Nous pouvons noter que le nombre de variables qui vont être optimisées en *simple* précision est inférieur ou égal au nombre de variables optimisées par PROMISE. Cela provient de l'implémentation de l'algorithme de delta debug au sein de Precimonious. Si la granularité est plus importante que le nombre de variables restant à vérifier, Precimonious ne testera pas l'optimisation sur ces variables, tandis que PROMISE va essayer de modifier ces variables en *simple* précision une par une. Cela permet donc d'obtenir plus de variables en *simple* avec de PROMISE.

En ce qui concerne le temps d'exécution, avec le programme arclength, PROMISE est plus rapide que Precimonious, même lors de l'utilisation de la version `entièrement stochastique`. Le ratio des temps d'exécution peut aller jusqu'à 22. Avec le programme rectangle, ce ratio n'est pas aussi important. Cependant, nous pouvons observer que la version `référence stochastique` est plus rapide que Precimonious pour trouver la configuration résultat. Les performances de PROMISE sont à mettre en parallèle avec le plus faible nombre de compilations et d'exécutions effectuées. Ainsi, Precimonious a nécessité jusqu'à trois fois plus de compilations et d'exécutions que PROMISE. De nouveau cela provient de l'implémentation de l'algorithme de delta debug. Precimonious utilise la version provenant de [142]. Dans cette version, l'algorithme va tester un sous-ensemble en *simple* précision, puis tester le complémentaire (voir algorithme III.6). PROMISE implémente la version provenant de [141] (voir algorithme III.1) qui ne prend pas en compte le complémentaire, ce qui permet de réduire le nombre de configurations testées. Nous pouvons également constater que lorsque le temps d'exécution n'est pas pris en compte par Precimonious, alors le facteur d'accélération est similaire à celui obtenu avec PROMISE.

Le programme arclength, avec les variables initialement en précision *long double*, est analysé par Precimonious dans [124]. En effet, Precimonious gère les *long double* dans les types qui peuvent être optimisés au sein du programme. Les résultats dans [124] sont obtenus en prenant en compte le temps d'exécution dans le choix de la configuration. Nous pouvons constater que le programme arclength résultat a d'avantage de variables transformées en *simple* précision lorsque le temps d'exécution est pris en compte. En effet, plus de sous-ensemble sont testés au sein de Precimonious en utilisant le temps d'exécution pour valider une configuration. Or Precimonious cherche à maximiser localement le nombre de variables en *simple* précision et non de manière globale, ainsi

Algorithme III.6 Precimonious : version récursive de l'algorithme

Soit C_i^d une p partition de C^d telle que $\cup_{i=1}^{p} C_i^d = C^d$ avec $|C_i^d| \approx \frac{|C^d|}{p}$

$Precimonious(C^s, C^d, p)$=

$$\begin{cases} \mathbf{1} - Precimonious(C^s \cup C_i^d, C^d \backslash C_i^d, \max(p-1, 2)) \\ \quad \text{si } \exists i \in \{1, \cdots, p\} \text{ tel que } \texttt{test}(C^s \cup C_i^d,\ C^d \backslash C_i^d) = \checkmark\,; \\ \mathbf{2} - Precimonious(C^s \cup (C^d \backslash C_i^d), C_i^d, \max(p-1, 2)) \\ \text{si } \exists i \in \{1, \cdots, p\} \text{ tel que } \texttt{test}(C^s \cup (C^d \backslash C_i^d), C_i^d) = \checkmark\,; \\ \mathbf{3} - Precimonious(C^s, C^d, \min(2p, |C^d|)) \quad \text{si } p < |C^d|\,; \\ \mathbf{4} - \text{retourne } C^s \qquad\qquad\qquad\qquad\qquad\qquad \text{sinon.} \end{cases}$$

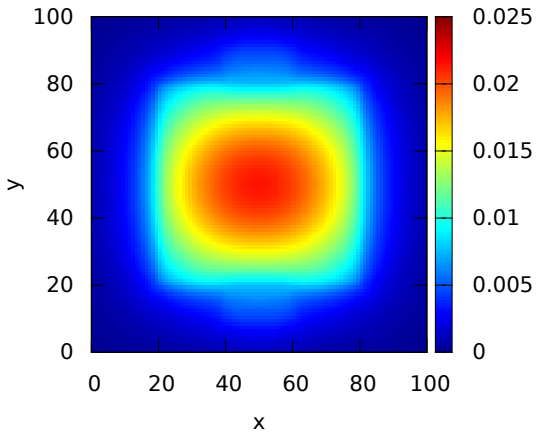
effectuer d'autres tests peut permettre l'obtention d'un maximum local ayant plus de variables en *simple* précision. De plus, le facteur d'accélération, par rapport à la version en précision *long double*, est meilleur (entre 11,0% et 41,7%). À l'exception du cas où nous souhaitons avoir 4 chiffres significatifs exacts, le nombre de configurations testées est plus important dans [124] que dans le tableau III.3, le ratio pouvant être de 4,75. Cela a nécessairement un impact sur le temps d'exécution de Precimonious.

Ces différences sont les conséquences des objectifs de chaque outil. Precimonious fournit un programme plus rapide ; il ne va donc pas prendre en compte les configurations qui ne produisent pas d'accélération. Ainsi à chaque itération de l'algorithme de delta debug, il va tester toutes les configurations pour prendre en compte la branche possédant le meilleur facteur d'accélération, alors que PROMISE va s'arrêter lorsque le résultat est *valide*.

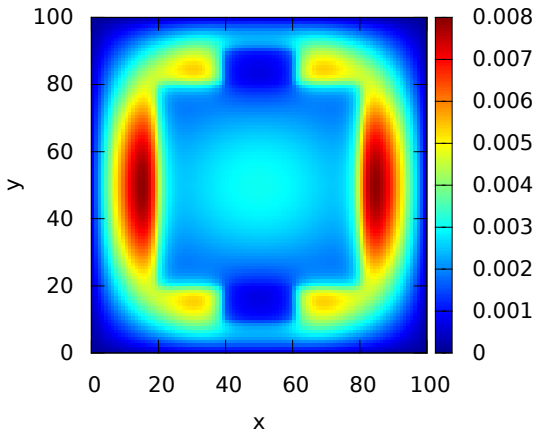
III.6.5 Cas d'un code industriel : MICADO

MICADO [41] est un code de simulation d'un cœur nucléaire dans une centrale. Il est développé à EDF. Il s'agit d'un code permettant de résoudre l'équation de transport des neutrons, dans une géométrie en 2D ou 3D, en utilisant la méthode des caractéristiques [63]. Il s'agit d'un code en C++ composé de 11,000 lignes. MICADO va créer des lignes caractéristiques dans un plan et calculer les intersections entre ces lignes et la géométrie du cœur de la centrale nucléaire. Les segments ainsi formés sont balayés pour calculer le flux de neutrons. Ce code permet de savoir si la réaction de fission est stable, c'est-à-dire si le nombre de neutrons ne varie pas dans le temps, ou encore de déterminer le taux de combustion du combustible nucléaire.

MICADO résout l'équation de Boltzmann. Elle admet une solution non nulle en régime stationnaire si la réaction est stable et auto-entretenue. Plusieurs discrétisations sont nécessaires pour résoudre cette équation et en particulier en fonction du groupe d'énergie des neutrons. Ces groupes d'énergies sont liés, en effet un neutron peut passer d'un groupe d'énergie à l'autre en fonction de sa vitesse. Le cas d'étude utilisé est un benchmark 2D simulant un petit cœur nucléaire avec 20 000 inconnues. Celui-ci calcule le flux de neutrons de deux groupes d'énergies. En utilisant la précision *double*, nous obtenons les résultats graphiques présentés dans les figures III.5a et III.5b.



(a) Flux neutronique du groupe d'énergie 0 en fonction de la position dans le plan



(b) Flux neutronique du groupe d'énergie 1 en fonction de la position dans le plan

FIGURE III.5 – Flux neutronique ($cm^{-2}s^{-1}$) dans le cas test étudié en fonction du groupe d'énergie et de sa position dans le cœur

# digits	# comp - # exec	# double - # float	Time (mm:ss)	facteur d'accélération	gain mémoire
10	75-36	20-31	276:35	1,01	1,00
8	72-33	19-32	255:17	1,01	1,01
6	75-33	17-34	251:57	1,06	1,43
5	61-22	11-40	177:12	1,20	1,43
4	60-21	11-40	171:58	1,24	1,50

(a) Version `entièrement stochastique`

# digits	# comp - # exec	# double - # float	Time (mm:ss)	facteur d'accélération	gain mémoire
10	83-51	19-32	88:56	1,01	1,00
8	80-48	18-33	85:10	1,01	1,01
6	69-37	13-38	71:32	1,20	1,44
5 4	3-3	0-51	9:58	1,32	1,62

(b) Version `référence stochastique`

Table III.4 – Résultats expérimentaux de PROMISE sur le solveur MICADO

MICADO se décompose en deux parties :

— un traceur qui calcule les lignes caractéristiques ($\approx 1\%$ du temps de calcul) ;
— un solveur basé sur l'algorithme de puissance inverse ($\approx 99\%$ du temps de calcul) qui calcule un scalaire λ et un vecteur ϕ.

Nous avons testé PROMISE sur le solveur de MICADO, le traceur impactant peu les performances globales du programme. MICADO est basé sur un solveur itératif qui s'arrête lorsque la différence relative entre deux itérés successifs est inférieure à 10^{-5}, cette valeur étant configurée par l'utilisateur. Un total de 63 itérations est nécessaire pour obtenir une solution. Le tableau III.4 présente les résultats obtenus.

Quelle que soit la version de PROMISE utilisée, une nouvelle configuration est trouvée. La version `entièrement stochastique` nécessite entre 3 et 5 heures pour déterminer cette configuration. La version `référence stochastique` nécessite moins de 90 minutes.

Lorsque nous souhaitons obtenir 8 ou 10 chiffres significatifs exacts, les gains mémoire et les facteurs d'accélération sont inférieurs à 2%. En effet, les modifications de types se font alors dans des fonctions qui impactent peu les performances du code. Dans les autres cas, nous avons un facteur d'accélération et un gain mémoire plus importants allant jusqu'à un facteur d'accélération de 1,32 et un gain mémoire de 1,62.

Selon la version `référence stochastique` de PROMISE, une version du code contenant uniquement des *simple* est *valide* lorsque nous demandons une précision de 5 chiffres significatifs exacts. Une précédente étude sur le solveur de MICADO a montré qu'utiliser des instructions SSE en *simple* permettait d'obtenir un facteur d'accélération de 1,52 par rapport à une exécution scalaire [105]. L'utilisation de PROMISE permet donc de valider cette configuration dans le cadre du cas test étudié.

PROMISE a un intérêt supérieur lorsque la configuration entièrement en *simple* précision n'est pas *valide*. Il s'agit en effet des configurations difficile à obtenir et

nécessitant différent tests et non uniquement la vérification que la configuration en *simple* précision est assez précise. C'est le cas ici lorsque l'on demande 6 chiffres significatifs exacts avec les deux versions. Or nous pouvons tout de même constater que les configurations résultats permettent d'obtenir un facteur d'accélération et un gain mémoire avant tout travail sur la vectorisation.

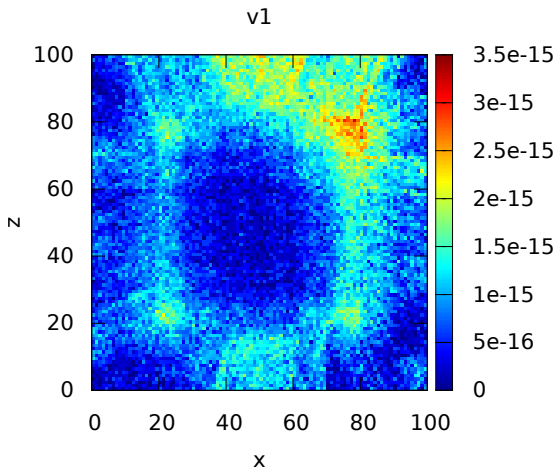
Nous avons vérifié les résultats de l'optimisation graphiquement. Pour cela nous avons affiché la différence relative entre le résultat après optimisation (ici pour 8 chiffres significatifs exacts) res_{opt} et le résultat de référence ref.

Les résultats sont visibles dans les figures III.6a et III.6b. La zone comportant le plus d'erreurs est un carré dont les coins se situent aux points [75,75], [75,100], [100,75] et [100,100] pour les deux figures. Nous pouvons voir que la différence relative est au maximum 10^{-14}, ce qui est inférieur à 10^{-8}.

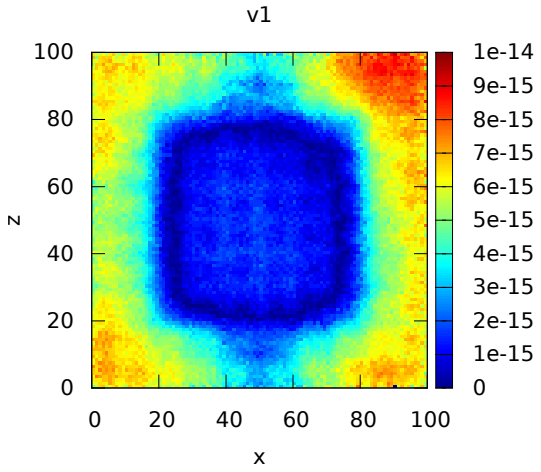
III.7 Conclusion

Nous avons développé PROMISE, un outil qui à partir d'une précision souhaitée par l'utilisateur va maximiser le nombre de variables en *simple* précision dans un programme ayant initialement ses variables déclarées en *double* précision. Nous vérifions les résultats grâce à l'Arithmétique Stochastique Discrète. PROMISE a été testé sur plusieurs codes et en particulier une application industrielle. Durant ces tests, une nouvelle configuration a été trouvée avec des variables transformées en *simple* précision. Ces configurations ont été validées grâce à l'Arithmétique Stochastique Discrète et les résultats sont connus avec un nombre minimum de chiffres significatifs exacts. Le temps d'exécution des programmes testés n'est pas toujours plus faible. Néanmoins, le coût mémoire des programmes a diminué. De plus, les nouvelles configurations fournies peuvent aider un développeur dans l'optimisation de son code grâce à une meilleure utilisation des instructions SIMD.

Plusieurs améliorations de PROMISE sont possibles. Tout d'abord, nous n'utilisons ici que les types *simple* et *double*. La norme IEEE 754 introduit dans la révision de 2008 [73] le type *quad* (deux fois plus précis qu'un *double*). Pour pouvoir utiliser ces types, il faut prévoir certaines modifications au sein de CADNA que nous introduirons dans le chapitre IV. De plus, nous pourrions améliorer le temps d'exécution de PROMISE en parallélisant les tests sur différentes configurations. Nous pourrions également prévoir une analyse statique du code dans le but de lier certaines variables et ainsi diminuer le nombre d'erreurs de compilation. Cela diminuerait le nombre de variables à tester également et donc améliorerait le temps d'exécution de PROMISE. Nous pourrions également prendre en compte des améliorations récentes sur l'algorithme de delta debug qui ont permis d'améliorer les performances pour le debug de 75 à 80% [69]. Nous devons pour cela vérifier l'intérêt des modifications introduites pour notre cas d'utilisation qui reste différent de l'objectif de base.



(a) Différence relative entre le flux neutronique du groupe d'énergie 0 entre la version de référence et la version modifiée (8 chiffres significatifs exacts) en fonction de la position dans le plan



(b) Différence relative entre le flux neutronique du groupe d'énergie 1 entre la version de référence et la version modifié (8 chiffres significatifs exacts) en fonction de la position dans le plan

FIGURE III.6 – Différence entre le flux neutronique ($cm^{-2}s^{-1}$) calculé dans la version de référence et celui calculé dans la version modifiée (8 chiffres significatifs exacts) dans le cas test étudié en fonction du groupe d'énergie et de sa position dans le cœur

Chapitre IV

Validation numérique de calculs en *quadruple* précision

IV.1 Introduction

Le but d'une simulation numérique d'obtenir un résultat valide disposant d'assez de précision pour pouvoir être exploité. Dans le chapitre II, les algorithmes compensés qui permettent d'améliorer la précision des résultats ont été présentés. Néanmoins, dans le cas de certains problèmes mal posés, ils peuvent s'avérer inefficaces ou difficile à mettre en place.

Par simplicité, un développeur peut choisir d'utiliser un type numérique plus précis. En particulier, depuis sa révision, la norme IEEE 754 [73] intègre la *quadruple* précision. Il faut dans ce cas pouvoir valider les résultats obtenus avec ce type.

Nous présenterons tout d'abord, section IV.2, comment est gérée la *quadruple* précision au sein de GCC. Nous comparerons ensuite, en section IV.3, plusieurs types disposant d'une précision équivalente, à savoir les *double-double* de la bibliothèque QD [67] et les nombres avec une mantisse de 113 bits fournis par MPFR [45]. Nous décrirons, en section IV.4, les algorithmes que nous avons implémentés pour les opérations stochastiques en quadruple précision. Nous présenterons des expérimentations numériques effectuées avec la bibliothèque CADNA en *quadruple* précision. Enfin, nous montrerons, en section IV.5, comment le logiciel PROMISE, qui a été décrit dans le chapitre III, a été modifié pour intégrer la *quadruple* précision.

IV.2 Analyse de la *quadruple* précision dans GCC

IV.2.1 Définition et stockage mémoire

Peu de processeurs 128 bits existent, comme nous l'avons mentionné dans le chapitre III, seuls les `SPARC` V8, V9 [131, 132] et les `POWER9` [115] disposent de registres de ce type. Sur les autres architectures, ce type doit être émulé au niveau logiciel pour pouvoir être utilisé. Nous allons décrire ici sa représentation au sein de GCC.

Nous supposons par la suite que nous travaillons sur un processeur 64 bits [1]. Alors

1. Dans le cas d'un processeur 32 bits la taille des registres est limitée à 32 bits. Les éléments

le type `binary128` de la norme IEEE 754 [73], aussi appelé *quadruple* précision, est implementé par la structure *bit field* suivante au sein de GCC [48] :

— 1 bit de signe ;
— 15 bits d'exposant ;
— 48 bits pour la partie "haute" de la mantisse ;
— 64 bits pour la partie "basse" de la mantisse.

Les parties "haute" et "basse" de la mantisse définissent les 112 bits explicites et consécutifs de celle-ci. Il s'agit ici de la représentation *big endian*. Si le processeur utilise la représentation *little endian*, alors le *bit field* débute par la partie "basse" de la mantisse, puis la partie "haute", l'exposant et enfin le signe.

Tout comme pour la *simple* et la *double* précision, un bit implicite est pris en compte en plus des 112 bits explicites pour former la mantisse. Sa valeur est de 1 sauf dans le cas où l'exposant est composé de quinze zéros. Sa valeur sera alors 0 et la variable représentera un nombre dénormalisé.

Nous disposons donc pour définir un nombre en *quadruple* précision d'un ensemble d'entiers sous forme binaire sur lesquels nous pouvons effectuer des manipulations pour obtenir le résultat des opérations arithmétiques. Celles-ci se déroulent selon un schéma identique. Tout d'abord, les quatre éléments de la structure de stockage sont extraits. Les exposants permettent de savoir si le calcul est un cas particulier spécifié par la norme IEEE 754 [73], c'est-à-dire un calcul avec un `NaN`[2], un infini ou un zéro. Le résultat est alors obtenu rapidement. Dans les autres cas, des opérations entières sont effectuées sur les exposants et les mantisses pour obtenir le résultat. Un arrondi est réalisé avant de créer la structure *bit field* du résultat et d'éventuelles exceptions sont levées[3].

Nous allons maintenant nous intéresser aux opérations arithmétiques implémentées dans GCC : l'addition, la soustraction, la multiplication, la division ainsi que la racine carrée.

IV.2.2 Addition et soustraction

La première étape de l'opération consiste à aligner les deux mantisses en fonction de la différence entre les deux exposants. Les deux parties "basses" sont ajoutées puis les parties "hautes". Cette dernière opération peut se faire avec une retenue en cas d'*overflow* survenu lors de l'addition des parties "basses". L'exposant du résultat sera quant à lui soit l'exposant le plus élevé, soit le plus élevé plus un en fonction de la valeur de la mantisse résultat. L'algorithme IV.1 montre les calculs effectués dans le cas de l'addition de deux nombres x et y. Les décalages de mantisse ayant été faits précédemment, ils ne sont pas présents dans le calcul de la mantisse résultat. Les entiers x_l et y_l (resp. x_h et y_h) sont les éléments de la partie "basse" (resp. "haute") des mantisses. Le test de la ligne 2 permet de savoir si un *overflow* est survenu et donc si une retenue est nécessaire.

L'algorithme de la soustraction est similaire, voir algorithme IV.2 qui calcule $x - y$.

nécessitant plus de 32 bits en mémoire doivent être scindés de manière à ne pas dépasser cette valeur (par exemple un élément de 64 bits deviendra deux de 32 bits, un de 48 bits deviendra un de 32 et un

Algorithme IV.1 Calcul de la mantisse résultat lors de l'addition $r = x + y$

1: $r_l = x_l + y_l$
2: **si** $r_l < x_l$ **alors**
3: $\quad r_h = 1$
4: **sinon**
5: $\quad r_h = 0$
6: **fin si**
7: $r_h = r_h + x_h + y_h$

Algorithme IV.2 Calcul de la mantisse résultat lors de la soustraction $r = x - y$

1: $r_l = x_l - y_l$
2: **si** $r_l > x_l$ **alors**
3: $\quad r_h = 1$
4: **sinon**
5: $\quad r_h = 0$
6: **fin si**
7: $r_h = x_h$ - y_h - r_h

IV.2.3 Multiplication

Dans le but de simplifier l'écriture et la lisibilité des algorithmes, nous utiliserons la fonction `ADD_3`. Celle-ci permet d'additionner deux nombres écrits sous la forme d'une somme de trois entiers non signés : $(x0, x1, x2)$ et $(y0, y1, y2)$ voir algorithme IV.3. La fonction `mul_long` prend en entrée deux entiers x et y et fournit en résultat deux nombres $r1$ et $r0$ tels que $x \times y = r1 + r0$, $r1$ et $r0$ ne se recouvrant pas.

Lors du calcul de la multiplication de x par y, l'exposant sera soit la somme des exposants de x et y ou la somme plus 1. Toutes les combinaisons de multiplications entre les parties hautes et les parties basses des deux mantisses sont réalisées. Elles sont ensuite additionnées pour obtenir le résultat avant un arrondi final, voir algorithme IV.4.

IV.2.4 Division

Le calcul de la mantisse résultat d'une division est plus complexe. La fonction `div_long(quotient, reste, numerateur1, numerateur0, denominateur)` permet de diviser la somme de deux entiers non signés `numerateur1` et `numerateur0` par `denominateur`. Les résultats sont le quotient et le reste de cette division. L'ensemble des opérations présentées dans l'algorithme IV.5 permet d'obtenir la mantisse résultat de la division de x par y. L'exposant est obtenu grâce à la soustraction de l'exposant de x par l'exposant de y ou cette soustraction moins 1.

---

de 16 bits).

2. *Not A Number*
3. Les différentes exceptions sont mentionnées dans la section I.1.2.

Algorithme IV.3 Fonction ADD_3($r2, r1, r0, x2, x1, x0, y2, y1, y0$), calcul de $r0 = y0 + x0$, $r1 = y1 + x1 + \delta_1$ et $r2 = y2 + x2 + \delta_2$. δ_1 (resp. δ_2 vaut 1 ou 0 en fonction du calcul de $r0$ (resp. $r1$)).

1: $r0 = y0 + x0$
2: **si** $r0 < x0$ **alors**
3: $tmp0 = 1$
4: **sinon**
5: $tmp0 = 0$
6: **fin si**
7: $r1 = y1 + x1$
8: **si** $r1 < x1$ **alors**
9: $tmp1 = 1$
10: **sinon**
11: $tmp1 = 0$
12: **fin si**
13: $r1 \mathrel{+}= tmp0$
14: **si** $r1 < tmp0$ **alors**
15: $tmp1 = 1 \mid tmp1$
16: **sinon**
17: $tmp1 = 0 \mid tmp1$
18: **fin si**
19: $r2 = y2 + x2 + tmp1$

Algorithme IV.4 Calcul de la mantisse résultat lors de la multiplication $r = x \times y$

1: mul_long($r1, r0, x_l, y_l$)
2: mul_long($tmpA\_1, tmpA\_0, x_l, y_h$)
3: mul_long($tmpB\_1, tmpB\_0, x_h, y_l$)
4: mul_long($r3, r2, x_h, y_h$)
5: ADD_3($r3, r2, r1, 0, tmpA\_1, tmpA\_0, r3, r2, r1$)
6: ADD_3($r3, r2, r1, 0, tmpB\_1, tmpB\_0, r3, r2, r1$)

Algorithme IV.5 Calcul de la mantisse résultat lors de la division $r = x/y$

1: **si** $x_h > y_h$ ou ($x_h == y_h$ et $x_l \geq y_l$) **alors**
2: $nf_2 = x_h >> 1$
3: $nf_1 = x_h << 63 | x_l >> 1$
4: $nf_0 = x_l << 63$
5: **sinon**
6: $nf_2 = x_h$
7: $nf_1 = x_l$
8: $nf_0 = 0$
9: **fin si**
10: Décaler sur la droite y_h et y_l jusqu'à avoir le bit le plus significatif de y_h égale à un
11: div_long($R_h, r_1, nf_2, nf_1, y_h$)
12: mul_long(mf_1, mf_0, R_h, Y_l)
13: $r_0 = nf_0$
14: **si** $mf_1 > y_h$ ou ($mf_1 == y_h$ et $mf_0 > y_l$) **alors**
15: $R_h = R_h - 1$
16: $r = r + y$
17: **si** ($r_h > y_h$ ou ($r_h == y_h$ et $r_l \geq y_l$)) et ($mf_1 > y_h$ ou ($mf_1 == y_h$ et $mf_0 > y_l$)) **alors**
18: $R_h = R_h - 1$
19: $r = r + y$
20: **fin si**
21: **fin si**
22: $savex_l = x_l$
23: $x_l = x_l - y_l$
24: **si** $x_l > savex_l$ **alors**
25: $tmp = 1$
26: **sinon**
27: $tmp = 0$
28: **fin si**
29: $x_h = x_h - y_h + tmp$
30: **si** $r_1 == Y_h$ **alors**
31: $R_0 = -1$
32: **sinon**
33: div_long(R_l, r_1, r_1, r_0, y_h)
34: mul_long(mf_1, mf_0, R_l, y_l)
35: **si** $mf_1 > r_1$ ou ($mf_1 == r_1$ et $mf_0 > r_0$) **alors**
36: $R_l = R_l - 1$
37: $r = r + y$
38: **si** ($r_h > y_h$ ou ($r_h == y_h$ et $r_l \geq y_l$)) et ($mf_1 > r_1$ ou ($mf_1 == r_1$ et $mf_0 > r_0$)) **alors**
39: $R_l = R_l - 1$
40: $r = r + y$
41: **fin si**
42: **fin si**
43: **si** $r \neq m$ **alors**
44: $R_0 = R_0 | 1$
45: **fin si**
46: **fin si**

IV.2.5 Racine carrée

L'algorithme IV.6, qui décrit le calcul de la racine carrée d'un nombre en *quadruple* précision, se découpe en trois parties distinctes.

Tout d'abord de la ligne 1 à 4 l'exposant du résultat est calculé. Deux parties similaires se suivent par la suite. Elles permettent de calculer les valeurs de la partie haute (lignes 9 - 18) et de la partie basse (lignes 20 - 31) de la mantisse.

Enfin, le calcul de la racine carrée consiste à parcourir l'ensemble des bits de la partie haute et de la partie basse de la mantisse en effectuant à chaque fois un test. Cette approche est donc coûteuse en nombre d'opérations mais permet de ne pas utiliser de divisions qui sont elles coûteuses en temps.

IV.3 Comparaison de la *quadruple* précision avec les *double-double* et MPFR

IV.3.1 Les bibliothèques *Double-Double*

Les *double-double* sont des expansions de taille fixe composées de deux nombres en *double* précision. Les opérations impliquant des *double-double* sont fondées sur les transformations exactes [32, 85, 107, 120]. Néanmoins, les algorithmes utilisés ne permettent pas d'obtenir un résultat avec un arrondi tel que défini dans la norme IEEE 754 [73]. En effet, les algorithmes utilisés pour calculer les *double-double* sont sujets aux erreurs et une erreur sur l'élément le plus petit est observable dans certains cas [107, section 14.1].

De plus, la taille de l'exposant étant différente entre un nombre en *double* et un nombre en *quadruple* précision, les *double-double* ne peuvent pas représenter une plage de nombres aussi importante qu'avec la précision *quadruple*. Enfin, la taille de la mantisse d'un *double-double* est de 106 bits au total alors qu'elle est de 113 bits pour la *quadruple* précision.

Par la suite, nous avons étudié plus particulièrement les *double-double* de la bibliothèque QD [67] [4]. Plusieurs implémentations des opérations existent dans celle-ci. Il y a ainsi les versions *sloppy* pour l'addition et la division qui permettent d'obtenir le résultat en moins d'opérations en s'autorisant une erreur plus importante sur le résultat.

Un *double-double* de la bibliothèque QD s'écrit sous la forme $x = x_h + x_l$ avec $|x_h| \geq |x_l|$. x_h (resp. x_l) peut être considérée comme la partie haute (resp. basse) de x.

IV.3.1.a Addition

Les algorithmes IV.7 et IV.8 présentent les deux implémentations dans la bibliothèque QD [67] de l'addition de x et y.

La version *sloppy*, algorithme IV.7, est à utiliser avec parcimonie. En effet, l'erreur relative du résultat n'est pas bornée lorsque x et y sont de signes opposés. Ainsi si $x_h = 1 + 2^{-p+3}$, $x_l = -2^{-p}$, $y_h = -1 - 6 \times 2^{-p}$ et $y_l = -2^{-p} + 2^{-2p}$ alors le résultat calculé sera 0 alors que le résultat exact est 2^{-2p} [83].

4. Il existe d'autres implémentations similaires(par exemple [13])

Algorithme IV.6 Calcul de la racine carrée de x en *quadruple* précision, *exponent_x* est l'exposant de x, x_h la partie haute de la mantisse et x_l la partie basse de la mantisse

1: **si** $exponent\_x == 1$ **alors**
2: Décaler tous les bits de la mantisse sur la gauche
3: **fin si**
4: $exponent\_r = exponent\_x >> 1$
5: $tmp\_sqrt\_s = (0,0)$
6: $r = (0,0)$
7: $tmp\_sqrt\_q = 53 >> 1$
8: $q = 1 << 63$
9: **tant que** q **faire**
10: $tmp\_t_h = tmp\_sqrt\_s_h + q$
11: **si** $tmp\_t_h < x_h$ **alors**
12: $tmp\_sqrt\_s_h = tmp\_t_h + q$
13: $x_h = x_h - tmp\_t_h$
14: $r_h = r_h + q$
15: **fin si**
16: Décaler tous les bits de x_l et x_h sur la gauche
17: $q = q >> 1$
18: **fin tant que**
19: $q = 1 << 63$
20: **tant que** $q \neq 1 << 2$ **faire**
21: $tmp\_t_l = tmp\_sqrt\_s_l + q$
22: $tmp\_t_h = tmp\_sqrt\_s_h$
23: **si** $tmp\_t_h < x_h$ ou ($tmp\_t_h == x_h$ et $tmp\_t_l \leq x_l$) **alors**
24: $tmp\_sqrt\_s_l = tmp\_t_h + q$
25: $tmp\_sqrt\_s_h += (tmp\_t_l > tmp\_sqrt\_s_l)$
26: $x_h -= tmp\_t_h + ((x_l -= tmp\_t_l) > x_l)$
27: $r_l += q$
28: **fin si**
29: Décaler tous les bits de x_l et x_h sur la gauche
30: $q = q >> 1$
31: **fin tant que**
32: **si** $x_h \neq 0$ and $x_l \neq 0$ **alors**
33: **si** $tmp\_sqrt\_s_h < x_h$ ou ($tmp\_sqrt\_s_h == x_h$ et $tmp\_sqrt\_s_l \leq x_l$) **alors**
34: $r_l = r_l|1 << 2$
35: $r_l = r_l|1 << 0$
36: **fin si**
37: **fin si**

Algorithme IV.7 Addition *sloppy* de deux *double-double* x et y

1: $s_h, s_l = \texttt{TwoSum}(x_h, y_h)$
2: $v = x_l + y_l$
3: $w = s_l + v$
4: $z_h, z_l = \texttt{FastTwoSum}(s_h, w)$

Dans le cas où nous ne connaissons pas à l'avance les signes des opérandes, il faut donc utiliser un algorithme permettant d'obtenir un résultat avec une erreur relative bornée entre le résultat exact et le résultat obtenu. C'est le cas de l'algorithme IV.8.

Algorithme IV.8 Addition de deux *double-double* x et y

1: $s_h, s_l = \texttt{TwoSum}(x_h, y_h)$
2: $t_h, t_l = \texttt{TwoSum}(x_l, y_l)$
3: $c = s_l + t_h$
4: $v_h, v_l = \texttt{FastTwoSum}(s_h, c)$
5: $w = t_l + v_l$
6: $z_h, z_l = \texttt{FastTwoSum}(v_h, w)$

IV.3.1.b Multiplication

Dans la librairie QD, la multiplication entre deux *double-double* ne dispose que d'une seule implémentation, voir algorithme IV.9, la version *sloppy* n'existant que pour le type *quad-double*. Dans le cas où l'opération FMA est disponible, il est possible de l'améliorer en n'effectuant qu'une seule opération entre la ligne 3 et 4 : $c_{l2} = \texttt{FMA}(x_l, y_h, t_{l1})$.

Algorithme IV.9 Multiplication de deux *double-double* x et y

1: $c_h, c_{l1} = \texttt{TwoProd}(x_h, y_h)$
2: $t_{l1} = x_h \times y_l$
3: $t_{l2} = x_l \times y_h$
4: $c_{l2} = t_{l1} + t_{l2}$
5: $c_{l3} = c_{l1} + c_{l2}$
6: $z_h, z_l = \texttt{FastTwoSum}(c_h, c_{l3})$

IV.3.1.c Division

Dans le but de simplifier l'écriture des algorithmes, nous utilisons l'algorithme IV.10 permettant de retourner un *double-double* résultat de l'opération $(x + y) \times z$, avec x,y et z des variables en *double* précision.

Algorithme IV.10 Fonction $addMul(y_h, y_l, t_h)$: évaluation de $(y_h + y_l) \times t_h$

1: $c_h, c_{l1} = \texttt{TwoProd}(x_h, y)$
2: $c_{l2} = x_l \times y$
3: $t_h, t_{l1} = \texttt{FastTwoSum}(c_h, c_{l2})$
4: $t_{l2} = t_{l1} + c_{l1}$
5: $z_h, z_l = \texttt{FastTwoSum}(t_h, t_{l2})$

L'algorithme IV.11 permet de calculer la division de x par y de manière *sloppy*. Le but est encore une fois de calculer rapidement en s'autorisant une erreur. La version plus précise, algorithme IV.12, effectue une division supplémentaire. Or la division est une des opérations nécessitant le plus de cycles d'horloge.

Algorithme IV.11 Division *sloppy* de deux *double-double* x et y

1: $t_h = x_h/y_h$
2: $r_h, r_l = addMul(y_h, y_l, t_h)$
3: $\pi_h, \pi_l = \texttt{TwoSum}(x_h, -r_h)$
4: $\delta_h = \pi_l - r_l$
5: $\delta_l = \delta_h + x_l$
6: $\delta = \pi_h + \delta_l$
7: $t_l = \delta/y_h$
8: $z_h, z_l = \texttt{FastTwoSum}(t_h, t_l)$

Algorithme IV.12 Division de deux *double-double* x et y

1: $q_h = x_h/y_h$
2: $r = x - q_h \times y$
3: $q_l = r_h/y_h$
4: $r = r - (q_l \times y)$
5: $tmp = r_h/y_h$
6: $(q_h, q_l) = \texttt{FastTwoSum}(q_h, q_l)$
7: $r = q_h + q_l + tmp$

IV.3.1.d Racine carrée

Le calcul de la racine carrée d'un nombre en *double-double* dans la bibliothèque QD se fonde sur l'idée suivante de Karp et Markstein [84] :

Si x est une approximation de $1/\sqrt{a}$, alors :

$$\sqrt{a} \approx a \times x + [a - (a \times x)^2] \times \frac{x}{2}.$$

Cette approximation est deux fois plus précise que x. De plus, les multiplications $a \times x$ et $(-1) \times x$ peuvent être effectuées avec la moitié de la précision souhaitée. Il est ainsi possible d'estimer la valeur de la racine carrée en *double-double* tout en utilisant majoritairement la *double* précision, ce qui permet d'améliorer grandement les performances du calcul.

Algorithme IV.13 Calcul de la racine carrée d'un *double-double* A. Les variables en majuscules sont des *double-double*, les variables en minuscules sont des *double*.

1: $x = 1/\sqrt{A_h}$
2: $ax = x \times A_h$
3: $tmp = 0.5 \times x$
4: $TMP1 = ax \times ax$
5: $TMP1 = A - TMP1$
6: $tmp2 = TMP1_h \times tmp$
7: $RES = ax + tmp2$

IV.3.2 Évaluation expérimentale

Les tests sont effectués sur une station de travail disposant d'un processeur Intel Xeon CPU E5-2660 cadencé à 2,2 GHz et de 32 Go de RAM. Tous les codes sont écrits en C++ et compilés avec G++ 4.8.5 ou ICC 17. La bibliothèque MPFR 3.1.1 est utilisée avec une précision de 113 bits pour la mantisse. La taille de l'exposant n'est pas fixée au sein de MPFR et peut donc amener au traitement de nombres infimes qui ne pourraient pas être représentés avec la *quadruple* précision ou le type *double-double*.

Nous utiliserons pour les tests deux *benchmarks* :

— Matrix : multiplication naïve de matrices de taille 1,000×1,000 ;

— Map : suite récurrente définie dans l'algorithme IV.14 avec 128,000,000 itérations.

Algorithme IV.14 Suite récurrente utilisée pour les comparaisons de performances

$U_0 = 1.1$
pour $i = 1$ à n **faire**
  $U_i = (0.1 * U_{i-1} - (1/3 + U_{i-1})^2)/(1 - U_{i-1})^3$
fin pour

La suite présentée dans l'algorithme IV.14 a été définie par nos soins. Elle prend en compte les différentes opérations arithmétiques existantes et permet d'éviter certaines optimisations telles que la vectorisation. La fonction `pow` n'est pas utilisée, des multiplications sont effectuées à la place.

Les figures IV.1a et IV.1b présentent le ratio des différents types étudiés par rapport à la *double* précision sans aucune optimisation (avec l'option de compilation *O0*) : $ratio = \frac{Temps_{type}}{Temps_{double}}$. Nous pouvons voir que dans tous les cas, MPFR est l'approche la plus coûteuse. Cela vient du fait que cette bibliothèque permet d'utiliser des mantisses ayant des tailles supérieures à 1000 bits. Les algorithmes de calcul sont donc des algorithmes généraux non optimisés pour une taille de mantisse de 113 bits.

Nous remarquons également que, dans nos expérimentations effectuées avec l'option de compilation *O0*, la différence de ratio entre les deux versions des *double-double* sont proches [5].

Enfin nous pouvons voir, qu'avec l'option *O0*, la *quadruple* précision permet d'obtenir les meilleures performances mais qui correspondent tout de même à un facteur dix par rapport à la *double* précision.

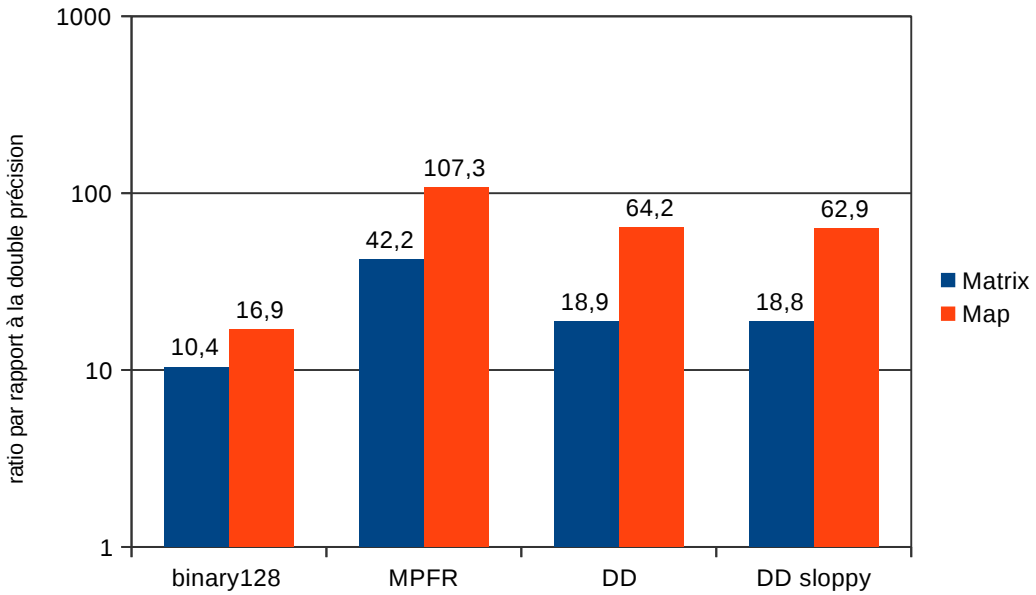
Les figures IV.2a et IV.2b présentent ces ratios lorsque l'option de compilation *O3* est utilisée. De nouveau, MPFR est l'approche la plus coûteuse.

La version *sloppy* des *double-double* permet d'obtenir de meilleures performances que la version classique des *double-double*. Néanmoins, la différence est assez faible dans la plupart des cas [6].

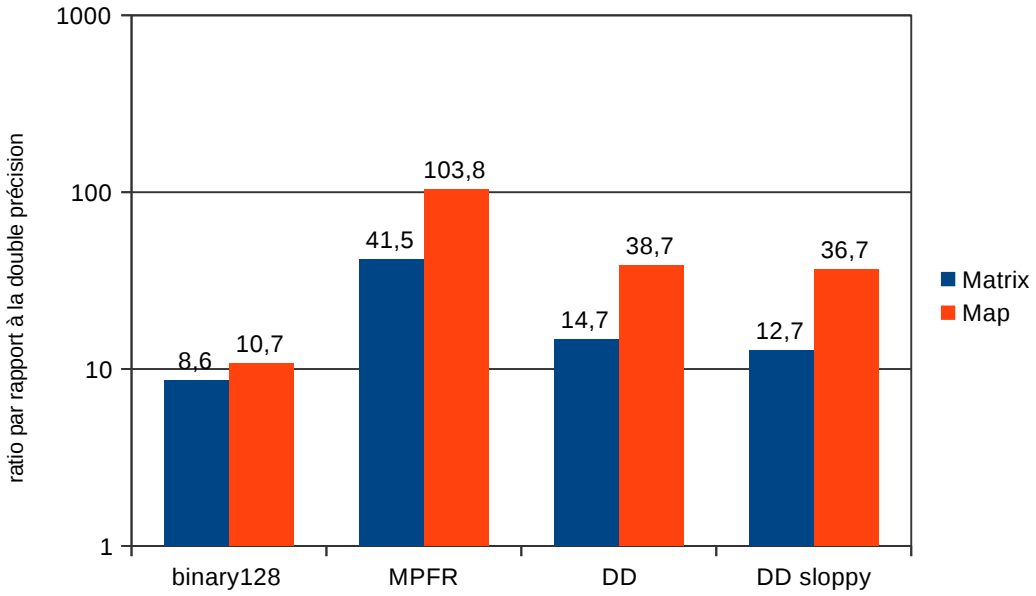
Avec la suite récurrente, la *quadruple* précision et les *double-double* ont des ratios similaires. La *quadruple* précision est néanmoins beaucoup plus coûteuse lors du calcul

5. Avec GCC les ratios sont de 18,9 et 18,8 pour Matrix et de 64 et 62 pour Map. Avec ICC, ils sont de 14 et 12 pour Matrix et 38 et 36 pour Map.

6. Avec GCC, nous passons d'un ratio de 13 à 9 pour Matrix et 13 à 12 pour Map. Avec ICC, il est de 47 contre 38 pour Matrix et 10 contre 7 pour Map.



(a) Ratio de différents types par rapport à la *double* précision – compilation avec GCC et l'optimisation *O0*



(b) Ratio de différents types par rapport à la *double* précision – compilation avec ICC et l'optimisation *O0*

Figure IV.1 – Ratio par rapport à la double précision de la *quadruple* précision, du type *double-double* et de MPFR avec 113 bits de mantisse avec l'option *O0*

matriciel [7]. Il se trouve que les calculs matriciels en *simple* et *double* précision sont régulièrement étudiés lors du développement des compilateurs et sont optimisés. Les calculs en *double-double* sont également optimisés grâce à l'utilisation de l'option *O3*. La *quadruple* précision ne possède pas ces avantages. Chaque calcul en *quadruple* précision implique des accès à la structure de données pour la lecture des opérandes et l'écriture du résultat.

Nous pouvons toutefois rappeler que la *quadruple* précision respecte la norme IEEE 754, ce qui n'est pas le cas des *double-double* de la bibliothèque QD. En particulier aucun des arrondis définis dans la norme IEEE 754 n'est utilisable avec les *double-double*. Il est plus donc judicieux d'étendre la bibliothèque CADNA avec la *quadruple* précision, plutôt qu'avec le type *double-double*.

IV.4 Arithmétique stochastique et *quadruple* précision

IV.4.1 Prise en compte de la *quadruple* précision dans CADNA

Pour pouvoir implémenter dans la bibliothèque CADNA le type `quad_st`, c'est-à-dire la *quadruple* précision stochastique, certains points sont à prendre en compte. Tout d'abord nous pouvons remarquer que le nom définissant la *quadruple* précision n'est pas le même avec les compilateurs GCC (`__float128`) et ICC (`_Quad`). Pour uniformiser les déclarations, nous avons utilisé le *wrapper* de la bibliothèque multiprécision de Boost [11]. Cela n'amène aucun surcoût et permet également la définition des prototypes des fonctions spécifiques à la *quadruple* (telles que les fonctions mathématiques) de manière identique entre ICC et GCC. Le *wrapper* utilisé est visible dans le code 4.

Les fonctions mathématiques implémentées dans la bibliothèque mathématique en *quadruple* précision de GCC nécessitent un suffixe `q` en *quadruple* précision (par exemple `cosq` ou `logq`). Nous avons donc surchargé ces fonctions pour qu'elles soient accessibles grâce à leur nom standard. Il est ainsi possible d'utiliser la fonction `cosq` ou `cos` avec le type `quad_st`.

Après avoir ajouté les instructions permettant de mêler la *quadruple* précision classique avec les types stochastiques en *simple* et *double* précision, nous avons introduit le type `quad_st` au sein de CADNA. Pour ce faire, nous avons utilisé la même approche que dans la version 2.0.0 de CADNA [39]. Aucune modification explicite du mode d'arrondi n'est effectuée durant les calculs. Le mode d'arrondi est fixé de façon arbitraire à plus ou moins l'infini au début de l'exécution du programme grâce à une fonction d'initialisation. Le mode d'arrondi opposé est alors obtenu grâce aux formules :

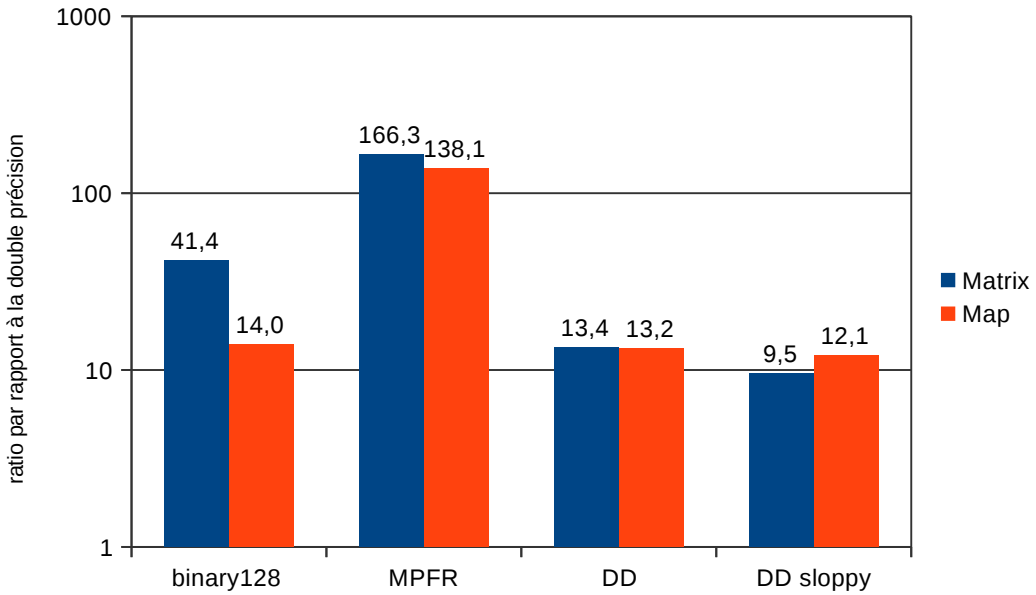
— $a \oplus_{+\infty} b = -\left(-a \oplus_{-\infty} -b\right)$ (similaire pour $\ominus$)

— $a \otimes_{+\infty} b = -\left(a \otimes_{-\infty} -b\right)$ (similaire pour $\oslash$)

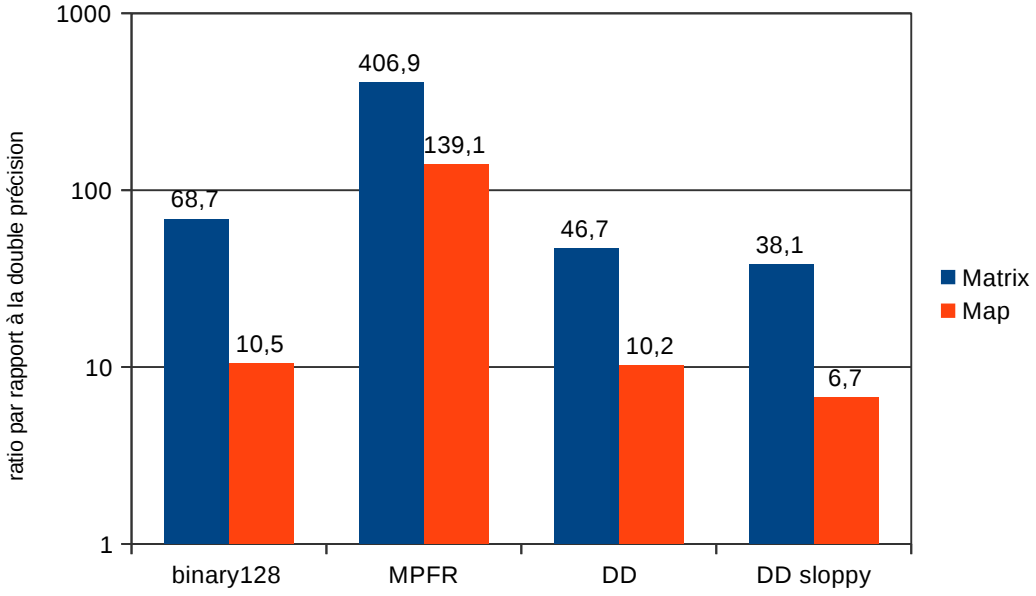
avec $\oplus_{+\infty}$ et $\otimes_{+\infty}$ (resp. $\oplus_{-\infty}$ et $\otimes_{-\infty}$) les opérations flottantes arrondies vers $+\infty$ (resp. $-\infty$). Les opérateurs bit à bit sont disponibles pour les variables en *quadruple* précision. Il est ainsi possible de modifier directement le bit de signe grâce à une opération "ou exclusive" (`XOR`), comme pour la *simple* et la *double* précision. Il est à noter qu'il faut utiliser un entier de 128 bits pour effectuer cette opération [8]. Cette

7. Elle est 3 fois plus coûteuse avec GCC et $1,5$ fois avec ICC

8. Tout comme le changement du bit de signe nécessite un *unsigned int* (32 bits) pour un un nombre en *simple* précision et un *unsigned long* (64 bits) pour un nombre en *double* précision.



(a) Ratio de différents types par rapport à la *double* précision – compilation avec GCC et l'optimisation *O3*



(b) Ratio de différents types par rapport à la *double* précision – compilation avec ICC et l'optimisation *O3*

Figure IV.2 – Ratio par rapport à la double précision de la *quadruple* précision, du type *double-double* et de MPFR avec 113 bits de mantisse avec l'option O3

Code 4 Code permettant d'uniformiser les notations relatives à la quadruple précision entre GCC et ICC

```
#if defined(__GNUC__)
extern "C"{
  #include <quadmath.h>
}
typedef __float128 float128;
#elif defined(__INTEL_COMPILER)
typedef _Quad float128;
extern "C" {
_Quad __ldexpq(_Quad, int);
_Quad __frexpq(_Quad, int*);
_Quad __fabsq(_Quad);
_Quad __floorq(_Quad);
_Quad __ceilq(_Quad);
_Quad __sqrtq(_Quad);
_Quad __truncq(_Quad);
_Quad __expq(_Quad);
_Quad __powq(_Quad, _Quad);
_Quad __logq(_Quad);
_Quad __log10q(_Quad);
_Quad __sinq(_Quad);
_Quad __cosq(_Quad);
_Quad __tanq(_Quad);
_Quad __asinq(_Quad);
_Quad __acosq(_Quad);
_Quad __atanq(_Quad);
_Quad __sinhq(_Quad);
_Quad __coshq(_Quad);
_Quad __tanhq(_Quad);
_Quad __fmodq(_Quad, _Quad);
_Quad __atan2q(_Quad, _Quad);
#define ldexpq __ldexpq
#define frexpq __frexpq
#define fabsq __fabsq
#define floorq __floorq
#define ceilq __ceilq
#define sqrtq __sqrtq
#define truncq __truncq
#define expq __expq
#define powq __powq
#define logq __logq
#define log10q __log10q
#define sinq __sinq
#define cosq __cosq
#define tanq __tanq
#define asinq __asinq
#define acosq __acosq
#define atanq __atanq
#define sinhq __sinhq
#define coshq __coshq
#define tanhq __tanhq
#define fmodq __fmodq
#define atan2q __atan2q}
inline _Quad isnanq(_Quad v)
{
   return v != v;
}
inline _Quad isinfq(_Quad v)
{
   return __fabsq(v) > 1.189731495357231765085759326628007 02e4932Q;
}
#endif // compiler
```

approche est plus rapide qu'une multiplication par -1.

Enfin certaines fonctions ne sont pas prévues pour être utilisées en *quadruple* précision, en particulier les fonctions d'affichage du langage C (`printf`) ou de redirection du C++ (l'opérateur `<<`). Il a donc fallu redéfinir ces opérateurs en utilisant la fonction adéquate : `quadmath_snprintf`. Cette dernière transforme un nombre en *quadruple* précision en une chaîne de caractères.

Les fonctions internes de CADNA utilisant les fonctions mathématiques, par exemple la fonction permettant de calculer le nombre de chiffres significatifs exacts d'un nombre, ont également été modifiées. L'utilisation des fonctions mathématiques en *quadruple* précision est parfois nécessaire.

Enfin, dans le but d'améliorer l'alignement mémoire nous avons introduit un *padding* pour la *quadruple* précision stochastique, tout comme il en existe un pour la *double* précision stochastique. Nous avons choisi d'utiliser un alignement sur 64 bits. En effet, comme nous l'avons déjà précisé, la plupart des processeurs actuels disposent de registres 64 bits. Il est donc inutile de faire un alignement sur 128 bits car cela nous ferait perdre de l'espace mémoire. Ainsi la *quadruple* précision stochastique se compose de 3 nombres en *quadruple* précision (3×128 bits), un entier permettant de garder en mémoire le nombre de chiffres significatifs exacts (32 bits) et d'un entier servant au décalage mémoire (32 bits). Il faut donc 448 bits de mémoire pour stocker les différents éléments d'un nombre en *quadruple* précision stochastique.

Enfin, nous laissons la possibilité à l'utilisateur d'activer ou non la *quadruple* précision stochastique. Ainsi lors de la configuration, il faut utiliser l'option `--enable-float128` si nous souhaitons utiliser la *quadruple* précision (stochastique ou non) avec CADNA.

IV.4.2 Comparaison entre les implémentations stochastiques et les types non stochastiques

Nous allons maintenant étudier le surcoût introduit avec CADNA sur les trois précisions : *simple*, *double* et *quadruple* avec les *benchmarks* Matrix et Map. Nous mesurons ici le ratio de temps entre la précision stochastique et la précision fournie par la norme IEEE 754. Les tests sont réalisés avec les différents niveaux de détection d'instabilités définis dans la section I.4.3 (page 22) :

— *aucune instabilité* ;
— *auto-validation* ;
— *toutes les instabilités.*

Des instabilités sont détectées dans tous les codes quelle que soit la précision. Ainsi les codes matrix et map génèrent des multiplications instables ainsi que des cancellations.

Le tableau IV.1 présente, pour les benchmarks matrix et map, le coût de CADNA par rapport à l'exécution en arithmétique à virgule flottante classique. Nous pouvons tout d'abord noter que la détection des instabilités augmente le temps d'exécution en raison des tests effectués.

De manière générale nous pouvons remarquer que le surcoût de la *quadruple* précision stochastique est inférieur à celui de la *simple* et *double* précision. Cela est particulièrement visible pour le benchmark matrix. Ce surcoût est compris entre 3,6 et 20,8 pour

		aucune instabilité		*auto-validation*		*toutes les instabilités*	
		GCC	ICC	GCC	ICC	GCC	ICC
simple	matrix	15	16,3	16	17,7	34	43,9
	map	10	6,8	14,5	7,3	20	10,6
double	matrix	20	11	22	12,3	35	19,9
	map	11	7,2	14	8,8	19,6	10,3
quadruple	matrix	5	3,6	5,4	3,8	20,8	17,4
	map	7,8	5	12,2	7,4	19,4	8,8

Table IV.1 – Coût de CADNA par rapport à l'exécution en arithmétique à virgule flottante classique, avec optimisation en *O3*

la *quadruple* précision.

Ce faible surcoût en *quadruple* précision s'explique grâce à plusieurs facteurs. Les compilateurs sont optimisés pour travailler avec la *simple* et la *double* précision. La *quadruple* précision est moins performante en raison de l'appel à la structure avant et après chacune des opérations arithmétiques. L'utilisation des types stochastiques ne permet pas certaines optimisations effectuées avec la *simple* et la *double* précision. Le surcoût est donc important dans ce cas. La *quadruple* précision ne disposant pas de ces optimisations également, le fait d'utiliser la *quadruple* précision stochastique amène un surcoût moins important qu'en *simple* ou en *double* précision.

Nous pouvons ainsi voir que la *quadruple* précision stochastique amène un surcoût raisonnable par rapport aux surcoûts obtenus en *simple* et *double* précision. Ainsi elle peut être utilisée dans un contexte industriel.

IV.4.3 Comparaison entre la *quadruple* précision de CADNA et de SAM

Nous comparons ici l'implémentation de la *quadruple* précision stochastique dans CADNA avec SAM [58], la bibliothèque stochastique multiprécision fondée sur MPFR, en définissant des mantisses de taille 113 bits.

Le tableau IV.2 présente les résultats obtenus sur les programmes de tests Matrix et Map compilés avec GCC et ICC.

Nous pouvons voir que dans tous les cas, la *quadruple* précision stochastique permet

	aucune instabilité		*auto-validation*		*toutes les instabilités*	
	CADNA	SAM	CADNA	SAM	CADNA	SAM
	GCC					
matrix	240	679	251	754	1144	7387
map	133	547	207	2148	409	4684
	ICC					
matrix	163	698	170	724	791	7593
map	90	597	133	2145	160	4754

Table IV.2 – Temps d'exécution en secondes de la *quadruple* précision stochastique et de SAM avec une mantisse de 113 bits, compilation avec GCC et ICC

d'obtenir de meilleures performances que SAM avec une taille de mantisse de 113 bits. Nous avons au minimum un facteur 3 et qui peut aller jusqu'à 30 dans le cas de Map avec détection de toutes les instabilités et compilé avec ICC. La bibliothèque SAM est fondée sur MPFR, or, comme nous l'avons vu dans la section IV.3.2, MPFR est plus lent que la *quadruple* précision.

L'intérêt de cette extension de CADNA par rapport à une utilisation de SAM avec une taille de mantisse de 113 bits est donc réel.

IV.5 Utilisation de la *quadruple* précision

IV.5.1 Amélioration de la précision des résultats pour un système chaotique : l'attracteur de Hénon

L'attracteur de Hénon est un système dynamique discret [71]. Il s'agit d'une suite $(x_i, y_i) \in \mathbb{R}^2$ avec

$$\begin{cases} x_{i+1} &= 1 + y_i - a \times x_i^2 \\ y_{i+1} &= b \times x_i \end{cases} \tag{IV.5.1}$$

En fonction des paramètres a et b, il s'agit d'un système pouvant être chaotique, intermittent ou converger vers un point fixe. Nous nous intéresserons ici à l'attracteur de Hénon avec les paramètres les plus classiques : $a = 1,4$ et $b = 0,3$ et ayant pour valeurs initiales $x_0 = 1$ et $y_0 = 0$. Avec ces paramètres, le système est considéré comme chaotique. La figure IV.3 montre l'évolution de l'attracteur de Hénon obtenu en calcul symbolique pour 1,000,000 itérations.

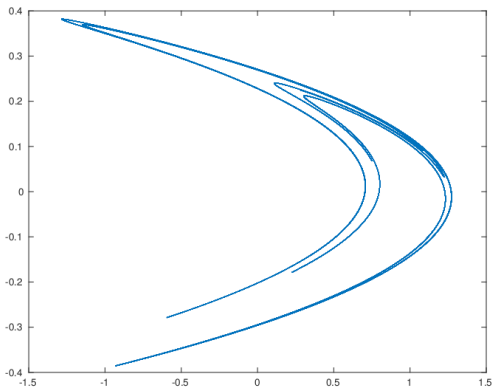


Figure IV.3 – Résultats obtenus lors du calcul de 1,000,000 itérations de l'attracteur de Hénon avec $a = 1,4$, $b = 0,3$, $x_0 = 1$, et $y_0 = 0$.

Nous présentons maintenant les résultats obtenus avec CADNA. La figure IV.4 montre le nombre de chiffres significatifs exacts estimé par CADNA sur les x_i en *simple*, *double* et *quadruple* précision. Nous pouvons voir que la précision des x_i décroît à mesure que le nombre d'itérations i augmente. De même la figure IV.5 montre un comportement similaire sur les y_i. À partir d'une itération dépendant du type utilisé, le résultat devient non significatif.

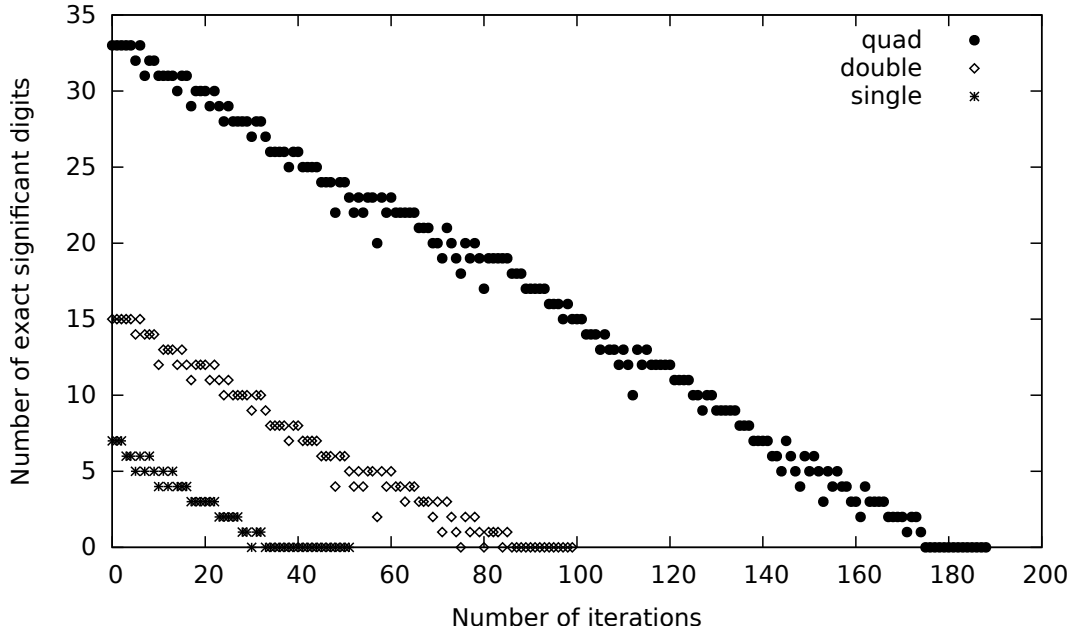


FIGURE IV.4 – Précision estimée par CADNA sur les x_i de l'attracteur de Hénon avec $a = 1,4$, $b = 0,3$, $x_0 = 1$, et $y_0 = 0$.

Le tableau IV.3 montre pour la *simple*, *double* et *quadruple* précision le nombre d'itérations effectuées avant qu'un des éléments de la suite soit non significatif. Ainsi, plus la précision est élevée, plus le nombre d'itérations est important avant que le résultat soit non significatif.

Le tableau IV.4 montre pour différentes itérations les résultats (x_i, y_i) calculés avec CADNA en *simple*, *double* et *quadruple* précision. Pour rappel, CADNA n'affiche que le nombre de chiffres significatifs exacts du résultat ou @.0 lorsque le résultat est non significatif.

IV.5.2 Calcul d'une racine multiple d'un polynôme

La méthode de Newton permet de calculer une racine d'une fonction f. À partir d'une approximation de la racine x_0, nous calculons la suite $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Le critère d'arrêt est généralement un seuil ε tel que $|x_{n+1} - x_n| < \varepsilon$ ou $\left|\frac{x_{n+1}-x_n}{x_{n+1}}\right| < \varepsilon$ si $x_{n+1} \neq 0$. L'utilisation de l'Arithmétique Stochastique Discrète permet de stopper les

type	nombre d'itérations
simple	29
double	74
quadruple	174

Table IV.3 – Nombre d'itérations avant qu'un des éléments de l'attracteur de Hénon soit non significatif avec $a = 1,4$, $b = 0,3$, $x_0 = 1$, et $y_0 = 0$.

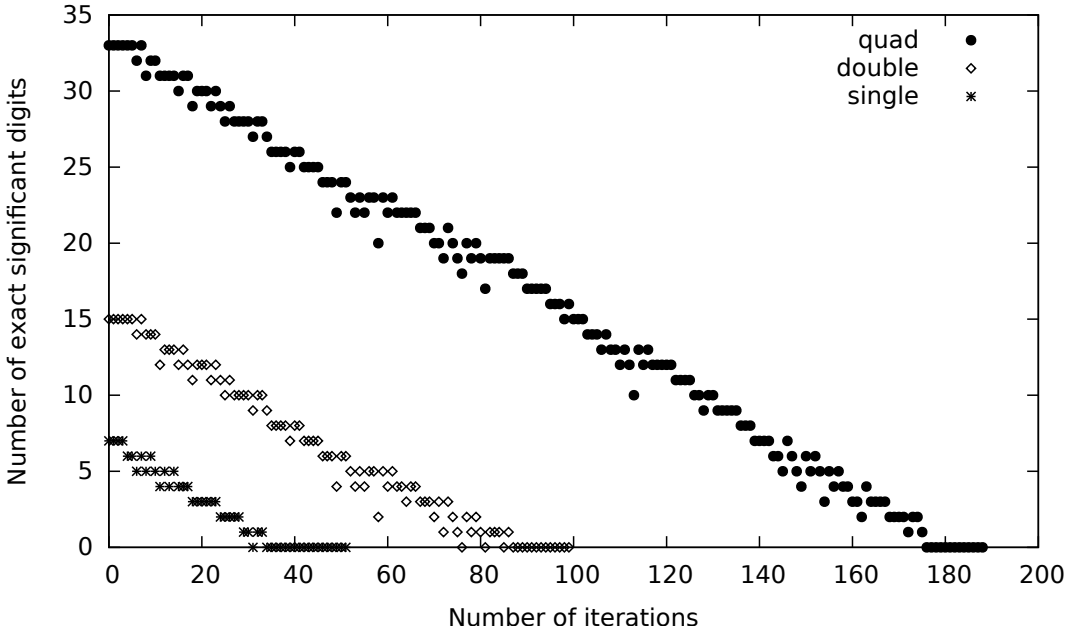


FIGURE IV.5 – Précision estimée par CADNA sur les y_i de l'attracteur de Hénon avec $a = 1,4$, $b = 0,3$, $x_0 = 1$, et $y_0 = 0$.

itérations lorsque la différence entre deux itérés successifs x_n et x_{n+1} est non significative, c'est-à-dire lorsque x_n est sotchastiquement égal à x_{n+1}. Dans ce cas, le passage de x_n à x_{n+1} est dû aux erreurs d'arrondi et augmenter le nombre d'itérations est inutile.

La qualité numérique du résultat obtenu grâce à la méthode de Newton dépend de la multiplicité de la racine. La précision d'une racine multiple en *double* précision peut être insuffisante. Il faut alors utiliser la *quadruple* précision.

Le théorème IV.5.1 établi dans [53] donne la relation entre le nombre de chiffres communs entre deux itérés successifs d'une racine multiple calculée grâce à la méthode de Newton et les chiffres en commun entre une approximation et la racine. Le théorème IV.5.1 est fondé sur la définition IV.5.1 qui explicite la notion de chiffres décimaux significatifs en commun entre deux nombres.

Définition IV.5.1. Le nombre de chiffres décimaux significatifs en commun entre deux nombres réels a et b est défini dans $\mathbb{R}$ par

— pour $a \neq b$, $C_{a,b} = \log_{10} \left| \dfrac{a+b}{2(a-b)} \right|$;

— pour tout $a \in \mathbb{R}$, $C_{a,a} = +\infty$;

avec $\oplus_{+\infty}$ et $\otimes_{+\infty}$ (resp. $\oplus_{-\infty}$ et $\otimes_{-\infty}$) les opérations flottantes en arrondi vers plus l'infini (resp. moins l'infini). Alors $|a-b| = \left|\frac{a+b}{2}\right| 10^{-C_{a,b}}$. Par exemple, si $C_{a,b} = 3$, la différence relative entre a et b est de l'ordre de 10^{-3}. Ainsi, a et b ont trois chiffres significatifs décimaux en commun.

Remarque 1. La notion de chiffres significatifs exacts qui a été introduite en page 4 (section I.1.3.a) est fondée sur la définition IV.5.1.

itération	précision	point (x_i, y_i) calculé avec CADNA	
1	*simple*	x_i	-0,3999999E+000
		y_i	0,3000000E+000
1	*double*	x_i	-0,399999999999999E+000
		y_i	0,300000000000000E+000
1	*quadruple*	x_i	-0,399999999999999999999999999999999E+000
		y_i	0,300000000000000000000000000000000E+000
30	*simple*	x_i	@.0
		y_i	0,2E+000
30	*double*	x_i	-0,13848191E+000
		y_i	0,2856319104E+000
30	*quadruple*	x_i	-0,13848191914679246248648931 2E+000
		y_i	0,28563191040030071809805899 04E+000
75	*double*	x_i	@.0
		y_i	-0,1E+000
75	*quadruple*	x_i	0,115649947336564503E+000
		y_i	-0,183998067245880684 0E+000
175	*quadruple*	x_i	@.0
		y_i	-0,2E+000

Table IV.4 – Valeurs (x_i, y_i) de l'attracteur de Hénon obtenues avec CADNA à différentes itérations en *simple*, *double* ou *quadruple* précision ($a = 1,4$, $b = 0,3$, $x_0 = 1$, et $y_0 = 0$).

Théorème IV.5.1. *Soit x_n et x_{n+1} deux itérés successifs calculés en utilisant la méthode de Newton pour approcher la racine α d'un polynôme ayant une multiplicité $m \geq 2$.*

Alors

$$C_{x_n,x_{n+1}} \sim_\infty C_{x_{n+1},\alpha} + \log_{10}(m-1).$$

Grâce au théorème IV.5.1, si la suite a convergé, alors les chiffres communs entre x_n et x_{n+1} sont aussi en commun avec la racine exacte α, à $\log_{10}(m-1)$ près. Si les itérations sont effectuées jusqu'à ce que la différence entre x_n et x_{n+1} soit non significative, alors les chiffres significatifs de x_{n+1} non affectés par les erreurs d'arrondi sont en commun avec la racine exacte α, à $\delta = \lceil \log_{10}(m-1) \rceil$ près, comme montré dans la figure IV.6.

Le tableau IV.5 montre pour les racines α_i du polynôme $P(x) = (x-1)^2(3x-1)^3$, la valeur $\delta_i = \lceil \log_{10}(m_i - 1) \rceil$, le nombre de chiffres significatifs exacts fourni par CADNA de l'approximation des racines obtenu grâce à la méthode de Newton et le nombre de chiffres significatifs en commun avec la valeur exacte de la racine selon la définition IV.5.1.

Le tableau IV.6 montre pour différents polynômes $P_m(x) = (x-1)^m$, la valeur $\delta = \lceil \log_{10}(m-1) \rceil$, le nombre de chiffres significatifs de la racine non affecté par les erreurs d'arrondi estimé par CADNA et le nombre de chiffres significatifs en commun avec la valeur exacte de la racine selon la définition IV.5.1.

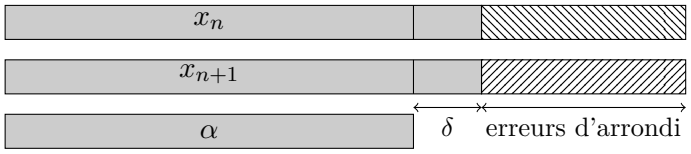


Figure IV.6 – Représentation des derniers itérés x_n et x_{n+1} de la méthode de Newton, et des premiers chiffres de la racine multiple α : si $x_n - x_{n+1}$ est non significatif, alors les chiffres de x_{n+1} non affectés par les erreurs d'arrondi sont en commun avec α, à δ près.

type	$\alpha_1 = 1$, $\delta_1 = 0$		$\alpha_2 = 1/3$, $\delta_2 = 1$	
	CADNA	exact	CADNA	exact
simple	4	3,1	3	2,2
double	7	7,3	5	5,0
quadruple	17	16,6	12	11,0

Table IV.5 – Pour chaque racine de $P(x) = (x-1)^2(3x-1)^3$, nombre de chiffres significatifs exacts estimés par CADNA et nombre de chiffres en commun avec la valeur exacte de la racine.

précision	$m = 6$, $\delta = 1$		$m = 8$, $\delta = 1$		$m = 18$, $\delta = 2$	
	CADNA	exact	CADNA	exact	CADNA	exact
simple	2	1,0	1	0,7	1	0,5
double	3	2,4	2	1,8	1	0,7
quad	7	5,5	5	4,0	3	1,6

Table IV.6 – Pour chaque racine de $P(x) = (x-1)^m$, nombre de chiffres significatifs exacts estimés par CADNA et nombre de chiffre en commun avec la valeur exacte de la racine.

Dans les tableaux IV.5 et IV.6, nous pouvons voir que si la multiplicité d'une racine augmente, la précision de l'approximation de la racine obtenue diminue. Ainsi il peut être nécessaire d'utiliser une précision supérieure à savoir la *quadruple* précision.

De plus, d'après les tableaux IV.5 et IV.6 nous pouvons dire que, lorsque $\delta = \lceil \log_{10}(m-1) \rceil$, m étant la valeur de la multiplicité de la racine, la précision estimée par CADNA est la précision exacte, à δ près dans 12 des 15 cas et dans les 3 cas restants jusqu'à $\delta+1$. Ce résultat est en accord avec le théorème IV.5.1 et le fait que la précision estimée par CADNA peut être considérée comme correcte à un chiffre près, comme indiqué dans la section I.4.1.b.

IV.5.3 PROMISE avec trois types

IV.5.3.a Description de l'algorithme

En raison de son coût d'utilisation, un développeur limitera au maximum le nombre de variables en *quadruple* précision. Cette précision sera utilisée uniquement lorsque la précision *double* ne suffit pas et uniquement sur les variables critiques. Si le développeur ne connaît pas le niveau de criticité des variables, l'outil PROMISE, décrit dans le chapitre III, permettra de l'aider à le déterminer.

L'outil PROMISE peut être amélioré dans le but de gérer trois types flottants : *simple*, *double* et *quadruple* précisions. L'ensemble C des variables d'un programme sera partitionné en trois sous-ensembles : C^s, C^d et C^q représentant respectivement les variables pouvant être déclarées en précision *simple*, *double* et *quadruple* tout en fournissant un résultat *valide* au sens de la définition III.4.2.

L'algorithme de Delta-Debug [141], sur lequel est fondé PROMISE, ne permet qu'une subdivision en deux sous-ensembles et non en trois comme nous le souhaitons ici. Notre approche est d'utiliser deux appels successifs à l'algorithme de Delta-Debug comme nous pouvons le voir dans l'algorithme récursif de PROMISE utilisant trois types (algorithme IV.15). Cet algorithme prend en entrée l'ensemble des variables d'un programme, toutes délarées en *quadruple* précision. Nous rappelons que la fonction `test` vérifie si le résultat est *valide*. La première étape consiste à dégrader un maximum de variables de la *quadruple* à la *double* précision. L'ensemble C^q des variables en *quadruple* précision, qui est le résultat de cette étape, est le sous-ensemble des variables devant rester en *quadruple* précision. Ensuite, l'ensemble C_0^d des variables pouvant être définies en *double* précision est partitionné en deux sous-ensembles grâce à un nouvel appel à l'algorithme de PROMISE qui fournit :

— les variables devant être définies en *double* précision C^d ;
— les variables pouvant être définies en *simple* précision C^s.

IV.5.3.b Résultats expérimentaux

Contexte expérimental :

Tous les tests ont été effectués sur une station de travail équipée d'un processeur Intel Xeon CPU E5-2670 à 2,6 GHz avec 128 Go de RAM. Les codes utilisés sont écrits en C++ et compilés avec GCC 4.9.2 avec une optimisation en O3. Nous utilisons la

Algorithme IV.15 Algorithme récursif de PROMISE avec trois types, l'initialisation se fait par l'appel : $PROMISE(\emptyset, \emptyset, C, 1, 1)$

$PROMISE(C^s, C^d, C^q, p, step)$=

$$\begin{cases} \textbf{1} - PROMISE(\emptyset, C^d \cup C_i^q, C^q \backslash C_i^q, \max(p-1,2), 1) \\ \quad \text{si } \exists i \in \{1, \cdots, p\} \ \texttt{test}(\emptyset, C^d \cup C_i^q, C^q \backslash C_i^q) = \checkmark \text{ et } step = 1\,; \\ \textbf{2} - PROMISE(\emptyset, C^d, C^q, \max(2p, |C^q|), 1) \\ \qquad \text{si } p < |C^q \text{ et } step = 1\,; \\ \textbf{3} - PROMISE(C^s \cup C_i^d, C^d \backslash C_i^d, C^q, \max(p-1,2), 2) \\ \qquad \text{si } \exists i \in \{1, \cdots, p\} \ \texttt{test}(C^s \cup C_i^d, C^d \backslash C_i^d, C^q) = \checkmark\,; \\ \textbf{4} - PROMISE(C^s, C^d, C^q, \min(2p, |C^d|), 2) \ \text{ si } p < |C^d|\,; \\ \textbf{5} - \text{retourne } (C^s, C^d, C^q) \qquad \text{sinon.} \end{cases}$$

version 2.0.0 de CADNA [39] à laquelle nous avons ajouté la *quadruple* précision. Nous testons PROMISE sur les codes suivants :

— multiplication de matrice (MatMul) ;
— méthode des Babyloniens pour le calcul d'une racine carrée (squareRoot) ;
— méthode des rectangles pour le calcul d'une intégrale (rectangle).

Dans ces cas tests, toutes les variables flottantes sont prises en compte pour une éventuelle optimisation. Lorsque le résultat est une matrice alors tous les éléments de la matrice doivent satisfaire le critère de précision choisi par l'utilisateur.

Résultat expérimentaux :

Le tableau IV.7 présente les résultats obtenus avec PROMISE lors d'une optimisation à trois types en utilisant la version `référence stochastique` de PROMISE, voir section III.5.1. Pour chaque cas test, nous demandons les niveaux de précision suivants : 4, 6, 8, 10, 12, 14, 16, 18 et 20 chiffres significatifs exacts. Nous reportons les résultats suivants :

— le nombre d'exécutions effectuées, dans les cas étudiés le nombre de compilations est égal au nombre d'exécutions (# exec) ;
— le nombre de variables devant rester en *quadruple* précision (# quad) ;
— le nombre de variables pouvant être passées en *double* précision (# double) ;
— le nombre de variables pouvant être passées en *simple* précision (# float) ;
— le temps d'exécution total de PROMISE ;
— le facteur d'accélération entre la version initiale et la version finale ($\frac{Temps_{initial}}{Temps_{final}}$) [9].

Dans tous les cas tests étudiés, PROMISE a pu trouver une nouvelle configuration avec des variables voyant leurs précisions dégradées tout en respectant le critère de précision souhaité.

Les facteurs d'accélération obtenus sont entre 1,09 et 5,52. Ces valeurs sont meilleures que lorsque nous optimisions avec deux types (voir tableau III.1). Cela s'explique en particulier par les coûts importants de la *quadruple* précision. Ainsi en réduisant

9. Les temps sont une moyenne de plusieurs exécutions des programmes.

Programme	# chiffres	# exec	# quad - # double - # float	Temps (mm:ss)	facteur d'accélération
MatMul	20	9	1 - 1 - 1	0:30	1,45
	18	9	1 - 1 - 1	0:30	1,45
	16	9	1 - 1 - 1	0:30	1,45
	14	8	0 - 2 - 1	0:24	3,63
	12	8	0 - 2 - 1	0:24	3,63
	10	8	0 - 2 - 1	0:24	3,63
	8	8	0 - 2 - 1	0:24	3,63
	6	8	0 - 2 - 1	0:24	3,63
	4	4	0 - 0 - 3	0:15	5,52
squareRoot	20	22	6 - 0 - 2	0:12	1,14
	18	22	6 - 0 - 2	0:12	1,14
	16	25	5 - 1 - 2	0:12	2,61
	14	22	0 - 6 - 2	0:10	2,70
	12	22	0 - 6 - 2	0:10	2,70
	10	22	0 - 6 - 2	0:10	2,70
	8	22	0 - 6 - 2	0:10	2,70
	6	4	0 - 0 - 8	0:04	2,68
	4	4	0 - 0 - 8	0:04	2,68
rectangle	20	18	6 - 1 - 0	0:11	1,09
	18	18	6 - 1 - 0	0:11	1,09
	16	20	2 - 5 - 0	0:12	1,40
	14	18	1 - 6 - 0	0:10	1,42
	12	16	0 - 7 - 0	0:10	1,42
	10	16	0 - 7 - 0	0:10	1,42
	8	12	0 - 2 - 5	0:08	1,42
	6	12	0 - 1 - 6	0:08	1,43
	4	4	0 - 0 - 7	0:04	1,44

Table IV.7 – Résultats expérimentaux de PROMISE avec trois types sur plusieurs cas tests.

son utilisation, les codes sont plus performants. Nous pouvons noter que plus nous demandons une précision faible, plus le facteur d'accélération est important.

IV.5.3.c Autres implémentations possibles

Lors du développement de PROMISE à trois types, différentes implémentations ont été pensées. Nous avons choisi celle décrite précédemment car il s'agit de la version permettant d'obtenir le moins de variables en *quadruple* précision. L'utilisation de ce type limite grandement la performance des codes et doit donc être la plus réduite possible.

Une autre approche possible aurait été de maximiser le nombre de variables en *simple* précision. Ainsi, dès qu'une variable a son type dégradé de la *quadruple* à la *double* précision, il faudrait essayer de la transformer en *simple* précision. Cela se ferait par l'exécution de deux passes de l'algorithme de Delta-Debug (DD) imbriquées. Dans la figure IV.7, nous considérons qu'il est possible d'examiner l'ensemble des variables en *double* précision obtenu après la première application du Delta-Debug. Cet ensemble sera optimisé immédiatement avec un appel à l'algorithme de Delta-Debug pour tenter de transformer les variables en *simple* précision. Ainsi cette approche cherche à obtenir rapidement des variables en *simple* précision au détriment de la réduction du nombre de variables en *quadruple* précision.

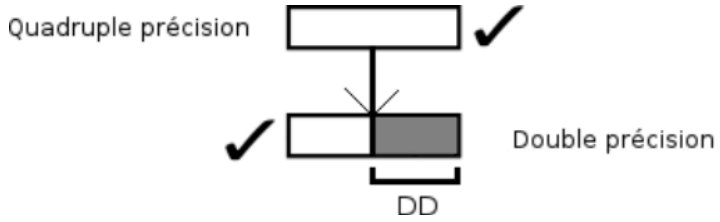


Figure IV.7 – PROMISE à trois niveaux en utilisant deux appels à l'algorithme de Delta-Debug imbriqués. Les zones blanches (resp. grises) correspondent à des variables en *quadruple* (resp. *double*) précision.

Pour maximiser le nombre de variables en *simple* précision, il est possible de changer l'ordre des optimisations. Ainsi actuellement nous proposons d'optimiser de la *quadruple* vers la *double*, puis de la *double* vers la *simple*. Cela peut être remplacé par une optimisation de la *quadruple* vers la *simple* puis de la *quadruple* vers la *double* des variables restantes. Cela devrait permettre d'obtenir plus de variables en *simple* précision que précédemment.

Néanmoins, ces possibilités ne sont pour le moment pas viables et ne pourraient le devenir qu'à partir du moment où la *quadruple* précision sera disponible au niveau matériel et non plus logiciel. En effet, il est préférable actuellement de minimiser le nombre de variables en *quadruple* précision plutôt que de maximiser le nombre de variables en *simple* précision. Si dans le futur le coût d'utilisation de la *quadruple* précision se rapproche de celui de la *double*, il pourrait être intéressant de tester ces approches. En effet, une vectorisation en *simple* précision pourrait compenser le surcoût lié à l'utilisation de la *quadruple* précision dans une partie du code.

IV.6 Conclusion

Nous avons développé une extension à la bibliothèque CADNA permettant de gérer la *quadruple* précision stochastique à partir des implémentations logicielles de la *quadruple* précision existantes. Cela nous permet de valider des codes nécessitant un type plus précis que la *double* précision. Cette validation du résultat est importante. En effet, augmenter la précision des types ne garantit pas nécessairement un résultat de meilleure qualité numérique, comme nous l'avons vu dans la section I.1.4. Deux expérimentations numériques utilisant des variables stochastiques en *quadruple* précision ont été présentées : l'attracteur de Hénon (section IV.5.1) et la recherche d'une racine multiple d'un polynôme (section IV.5.2).

De nombreuses améliorations peuvent être apportées à cette extension de la bibliothèque CADNA en *quadruple* précision. En effet, cette extension pourrait être enrichie grâce à OpenMP et MPI. Les travaux visant à permettre le contrôle avec CADNA d'un code utilisant OpenMP et MPI ont été réalisés pour la *simple* et la *double* précision [40, 100, 101] et doivent donc être étendus à l'utilisation de la *quadruple* précision. Les dernières spécifications de MPI [44] et d'OpenMP [8] n'imposent pas la gestion de la *quadruple* précision. Dans les deux cas l'utilisation d'un autre type est possible. Il est néanmoins nécessaire de l'implémenter entièrement, puis de développer sa version stochastique.

Enfin, si la bibliothèque MPFR se trouve être fortement améliorée avec l'équivalent de la *quadruple* précision, il pourrait devenir intéressant d'utiliser SAM pour valider les calculs nécessitant cette précision.

Chapitre V
Conclusion et perspectives

V.1 Conclusion

Durant cette thèse nous avons travaillé sur la validation numérique des codes de calcul et en particulier sur la précision des résultats estimée grâce à l'Arithmétique Stochastique Discrète. Des algorithmes existent pour améliorer la précision d'un résultat grâce à l'utilisation de transformations exactes. Ceux-ci permettent d'obtenir le résultat d'une opération et l'erreur commise sur ce résultat. Après avoir les avoir mis en place, il faut être capable de valider les nouveaux résultats obtenus. Malheureusement les algorithmes permettant d'obtenir plus de précision sur le résultat nécessitent un arrondi au plus près ce qui est *a priori* incompatible avec la méthode CESTAC.

Nous avons étudié des transformations exactes, de l'addition et de la multiplication, avec un arrondi dirigé. En raison de l'utilisation de cet arrondi, le terme de compensation n'est plus exact et une erreur est introduite dans le cas de la compensation de la somme. Nous avons ensuite approfondi les travaux en étudiant l'impact de l'arrondi dirigé sur différents algorithmes compensés permettant de calculer la somme des éléments d'un vecteur, un produit scalaire ou encore une évaluation polynomiale. Les bornes d'erreurs obtenues nous ont permis de montrer que la précision des résultats obtenues avec ces algorithmes est améliorée même lors de l'utilisation d'un arrondi dirigé. Nous avons alors introduit l'ensemble de ces algorithmes au sein de CADNA pour les tester expérimentalement. Nous avons ainsi obtenu un résultat similaire à celui qui aurait été obtenu avec une précision doublée ou multipliée par K en fonction des algorithmes [1]. Cette implémentation a nécessité des modifications au sein de CADNA dans le but de limiter la synchronisation inhérente à l'Arithmétique Stochastique Discrète.

Néanmoins, avoir plus de précision peut être contre-productif en raison d'une baisse de performance. De plus, l'utilisateur peut n'avoir besoin que d'un nombre limité de chiffres significatifs exacts. Partant de ce constat, il peut être intéressant de limiter la précision des résultats en utilisant de la précision mixte, c'est-à-dire avoir un mélange de variables flottantes utilisant différents types. Les outils effectuant automatiquement cette tâche nous ont paru limités. En effet, les tests effectués se basent sur des comparaisons à des résultats non validés. Nous avons donc développé l'outil PROMISE qui permet de tester automatiquement plusieurs configurations de types, grâce à un algorithme de type

1. K étant défini par l'utilisateur

"diviser pour régner", le Delta-Debug, tout en validant les résultats avec CADNA. Grâce à cet outil, nous obtenons un ensemble de variables flottantes pouvant être passées en précision inférieure. Nous avons utilisé notre outil sur plusieurs cas tests simples dans le but de valider notre approche. Cela nous a permis d'obtenir des configurations mixtes disposant de la précision souhaitée. Nous avons ensuite mis en pratique notre outil sur un code industriel simulant un cœur nucléaire : nous avons alors observé un gain en temps de calcul ainsi qu'en utilisation mémoire.

Enfin, nous avons étendu la bibliothèque CADNA avec l'ajout d'un type stochastique plus précis défini dans la nouvelle norme IEEE 754 : la *quadruple* précision. En effet, des développeurs peuvent privilégier l'utilisation d'un type plus précis à des modifications algorithmiques telles que l'utilisation d'algorithmes compensés. Certaines adaptations ont été effectuées dans le but d'utiliser au mieux ce nouveau type. Des comparaisons de performance ont été effectuées entre cette implémentation et l'Arithmétique Stochastique Discrète en multiprécision [2]. Nous avons ainsi vu l'intérêt de cette version en terme de performance. Nous avons également montré le gain de précision obtenu sur le résultat dans le cas du calcul des racines multiples d'un polynôme par la méthode de Newton ainsi que pour le calcul des coordonnées de l'attracteur de Hénon. La *quadruple* précision étant moins performante que la *double* et la *simple* précision, nous avons amélioré PROMISE pour que celui-ci puisse gérer plus de deux types flottants. Nous sommes ainsi capables d'obtenir des configurations mixtes composées de variables en *simple*, *double* et *quadruple* précision.

V.2 Perspectives

Nous avons identifié plusieurs perspectives à nos travaux.

Une limitation de la bibliothèque CADNA se trouve être la gestion des bibliothèques externes pour lesquelles les sources ne sont pas disponibles. Avec le développement de verrou, un outil de validation numérique utilisant la méthode CESTAC et gérant les bibliothèques externes, il devient possible de combiner les deux outils pour profiter des avantages de l'un et de l'autre. Ainsi CADNA permettrait la recherche des instabilités numériques et verrou perturberait les résultats produits par les bibliothèques externes. Il faut pour cela permettre un échange de données entre les deux outils et gérer une étape de synchronisation et de désynchronisation. Cette étape peut être source de pertes de performances importantes et doit être gérée de manière efficace.

Plusieurs améliorations de PROMISE sont possibles. Tout d'abord nous pourrions effectuer une analyse statique du code pour diminuer le nombre d'erreurs de compilation. En effet, une gestion commune du type de certaines variables limiterait les erreurs en raison de l'incompatibilité de type entre l'appel d'une fonction et son prototype. De plus, cela réduirait également l'ensemble des variables à tester et donc le nombre de configurations testées en moyenne.

Bien que CADNA puisse être utilisée pour contrôler les codes parallèles fondées sur MPI, PROMISE n'est pas capable d'optimiser ce genre de code. Les communications dépendent d'un type MPI prédéfini pour connaître la taille des données à transmettre.

2. http://www-pequan.lip6.fr/~jezequel/SAM/

Nous ne modifions pas le mot clé le définissant [3] avec la version actuelle de PROMISE. Il y a donc des problèmes de compatibilité entre les types MPI et les types des variables associés qui vont empêcher une optimisation efficace et même créer des décalages mémoires. L'analyse statique permettrait de gérer les types des données transmises lors des communications.

Il est également possible de penser à une amélioration de PROMISE grâce à des modifications de l'algorithme "diviser pour régner". Sans remettre en cause l'algorithme, il est possible de le paralléliser de manière évidente. En effet, nous connaissons le nombre maximum de sous-ensembles pouvant être testés avant de diminuer le nombre d'éléments dans ceux-ci. Il est donc possible de lancer plusieurs tests de configurations en parallèle. Comme à une étape plusieurs configurations peuvent fournir un ensemble de variables pouvant être passées en précision inférieure, le résultat final peut ne pas être reproductible. Néanmoins la reproductibilité peut être forcée en choisissant prioritairement certaines configurations par rapport aux autres.

Comme nous l'avons indiqué, notre extension de la bibliothèque CADNA à la *quadruple* précision n'est pas complète. Nous devons ainsi ajouter la gestion d'OpenMP et MPI. Pour cela, il est nécessaire dans les deux cas de définir des fonctions utilisées fréquemment telles que les réductions. En effet, beaucoup de fonctions sont définies pour les types flottants *simple* et *double*. Néanmoins les dernières spécifications pour OpenMP et MPI n'imposent pas la définition de ces fonctions pour la *quadruple* précision. Il faut donc définir l'ensemble de ces fonctions pour la *quadruple* précision. Une fois cela fait, nous pourrons étendre ces modifications à la *quadruple* précision stochastique comme cela a été fait pour la *simple* et la *double* précision stochastique.

Lors de nos tests de performances, nous avons pu observer que le type *double-double* pouvait être plus performant que la *quadruple* précision. Nous avons néanmoins choisi de développer l'extension de la bibliothèque CADNA avec la *quadruple* précision. Cette dernière respecte en effet la norme IEEE-754 et ne remet pas en cause les hypothèses de l'Arithmétique Stochastique Discrète. À l'opposé, les bibliothèques définissant les *double-double* ne fournissent aucun arrondi défini dans cette même norme. L'erreur commise peut être supérieure à l'erreur relative minimale du nombre. Dans un premier temps il serait possible d'expérimenter des calculs entre les types stochastiques de CADNA et les *double-double* non stochastiques. Des tests effectués avec les *double-double* et l'outil de validation verrou montrent des résultats expérimentaux encourageants pour la création d'une extension stochastique aux *double-double*, à condition de pouvoir valider l'approche grâce à la théorie.

Enfin, nous pouvons imaginer à plus long terme un outil d'optimisation capable de réécrire certaines parties d'un code dans le but d'améliorer la précision du résultat ou d'améliorer les performances. Les algorithmes itératifs peuvent effectuer un grand nombre de fois la même boucle dans le but d'obtenir un résultat ayant la précision souhaitée. Lors des premières itérations le gain de précision peut être faible, les exécuter plus rapidement grâce à l'utilisation de la précision mixte est donc une option n'altérant pas *a priori* le résultat. Il est ainsi possible de trouver certains codes effectuant un préconditionnement d'un algorithme itératif en exécutant ce même algorithme dans une précision inférieure. Le développement d'un outil pouvant gérer entièrement la

3. Il faut par exemple utiliser `MPI_DOUBLE` pour le type `double` et `MPI_FLOAT` pour le type `float`

duplication de boucles, l'optimisation des types, ainsi que le nombre d'itérations à effectuer permettrait d'obtenir des gains de performance.

Annexe A

Relation entre les valeurs de γ_n

Nous utilisons ici les notations de la section II.2. Nous nous plaçons dans le cadre de l'arithmétique à virgule flottante binaire de avec p bits de précision. Soient $\mathbf{u} = 2^{-p}$ et $\gamma_n(2\mathbf{u}) = \frac{2n\mathbf{u}}{1-2n\mathbf{u}}$. Supposons que $n\mathbf{u} < \frac{1}{2}$.

Proposition A.0.1.

$$\gamma_{n-1}(2\mathbf{u}) + 2\mathbf{u}(1 + \gamma_n(2\mathbf{u})) \leq \gamma_n(2\mathbf{u}) \tag{A.0.1}$$

Démonstration.

$$\gamma_{n-1}(2\mathbf{u}) \leq \frac{2(n-1)\mathbf{u}}{1-2n\mathbf{u}} \tag{A.0.2}$$

et

$$2\mathbf{u}(1 + \gamma_n(2\mathbf{u})) = \frac{2\mathbf{u}}{1-2n\mathbf{u}} \tag{A.0.3}$$

C'est pourquoi nous pouvons déduire des équations (A.0.2) et (A.0.3) que

$$\gamma_{n-1}(2\mathbf{u}) + 2\mathbf{u}(1 + \gamma_n(2\mathbf{u})) \leq \frac{2n\mathbf{u}}{1-2n\mathbf{u}} \tag{A.0.4}$$

□. □

Proposition A.0.2.

$$\gamma_n(2\mathbf{u}) + 2\mathbf{u}\gamma_n(2\mathbf{u}) \leq \gamma_{n+1}(2\mathbf{u}) \tag{A.0.5}$$

Démonstration. Comme $n\mathbf{u} < \frac{1}{2}$,

$$\gamma_n(2\mathbf{u}) < \frac{1}{1-2n\mathbf{u}} \tag{A.0.6}$$

C'est pourquoi

$$\gamma_n(2\mathbf{u}) + 2\mathbf{u}\gamma_n(2\mathbf{u}) < \gamma_n(2\mathbf{u}) + \frac{2\mathbf{u}}{1-2n\mathbf{u}} \tag{A.0.7}$$

et

$$\gamma_n(2\mathbf{u}) + 2\mathbf{u}\gamma_n(2\mathbf{u}) < \frac{2(n+1)\mathbf{u}}{1-2n\mathbf{u}} \tag{A.0.8}$$

Nous pouvons donc déduire l'équation (A.0.5). □

Proposition A.0.3.

$$\gamma_{n-2}(2\mathbf{u})\gamma_n(2\mathbf{u}) + 2\mathbf{u}\gamma_{n-1}(2\mathbf{u}) \leq 2\gamma_n^2(2\mathbf{u}) \tag{A.0.9}$$

Démonstration.

$$\gamma_n(2\mathbf{u}) - 2\mathbf{u} = \frac{2n\mathbf{u} - 2\mathbf{u} + 4n\mathbf{u}^2}{1 - 2n\mathbf{u}} \tag{A.0.10}$$

Comme $1 - 2n\mathbf{u} > 0$ et $2n\mathbf{u} + 4n\mathbf{u}^2 > 2\mathbf{u}$, $\gamma_n(2\mathbf{u}) - 2\mathbf{u} > 0$, alors

$$2\mathbf{u} < \gamma_n(2\mathbf{u}) \tag{A.0.11}$$

De plus, comme $\gamma_{n-1}(2\mathbf{u}) \leq \gamma_n(2\mathbf{u})$, nous pouvons déduire l'équation (A.0.9). □

Table des figures

Table des algorithmes

Liste des tableaux

Bibliographie

[1] ANDRA. *Projet CIGÉO: centre industriel de stockage réversible profond de déchets radioactifs en Meuse/Haute-Marne*. Dossier du maître d'ouvrage. (Cité page v.)

[2] H. Anzt, B. Rocker, and V. Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. *Computer Science-Research and Development*, 25(3-4):141–148, 2010. (Cité page 76.)

[3] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526 – 2533, 2009. (Cité page 76.)

[4] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. ARPREC: An arbitrary precision computation package. Technical report, Lawrence Berkeley National Laboratory, 2002. (Cité page 19.)

[5] M. Berz, J. Hoefkens, and K. Makino. COSY INFINITY Version 8.1 — Programming Manual. Technical Report MSUHEP–20703, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2001. http://cosy.pa.msu.edu. (Cité page 11.)

[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001. (Cité page vii.)

[7] M. Blair, S. Obenski, and P. Bridickas. GAO/IMTEC-92-26 Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Technical report, GAO/IMTEC, February 1992. (Cité page vi.)

[8] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.5, Novembre 2015. (Cité page 124.)

[9] S. Boldo, S. Graillat, and J.-M. Muller. On the Robustness of the 2Sum and Fast2Sum Algorithms. *ACM Trans. Math. Softw.*, 44(1):4:1–4:14, July 2017. (Cité page 40.)

[10] S. Boldo and T. Nguyen. Proofs of numerical programs when the compiler optimizes. *Innovations in Systems and Software Engineering*, 7(2):151–160, Jun 2011. (Cité page vii.)

[11] Boost. Boost C++ Libraries. http://www.boost.org/, 2016. (Cité pages 10 et 110.)

[12] J. Brajard, P. Li, F. Jézéquel, H.-S. Benavidès, and S. Thiria. Numerical Validation of Data Assimilation Codes Generated by the YAO Software. In *SIAM Annual Meeting, San Diego, California (USA)*, July 2013. (Cité page 22.)

[13] K. Briggs. The doubledouble library. http://keithbriggs.info/doubledouble.html, 1998. (Cité page 104.)

[14] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000. (Cité page 77.)

[15] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance While Achieving 64-bit Accuracy. *ACM Trans. Math. Softw.*, 34(4):17:1–17:22, July 2008. (Cité page 76.)

[16] A. Calloo, D. Couyras, F. Févotte, and M. Guillo. COCAGNE: EDF new Neutronic Core Code for ANDROMÈDE Calculation Chain. In *International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering (M&C)*, Jeju, Korea, April 2017. (Cité page vi.)

[17] J.-M. Chesneaux. *étude théorique et implémentation en ADA de la méthode CESTAC*. PhD thesis, Université Pierre et Marie Curie, 1988. (Cité pages 5 et 20.)

[18] J.-M. Chesneaux. Study of the computing accuracy by using probabilistic approach. In C. Ullrich, editor, *Contribution to Computer Arithmetic and Self-Validating Numerical Methods*, pages 19–30, IMACS, New Brunswick, New Jersey, USA, 1990. (Cité page 6.)

[19] J.-M. Chesneaux. *L'arithmétique stochastique et le logiciel CADNA*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, France, November 1995. (Cité page 21.)

[20] J.-M. Chesneaux, L.-S. Didier, and F. Rico. The Fixed CADNA library. In *Proceedings of RNC5*. UMPR7606, 2003. ANP 3-5 september LIP6. (Cité page 22.)

[21] J.-M. Chesneaux, S. Graillat, and F. Jézéquel. *Encyclopedia of Computer Science and Engineering*, volume 4, chapter Rounding Errors, pages 2480–2494. Wiley, 2009. (Cité page 35.)

[22] J.-M. Chesneaux and J. Vignes. Sur la robustesse de la méthode CESTAC. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 307:855–860, 1988. (Cité page 21.)

[23] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 300–315, New York, NY, USA, 2017. ACM. (Cité page 76.)

[24] CIGÉO. http://www.xn--cigo-dpa.com/. (Cité page v.)

[25] Code_Saturne. http://code-saturne.org/cms/, 2017. (Cité page v.)

[26] Contributors, GSL Project. GSL - GNU Scientific Library - GNU Project - Free Software Foundation (FSF). http://www.gnu.org/software/gsl/, 2010. (Cité page 87.)

[27] Contributors of Center for Manycore Programming, Seoul. SNU NPB Suite, 2010. (Cité page 89.)

[28] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, Last 2010. (Cité page v.)

[29] Electricité de France. Finite element *code_aster*, Analysis of Structures and Thermomechanics for Studies and Research. Open source on www.code-aster.org, 1989–2017. (Cité page v.)

[30] Electricité de France. *code_aster* Analyse des Structures et Thermo-mécanique pour des Études et des Recherches: plaquette de présentation. http://www.code-aster.org/UPLOAD/DOC/Presentation/plaquette_aster_fr.pdf, 2017. (Cité page v.)

[31] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. (Cité pages 11 et 78.)

[32] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971. (Cité pages vii, viii, 16, 17, 37, 47, 60, et 104.)

[33] Delta-Debugging. https://www.st.cs.uni-saarland.de/dd/. (Cité pages 23 et 87.)

[34] J. Demmel and H. D. Nguyen. Fast Reproducible Floating-Point Summation. In *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, pages 163–172, 2013. (Cité page vii.)

[35] C. Denis, P. de Oliveira Castro, and E. Petit. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, pages 55–62, 2016. (Cité page 14.)

[36] dragonegg, 2014. https://dragonegg.llvm.org/. (Cité page 23.)

[37] N. Débarbouillé. Etude des outils de validation de codes scientifiques: VERIFICARLO et CADNA. Master's thesis, Université Paris-Saclay, 2017. (Cité page 23.)

[38] J. W. Eaton et al. GNU Octave. (Cité page 18.)

[39] P. Eberhart, J. Brajard, P. Fortin, and F. Jézéquel. High performance numerical validation using stochastic arithmetic. *Reliable Computing*, 21:35–52, 2015. (Cité pages vii, viii, 22, 87, 110, et 121.)

[40] P. Eberhart, J. Brajard, P. Fortin, and F. Jézéquel. Estimation of Round-off Errors in OpenMP Codes. In *IWOMP 2016 - 12th International Workshop on OpenMP*, volume 9903 of *Lecture Notes in Computer Science*, pages 3–16, Nara, Japan, October 2016. Riken AICS, Springer International Publishing. (Cité pages 22, 23, et 124.)

[41] F. Févotte and B. Lathuilière. MICADO: Parallel Implementation of a 2D–1D Iterative Algorithm for the 3D Neutron Transport Problem in Prismatic

Geometries. In *Proceedings of Mathematics, Computational Methods & Reactor Physics*, May 2013. (Cité page 94.)

[42] F. Févotte and B. Lathuilière. VERROU: a CESTAC evaluation without recompilation. In *17th international symposium on Scientific Computing, Computer Arithmetic and Verified Numerics (SCAN 2016)*, UPPSALA, Sweden, September 2016. (Cité pages vii et 14.)

[43] F. Févotte and B. Lathuilière. Studying the Numerical Quality of an Industrial Computing Code: A Case Study on Code_aster. In *Numerical Software Verification: 10th International Workshop, NSV 2017, Heidelberg, Germany, July 22-23, 2017, Proceedings*, pages 61–80, Cham, 2017. Springer International Publishing. (Cité page 15.)

[44] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, Juin 2015. (Cité page 124.)

[45] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13:1–13:15, 2007. http://www.mpfr.org. (Cité pages 14, 19, et 99.)

[46] M. Frechtling and P. H. W. Leong. MCALIB: Measuring Sensitivity to Rounding Error with Monte Carlo Programming. *ACM Trans. Program. Lang. Syst.*, 37(2):5:1–5:25, April 2015. (Cité page 14.)

[47] Gelpia. Gelpia: Global function optimizer based on branch and bound for noncontinuous functions. https://github.com/keram88/gelpia-cs6230. (Cité page 11.)

[48] GNU Project. GCC, the GNU Compiler Collection. (Cité pages 14 et 100.)

[49] E. Goubault and S. Putot. Robustness Analysis of Finite Precision Implementations. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 50–57, New York, NY, USA, 2013. Springer-Verlag New York, Inc. (Cité page 11.)

[50] S. Graillat. Accurate simple zeros of polynomials in floating point arithmetic. *Comput. Math. Appl.*, 56(4):1114–1120, 2008. (Cité pages viii et 71.)

[51] S. Graillat. Accurate Floating Point Product and Exponentiation. *IEEE Transactions on Computers*, 58(7):994–1000, 2009. (Cité page 71.)

[52] S. Graillat. *Contribution à l'amélioration de la précision et à la validation des algorithmes numériques*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, France, December 2013. (Cité page 15.)

[53] S. Graillat, F. Jézéquel, and M. S. Ibrahim. Dynamical control of Newton's method for multiple roots of polynomials. *Reliable Computing*, 21, October 2016. (Cité pages viii et 117.)

[54] S. Graillat, F. Jézéquel, and R. Picot. Numerical validation of compensated summation algorithms with stochastic arithmetic. *Electronic Notes in Theoretical Computer Science*, 317:55–69, 2015. (Cité pages viii, 38, et 41.)

[55] S. Graillat, F. Jézéquel, and R. Picot. Numerical Validation of Compensated Algorithms with Stochastic Arithmetic. working paper or preprint, September 2016. (Cité page viii.)

[56] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière. Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic. working paper or preprint, June 2016. (Cité page viii.)

[57] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière. PROMISE: floating-point precision tuning with stochastic arithmetic. In *17th international symposium on Scientific Computing, Computer Arithmetic and Verified Numerics (SCAN 2016)*, pages 98–99, UPPSALA, Sweden, September 2016. (Cité pages viii et 22.)

[58] S. Graillat, F. Jézéquel, S. Wang, and Y. Zhu. Stochastic Arithmetic in Multiprecision. *Mathematics in Computer Science*, 5(4):359–375, 2011. (Cité pages 22 et 114.)

[59] S. Graillat, Ph. Langlois, and N. Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan J. Indust. Appl. Math.*, 2-3(26):191–214, 2009. Special issue on State of the Art in Self-Validating Numerical Computations. (Cité pages 52 et 53.)

[60] S. Graillat, N. Louvet, and Ph. Langlois. Compensated Horner Scheme. Research Report 04, Équipe de recherche DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, France, 52 avenue Paul Alduy, 66860 Perpignan cedex, France, July 2005. (Cité page 52.)

[61] T. Grandlund. GNU MP: The GNU Multiple Precision Arithmetic Library. http://gmplib.org. (Cité page 19.)

[62] Inc. Gurobi Optimization. Gurobi Optimizer Reference Manual, 2016. (Cité page 77.)

[63] M. J. Halsall. *CACTUS, a characteristics solution to the neutron transport equations in complicated geometries*. AEEW / R: AEEW. Atomic Energy Establishment, Winfrith, 1980. (Cité page 94.)

[64] Y. Hardy, K. S. Tan, and W.-H. Steeb. *Computer Algebra With Symbolic C++*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2008. (Cité page 18.)

[65] J.-M. Hervouet. *Hydrodynamics of Free Surface Flows - modelling with the finite element method*. John Wiley & Sons Ltd, 2007. (Cité page vii.)

[66] J.-M. Hervouet and R. Ata. User manual of opensource software TELEMAC-2D. Report, EDF-R&D, www.opentelemac.org, 2017. V7P2. (Cité page v.)

[67] Y. Hida, X. S Li, and D. H Bailey. Library for double-double and quad-double arithmetic. *NERSC Division, Lawrence Berkeley National Laboratory*, 2007. (Cité pages 19, 99, et 104.)

[68] N.J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002. (Cité pages 8, 35, 36, 40, 42, 46, et 52.)

[69] Renáta Hodován and Ákos Kiss. Practical Improvements to the Minimizing Delta Debugging Algorithm. In *ICSOFT-EA*, 2016. (Cité page 97.)

[70] W. Hofschuster and W. Krämer. *C-XSC 2.0 – A C++ Library for Extended Scientific Computing*, pages 15–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. (Cité page 10.)

[71] M. Hénon. A two-dimensional mapping with a strange attractor. *Comm. Math. Phys.*, 50(1):69–77, 1976. (Cité pages viii et 115.)

[72] IEEE Computer Society. *IEEE standard for binary floating-point arithmetic.* IEEE Standard 754, 1985. Note: Standard 754–1985. (Cité page 2.)

[73] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic.* IEEE Standard 754-2008, August 2008. (Cité pages 2, 35, 66, 75, 79, 97, 99, 100, et 104.)

[74] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic.* IEEE Standard 1788-2015, August 2015. (Cité page 9.)

[75] Intel. *Intel C++ Compiler XE User and Reference Guides.* (Cité page 14.)

[76] F. Jézéquel, J.-M. Chesneaux, and J.-L. Lamotte. A new version of the CADNA library for estimating round-off error propagation in Fortran programs. *Computer Physics Communications*, 181(11):1927–1928, 2010. (Cité pages vii, viii, et 22.)

[77] F. Jézéquel and J.-L. Lamotte. Numerical validation of Slater integrals computation on GPU. In *The 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN'10)*, pages 78–79, Lyon, France, September 2010. (Cité pages 22 et 23.)

[78] F. Jézéquel, J.-L. Lamotte, and O. Chubach. Parallelization of Discrete Stochastic Arithmetic on multicore architectures. In *10th International Conference on Information Technology: New Generations (ITNG), Las Vegas, Nevada (USA)*, April 2013. (Cité page 22.)

[79] F. Jézéquel, J.-L. Lamotte, and I. Said. Estimation of numerical reproducibility on CPU and GPU. In *8th Workshop on Computer Aspects of Numerical Algorithms (CANA), Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 687–692, September 2015. (Cité pages 22 et 23.)

[80] F. Jézéquel, F. Rico, J.-M. Chesneaux, and M. Charikhi. Reliable computation of a multiple integral involved in the neutron star theory. *Math. Comput. Simulation*, 71(1):44–61, 2006. (Cité page 22.)

[81] H. Jiang, S. Graillat, and R. Barrio. Accurate and Fast Evaluation of Elementary Symmetric Functions. In *Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, TX, USA, April 7-10*, pages 183–190, 2013. (Cité page 71.)

[82] H. Jiang, S. Graillat, C. Hu, S. Lia, X. Liao, L. Cheng, and F. Su. Accurate evaluation of the k-th derivative of a polynomial. *J. Comput. Appl. Math.*, 191:28–47, 2013. (Cité pages viii et 71.)

[83] M. Joldes, J.-M. Muller, and V. Popescu. Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. *ACM Trans. Math. Softw.*, 44(2):15res:1–15res:27, October 2017. (Cité pages 19 et 104.)

[84] A. H. Karp and P. Markstein. High-precision Division and Square Root. *ACM Trans. Math. Softw.*, 23(4):561–589, December 1997. (Cité page 107.)

[85] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. (Cité pages vii, viii, 16, 37, 47, 57, 60, et 104.)

[86] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre. Automatically Adapting Programs for Mixed-precision Floating-point Computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 369–378, New York, NY, USA, 2013. ACM. (Cité page 77.)

[87] J.-L. Lamotte, J.-M. Chesneaux, and F. Jézéquel. CADNA_C: A version of CADNA for use with C or C++ programs. *Computer Physics Communications*, 181(11):1925–1926, 2010. (Cité page 22.)

[88] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. (Cité pages 14 et 78.)

[89] W. Li. *Numerical accuracy analysis in simulations on hybrid high-performance computing systems*. Phd thesis, Stuttgart University, Germany, July 2012. (Cité pages 21 et 85.)

[90] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. Design, Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002. (Cité page 76.)

[91] N. Louvet. *Algorithmes compensés en arithmétique flottante : précision, validation, performances*. PhD thesis, Université de Perpignan Via Domitia, nov 2007. (Cité page 71.)

[92] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. (Cité page 77.)

[93] K. Makino and M. Berz. Remainder Differential Algebras and their Applications. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 63–74. SIAM, Philadelphia, PA, 1996. (Cité page 10.)

[94] K. Makino and M. Berz. COSY INFINITY Version 9. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 558(1):346 – 350, 2006. Proceedings of the 8th International Computational Accelerator Physics Conference ICAP 20048th International Computational Accelerator Physics Conference. (Cité page 11.)

[95] M. Martel. Floating-Point Format Inference in Mixed-Precision. In C. Barrett, M. Davies, and T. Kahsai, editors, *NASA Formal Methods*, pages 230–246, Cham, 2017. Springer International Publishing. (Cité page 77.)

[96] MATLAB. *Symbolic Math Toolbox*. The MathWorks Inc., Natick, Massachusetts, 2017. (Cité page 18.)

[97] Maxima. Maxima, a Computer Algebra System. Version 5.34.1, 2014. (Cité page 11.)

[98] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017. (Cité page 18.)

[99] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218(5136):19–22, April 1968. (Cité page 81.)

[100] M. Montagnac. *Contrôle dynamique d'algorithmes itératifs de résolution de systèmes linéaires*. PhD thesis, Université Pierre et Marie Curie (Paris VI), October 1999. (Cité pages 22, 23, et 124.)

[101] S. Montan. *Sur la validation numérique des codes de calcul industriels*. Theses, Université Pierre et Marie Curie - Paris VI, October 2013. (Cité pages vii, 22, 23, et 124.)

[102] S. Montan, J.-M. Chesneaux, C. Denis, and J.-L. Lamotte. Towards an efficient implementation of CADNA in the BLAS : Example of DgemmCADNA routine. In *15th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN)*, Novosibirsk, Russia, September 2012. (Cité page vii.)

[103] S. Montan and C. Denis. Numerical Verification of Industrial Numerical Codes. In *ESAIM: Proc.*, volume 35, pages 107–113, March 2012. (Cité page vii.)

[104] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009. (Cité page 9.)

[105] S. Moustafa, F. Févotte, B. Lathuilière, and L. Plagne. Vectorization of a 2D-1D Iterative Algorithm for the 3D Neutron Transport Problem in Prismatic Geometries. In *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)*, 2013. (Cité page 96.)

[106] J.-M. Muller. *Arithmétique des Ordinateurs*. Masson, 1989. (Cité page 27.)

[107] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston, 2010. (Cité pages vii, viii, 18, et 104.)

[108] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. (Cité pages vii, 14, et 23.)

[109] C. Nguyen, C. Rubio-González, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-Point Precision Tuning Using Blame Analysis. Technical report, LBNL TR, 2015. (Cité page 78.)

[110] R. Nheili. *How to improve the numerical reproducibility of hydrodynamics simulations: analysis and solutions for one open-source HPC software*. Theses, Université de Perpignan Via Domita, December 2016. (Cité page vii.)

[111] R. Nheili, Ph. Langlois, and C. Denis. Numerical Reproducibility in open TELEMAC: A Case Study within the Tomawac Library. In *2nd International Workshop on High Performance Computing Simulation in Energy/Transport Domains (HPCSET 2015), ISC High Performance 2015 Conference.*, Frankfurt, Germany, July 2015. (Cité page vii.)

[112] R. Nheili, Ph. Langlois, and C. Denis. First improvements toward a reproducible Telemac-2D. In *XXIIIrd TELEMAC-MASCARET User Conference* , Paris, France, October 2016. (Cité page vii.)

[113] Nvidia. *Nvidia Tesla p100 datasheet*, 2016. http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf. (Cité page 76.)

[114] T. Ogita, S. M. Rump, and S. Oishi. Accurate Sum And Dot Product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005. (Cité pages vii, 38, 44, 47, 57, 58, 60, 61, 66, 67, 69, et 70.)

[115] OpenPOWER foundation. *Power ISA Version 3.0*, 2015. (Cité pages 75 et 99.)

[116] P. Guérin and R. Jonchière. Indicateurs de visibilité dans MODERATO. Présentation Scientifique et Technique I23, 2 2017. (Cité page v.)

[117] D. S. Parker. Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic. Technical report, University of California, Los Angeles, 1997. Tech. Rep. CSD-970002. (Cité pages 12 et 13.)

[118] M. Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, 19:400–406, 1972. (Cité page 37.)

[119] R. Picot. Conception et Développement d'un Outil de Post-Traitement de la Bibliothèque CADNA. Master's thesis, Polytech Paris-UPMC, 2014. (Cité page vii.)

[120] D.M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, Mathematics Department, University of California, Berkeley, CA, USA, November 1992. ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z. (Cité pages vii, viii, 16, 19, 37, 71, et 104.)

[121] N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. *The Journal of Logic and Algebraic Programming*, 64(1):135 – 154, 2005. (Cité page 11.)

[122] N. Revol and F. Rouillier. *MPFI (Multiple Precision Floating-point Interval library)*, 2009. Available at http://gforge.inria.fr/projects/mpfi. (Cité page 19.)

[123] A. Ribes and C. Caremoli. Salome Platform Component Model for Numerical Simulation. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 553–564, Washington, DC, USA, 2007. IEEE Computer Society. (Cité page vi.)

[124] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on*

High Performance Computing, Networking, Storage and Analysis, SC'13, pages 27:1–27:12, New York, NY, USA, 2013. ACM. (Cité pages 78, 89, 93, et 94.)

[125] S.M. Rump. Algorithms for Verified Inclusions - Theory and Practice. In Ramon E. Moore, editor, *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, pages 109–126. Academic Press, Boston, 1988. (Cité pages 6, 24, et 78.)

[126] S.M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. http://www.ti3.tuhh.de/rump/. (Cité page 10.)

[127] S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing*, 31(2):1269–1302, 2008. (Cité page 19.)

[128] N.S. Scott, F. Jézéquel, C. Denis, and J.-M. Chesneaux. Numerical 'health check' for scientific codes: the CADNA approach. *Computer Physics Communications*, 176(8):507–521, April 2007. (Cité page 22.)

[129] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997. (Cité page 19.)

[130] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods: 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 532–550, Cham, 2015. Springer International Publishing. (Cité pages 11 et 77.)

[131] SPARC. *SPARC V8 datasheet*, 1992. https://web.archive.org/web/20050204100221/http://www.sparc.org/standards/V8.pdf. (Cité pages 75 et 99.)

[132] SPARC. *SPARC V9 datasheet*, 1994. https://web.archive.org/web/20110728044139/http://www.sparc.org/standards/SPARCV9.pdf. (Cité pages 75 et 99.)

[133] P.H. Sterbenz. *Floating-point computation*. Prentice-Hall series in automatic computation. Prentice-Hall, 1973. (Cité pages 15 et 39.)

[134] Open TELEMAC-MASCARET v7.0, Release notes, 2014. V7P2. (Cité page vii.)

[135] F. Tisseur. Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 22(4):1038–1057, 2001. (Cité page viii.)

[136] J. Vignes. Zéro mathématique et zéro informatique. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 303:997–1000, 1986. also: La Vie des Sciences, 4 (1) 1-13, 1987. (Cité page 21.)

[137] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and Computers in Simulation*, 35:233–261, 1993. (Cité page 21.)

[138] J. Vignes. Discrete Stochastic Arithmetic for Validating Results of Numerical Software. *Numerical Algorithms*, 37(1–4):377–390, December 2004. (Cité pages vii et 79.)

[139] J. Vignes and M. La Porte. Error analysis in computing. In *Information Processing 1974*, pages 610–614. North-Holland, 1974. (Cité page 12.)

[140] J. Weidendorfer. Kcachegrind home page, 1998. https://kcachegrind.github.io/html/Home.html. (Cité page vii.)

[141] A. Zeller. *Why Programs Fail.* Morgan Kaufmann, Boston, second edition, 2009. (Cité pages 15, 78, 79, 80, 93, et 120.)

[142] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002. (Cité pages 78, 79, et 93.)

Romain PICOT

Sujet : Amélioration de la fiabilité numérique de codes de calcul industriels

Résumé : De nombreux travaux sont consacrés à la performance des simulations numériques, or il est important de tenir compte aussi de l'impact des erreurs d'arrondi sur les résultats produits. Ces erreurs d'arrondi peuvent être estimées grâce à l'Arithmétique Stochastique Discrète (ASD), implantée dans la bibliothèque CADNA.

Les algorithmes compensés permettent d'améliorer la précision des résultats, sans changer le type numérique utilisé. Ils ont été conçus pour être généralement exécutés en arrondi au plus près. Nous avons établi des bornes d'erreur pour ces algorithmes en arrondi dirigé et montré qu'ils peuvent être utilisés avec succès avec le mode d'arrondi aléatoire de l'ASD.

Nous avons aussi étudié l'impact d'une précision cible des résultats sur les types numériques des différentes variables. Nous avons développé l'outil PROMISE qui effectue automatiquement ces modifications de types tout en validant les résultats grâce à l'ASD. L'outil PROMISE a ainsi fourni de nouvelles configurations de types mêlant simple et double précision dans divers programmes numériques et en particulier dans le code MICADO développé à EDF.

Nous avons montré comment estimer avec l'ASD les erreurs d'arrondi générées en quadruple précision. Nous avons proposé une version de CADNA qui intègre la quadruple précision et qui nous a permis notamment de valider le calcul de racines multiples de polynômes. Enfin nous avons utilisé cette nouvelle version de CADNA dans l'outil PROMISE afin qu'il puisse fournir des configurations à trois types (simple, double et quadruple précision).

Mots clés : algorithmes compensés; Arithmétique Stochastique Discrète; bibliothèque CADNA; erreur d'arrondi; optimisation des types; validation numérique

Subject: Improved numerical reliability of industrial software

Abstract: Many studies are devoted to performance of numerical simulations. However it is also important to take into account the impact of rounding errors on the results produced. These rounding errors can be estimated with Discrete Stochastic Arithmetic (DSA), implemented in the CADNA library.

Compensated algorithms improve the accuracy of results, without changing the numerical types used. They have been designed to be generally executed with rounding to nearest. We have established error bounds for these algorithms with directed rounding and shown that they can be used successfully with the random rounding mode of DSA.

We have also studied the impact of a target precision of the results on the numerical types of the different variables. We have developed the PROMISE tool which automatically performs these type changes while validating the results thanks to DSA. The PROMISE tool has thus provided new configurations of types combining single and double precision in various programs and in particular in the MICADO code developed at EDF.

We have shown how to estimate with DSA rounding errors generated in quadruple precision. We have proposed a version of CADNA that integrates quadruple precision and that allowed us in particular to validate the computation of multiple roots of polynomials. Finally we have used this new version of CADNA in the PROMISE tool so that it can provide configurations with three types (single, double and quadruple precision).

Keywords: CADNA library; compensated algorithms; Discrete Stochastic Arithmetic; numerical validation; precision tuning; rounding error