

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITÉ DE 8 MAI 1945 - GUELMA -
FACULTÉ DES MATHÉMATIQUES, D'INFORMATIQUE ET DES SCIENCES DE LA MATIÈRE

Département d'Informatique



Mémoire de Fin d'études Master

Filière : Informatique

Option : Systèmes Informatiques

Thème _____

**Étude comparative des algorithmes de sommation
flottante**

Encadré Par :

DR. CHOHRRA Chemseddine

Présenté par :

BOUCHBOUT Yehya

Octobre 2020

Remerciements

Alhamdoulillah qui m'a facilité mon périple, et qui m'a fait sortir des ténèbres de l'ignorance à la lumière de la science et de la connaissance. Alhamdoulillah que j'ai pu finalement arriver ici, et que j'ai pu aboutir à un de mes objectifs dans la vie. Arriver ici n'était pas vraiment facile pour moi, mais grâce à l'aide des bonnes personnes que j'ai la chance de les avoir dans ma vie, je suis finalement ici.

Je commence par remercier sincèrement mon encadreur Dr. Chemseddine CHOHRA pour ses efforts géants. Ce jeune docteur me motivait, m'encourageait, et même influençait ma personne avec ses pensées et ses principes. Je lui remercie infiniment.

Mon encadreur n'était pas le seul à m'aider dans mon parcours, j'avais de la chance d'avoir des enseignants respectés, certains d'entre eux m'ont même donné plus que la simple information, leurs conseils étaient éclairants à plusieurs stages de mon parcours, et je suis certain que ces conseils vont encore m'aider dans ma vie poste-universitaire. Je tiens à leur remercier nominalement.

- *Dr Adel BENAMIRA*
- *Dr Nadjette BENHAMIDA*
- *Dr Karima BENHAMZA*
- *Dr Amina BENZERARA*
- *Dr Nabil BERRAHOUMA*
- *Dr Douadi BOUROUEIH*
- *Dr Lynda DJEKHDJAKHA*
- *M^{lle} Asma FRIHI*
- *Pr Salim HADDADI*
- *Dr Mourad HADJERIS*
- *Dr Samir HALLACI*
- *Dr Ali KHEBIZI*
- *Dr Mohamed Nadjib KOUAHLA*

Je tiens à remercier aussi le staff administratif : Soraya, Najdwa, Radja, et Mounira, pour tous qu'ils font pour garder la forteresse du département. Je remercie aussi le ravissant Radouan qui assure la sécurité de notre établissement.

Le meilleur pour la fin ! Ma mère la raison de mon existence, ma flamme inextinguible, ma source qui ne draine jamais. Je lui dédie mon succès et tous mes accomplissements. Soyez fière.

RÉSUMÉ

La sommation d'un vecteur de nombres flottants est un problème simple et très répandu dans les applications de calcul scientifique. À cause des opérations d'arrondi, des erreurs sont commises après chaque opération flottante. Par conséquent certains propriétés importantes de la somme comme l'associativité sont perdues. En pratique, la sommation d'un vecteur de nombres flottants rencontre principalement deux problèmes majeurs : la précision, qui dépend de la taille du vecteur, et du conditionnement de la somme. La reproductibilité, qui dépend de l'ordre des opérations d'accumulation. Plusieurs solutions ont été proposées pour répondre à ces deux problèmes. Dans ce mémoire, nous allons présenter et faire une étude comparative sur certaines solutions, et conclure sur quelle solution doit être utilisée selon la nature et le contexte du problème à résoudre.

ABSTRACT

Floating-point summation is a simple problem, that is widely used by different numerical scientific applications. Due to rounding operations, errors are committed after each floating-point operation. Subsequently, some important properties of the sum such as the associativity are lost. In practice, the summation of a floating-point vector encounters two major problems : accuracy, which depends on the size of the vector, and its conditioning number. Reproducibility, which depends on the order of the accumulation operations. Many solutions have been introduced to solve these two problems. In this thesis, we are going to present and compare some of the proposed solutions, and conclude with which algorithm should be used depending on the nature and the context of the problem.

TABLE DES MATIÈRES

Liste des figures		ix
Liste des tableaux		x
1 L'arithmétique flottante dans le standard IEEE-754		1
1.1 Introduction		1
1.2 Définition du standard IEEE-754		1
1.3 Les nombres flottants dans le standard		2
1.3.1 Le format binaire à double précision		3
Codage du format <i>binary64</i>		3
1.3.2 UFP, ULP, et LNB		5
ufp		5
ulp		6
lnb		7
1.4 Les modes d'arrondi		8
1.4.1 L'arrondi correct et l'arrondi fidèle		9
1.5 Les opérations flottantes		9
1.6 Les erreurs sur les nombres flottants		10
1.6.1 Les erreurs d'arrondi		10

	L'erreur absolue	10
	L'erreur relative	11
1.6.2	L'élimination	12
1.7	Les transformations sans erreurs pour l'addition	13
1.7.1	L'algorithme Fast2Sum	13
1.7.2	L'algorithme 2Sum	14
1.8	Conclusion	15
2	La sommation flottante : Problèmes & solutions	16
2.1	Introduction	16
2.2	Les problèmes de la sommation flottante	17
	Problèmes du conditionnement, de la majoration, et de la précision	17
2.2.1	Problème de la reproductibilité	18
2.3	Des solutions proposées	20
2.3.1	Solutions pour la précision	20
	Les algorithmes de distillation	20
	L'algorithme SumK	20
	L'algorithme iFastSum	22
	L'algorithme HybridSum	25
2.3.2	Solutions pour la reproductibilité	26
	L'algorithme ReprodSum	26
2.4	Conclusion	27
3	Outils & Méthodologie	29
3.1	Introduction	29
3.2	Les outils de développement	30
3.2.1	Le langage C	30
	L'implémentation du standard dans le langage C	30
	Les types flottants	30

La bibliothèque math.h	31
Les exceptions dans le calcul flottant	32
3.2.2 Matlab	33
3.2.3 Makefiles	34
3.2.4 Ressources matérielles et compilation	35
3.3 Méthodologie	36
3.3.1 Génération des données	36
3.3.2 Tests et génération des résultats	38
Les tests de la précision	39
Les tests de la reproductibilité	39
Les tests des performances	39
3.4 Implémentation	40
3.5 Conclusion	42
4 Résultats & Interprétation	43
4.1 Introduction	43
4.2 Résultats de précision	43
4.3 Reproductibilité numérique	48
4.4 Performances	51
4.5 Conclusion	54

TABLE DES FIGURES

1.1 <i>ulp</i> , <i>ufp</i> et <i>lnb</i> d'un nombre flottant (seulement la mantisse est représentée)	7
2.1 Non-reproductibilité de la somme	19
2.2 Découpage d'un élément dans l'algorithme <i>ExtractVector</i>	27
3.1 Le processus de génération des résultats	41
4.1 L'évolution de l'erreur relative par rapport au conditionnement et la valeur de k pour un vecteur de taille 10^3	44
4.2 L'évolution de l'erreur relative par rapport au conditionnement et la valeur de k pour un vecteur de taille 10^6	45
4.3 L'évolution de l'erreur relative par rapport au conditionnement et la valeur de k pour un vecteur de taille 10^3	46
4.4 L'évolution de l'erreur relative par rapport à la taille du vecteur et pour un $k = 2$ et un conditionnement fixé	47
4.5 L'évolution de la différence entre les résultats des deux sommations par rapport au conditionnement du vecteur pour un $k = 2$ et une taille fixe	49

4.6	L'évolution de la différence entre les résultats des deux sommations par rapport au conditionnement du vecteur et à la valeur de k pour un vecteur de taille 10^3	50
4.7	L'évolution du temps d'exécution normalisé par rapport à la taille du vecteur et pour un <i>conditionnement</i> = 8 et pour des différentes valeurs de k	52
4.8	L'évolution du temps d'exécution normalisé par rapport à la taille du vecteur et pour un $k = 4$ et pour des conditionnements fixes	53

LISTE DES TABLEAUX

1.1	Format double IEEE-754	6
1.2	Propriétés du format double	6
3.1	Des propriétés des différents formats flottants dans le C	31
3.2	Quelques fonctions de la bibliothèque math.h	31
3.3	Les propriétés de la machine utilisée pour les tests	35

INTRODUCTION GÉNÉRALE

Le calcul numérique constitue aujourd'hui une partie vitale pour l'infrastructure scientifique. Presque tous les calculs scientifiques utilisent l'arithmétique flottante, et presque tous les ordinateurs mettent en oeuvre le standard IEEE-754 de la virgule flottante, publié en 1985. Ce standard est considéré comme l'un des plus importants, si ce n'est pas le plus important dans l'industrie informatique, puisqu'il est le résultat d'une coopération sans précédente entre les académiciens et les fabricants. Parmi les opérations basiques du calcul scientifique détaillées dans le standard est la sommation flottante, qui est une opération plus compliquée par rapport à la sommation normale, à cause de la nature sophistiquée des nombres flottants. La puissance de calcul des ordinateurs actuels est très immense, ce qu'il augmente les performances en terme de calcul flottant exponentiellement. Mais malheureusement, le nombre des erreurs générées par ces opérations augmente aussi. C'est pour cela, rendre des résultats exactes et reproductibles devient de plus en plus un défi. Ces deux propriétés sont indispensables pour certains domaines de calculs scientifiques, et leur perte peut être catastrophique. Par exemple, les simulations à large échelle est une application exécutable en parallèle, elle nécessite toujours plus de précision, et une reproductibilité de ses expérimentations [5, 20, 10]. La moindre erreur soit de précision, soit de reproductibilité peut être dans certaines situations fatale. En février 1985, durant la guerre du Golfe, les défenses contre-aérienne américaines ont échoué à intercepter un missile

irakien, tuant 28 soldats et blessant une centaine d'autres. L'erreur est due au manque de la précision nécessaire pour estimer le temps de l'activation des défenses. La précision des résultats dépend directement de la taille et du conditionnement du problème à résoudre. Alors que les erreurs reliées à la reproductibilité numérique sont principalement due à la la parallélisation de la sommation, ou au changement de l'architecture matérielle en utilisation (qui engendre généralement un changement dans l'ordre des opérations d'accumulations). La résolution de ces problèmes est tellement nécessaire pour l'avancée de l'arithmétique flottante et le calcul numérique en général, qu'elle est devenue un axe de recherche intéressant qui attire beaucoup de scientifiques et d'experts. Plusieurs chercheurs ont essayé d'innover des méthodes et des algorithmes capables de surmonter les obstacles qui gênent le progrès de la sommation flottante à une opération exacte et reproductible, telle que nécessite la science. Parmi les solutions proposées qui surpassent la solution itérative classique, nous avons choisi quatre que nous allons détailler tout au long du mémoire. SumK, un algorithme itératif, basé sur les transformations sans erreurs, afin de garder le maximum des erreurs générées par chaque opération d'accumulation, son paramètre k spécifie le nombre de distillations que l'algorithme va appliquer sur les données. iFastSum, nous le considérons comme une version plus intelligente de Sumk, puisque il fait le même processus, mais il ne prend pas le nombre de distillations comme entrée, il choisit intelligemment la valeur de k pour que ses résultats soient correctement arrondis. HybrdiSum un algorithme qui accumule avec précision les entrées ayant le même exposant dans un même accumulateur, puis il les additionne en utilisant un algorithme de sommation itératif. ReprodSum, qui fournit les mêmes résultats peu importe l'ordre des opérations, en considérant les données comme deux parties (haute et basse). Dans notre travail, nous allons comparer ces algorithmes en terme de leur précision, reproductibilité, et performance, par rapport à la solution itérative classique. Plusieurs testes auront lieu pour avoir une base solide pour notre comparaison.

Le mémoire est organisé comme suit :

— Chapitre 01 : L'arithmétique flottante dans le standard IEEE-754

- Chapitre 02 : La sommation flottante : Problèmes & solutions
- Chapitre 03 : Outils & Méthodologie
- Chapitre 04 : Résultats & Interprétation

CHAPITRE 1

L'ARITHMÉTIQUE FLOTTANTE DANS LE STANDARD

IEEE-754

1.1 Introduction

Les nombres flottants sont utilisés par les machines actuelles pour donner une approximation aux nombres réels. Dans ce chapitre nous allons mettre de la lumière sur le fameux standard IEEE-754, et découvrir comment il a défini les nombres flottants, ainsi que leurs propriétés les plus importantes. Nous allons ensuite discuter les erreurs qui peuvent se produire lors des opérations flottantes. Et à la fin nous allons aborder un concept très important qui est celui des transformations sans erreurs.

1.2 Définition du standard IEEE-754

Le standard IEEE pour l'arithmétique à virgule flottante (IEEE-754) est un standard technique pour les calculs à virgule flottante, qui a été établi en 1985 par l'institut des ingénieurs électriciens et électroniciens (IEEE)[6]. Ce standard est venu pour résoudre plusieurs problèmes que rencontraient les machines de l'époque, parmi lesquelles le manque de la fiabilité et celui de la portabilité. Aujourd'hui ce standard

représente la base sur laquelle appuient les machines modernes (PC basés sur Intel, Mac, Unix, ... etc) pour représenter les nombres réels. Le standard fixe trois exigences principales :

- * Une représentation consistante des nombres flottants par toutes les machines adoptantes le standard.
- * Des opérations flottantes correctement arrondies, en utilisant les divers modes d'arrondi.
- * Un traitement consistant des cas exceptionnels, tel que la division par zéro.

1.3 Les nombres flottants dans le standard

Le standard [6] a introduit une façon efficace pour représenter les nombres flottants, en les considérant comme un regroupement de trois parties principales :

- * Le signe, qui est un entier positif, qui peut prendre soit la valeur 0 (pour indiquer qu'il s'agit d'un nombre positif), soit la valeur 1 (pour indiquer qu'il s'agit d'un nombre négatif).
- * L'exposant biaisé, qui est un entier définissant le décalage de la virgule sur la mantisse d'un nombre flottant. Grâce à cet exposant dit *biaisé*, nous sommes capables de coder les exposants positifs ainsi que ceux négatifs.
- * La mantisse normalisée, qui est une chaîne binaire qui représente la partie significative des nombres flottants.

Signe	Exposant	Mantisse
-------	----------	----------

Le signe est représenté en un seul bit, pourtant la taille de l'exposant et la mantisse diffère d'un format à l'autre. Chaque format est caractérisé par une base, une précision P , et un intervalle pour l'exposant, plus un biais d'exposant. Le standard en sa version récente introduit cinq formats :

- * Trois formats binaires, avec une longueur de 32, 64, et 128 bits.
- * Deux formats décimaux, avec une longueur de 64, 128 bits.

1.3.1 Le format binaire à double précision

Dans ce mémoire, nous nous intéressons aux formats binaires seulement, donc pour la suite de ce mémoire, nous considérons que la base $\beta = 2$ toujours. En particulier nous allons utiliser le format *binary64* (aussi connu comme : format double). Ce format est (comme son nom l'indique) codé sur 64 bits avec :

- * 1 bit de signe.
- * 11 bits d'exposant.
- * 52 bits de mantisse.

Codage du format *binary64*

Pour coder un nombre réel X au format *binary64* il faut suivre les étapes suivantes :

1. Convertir le nombre en binaire.
2. Écrire le nombre binaire obtenu en notation scientifique en base 2. Si $X \neq 0$, on obtient un résultat sous la forme :

$$X = \pm b_0, b_1 b_2 b_3 b_4 \dots \times 2^e$$

avec $b_0 = 1$, et pour $i \geq 1$, $b_i \in \{0, 1\}$.

3. Coder le signe, la mantisse et l'exposant de la notation scientifique.

Le signe est la partie la plus facile à coder. Comme nous l'avons expliqué dans la section précédente, pour coder un nombre positif, nous utilisons 0 comme bit de signe, alors que pour coder un nombre négatif nous utilisons 1 comme bit de signe.

L'exposant par contre demande plus d'étapes pour être codé. Le standard IEEE-754 définit pour chaque format ce qu'on appelle un *biais d'exposant* qu'on va noter avec le mot *bias* par la suite. Pour une taille d'exposant a , le biais d'exposant est calculé par la formule :

$$bias = 2^{(a-1)} - 1. \tag{1.1}$$

Donc pour le format *binary64*, nous avons :

$$bias = 2^{(11-1)} - 1 = 1023. \quad (1.2)$$

Étant donné un exposant e à coder, nous devons d'abord calculer $E = e + bias$. Puis écrire l'entier E (majuscule) en binaire pour obtenir le code de e .

Finalement, pour coder la mantisse, il aurait suffi d'écrire le code binaire obtenu pour la mantisse en notation scientifique. Mais en pratique (pour $X \neq 0$), ce n'est pas nécessaire de coder le premier bit de la mantisse. parce qu'on sait que implicitement $b_0 = 1$ (appelé bit de normalisation). Donc les 52 bits disponible en format *binary64* nous permettent de coder des mantisses de taille 53 bits (52 bits codés + 1 bit de normalisation implicite).

Exemple :

Dans cet exemple, nous allons suivre la procédure de codage du nombre $X = -523.25$.

1. Nous allons d'abord écrire X en binaire. Nous avons

$$X = -(523.25)_{10} = -(1000001011.01)_2$$

2. Puis nous écrivons X en se basant sur la notation scientifique en base 2, nous obtenons :

$$X = -(1.00000101101)_2 \times 2^9$$

3. Finalement, nous allons coder le signe (négatif), l'exposant (9), et la mantisse (1.00000101101).

— Codage de signe : pour un nombre négatif, nous utilisons la valeur 1 pour le signe.

Exposant codé (E)	La valeur numérique représentée
$(0000000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3 \dots b_{52})_2 \times 2^{-1022}$ (dénormalisé)
$(0000000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^{-1022}$
$(0000000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^{-1021}$
$(0000000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^{-1020}$
↓	↓
$(0111111111)_2 = (1023)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^0$
$(1000000000)_2 = (1024)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^1$
↓	↓
$(1111111100)_2 = (2044)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^{1021}$
$(1111111101)_2 = (2045)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^{1022}$
$(1111111110)_2 = (2046)_{10}$	$\pm(1.b_1b_2b_3 \dots b_{52})_2 \times 2^{1023}$
$(1111111111)_2 = (2047)_{10}$	$\pm\infty$ Si $b_1 = \dots = b_{52} = 0$, NaN Sinon

TABLE 1.1 – Format double IEEE-754

Format	Précision	E_{min}	E_{max}	N_{min}	N_{max}	Epsilon machine
Double	P = 53	-1022	1023	$2^{-1022} \approx 2.2 \times 10^{-308}$	$\approx 2^{1024} \approx 1.8 \times 10^{308}$	$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$

TABLE 1.2 – Propriétés du format double

$$\begin{aligned}
 x \neq 0 &\rightarrow ufp(x) = 2^{\lceil \log_2 |x| \rceil} \\
 x = 0 &\rightarrow ufp(x) = 0
 \end{aligned}
 \tag{1.3}$$

Dans d'autres mots, nous pouvons définir $ufp(x)$ comme le poids du premier bit à gauche de la mantisse de x .

ulp

(Unit in the Last Place). Pour un nombre flottant normalisé x , elle vérifie :

$$ulp(x) = ufp(x) \cdot u \tag{1.4}$$

u étant la précision machine donnée par

$$u = 2^{-P} \quad (1.5)$$

En d'autres termes, $ulp(x)$ retourne le poids du dernier bit de la mantisse de x .

lnb

(Last Non-zero Bit). Introduit dans [15], $lnb(x)$ est le poids du dernier bit non nul dans la mantisse de x . Pour un nombre flottant x , nous savons que :

$$ulp(x) \leq lnb(x) \leq ufp(x)$$

Remarquez que ufp et lnb sont définies pour chaque nombre réel, alors que ulp dépend du format des nombre flottant utilisé. Étant donné deux nombres x, y :

$$lnb(x) \geq ufp(x).2u \leftrightarrow round(x) = x,$$

$$lnb(x) < ufp(x).2u \leftrightarrow round(x) \neq x$$

et

$$lnb(x + y) \geq \min(lnb(x), lnb(y)),$$

$$lnb(x \times y) = lnb(x) \times lnb(y)$$

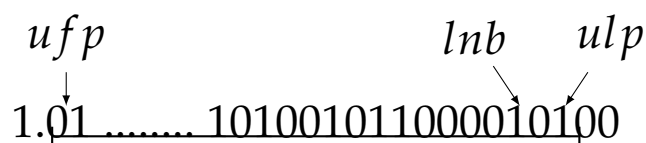


FIGURE 1.1 – ulp , ufp et lnb d'un nombre flottant (seulement la mantisse est représentée)

1.4 Les modes d'arrondi

En général, le résultat d'une opération (ou une fonction) sur un nombre flottant n'est pas exactement représentable dans le système à virgule flottante utilisé, alors il doit être *arrondi*. Dans les premiers systèmes à virgule flottante, la manière dont les résultats sont arrondis n'était pas toujours spécifiée. L'une des plus intéressantes idées amenées par IEEE 574-1985 [6] est le concept du *mode d'arrondi* : comment une valeur numérique est arrondie à un nombre flottant fini est spécifié par un *mode d'arrondi* (ou *attribut de la direction d'arrondi*) qui définit une fonction d'arrondi \circ . Par exemple, lorsque nous calculons $a + b$, où a et b sont des nombres flottants, le résultat retourné est $\circ(a + b)$.

L'utilisation de l'opération d'arrondi n'est pas limitée seulement aux opérations flottantes, mais il est aussi possible qu'on ait besoin d'arrondir un nombre qui est représentable en décimal avec un nombre fini de chiffres, mais pas en binaire. Par exemple, le nombre décimal 0.1 n'a pas de représentation binaire fini. Par conséquent, pour le représenter au format *binary64* il faut l'arrondir même si on a pas effectué une opération flottante.

Nous pouvons définir plusieurs modes d'arrondi possibles. Par exemple, les quatre modes mentionnés dans le standard IEEE 754-2008 [14] :

- * L'arrondi dirigé vers le bas (Arrondi vers $-\infty$) : $RD(x)$ est le plus large nombre flottant (possiblement, $-\infty$) inférieur ou égale à x .

$$RD(x) = x_-$$

- * L'arrondi dirigé vers le haut (Arrondi vers $+\infty$) : $RU(x)$ est le plus petit nombre flottant (possiblement $+\infty$) supérieur ou égale à x .

$$RU(x) = x_+$$

- * Arrondi vers zéro : $RZ(x)$ est le nombre flottant le plus proche à x qui n'est pas supérieur en magnitude à x .

$$x \geq 0 \rightarrow RZ(x) = RD(x)$$

$$x \leq 0 \rightarrow RZ(x) = RU(x)$$

- * Arrondi vers le plus proche : $RN(x)$ est le nombre flottant qui est le plus proche à x . La règle du tie-breaking doit être choisie lorsque x est exactement au milieu entre deux nombre flottant consécutif. Une règle tie-breaking fréquemment utilisée est appelée *round to nearest even* : x est arrondi à celui que sa mantisse est pair. C'est le mode par défaut dans le standard IEEE 754-2008.

Dans ce qui suit, nous supposons que le mode d'arrondi par défaut est l'arrondi vers le plus proche.

1.4.1 L'arrondi correct et l'arrondi fidèle

Pour un nombre réel non flottant x , son arrondi correct dépend du mode d'arrondi en utilisation, alors dans ce mémoire l'arrondi correct de x est son arrondi vers le plus proche. Alors que les deux nombres flottants qui entourent x de bas et de haut sont dits des arrondis fidèles.

Si x est exactement représentable au format flottant, son arrondi correct et arrondi fidèle sont x lui-même.

1.5 Les opérations flottantes

En appliquant une opération arithmétique sur deux nombres flottants, le résultat de cette opération est rarement un nombre flottant. Prenons l'exemple de 1 et 10 qui sont tous les deux des nombres flottants, et malgré cela le résultat de la division $1 \div 10$ ne l'est pas. Ce qui a causé un problème majeur dans le calcul arithmétique, puisque le résultat d'une opération peut se changer d'une machine à l'autre.

Avec l'apparition du standard IEEE-754, ce problème n'est plus à l'ordre du jour, puisque ce standard a introduit un nouveau concept, qui est celui des opérations flottantes [11].

Une opération flottante (\oplus , \ominus , \otimes , et \oslash) donne une approximation (une valeur arrondie) au résultat de l'opération standard (+, -, \times , /).

$$x \oplus y = \circ(x + y)$$

$$x \ominus y = \circ(x - y)$$

$$x \otimes y = \circ(x \times y)$$

$$x \oslash y = \circ(x / y)$$

1.6 Les erreurs sur les nombres flottants

1.6.1 Les erreurs d'arrondi

Les résultats fournis par des opérations flottantes sont généralement des approximations des résultats exacts, qui dépendent du mode d'arrondi. La différence entre le résultat exact et le résultat flottant est ce que nous appelons l'erreur d'arrondi.

L'erreur absolue

L'erreur absolue d'une opération flottante est la différence entre le résultat exacte x et le résultat flottant $\circ(x)$ étant l'arrondi correct de x .

$$AbsErr = |x - \circ(x)|$$

Sa valeur dépend de la précision et du mode d'arrondi en utilisation. Informellement, nous pouvons dire que l'erreur absolue est inférieure à $ulp(x)/2$.

$$AbsErr < ulp(x)/2$$

et

$$ulp(x) = 2^{-(P-1)} \times 2^e$$

e étant l'exposant de x .

↓

$$AbsErr < \frac{1}{2} \times 2^{-(P-1)} \times 2^e = 2^{-1} \times 2^{-(P-1)} \times 2^e$$

↓

$$AbsErr < 2^{-P} \times 2^e$$

Pour chaque nombre $x = \pm 1, b_1 b_2 \dots \times 2^e$, nous savons que $2^e \leq |x|$. Et nous avons défini dans l'équation 1.5 la précision machine $u = 2^{-P}$.

Donc nous pouvons plus simplement dire que

$$|x - \circ(x)| < u \times |x| \tag{1.6}$$

L'erreur relative

Sa valeur est le rapport entre l'erreur absolue et la valeur de x :

$$RelErr = |\sigma|$$

Où

$$\sigma = \frac{\circ(x) - x}{x}$$

↓

$$\begin{aligned}
 RelErr &= \left| \frac{\circ(x) - x}{x} \right| \\
 &\quad \downarrow \\
 RelErr &= \frac{|\circ(x) - x|}{|x|} < \frac{|u \times x|}{|x|}
 \end{aligned}$$

Vu que u est un nombre positif, nous pouvons directement déduire que

$$RelErr < u \tag{1.7}$$

1.6.2 L'élimination

(Cancellation en anglais) est un phénomène qui survient lors de la soustraction de deux nombres flottants qui sont très proches. Sterbenz a démontré dans [19] que pour $a/2 \leq b \leq 2 \cdot a$ la soustraction flottante de a et b est exacte ($a \ominus b = a - b$). Donc l'élimination n'introduit pas de nouvelles erreurs au calcul mais elle amplifie les erreurs générées par les calculs précédents. En d'autres termes, l'élimination augmente l'erreur relative de façon beaucoup plus significative que l'erreur absolue.

Cette élimination peut être catastrophique lorsque tous les bits significatifs s'éliminent, comme dans l'exemple suivant :

Soit

$$x = 3.141592653589793$$

$$y = 3.141592653585682$$

x représente une approximation de π sur 16 chiffres, et y ayant les mêmes douze premiers chiffres.

Leur différence est $z = 0000000000004111 = 4.111 \times 10^{-12}$. Pourtant, si nous calculons cette différence dans un programme C, en utilisant ici le format single (*binary32*) pour stocker x et y , ainsi que le résultat z , nous aurons $z = 0.000000e00$.

C'est simple, puisque les opérandes x et y doivent être premièrement convertis du format décimal au format *binary32*, ce qui nécessite certainement une opération

d'arrondi. Ici x et y partagent les douze premiers chiffres, ce qui veut dire qu'ils sont arrondis au même nombre flottant sous format single, ce qui induit une perte complète de précision lorsque x et y s'annulent lors de la soustraction.

1.7 Les transformations sans erreurs pour l'addition

a, b sont des nombres flottants, ainsi que $\circ(a + b)$, alors que le résultat de la simple addition $a + b$ est un réel (qui n'est pas généralement un flottant). Il est déjà connu que l'erreur d'approximation d'une opération flottante e est toujours un nombre flottant. Ce qui nous laisse conclure que :

$$s = \circ(a + b) \rightarrow a + b = s + e$$

Heureusement les quantités s , et e peuvent être calculées exactement dans l'arithmétique flottante.

Ci-dessous une sélection d'algorithmes qui nous permettent de le faire.

1.7.1 L'algorithme Fast2Sum

Cet algorithme a été introduit par Dekker [1] en 1971, mais les trois opérations de cet algorithme sont déjà apparues en 1965, comme partie de l'algorithme de la sommation compensée de Kahan.

L'algorithme afin de calculer s et e prend comme entrée deux nombre flottant a et b , tel que l'exposant de a est supérieur ou égale à celui de b (cette condition peut être difficile à vérifier, mais si $|a| \geq |b|$, la condition sera satisfaite). L'algorithme Fast2Sum fonctionne seulement pour un arrondi vers le plus proche.

L'algorithme 1 assure les deux propriétés suivantes :

Algorithme 1 : Algorithme Fast2Sum

Entrées: a, b : nombres flottants ($\text{exposant}(a) \geq \text{exposant}(b)$)**Sorties:** s, e : nombres flottants

1: $s = a \oplus b$

2: $t = s \ominus a$

3: $e = t \ominus b$

$$s = a \oplus b = \circ(a + b) \tag{1.8}$$

$$s + e = a + b$$

1.7.2 L'algorithme 2Sum

Basé aussi sur le mode d'arrondi vers le proche, introduit pour la première fois par Knuth [9] en 1969, cet algorithme calcule les mêmes valeurs s et e , en assurant les mêmes propriétés que Fast2Sum présentés dans l'équation 1.8. Contrairement à Fast2Sum, l'algorithme 2Sum n'exige pas un ordre particulier pour ces paramètres, mais de l'autre côté, il nécessite 6 opérations flottantes pour assurer la transformation sans erreur (trois opérations de plus par rapport à l'algorithme Fast2Sum). L'algorithme 2Sum fonctionne par contre dans n'importe quelle base de représentation, ce qui lui rend intéressant pour ceux qui travaillent dans le système décimale.

Si nous n'avons pas d'information sur l'ordre des paramètres a et b , il est très difficile de prédire en pratique s'il est plus rapide d'utiliser l'algorithme 2Sum qui fait six opérations flottantes pour assurer la transformation sans erreur, ou bien utiliser l'algorithme Fast2Sum avec un test préliminaire sur les valeurs de a et b . Le résultat de cette comparaison dépend largement de l'architecture du processeur sur lequel le code est exécuté. En effet, les processeurs récents, lorsqu'ils font face à une comparaison ou n'importe quelle autre condition, ils n'attendent pas que le résultat de la comparaison soient calculés, mais ils vont plutôt spéculer et prendre la valeur et charger les instructions suivantes avant que la condition soit testée. Cette technique peut

améliorer les performances dans certains cas, mais ça introduit un sur-coût considérable en cas de fausse prédiction. Quelques techniques de prédiction de branchement sont présentés dans [18], mais nous n'allons pas les détailler parce que ça va très loin des objectifs de ce mémoire.

Algorithme 2 : Algorithme 2Sum

Entrées: a, b : nombres flottants

Sorties: s, e : nombres flottants

1: $s = a \oplus b$

2: $t = s \ominus a$

3: $e = (a \ominus (s \ominus t)) \oplus (b \ominus t)$

1.8 Conclusion

Nous avons défini dans ce chapitre le standard IEEE-754, en abordant quelques notions importantes parmi celles introduites dans le standard sur les nombre flottants et leurs propriétés. À la fin du chapitre le concept avancé des transformations sans erreurs a été traité. Les notions présentées dans ce chapitre seront utilisées tout au long de ce mémoire.

CHAPITRE 2

LA SOMMATION FLOTTANTE : PROBLÈMES &
SOLUTIONS

2.1 Introduction

La sommation flottante est l'une des opérations basiques de calcul scientifique, et plusieurs algorithmes ont été déjà introduits et analysés pour calculer la somme. Vu que le concept de la sommation normale est simple, il est imaginable que les algorithmes de la sommation flottante ne demandent pas d'ingéniosité. C'est pour cela nous allons essayer dans ce chapitre d'éclairer un peu l'image, en spécifiant les problèmes majeurs qu'envisage la sommation, ainsi que les solutions qui ont été proposées pour les résoudre.

2.2 Les problèmes de la sommation flottante

Problèmes du conditionnement, de la majoration, et de la précision

Soit P un vecteur de n nombres flottants (p_1, p_2, \dots, p_n) , S est la somme exacte des éléments de P ($S = \sum_{i=1}^n p_i$), \hat{S} est la somme flottante des éléments de P

$$\hat{S} = p_1 \oplus p_2 \oplus p_3 \cdots \oplus p_n$$

L'algorithme 3 représente la méthode itérative naïve qui permet de calculer la somme \hat{S} , qui est connu sous le nom "la sommation classique".

Algorithme 3 : La somme classique

Entrées: p : un vecteur de n nombres flottants ;

Sorties: \hat{S} : la somme flottante des éléments de p ;

1: $\hat{S} = 0$

2: **for** ($i = 1; i \leq n; i = i + 1$) **do**

3: $\hat{S} = \hat{S} \oplus p_i$

4: **end for**

La différence entre \hat{S} et S est l'erreur absolue de cet algorithme de sommation classique. Il est montré dans [11] que pour les additions élémentaires arrondies vers le plus proche, nous avons :

$$|S - \hat{S}| \leq \gamma_{n-1} \cdot \sum_{i=1}^n |p_i|$$

Où

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

L'erreur relative de cet algorithme peut être calculée de la manière suivante :

$$\frac{|S - \hat{S}|}{|S|} \leq \gamma_{n-1} \cdot \frac{\sum_{i=1}^n |p_i|}{|S|} = \gamma_{n-1} \cdot \frac{\sum_{i=1}^n |p_i|}{|\sum_{i=1}^n p_i|} \quad (2.1)$$

L'erreur relative dépend donc de la précision machine u , la taille du vecteur n , et le conditionnement. Ici, le conditionnement de la somme est défini par l'expression :

$$\text{Cond}\left(\sum_{i=1}^n p_i\right) = \frac{\sum_{i=1}^n |p_i|}{\left|\sum_{i=1}^n p_i\right|} \quad (2.2)$$

Les éliminations peuvent rendre la valeur de $|\sum_{i=1}^n p_i|$ très petite comparé à $\sum_{i=1}^n |p_i|$, ce qui induit un problème mal conditionné. Par contre, s'il n'y a pas beaucoup d'éliminations, la valeur de $|\sum_{i=1}^n p_i|$ sera proche de la valeur de $\sum_{i=1}^n |p_i|$, le conditionnement sera petit, et nous pouvons dire que le problème est bien conditionné.

2.2.1 Problème de la reproductibilité

Au contraire de l'addition normale, l'addition flottante ne jouit pas de la propriété d'associativité à cause des erreurs d'arrondi. Ce qui implique des résultats différents lorsque nous changeons l'ordre des opérations, en d'autres mots, pour trois nombres flottants a, b et c

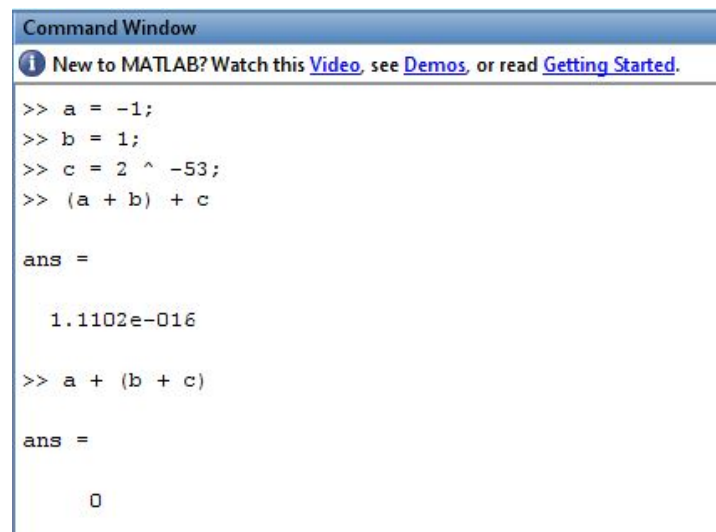
$$(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$$

L'exemple le plus souvent utilisé pour illustrer ce phénomène est l'addition des trois nombres $a = -1, b = 1$ et $c = u = 2^{-53}$.

Si nous calculons $(a \oplus b) \oplus c$, l'ordinateur calcule d'abord $a \oplus b$ ce qui donne évidemment 0 parce que $a = -b$ (pas besoin d'opération d'arrondi parce que le résultat est représentable exactement). puis $0 + c$ nous donne comme résultat 2^{-53} .

Par contre, si nous calculons $a \oplus (b \oplus c)$, l'ordinateur commence par l'évaluation de $b \oplus c$ ce qui donne $1 \oplus 2^{-53} = \circ(1 \oplus 2^{-53}) = 1 = b$. On dit dans ce cas que c a été *absorbé* par b . Après l'évaluation de la deuxième opération, nous obtenons comme résultat 0.

$$a \oplus (b \oplus c) = 0 \neq u = (a \oplus b) \oplus c$$



```
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

>> a = -1;
>> b = 1;
>> c = 2 ^ -53;
>> (a + b) + c

ans =

    1.1102e-016

>> a + (b + c)

ans =

    0
```

FIGURE 2.1 – Non-reproductibilité de la somme

Il est possible de tester l'exemple précédent dans n'importe quel langage de programmation, nous montrons dans la figure 2.1 quelques commandes Matlab qui illustrent le problème que nous venons d'expliquer. Due à cette non-associativité, le résultat d'une sommation flottante ne dépend plus seulement des données en entrée mais aussi de l'ordre de l'exécution des opérations. Nous pouvons dire que la sommation flottante est non-reproductible, puisqu'elle ne reproduit pas les mêmes résultats à chaque fois que nous changeons l'ordre des opérations. Deux choix intéressants pour l'ordre de la sommation sont l'ordre ascendant et l'ordre descendant. Si nous appliquons l'algorithme de la sommation classique dans les deux ordres susmentionnés, nous obtiendrons deux résultats différents.

En pratique, l'ordre d'accumulation peut changer due aux changements dans les détails architecturaux si le code est porté d'une machine à une autre, ou aussi à cause du changement dans l'ordonnement dynamique et le nombre de threads (dans une exécution parallèle). La reproductibilité est différente de la précision. La précision est mesurée par l'erreur relative du résultat numérique. En d'autres termes, en comparant combien de bits corrects apparaissent dans le résultat numérique en le comparant avec le résultat exact. Par contre la reproductibilité est le fait d'avoir des

résultats numériques identiques lors des différentes exécutions. Néanmoins, un résultat reproductible n'est pas forcément précis, tout dépend de l'algorithme utilisé. Nous pouvons même imaginer un résultat reproductible sans aucun bit correct comparé au résultat exacte si le problème est mal conditionné.

Le calcul du résultat correctement arrondi assure à la fois la reproductibilité et la précision.

2.3 Des solutions proposées

2.3.1 Solutions pour la précision

Les algorithmes de distillation

Le distillation est un processus accompli tout simplement en copiant le contenu du vecteur d'entrée S dans un vecteur temporaire D , et en remplaçant itérativement chaque pair (d_i, d_{i+1}) par $(d_i + d_{i+1} - \hat{d}, \hat{d})$, où \hat{d} est la somme $d_i \oplus d_{i+1}$. En d'autres termes, d_{i+1} est remplacé par la somme $d_i \oplus d_{i+1}$, alors que d_i est remplacé par l'erreur de cette somme. Cet objectif peut être atteint en utilisant l'algorithme 2Sum ou bien l'algorithme Fast2Sum (avec un test préliminaire sur les paramètres) comme le montre l'équation 1.8. La distillation est généralement utilisée pour améliorer la précision de la sommation, ou pour appliquer une transformation sans erreur sur un vecteur de nombres flottants.

L'algorithme SumK

Cet algorithme a été introduit dans [12], il est basé sur le principe de la distillation. Ogita et al. ont identifié l'algorithme VecSum qui consiste tout simplement à un seul pas de distillation.

L'algorithme VecSum garantit que

Algorithme 4 : algorithm VecSum**Entrées:** p : un vecteur flottant de taille n ;**Sorties:** q : une transformation sans erreur de p issue de la distillation ;1: **for** ($i = 2; i \leq n; i = i + 1$) **do**2: $(q_i, q_{i-1}) = TwoSum(p_i, p_{i-1})$;3: **end for**

$$\sum_{i=1}^n p_i = \sum_{i=1}^n q_i,$$

$$\text{et } q_n = ((\dots(p_1 \oplus p_2) \oplus \dots) \oplus p_n).$$

Si la somme du vecteur P est mal conditionnée, il y aura beaucoup d'éliminations dans le processus, donc $\sum_{i=1}^n |q_i| \ll \sum_{i=1}^n |p_i|$. Alors que $\sum_{i=1}^n q_i = \sum_{i=1}^n p_i$. Les auteurs de [12] ont prouvé que si $Cond(\sum_{i=1}^n p_i) > u^{-1}$, nous aurons :

$$\sum_{i=1}^n |q_i| \approx \sum_{i=1}^n |p_i| \times u$$

Alors calculer la somme du vecteur Q est significativement mieux conditionné par rapport à calculer la somme du vecteur P , puisque :

$$Cond\left(\sum_{i=1}^n q_i\right) = \frac{\sum_{i=1}^n |q_i|}{|\sum_{i=1}^n q_i|} \approx \frac{u \cdot \sum_{i=1}^n |p_i|}{|\sum_{i=1}^n p_i|} \approx u \cdot Cond\left(\sum_{i=1}^n p_i\right)$$

L'algorithme SumK utilise répétitivement l'algorithme VecSum pour améliorer la précision du résultat de la sommation.

Les auteurs ont prouvé que pour $k = 2$, l'erreur relative du résultat obtenu S_k est majorée comme suit :

$$\frac{|S_k - S|}{|S|} \leq u + \gamma_{n-1}^2 \cdot Cond\left(\sum_{i=1}^n p_i\right)$$

et pour $K \geq 3$, l'erreur relative est prouvée d'être améliorée tel que :

Algorithme 5 : Algorithme SumK

Entrées: p : un vecteur de n nombres flottants
 K : le nombre de distillations à faire.
Sorties: S_k : une somme plus précise des éléments de p

- 1: **for** ($k = 1; k \leq K - 1; k = k + 1$) **do**
- 2: $p = \text{vecSum}(p)$
- 3: **end for**
- 4: $S_k = 0$
- 5: **for** ($i = 1; i \leq n; i = i + 1$) **do**
- 6: $S_k = S_k \oplus p_i$
- 7: **end for**

$$\frac{|S_k - S|}{|S|} \leq u + 3 \cdot \gamma_{n-1}^2 + \gamma_{2n-2}^K \cdot \text{Cond} \left(\sum_{i=1}^n p_i \right) \quad (2.3)$$

L'effet du conditionnement dans le dernier terme de la relation précédente peut disparaître en choisissant un k suffisamment large.

L'algorithme iFastSum

Cet algorithme a été introduit par Zhu et Hayes dans [22]. Son principe n'est pas différent de celui de SumK. iFastSum adapte le nombre d'itérations k selon le conditionnement du problème sans le calculer, et il répète le processus de la distillation jusqu'à ce que le résultat final soit correctement arrondi. L'équation 2.3 montre que peu importe la valeur de k , nous ne pouvons pas assurer que l'erreur relative est plus petit que $u + 3 \cdot \gamma_{n-1}^2$. Donc augmenter la valeur de k n'est pas suffisant pour assurer un résultat correctement arrondi. L'algorithme 6 représente une version simplifiée de iFastSum. Ce dernier définit un contrôle dynamique d'erreur durant la distillation pour estimer une limite d'erreur et décider si le résultat est correctement arrondi. Les propriétés de iFastSum sont démontrées dans [22]. Cependant, nous présentons par la suite une brève description de cet algorithme.

Les lignes 6 - 12 définissent la boucle principale de la distillation. Le teste de la ligne 9 assure que seulement les erreurs non-zéro sont gardées pour les prochaines itérations, et la variable *count* compte ces erreurs. Le vecteur d'entrée P est modifié

Algorithme 6 : Algorithme iFastSum

Entrées: p : un vecteur de n nombres flottants
allowRec : indique si des appels recursives sont autorisés (*true* par défaut)

Sorties: $\widehat{S} : \circ(\sum_{i=1}^n p_i)$

- 1: $\widehat{S} = 0$
- 2: **while true do**
- 3: *count* = 1 {*Compter le nombre d'erreurs non nuls*}
- 4: $S_t = 0$
- 5: $S_{max} = 0$ {*La valeur intermédiaire maximale de S_t *}
- 6: **for** ($i = 1; i \leq n; i = i + 1$) **do**
- 7: $(S_t, p_{count}) = TwoSum(S_t, p_i)$
- 8: $S_{max} = \max(S_{max}, |S_t|)$
- 9: **if** $p_{count} \neq 0$ **then**
- 10: *count* = *count* + 1
- 11: **end if**
- 12: **end for**
- 13: *maxError* = (*count* - 1) $\times ulp(S_{max})/2$
- 14: $(\widehat{S}, S_t) = TwoSum(\widehat{S}, S_t)$
- 15: *n* = *count* {*Travailler sur les erreurs non nuls seulement*}
- 16: $p_{count} = S_t$ {*Enregistrer les erreurs dans p^* }
- 17: **if** *maxError* < $ulp(\widehat{S})/2$ **then**
- 18: **if not** *allowRec* **then**
- 19: **return** \widehat{S}
- 20: **end if**
- 21: $(S^+, e^+) = TwoSum(S_t, maxError)$
- 22: $(S_-, e_-) = TwoSum(S_t, -maxError)$
- 23: **if** $(\widehat{S} \oplus S^+ \neq \widehat{S})$ or $(\widehat{S} \oplus S_- \neq \widehat{S})$ or $(round3(\widehat{S}, S^+, e^+) \neq \widehat{S})$ or
 $(round3(\widehat{S}, S_-, e_-) \neq \widehat{S})$ **then**
- 24: $S_1 = iFastSum(x, false)$
- 25: $S_2 = iFastSum(x, false)$
- 26: $\widehat{S} = round3(\widehat{S}, S_1, S_2)$
- 27: **end if**
- 28: **return** \widehat{S}
- 29: **end if**
- 30: **end while**

par les erreurs générées. Considérant que la somme exacte est S , nous savons qu'à la fin de la boucle nous aurons : $S = \widehat{S} + S_t + \sum_{i=0}^{count-1} p_i$.

Vu que pour chaque évaluation intermédiaire de S_t , et selon la relation 1.6, l'erreur générée est plus petite que $S_t * u = ulp(S_t)/2$, et S_{max} est la valeur intermédiaire

maximale de la somme des erreurs générées, $\sum_{i=0}^{count-1} p_i$ est limitée par $(count - 1) \times (ulp(S_{max})/2)$. Si la condition de la ligne 17 est vérifiée, \widehat{S} est garantie d'être un arrondi fidèle de la valeur de S .

La condition dans la ligne 18 teste si les appels récursifs sont permis. Remarquez que si ce n'est pas le cas (`allowRec` est égale à `false`), `iFastSum` retourne un arrondi fidèle du résultat exacte. Autrement, les lignes 21 - 26 assurent que le résultat final est correctement arrondi. À ce niveau nous savons que

$$\widehat{S} + S_t - maxError \leq S \leq \widehat{S} + S_t + maxError$$

. Après les lignes 22 et 23, nous avons

$$\widehat{S} + S_- + e_- \leq S \leq \widehat{S} + S^+ + e^+$$

Ensuite, la fonction `round3` (présentée dans la figure 2.3 dans [22]) est utilisée pour arrondir $\widehat{S} + S^+ + e^+$ et $\widehat{S} + S_- + e_-$. Si les deux valeurs sont arrondies à \widehat{S} , on déduit que le dernier est l'arrondi correct de S . Autrement, S_1 et S_2 sont calculés, tel que S_1 est l'arrondi fidèle de $S - \widehat{S}$, et S_2 est le résultat l'arrondi fidèle de $S - \widehat{S} - S_1$. Finalement, la fonction `round3` est utilisée pour calculer un résultat correctement arrondi de S . Vu que \widehat{S} est l'arrondi fidèle de S , nous déduisons que $S_2 < ulp(S_1)$.

Nous sommes ici devant deux possibilités :

1. $|S_1| = ulp(\widehat{S})/2$, dans ce cas seulement le signe de S_2 est demandé. Si $S_2 > 0$ l'arrondi vers le haut de $\widehat{S} + S_1$ retourne l'arrondi correct de S ; Si $S_2 < 0$, l'arrondi vers le bas de $\widehat{S} + S_1$ retourne l'arrondi correct de S .

Si $S_2 = 0$, nous déduisons que S est exactement au milieu de deux nombres flottants, dans ce cas l'utilisation de l'arrondi vers le plus proche garantit que $\widehat{S} \oplus S_1$ est la valeur correctement arrondie de S .

2. $|S_1| \neq ulp(\widehat{S})/2$, $\widehat{S} \oplus S_1$ représente le résultat correctement arrondi de S .

Certains pas ont été simplifiés dans notre description. Cependant, vous trouvez plus de détails, ainsi qu'une démonstration formelle sur iFastSum dans [22].

L'algorithme HybridSum

L'inconvénient majeur des algorithmes déjà présentés est le fait qu'ils font plusieurs parcours des données d'entrée pour affiner le résultat de la sommation et fournir un résultat correctement arrondi ou au moins son arrondi fidèle. Plus d'affinements sont exigés si le problème est mal conditionné. HybridSum [22] surmonte ce problème, il fournit un processus similaire indépendamment du conditionnement du problème. Il accumule avec précision les entrées ayant le même exposant dans un même accumulateur.

Des accumulateurs à haute précision ne sont pas encore implémentés dans le matériel. Alors HybridSum divise la mantisse d'entrée en deux parties pour que les nombres flottants standards puissent jouer le rôle d'accumulateurs. L'algorithme 7 peut être utilisé pour diviser un nombre flottant en deux nombres avec une mantisse de demi longueur.

Algorithme 7 : Algorithme split

Entrées: $x, mask$: nombre flottant.

Sorties: x_h, x_l : la partie haute (premier 27 bits de la mantisse) et la partie basse (les derniers 26 bits de la mantisse) de x .

1: $x_h = \text{binaryAnd}(x, mask)$ {*remplacer les derniers 26 bits de la mantisse de x par des zéros*}

2: $x_l = x \ominus x_h$

Remarquez que nous avons besoin d'un accumulateur distinct pour le format *binary64* (2048 exposants possibles). Ce processus réduit la taille des données en entrée à un vecteur de 2048 éléments sans générer des erreurs d'arrondi. Ceci est très intéressant pour les vecteurs de grandes tailles, puisqu'il transforme sans erreur le vecteur d'entrée en un vecteur à petite taille en une seule itération avec un coût linéaire par rapport à la taille du vecteur, et qui ne dépend pas du conditionnement de la somme. Par la suite, un algorithme de sommation itératif peut parcourir ce petit

vecteur à un coût réduit. Les écrivains ont utilisé iFastSum pour calculer un résultat correctement arrondi. Cependant tout autre algorithme de sommation peut être utilisé.

L'algorithme 8 montre comment le vecteur d'entrée est transformé sans erreur en un vecteur de taille 2048.

Algorithme 8 : Transformation

Entrées: p : un vecteur de nombre flottants de taille n
Sorties: C : un vecteur de nombres flottants de taille 2048

- 1: **for** ($i = 1; i \leq n; i = i + 1$) **do**
- 2: $(p_{i,h}, p_{i,l}) = \text{split}(p_i)$
- 3: $e_h = \text{exponent}(p_{i,h})$
- 4: $e_l = \text{exponent}(p_{i,l})$
- 5: $C_{e_h} = C_{e_h} \oplus p_{i,h}$
- 6: $C_{e_l} = C_{e_l} \oplus p_{i,l}$
- 7: **end for**

2.3.2 Solutions pour la reproductibilité

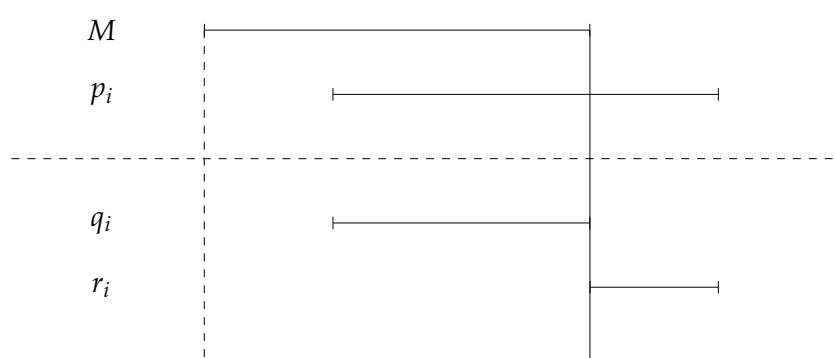
L'algorithme ReprodSum

ReprodSum [2] dépend de AccuSum qui est un algorithme qui assure que le résultat est un arrondi fidèle du résultat exacte (présenté dans [17]). L'algorithme Extract-Vector est utilisé pour extraire la partie haute du vecteur d'entrée. Donc, le résultat est dans ce cas indépendante de l'ordre d'accumulation. Alors, ReprodSum assure des résultats reproductible indépendamment de l'environnement (jeu d'instruction, nombre de threads ... etc).

L'algorithme 9 montre la procédure d'extraction et l'accumulation de la partie haute du vecteur. Chaque élément est découpé comme le montre la figure 2.2. La valeur de M est choisie selon la taille du vecteur et le maximum des valeurs absolues de ces éléments ($\max |p_i|$)s

Algorithme 9 : L'algorithme *ExtractVector***Entrées:** p : un vecteur de nombres flottants.**Sorties:** T : la somme de la partie haute des éléments du vecteur p .

- 1: $T = 0$
- 2: **for** ($i = 1; i \leq n; i = i + 1$) **do**
- 3: $q_i = (M \oplus p_i) \ominus M$
- 4: $r_i = p_i \ominus q_i$
- 5: $T = T \oplus q_i$
- 6: **end for**

FIGURE 2.2 – Découpage d'un élément dans l'algorithme *ExtractVector*

Alors que *AccSum* répète le processus jusqu'à obtenir l'arrondi fidèle, *ReprodSum* prend le nombre d'itérations k comme paramètre. Comme dans le cas de *SumK*, augmentant la valeur de k augmente la précision du résultat de *Reprodsum*. Si le conditionnement de la somme est connu, nous devons choisir un k suffisamment large pour surmonter les plusieurs éliminations ou même assurer un arrondi fidèle du résultat exacte. Réciproquement, diminuant la valeur de k diminue la précision, mais augmente les performances (réduit le temps de calcul).

2.4 Conclusion

Dans la première partie de ce chapitre nous avons présenté les problèmes majeurs que rencontre la sommation flottante de la précision à la reproductibilité numérique. Plusieurs algorithmes ont été proposés pour résoudre ces problèmes, parmi ceux une

Algorithme 10 : L'algorithme ReprodSum

Entrées: p : un vecteur de n nombres flottants. k : le nombre d'itérations à faire.

Sorties: T : Somme reproductible des éléments de p .

```

1:  $m = \max(v_i)$ 
2:  $\sigma_1 = (n \otimes m) \otimes (1 \ominus 2n\epsilon)$ 
3:  $M_1 = 2^{\lceil \log_2(\sigma_1) \rceil}$ 
4: for ( $i = 1; i \leq k - 1; i = i + 1$ ) do
5:    $[T_i, v] = \text{ExtractVector}(M_i, v)$ 
6:    $\sigma_{i+1} = (n \otimes (2 \otimes \epsilon \otimes M_i)) \otimes (1 \ominus 2 \otimes n \otimes \epsilon)$ 
7:    $M_{i+1} = 2^{\lceil \log_2(\sigma_{i+1}) \rceil}$ 
8: end for
9:  $T_k = 0$ 
10: for ( $i = 1; i \leq n; i = i + 1$ ) do
11:    $q_i = (M_k \oplus v_i) \ominus M_k$ 
12:    $T_k = T_k \oplus q_i$ 
13: end for
14:  $T = 0$ 
15: for ( $i = 1; i \leq k; i = i + 1$ ) do
16:    $T = T \oplus T_i$ 
17: end for

```

sélection d'algorithmes récents ont été présentés dans la deuxième partie de ce chapitre. Ces algorithmes sont le coeur de notre étude, et nous allons dans les prochains chapitres les comparer en se basant sur les résultats des testes intensifs à lesquels ils seront remis.

CHAPITRE 3

OUTILS & MÉTHODOLOGIE

3.1 Introduction

Le standard IEEE-754 a été établi pour résoudre entre autres les problèmes de la fiabilité et de la probabilité dans le domaine du calcul numérique. Mais, la publication du standard n'était pas suffisante pour régler ces problèmes. Il fallait une adoption globale de ce standard par les constructeurs des machines, en concevant de nouvelles architectures matérielles adoptant les nouveaux concepts qui ont été introduits. Ainsi qu'une révision générale des langages de programmations existants pour qu'ils soient capables de gérer ces nouveaux concepts (les différents formats, les différents modes d'arrondi, les exceptions, ... etc). L'adoption du standard n'était pas instantanée, il fallait attendre plusieurs années pour voir finalement des langages l'adopter entièrement. Alors que d'autres langages restent jusqu'à aujourd'hui non concernés par le standard, Le C et Fortran étaient les premiers à adopter entièrement le standard (1999 et 2003 respectivement) [14].

Afin de faire notre étude comparative, nous avons choisi le C pour implémenter les différents algorithmes à comparer, ainsi que Matlab pour avoir une visualisation graphique des résultats obtenus.

3.2 Les outils de développement

3.2.1 Le langage C

Le C est un langage de programmation à usage général qui suit le paradigme impératif. Il supporte la programmation structurée, la portée lexicale des variables, et les types statiques, ... etc. Le C fournit au programmeur la puissance totale de la machine, ainsi qu'un contrôle très considérable sur les programmes qu'il écrit. Ce qui a lui permis de prendre la place du langage assembleur pour résoudre des problèmes ou le dernier a été la solution exclusive [16]. À l'origine, le C a été développé par Dennis Ritchie entre 1969 et 1973 dans les laboratoires AT&T Bell. Il a été utilisé pour ré-implémenter le système d'exploitation Unix. Le C est devenu l'un des langages de programmation les plus utilisés dans le monde, avec ses compilateurs des différents vendeurs et disponibles pour la majorité des architectures systèmes. Le C a été normalisé par l'ANSI (American National Standards Institute) et par la suite par l'organisation internationale de la normalisation (ISO). Plusieurs langages ont emprunté par la suite du C (directement ou indirectement), tel que C++, D, Go, Rust, Java, JavaScript, Limbo, LPC, C#, Objective-C, Perl, PHP, Python, Verilog, et le shell du système d'exploitation Unix. Ces langages ont conçu plusieurs de leurs structures de contrôle et d'autres caractéristiques de bases du C, habituellement avec des similarités syntaxiques. Le C est aussi utilisé comme un langage intermédiaire pour d'autres langages, ainsi que pour construire des bibliothèques standards et des systèmes d'exécution pour les langages à haut niveau, tel que CPython [8].

L'implémentation du standard dans le langage C

Les types flottants

Dans le langage C, les trois formats binaires principaux sont prises en charge comme suit :

- Le type *float* correspond au format single du standard.

- Le type *double* correspond au format double du standard.
- Le type *long double* correspond au format étendu du standard.

Type	Taille (Octets)	Identifiant	Valeur de Π
Float	4	%f	3.141592
Double	8	%lf	3.141592653589793
Long double	10 - 16	%Lf	3.141592653589793213

TABLE 3.1 – Des propriétés des différents formats flottants dans le C

La bibliothèque math.h

Math.h est une bibliothèque disponible sur tous les compilateurs C. Elle fournit au programmeur un ensemble de fonctions mathématiques traditionnelles, ainsi que d'autres plus avancées qui prennent des nombres flottants comme paramètres, et retournent leurs résultats sous format flottant aussi.

Ci-dessous est un tableau contenant quelques fonctions de cette bibliothèque.

<i>fabs</i>	La valeur absolue : <i>fabs(x)</i> retourne $ x $
<i>sqrt</i>	La racine carrée : <i>sqrt(x)</i> retourne \sqrt{x}
<i>exp</i>	Exponentiel (base e) : <i>exp(x)</i> retourne e^x
<i>log</i>	Logarithme (base e) : <i>log(x)</i> retourne $\log_e(x)$
<i>log10</i>	Logarithme (base 10) : <i>log10(x)</i> retourne $\log_{10}(x)$
<i>sin</i>	Sinus (l'argument en radian)
<i>Cos</i>	Cosinus (l'argument en radian)
<i>pow</i>	La puissance : <i>pow(x,y)</i> retourne x^y
<i>ceil</i>	L'arrondi entier dirigé vers le haut

TABLE 3.2 – Quelques fonctions de la bibliothèque math.h

Les exceptions dans le calcul flottant

Le standard définit cinq exceptions [3] qui peuvent se produire lors des calculs. Chacune correspond à un type particulier d'erreur. Lorsque les exceptions se produisent, il y aura deux conséquences possibles. Par défaut, l'exception est tout simplement marquée dans le drapeau *status*, et le programme continue si rien ne s'est passé. L'opération produit un résultat par défaut, qui dépend de l'exception. Le programme peut vérifier le *status* pour trouver les exceptions produites. Alternativement, nous pouvons activer les *traps* pour les exceptions. Dans ce cas, lorsqu'une exception se produit le programme reçoit un signal SIGFPE (SIGnal : Floating Point Exception). L'action par défaut pour ce signal est de mettre fin au programme, mais nous pouvons la reconfigurer. Les exceptions définies par le standard IEEE-754 sont :

- Opération invalide, cette exception se produit si un des opérandes donnés est invalide pour l'opération à exécuter.
 - L'addition ou la soustraction des infinités.
 - La multiplication $0 \cdot \infty$.
 - La division $0/0$ ou ∞/∞ .
 - Le reste de la soustraction $x \text{ REM } y$ lorsque y est un 0 ou x est une infinité.
 - La racine carrée si l'opérande est négatif. Plus généralement, chaque fonction évaluée en dehors de son domaine de définition produit cette exception.
 - La conversion d'un nombre flottant en un entier ou une chaîne de caractères décimaux, lorsque le nombre ne peut pas être représenté dans format ciblé.
- Division par zéro, cette exception se produit lorsqu'un nombre fini non-nul est divisé par zéro. Si aucun *traps* n'est signalé le résultat est $+\infty$ ou $-\infty$, selon les signes des opérandes.
- Dépassement, cette exception se produit lorsque le résultat ne peut pas être représenté comme une valeur finie dans le format de destination. Si aucun *traps* n'est signalé le résultat dépend du signe du résultat intermédiaire et le mode

d'arrondi actuel. En d'autres termes, si l'exposant est trop grand pour être représenté dans le format utilisé.

— Soupassement, cette exception se produit lorsque le résultat intermédiaire est tellement petit qu'il ne peut pas être calculé précisément ou normalisé. En d'autres termes, si l'exposant est trop petit pour être représenté dans le format utilisé.

— Inexact, cette exception est signalée si le résultat arrondi n'est pas exacte.

Le C utilise des macros [7] pour gérer quelques exceptions, parmi lesquels nous mentionnons :

— **INFINITY**, est une expression qui représente l'infinité positive. Elle est égale à la valeur produite par des opérations mathématiques similaires à 1.0/0.0. Alors que la macro **-INFINITY** représente l'infinité négative. Nous pouvons comparer une valeur flottante avec cette macro pour vérifier si cette valeur est infinie. Mais ce n'est pas très conseillé. Alternativement, nous pouvons utiliser la macro **isfinite**.

— **NAN**, est une expression qui représente une valeur qui n'est pas un nombre. Nous pouvons utiliser **#ifdef NAN** pour tester si la machine supporte **NaN**.

— L'environnement flottant défini dans **<fenv.h>** inclut les drapeaux de *status* et les *traps* des exceptions, les contrôle du mode d'arrondi, ainsi que le contrôle de la précision.

3.2.2 Matlab

Matlab est un langage de programmation interprété et structuré, utilisé principalement pour le calcul numérique. MATLAB a été conçu par Cleve Moler à la fin des années 1970. Matlab 1.0 écrit en C et commercialisé en 1984 par la société The MathWorks qui vient d'être crée pour cet objectif. Ce langage a ensuite évolué en intégrant des boites d'outils qui permettent entre autre la coopération entre lui et d'autres langages. Aujourd'hui Matlab est très connu grâce aux différentes options de manipulation de données qu'il offre. Il manipule les vecteurs et les matrices, et dessine les

courbes. Matlab offre la possibilité de travailler même sur des données fournies par d'autres langages comme le C dans notre cas. Nous avons utilisé MATLAB pour lire les données générées par notre code C, et en générer des figures facilement lisibles. Nous avons choisi d'utiliser Matlab principalement parce qu'il est très pratique lorsqu'il s'agit de manipuler des vecteurs et des matrices. Donc nous pourrions facilement lire, filtrer, trier ou effectuer n'importe quelle autre opération sur la matrice contenant nos résultats. D'un autre côté, Matlab permet aussi de générer et enregistrer des figures contenant des courbes facilement et sans avoir besoin d'utiliser une bibliothèque externe.

3.2.3 Makefiles

Les makefiles sont des petit script écrits généralement dans le but d'effectuer des actions simples (compilation d'une bibliothèque, Lancer plusieurs programmes dans un ordre spécifié ... etc.). Un fichier "makefile" est constitué de plusieurs règles de la forme [4] :

```
cible: dépendances
    actions
```

La cible est généralement un fichier à créer, les dépendances sont les fichiers que nous devons utiliser pour générer la cible, et les actions sont les instructions qui permettent de créer la cible à partir de ces dépendances. Si les dépendances ne sont pas encore satisfaites (leurs fichiers n'existent pas), le makefile va d'abord chercher des règle où les dépendances actuelles sont des cibles. Si les dépendances ont d'autres dépendances, le même processus est répété récursivement. Par exemple la règle qui permet de générer un exécutable à partir d'un code source ".c" est la suivante.

```
main.exe: main.c
    gcc main.c -o main.exe
```


3.2.4 Ressources matérielles et compilation

L'implémentation et les différents tests des cinq algorithmes ont été fait sur un PC dont les propriétés sont spécifiées dans le tableau ci-dessous :

Processeur	Intel® Core™ i5-2520M CPU @ 2.50GHz ×2
Cache	3 Mega Bytes
Systeme d'exploitation	Ubuntu 20.04.1 LTS
Compilateur	gcc version 9.3.0
Options	-O3 -march=native -funroll-all-loops -Wall

TABLE 3.3 – Les propriétés de la machine utilisée pour les tests

Nous nous sommes limité à l'utilisation de l'option de compilation "-O3" au lieu de "-Ofast" parce que la dernière introduit sur le code des optimisations très agressives qui ignorent l'arithmétique flottante. Par exemple si l'option "-Ofast" est choisi, et que le compilateur est en train d'optimiser le code de la fonction Fast2Sum (Algorithme 1), lorsqu'il trouve que $s = a + b$, $t = s - a$, et $e = t - b$. Il estime que :

$$e = t - b$$

$$e = s - a - b$$

$$e = a + b - a - b$$

$$e = 0$$

Donc, pour le compilateur, d'un point de vu mathématique, les instructions 2 et 3 sont inutiles, et il peut les éliminer. Alors, au lieu de faire une transformation sans erreur, il remplace le code tout simplement par :

$$s = a \oplus b$$

$$e = 0$$

La documentation officielle du langage [13] fournit plus d'informations sur les options de compilation à utiliser.

3.3 Méthodologie

3.3.1 Génération des données

Afin de comparer notre sélection d'algorithmes de sommation, nous avons besoin des données comme entrée de ces algorithmes. Nous savons déjà que les algorithmes de sommation opèrent sur des vecteurs de nombre flottants. Alors les données que nous devons générer seront des vecteurs de n nombres flottants, chacun associé avec sa somme correctement arrondie. Pour que l'étude soit plus précise, nous avons besoin de générer plusieurs échantillons, en variant à chaque fois la taille du vecteur, ainsi que son conditionnement. Rump [12] a utilisé dans un travail précédent un algorithme pour générer des produits scalaires mal conditionnés. Nous avons utilisé une version légèrement différente pour générer des sommes mal conditionnées, qui serviront comme des entrées pour les algorithmes que nous allons comparer. Ci-dessous l'algorithme de génération de données.

Le principe de l'algorithme 11 est de :

1. Générer la première partie du vecteur (lignes 2-10) en assurant que la somme de ses valeurs absolues est au même ordre de grandeur que le conditionnement (demandé par l'utilisateur).
2. Puis, la deuxième partie du vecteur est générée pour éliminer (cancelation) la somme de la première partie et assurer que la somme du vecteur est proche de 1, et que par conséquent son conditionnement est proche de sa somme des valeurs absolues (lignes 11-17).
3. Finalement, les trois instructions à la fin de l'algorithme (lignes 18, 19 et 20) calculent la somme correctement arrondie, le conditionnement de la somme générée, et applique une permutation aléatoire sur les éléments du vecteur.

Algorithme 11 : Générateur de sommes mal conditionnée

Entrées: n : La taille de la somme à générer
 CA : Le conditionnement (approximatif) de la somme
Sorties: S : L'arrondi correcte de la somme générée
 CE : Le conditionnement (exact) de la somme générée
 X : Un vecteur de nombres flottants de taille n tel que $round(\sum X_i) = S$ et $round(cond(\sum X_i)) = CE$

- 1: $n2 = n \text{ div } 2$
- 2: $b = \log_2(CA)$
- 3: $e[1] = b$
- 4: $e[n2] = 0$
- 5: **for** ($i = 2; i < n2; i = i + 1$) **do**
- 6: $e[i] = \text{randomInt}() \bmod n2$
- 7: **end for**
- 8: **for** ($i = 1; i \leq n2 - 1; i = i + 1$) **do**
- 9: $X[i] = \text{random}() * 2^{e[i]}$
- 10: **end for**
- 11: **for** ($i = n2 + 1; i \leq n; i = i + 1$) **do**
- 12: $X[i] = 0$
- 13: **end for**
- 14: **for** ($i = n2 + 1; i \leq n; i = i + 1$) **do**
- 15: $e[i] = \text{roundToInt}(b * ((n - i) / n2))$
- 16: $X[i] = \text{random}() * 2^{e[i]} - \text{sommeExacte}(X)$
- 17: **end for**
- 18: $S = \text{sommeExacte}(X)$
- 19: $CE = \text{sommeExacte}(\text{abs}(X)) / \text{abs}(\text{sommeExacte}(X))$
- 20: $X = \text{permutationAléatoire}(X)$

L'algorithme précédent suppose l'existence d'une fonction *sommeExacte* qui permet de calculer l'arrondi correcte de la somme du vecteur passé comme argument. Pour implémenter cette fonction nous avons utilisé la bibliothèque MPFR [21] qui permet de simuler une précision étendue qui dépasse les limites matériel de la plate-forme utilisée. La bibliothèque MPFR utilise des nombres entiers pour simuler la mantisse d'un accumulateur d'une taille théoriquement infinie. Nous avons utilisé cette bibliothèque pour générer les données seulement car elle introduit un surcoût de calcul très important ce qui rend son utilisation très limitée en pratique.

3.3.2 Tests et génération des résultats

En utilisant l'algorithme 11 présenté dans la section précédente, nous avons généré de nombreux vecteurs de données, chacun d'entre eux se distingue avec un conditionnement différent. Le conditionnement est défini dans le domaine suivant $\{10^5, 10^6, \dots, 10^{34}\}$. Vu que l'algorithme de génération est basé sur la bibliothèque MPFR, qui est très lente en terme de temps d'exécution, nous avons choisi de générer seulement des vecteur de taille 1000. Dans le cas où nous avons besoin d'utiliser des plus larges vecteurs, il suffit de dupliquer autant de fois que nécessaire. Étant donné un vecteur p ayant $\text{cond}(\sum p_i) = C$, et un vecteur q qui est composé par la duplication du vecteur p pour n fois. nous avons :

$$\begin{aligned}\sum q_i &= n \times \sum p_i \\ \sum |q_i| &= n \times \sum |p_i|\end{aligned}$$

Ce qui implique que :

$$\text{cond}(\sum q_i) = \frac{\sum |q_i|}{|\sum q_i|} = \frac{n \times \sum |p_i|}{n \times |\sum p_i|} = \frac{\sum |p_i|}{|\sum p_i|} = \text{cond}(\sum p_i) = C.$$

En utilisant ce petite astuce pour éviter la génération de grands vecteur nous avons pu gagner beaucoup de temps, tout en pouvant tester les algorithmes sur des instances de grandes tailles.

Nous avons testé des tailles définies dans le domaine suivant $\{1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, 1000000\}$. Il est important de tester les algorithmes sur de différentes tailles et de différents conditionnements pour observer leurs comportements en terme de précision, de reproductibilité et performances. Les cinq algorithmes de sommation sont donc appliqués sur tous les vecteurs générés, et des fichiers contenant les résultats sous format d'une matrice Matlab sont générés pour être réutilisé par Matlab par la suite.

Les tests de la précision

Pour tester la précision d'un algorithme, il est nécessaire d'analyser les résultats qu'il fournit, par rapport à la somme correctement arrondi déjà associé avec chaque vecteur donné (que nous désignons par la suite par la somme exacte). La meilleure solution pour faire cela est de calculer l'erreur relative. Nous avons appliqué les cinq algorithmes sur tous les vecteurs que nous avons déjà générés, et nous avons par la suite sauvegardé les résultats et la somme exacte dans un fichier. À partir de ces données nous pouvions calculer l'erreur relative pour chaque calcul et pour chaque algorithme. Par la suite nous avons tracé en utilisant la fonction *plot* de Matlab un graphe contenant cinq courbes, chaque courbe représente l'évolution de l'erreur relative d'un algorithme par rapport au conditionnement des données sur lesquelles il est appliqué.

Les tests de la reproductibilité

La reproductibilité numérique d'un algorithme dans notre cas d'étude consiste en avoir toujours la même somme calculée d'un vecteur indépendamment de l'ordre de la sommation de ses éléments. Pour tester cette propriété très importante, pour chaque vecteur nous avons calculé dans un premier temps la somme, puis nous avons appliqué une permutation aléatoire de tous ses éléments, et ensuite nous avons calculé la somme du vecteur résultant. Les deux sommes sont par la suite stockés dans un fichier. Un graphe est tracé en utilisant ce fichier. Une courbe montrant la différence entre les deux résultats est tracée pour chaque algorithme. Si cette différence est nulle nous pouvons dire que l'algorithme donne des résultats reproductibles.

Les tests des performances

La notion de la performance d'un algorithme est définie théoriquement par sa complexité, et en réalité par le temps qu'il prend pour retourner un résultat, en d'autres mots son temps d'exécution.

La mesure du temps d'exécution d'un algorithme est une tâche difficile, et elle n'est jamais exacte. Mais, cette estimation doit être au moins suffisamment proche pour qu'elle soit crédible. Les processeurs récents fournissent des compteurs matériels, qui donnent des mesures fiables des performances. Nous avons utilisé l'instruction assembleur *rdtsc* pour accéder au registre qui compte le nombre de cycles du processeur. Cette dernière instruction est appelée avant et après l'appel de l'algorithme, et la différence entre le premier et la dernier appel est le nombre de cycles nécessaires pour l'exécution de l'algorithme. Nous pouvons diviser cette valeur par la cadence du processeur pour estimer le temps d'exécution en secondes. Afin d'obtenir des résultats plus exactes nous n'avons utilisé aucun API en se limitant par les instructions assembleur. Pour chaque algorithme, pour une taille de vecteur et un conditionnement donné, nous lançons le test plusieurs fois, et nous prenons le temps d'exécution minimal pour minimiser les sur-coût due aux interruptions système.

3.4 Implémentation

Nous avons parlé dans la première section de ce chapitre du standard IEEE-754 et de son adoption par les langages de programmation. Et après une longue recherche, le C apparaît le choix le plus adéquat pour implémenter notre projet, et nous avons justifié ce choix dans la section 3.2.1.

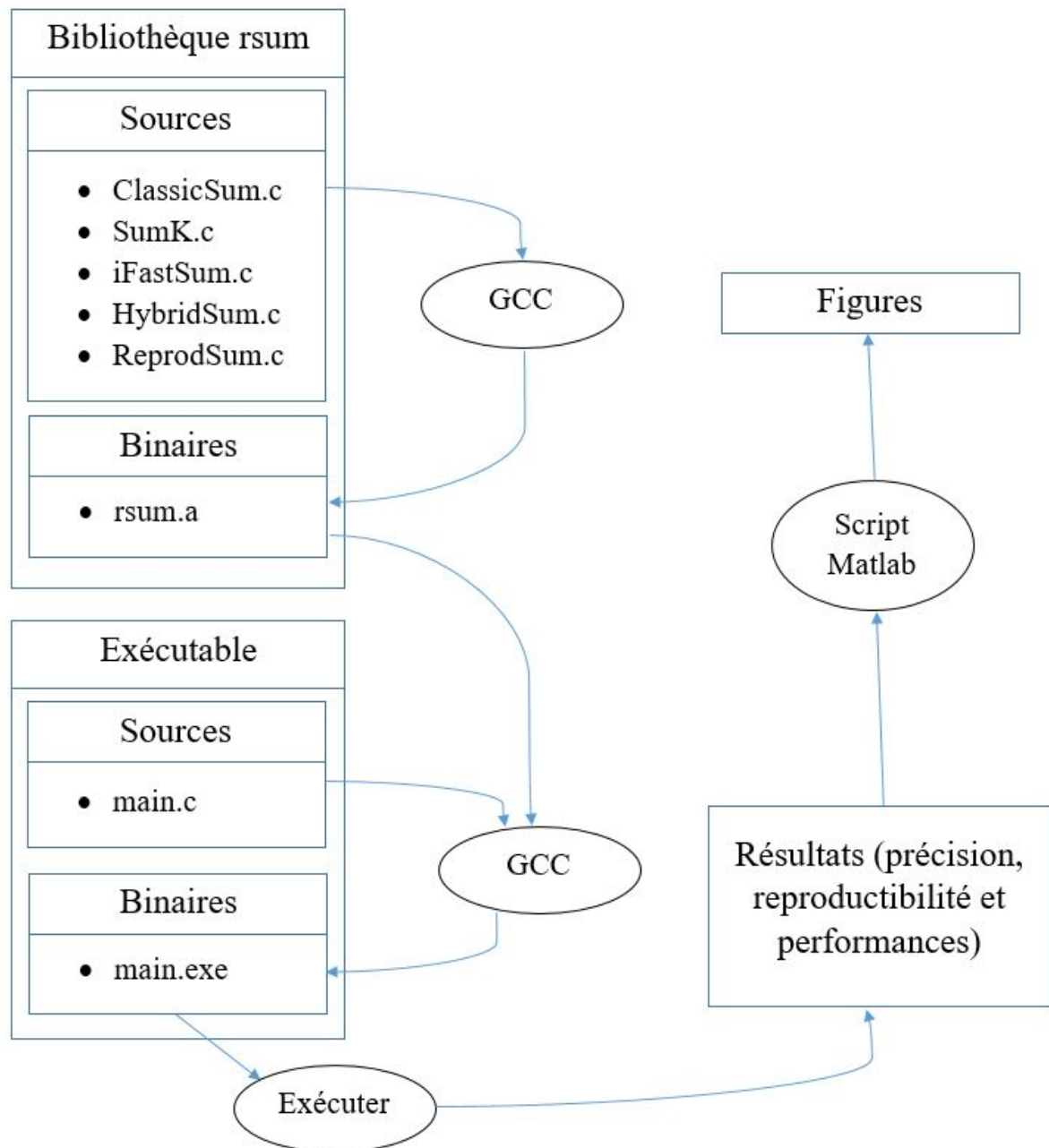


FIGURE 3.1 – Le processus de génération des résultats

Notre projet se compose de 4 parties :

1. Une bibliothèque C, qui définit les cinq algorithmes à tester.
2. Un exécutable (contenant la fonction main) utilisant la bibliothèque, qui parcourt et teste les algorithmes pour chaque taille et chaque conditionnement et

génère les résultats.

3. Un script Matlab, qui lit les résultats générés et trace les courbes pour faciliter la lecture et l'interprétation des résultats.
4. Des fichiers makefiles, qui permettent d'automatiser et simplifier tout ce processus. généralement ces makefiles sont générés automatiquement par des éditeurs comme code : :blocks ou Visual Studio. Mais dans notre cas, parce que nous utilisons deux différents langages de programmation, il nous a été plus pratique de lier les deux langages avec des makefiles personnalisés.

La figure 3.1 montre toutes les étapes par lesquelles passent nos algorithmes pour générer les figures que nous allons présenter dans le prochain chapitre.

3.5 Conclusion

Nous avons présenté dans ce chapitre une vue générale sur notre environnement de travail. Nous avons commencé par la présentation des outils que nous avons utilisés du début jusqu'à la fin de nos expérimentations. Nous avons ainsi présenté la méthode avec laquelle nous avons généré les données, et notre environnement de test (matériel et logiciel). Nous estimons que ces informations sont très utiles pour tout chercheur qui veut reproduire ou prendre nos résultats comme référence. Le dernier chapitre sera réservé pour la présentation et l'interprétation des résultats.

CHAPITRE 4

RÉSULTATS & INTERPRÉTATION

4.1 Introduction

Dans ce dernier chapitre, nous allons présenter les résultats des différents testes que nous avons faits. La première partie concentre sur les testes de la précision, la deuxième sur les testes de la reproductibilité numérique, et la troisième partie s'intéresse aux testes des performances. Le chapitre sera conclu avec une synthèse où nous allons essayer de résumer l'ensemble des informations extraites des différents résultats.

4.2 Résultats de précision

Dans cette section nous allons discuter les résultats des cinq algorithmes de sommation en terme de précision. La figure 4.1 contient deux graphes, avec chaque graphe comprenant cinq courbes. Chacun de ces courbes représente l'évolution de l'erreur relative (montrée sur l'axe des Y) d'un des cinq algorithmes par rapport au conditionnement du problème (sur l'axe des X), et cela pour un vecteur de taille 10^3 . Le graphe 4.1a illustre le cas où les algorithmes SumK et ReprodSum sont exécutés

pour un $K = 2$. Tout d’abord, nous pouvons constater que pour un conditionnement inférieur à 10^{11} tous les algorithmes à la part de ClassicSum retournent des résultats avec une erreur relative nulle (Dans le graphe l’erreur relative nulle est représentée par la droite définie par la fonction $Y = -17$). Ensuite, à partir du conditionnement 10^{11} l’erreur relative de ReprodSum commence à accroître linéairement. C’est le même cas pour Sumk à partir du conditionnement 10^{16} . iFastSum et HybridSum semblent pouvoir donner des résultats avec des erreurs relatives nulles même pour les conditionnements élevés. Enfin, nous pouvons constater clairement que ClassicSum ne rend aucun résultat correctement arrondi même pour les problèmes bien conditionnés.

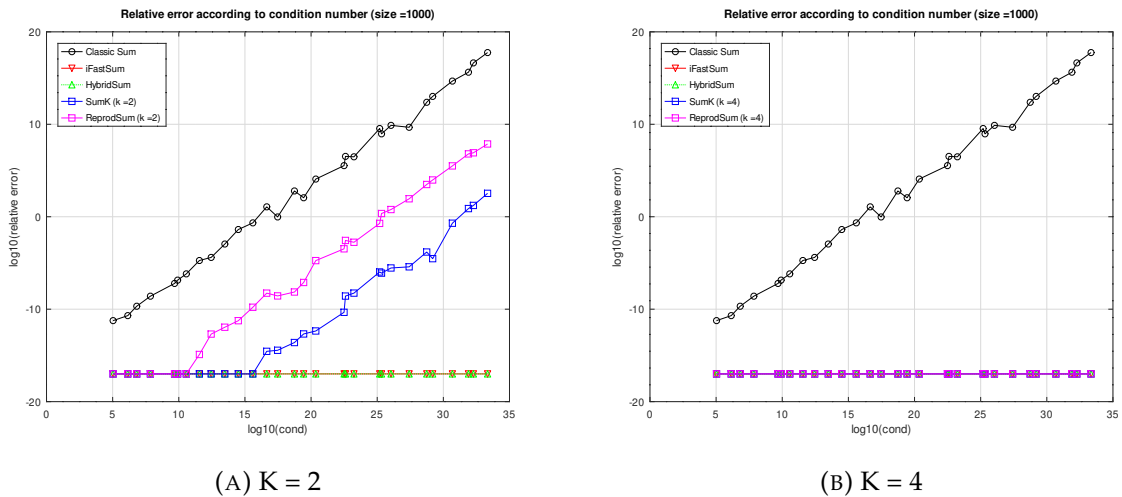


FIGURE 4.1 – L’évolution de l’erreur relative par rapport au conditionnement et la valeur de k pour un vecteur de taille 10^3

Le graphe 4.1b illustre le cas où les algorithmes SumK et ReprodSum sont exécutés pour un $K = 4$. Logiquement, les courbes de ClassicSum, iFastSum, ainsi que celui de HybridSum gardent le même comportement. Les seuls changements sont constatés au niveau des courbes de Sumk et ReprodSum qui dépendent directement de la valeur changée K . Nous constatons une stagnation dans les courbes de ces deux algorithmes pour tous les conditionnements testés.

Les deux graphes nous montrent donc que ClassicSum n’est pas un algorithme de sommation précis, et qu’il perd de la précision avec l’augmentation du conditionnement de ses données. En outre, Sumk et ReprodSum fournissent des résultats

correctement arrondis pour les petits conditionnements, et semblent perdre leur précision pour des conditionnements plus élevés. Par contre, iFastSum et HybridSum fournissent toujours des résultats fiables même pour les conditionnements élevés.

Ces conclusions restent valides lorsque le jeu de données est plus large. La seule différence est que les vecteurs les plus larges sont plus affectés par le conditionnement, comme le montre la figure 4.2. Partant de ce fait nous pouvons dire que l'augmentation de jeux de données augmente l'effet du conditionnement sur l'erreur relative.

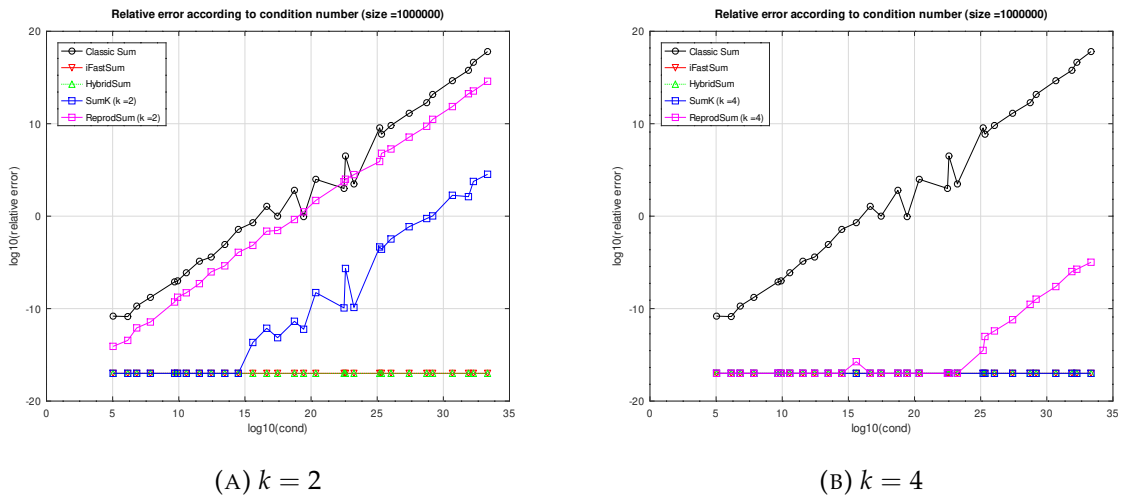


FIGURE 4.2 – L'évolution de l'erreur relative par rapport au conditionnement et la valeur de k pour un vecteur de taille 10^6

La figure 4.3 contient deux graphes, avec chaque graphe comprenant trois courbes. Chacun de ces courbes représente l'évolution de l'erreur relative par rapport au conditionnement du problème et la valeur de k , et cela pour un vecteur de taille 10^3 .

Le graphe 4.3a est dédié à l'algorithme Sumk. Tout d'abord nous remarquons une stabilité dans les trois courbes pour les conditionnements inférieurs à 10^{16} . Ensuite, à partir de conditionnement 10^{16} la courbe représentant les résultats pour un $k = 2$ entre dans une progression haussière. Pareil pour la courbe pour $k = 3$ qui suit la même tendance haussière à partir du conditionnement 10^{32} .

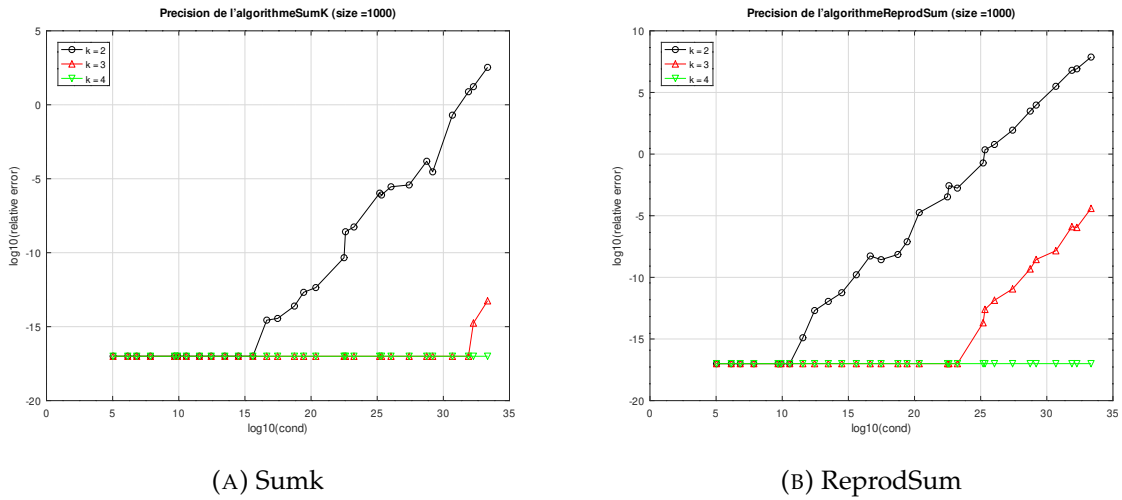


FIGURE 4.3 – L'évolution de l'erreur relative par rapport au conditionnement et la valeur de k pour un vecteur de taille 10^3

Le graphe 4.3b est dédié à l'algorithme ReprodSum. Son comportement est similaire à celui de Sumk. Tout d'abord tous les courbes se stabilisent sur la droite des erreurs relatives nulles. Ensuite, à partir du conditionnement 10^{11} la courbe pour $k = 2$ commence à accroître linéairement, pareil pour celle de $k = 3$ à partir du conditionnement 10^{24} .

Nous pouvons conclure de ces deux graphes précédents que l'augmentation de la valeur de k réduit la valeur de l'erreur relative. En d'autres termes, refaire le processus de distillation (pour le cas de Sumk) et le processus de l'extraction (pour le cas de ReprodSum) réduit l'erreur relative.

La figure 4.4 contient deux graphes. Le premier 4.4a compare l'erreur relative des cinq algorithmes pour des tailles variables et un conditionnement fixé à 10^8 , par contre le deuxième 4.4b les compare aussi pour des tailles différentes mais cette fois-ci avec le conditionnement fixé à 10^{16} . À partir du graphe 4.4a nous pouvons constater qu'au début tous les algorithmes à part de ClassicSum affichent un comportement stable pour les vecteurs de tailles inférieures ou égales à 10^4 . Puis la courbe de ReprodSum commence à évoluer pour les vecteurs de tailles supérieures, par contre iFastSum et HybridSum gardent la même progression stable pour le reste des tailles testées. Finalement, nous pouvons remarquer une stagnation de la courbe de ClassicSum mais

dans une grande valeur d'erreur relative.

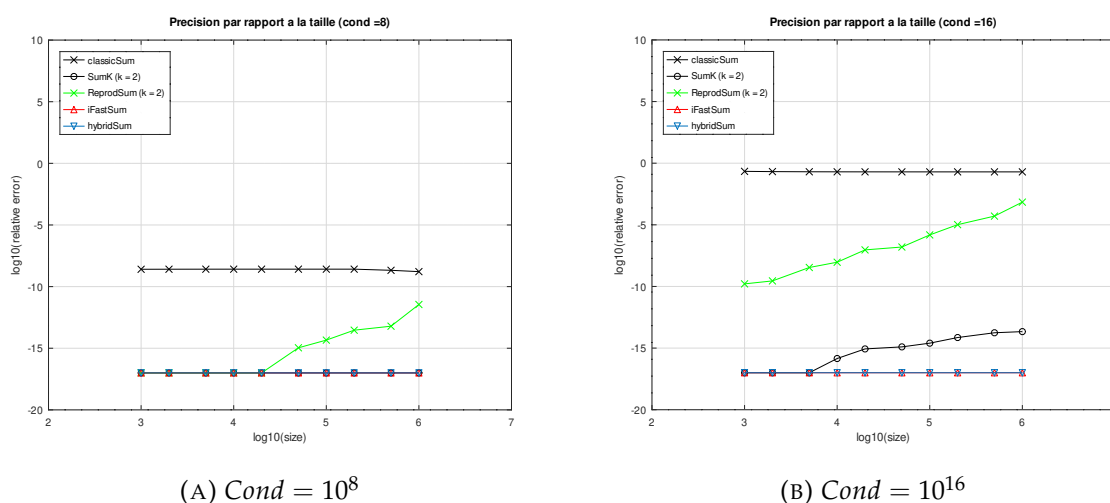


FIGURE 4.4 – L'évolution de l'erreur relative par rapport à la taille du vecteur et pour un $k = 2$ et un conditionnement fixé

Du graphe 4.4b, nous pouvons apercevoir un comportement similaire avec des différences dans les valeurs de l'erreur relatives enregistrées et les tailles dans lesquelles les courbes changent de comportement. Premièrement, Nous constatons pour toutes les tailles testées une stagnation dans la courbe de ClassicSum dans un niveau supérieur d'erreur relative, et une progression linéaire pour la courbe de ReprodSum. Alors que les courbes iFastSum et HybridSum stagnent sur la droite représentant l'erreur relative nulle pour l'ensemble des données testées. La courbe de SumK qui a montré une stabilité sur la droite représentant l'erreur relative nulle au début pour les vecteurs de petites tailles, mais elle a fini par accroître linéairement pour les vecteurs plus larges.

Bref, ClassicSum apparaît incapable de donner des résultats corrects même pour les vecteurs de petites tailles avec un bon conditionnement. Sumk et ReprodSum sont affectés par le volume de jeux du données comme pour son conditionnement, et ils tendent à perdre leur précision avec un rythme linéaire. iFastSum et HybridSum semblent capables de retourner des résultats fiables peu importe la taille ou le conditionnement des données sur lesquelles ils sont appliqués.

4.3 Reproductibilité numérique

Nous avons déjà défini la reproductibilité numérique de la sommation plusieurs fois dans ce mémoire, qu'elle s'agit d'avoir les mêmes résultats de la sommation des mêmes données indépendamment de l'ordre dans lequel nous effectuons l'accumulation. C'est pour cela les testes seront logiquement des comparaisons entre les résultats de deux sommations sur le même jeu de données, mais avec une permutation aléatoire des éléments du vecteur à chaque fois.

La figure 4.5 contient deux graphes, chacun contenant cinq courbes. Chaque courbe illustre la progression de la différence entre les résultats des deux sommations par rapport au conditionnement du vecteur sur lequel elles sont appliquées. Dans le premier graphe le vecteur est de taille 10^3 , alors que dans le deuxième la taille est fixée à 10^6 .

Nous pouvons constater que ClassicSum ne fournit jamais des résultats similaires peu importe la taille ou le conditionnement du vecteur. La différence entre les résultats des deux sommations affiche un comportement progressif linéairement avec le conditionnement. Ainsi, nous pouvons remarquer que les valeurs des différences sont plus grandes pour le vecteur de taille 10^6 que pour le vecteur de taille 10^3 . Par exemple : pour un conditionnement $= 10^8$ la première différence est égale à $3.0092 \cdot 10^{-9}$ alors que la deuxième est égale à $5.9815 \cdot 10^{-5}$.

Cela nous laisse conclure que ClassicSum n'est pas reproductible, et le gap entre ses résultats s'amplifie avec la l'augmentation de la taille et le conditionnement de la somme.

À partir du même figure, nous pouvons clairement remarquer une stabilité des trois courbes illustrants iFastSum, HybridSum, et ReprodSum. Ces trois courbes se correspondent entre eux ainsi qu'avec la droite qui représente la différence nulle ($Y = -17$), pour tous les conditionnements testés et pour les deux tailles des vecteurs. Cela nous permet d'affirmer que iFastSum, HybridSum, et ReprodSum sont toujours reproductibles.

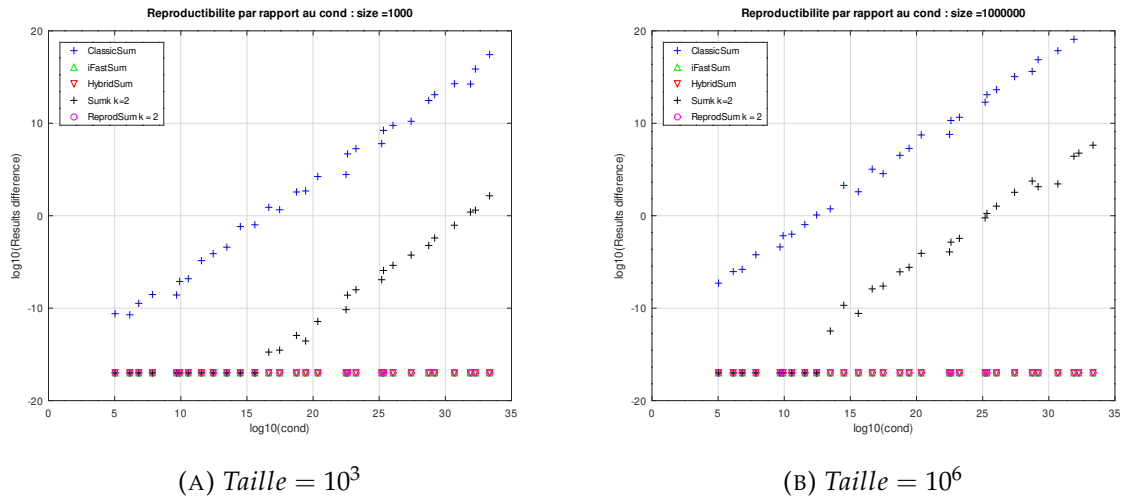


FIGURE 4.5 – L'évolution de la différence entre les résultats des deux sommations par rapport au conditionnement du vecteur pour un $k = 2$ et une taille fixe

La courbe de Sumk (pour un $k = 2$) stagne au début sur la droite représentant la différence nulle (la droite dont l'équation est $Y = -17$). Ensuite, à partir d'un certain conditionnement la courbe commence à progresser linéairement. Pour le vecteur de taille 10^3 ce point de changement est le conditionnement 10^{16} . Pourtant, pour le vecteur de taille 10^6 ce point est le conditionnement 10^{13} .

Sumk est un algorithme qui se caractérise par son paramètre k , qui est le nombre de d'itérations de distillations appliquées sur le vecteur. Afin de mieux analyser la reproductibilité de cet algorithme nous présentons dans la figure 4.6 deux graphes qui comparent l'influence de la valeur de k sur la différence entre les résultats des deux sommations. Les résultats du premiers graphe sont obtenus à partir d'un vecteur de 10^3 éléments, et ceux du deuxième sont à partir d'un vecteur de 10^6 éléments.

À partir du graphe 4.6a Nous pouvons remarquer que pour un $k = 2$ la différence stagne sur la barre de la différence nulle ($Y = -17$) jusqu'au conditionnement 10^{16} . Puis, elle commence à remonter linéairement. Pour un $k = 3$ la différence se stabilise toujours sur la barre de la différence nulle, puis elle remonte linéairement, mais cette fois-ci pour un conditionnement de 10^{32} . Pour un $k = 4$ la différence entre les résultats reste toujours nulle et la courbe ne s'éloigne pas de la droite $Y = -17$. Le graphe

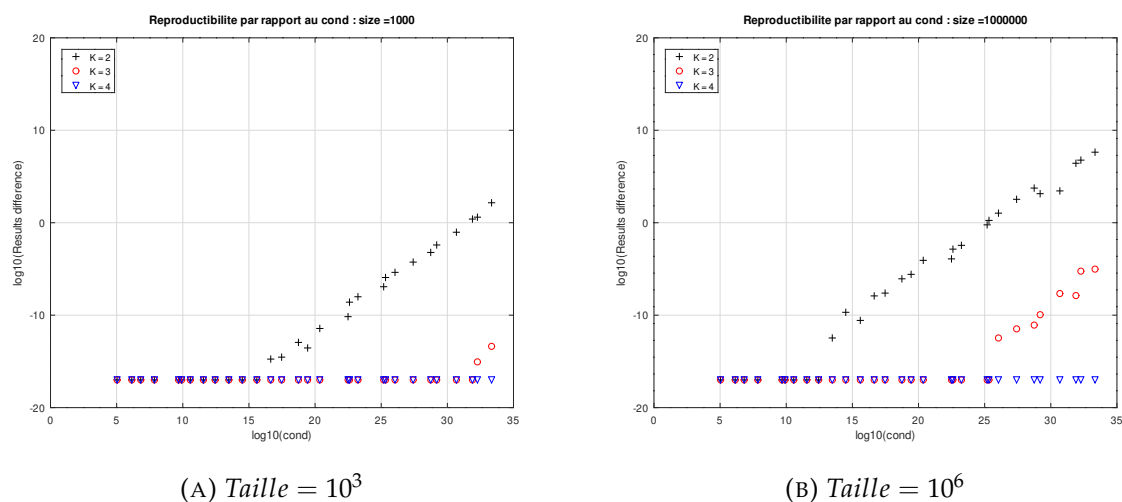
(A) Taille = 10^3 (B) Taille = 10^6

FIGURE 4.6 – L'évolution de la différence entre les résultats des deux sommations par rapport au conditionnement du vecteur et à la valeur de k pour un vecteur de taille 10^3

4.6b affiche un comportement similaire avec la seule différence dans les valeurs de conditionnement à partir de lesquelles les courbes commencent à remonter.

Nous pouvons conclure que Sumk peut fournir des résultats reproductibles, et il perd cette propriété importante à partir de certains conditionnements, leurs valeurs sont variables selon la valeur de k et la taille du vecteur de données (l'augmentation de la taille des données diminue la valeur du point de changement, alors que l'augmentation de la valeur de k l'augmente).

La précision vs la reproductibilité numérique

Dans les deux dernières sections nous avons étudié la précision et la reproductibilité numérique des cinq algorithmes individuellement. Alors que maintenant nous allons étudier la relation qui réside entre ces deux propriétés importantes. Commençons par iFastSum et HybridSum qui ont prouvé durant les tests intensifs qu'ils ont subi, d'être à la fois précis et reproductibles. Ici avoir l'arrondi correct a garanti la propriété de la reproductibilité. Vu que les deux algorithmes fournissent toujours des résultats correctement arrondis, peu importe l'ordre des opérations le même résultat sera toujours reproduit. La preuve de cela peut être remarquée lors de l'analyse du comportement de l'algorithme Sumk. Sumk qui est plus précis que la sommation

naïve, mais qui n'est pas toujours correctement arrondi, et qui gagne plus de précision à chaque fois la valeur de son paramètre k augmente. Pour des grandes valeurs de k , Sumk peut assurer des résultats correctement arrondis et par conséquent reproductibles (pour certaines tailles et certains conditionnements). D'après nos résultats ReprodSum est un algorithme reproductible, mais qui n'est pas toujours précis. Ici, avoir la reproductibilité ne lui a pas garanti la précision, mais pour certaines application, la précision est demandée pour un certains nombre de chiffres significatifs seulement et ce n'est pas nécessaires d'avoir l'arrondi correcte du résultat, par contre la non-reproductibilité des résultat reste considérée comme une anomalie dans le code, ce qui complique certaines étapes dans le cycle de vie d'un logiciel (certification, débogage ... etc.), surtout si nous n'arrivons pas à faire la différence entre les inconsistances liées à des problèmes numériques, et celles qui sont liées à des erreurs dans le code. Ce type d'applications montre l'utilité d'un algorithme comme ReporodSum qui assure la reproductibilité sans garantir l'arrondi correct du résultat. Finalement, nous arrivons à ClassicSum qui n'est ni précis ni reproductible.

Nous pouvons déduire qu'un algorithme qui garantit l'arrondi correct est un algorithme reproductible, mais l'inverse n'est pas vrai.

4.4 Performances

Afin de comparer les performances des cinq algorithmes, nous avons utilisé la notion du temps d'exécution normalisé. Ce dernier est un terme avec lequel nous désignons le quotient du temps d'exécution d'un algorithme quelconque par le temps d'exécution de ClassicSum. Nous l'avons utilisé pour pouvoir comparer le sur-coût de chaque algorithme par rapport à ClassicSum. Notre comparaison sera basée principalement sur des graphes représentant l'évolution du temps d'exécution normalisé par rapport à la taille du vecteur. Un couple de paramètres sera aussi pris en compte, qui est le conditionnement du problème, ainsi que la valeur de k pour les deux algorithmes Sumk et ReprodSum. Alors chaque graphe va illustrer le phénomène pour un

conditionnement et un k spécifiques.

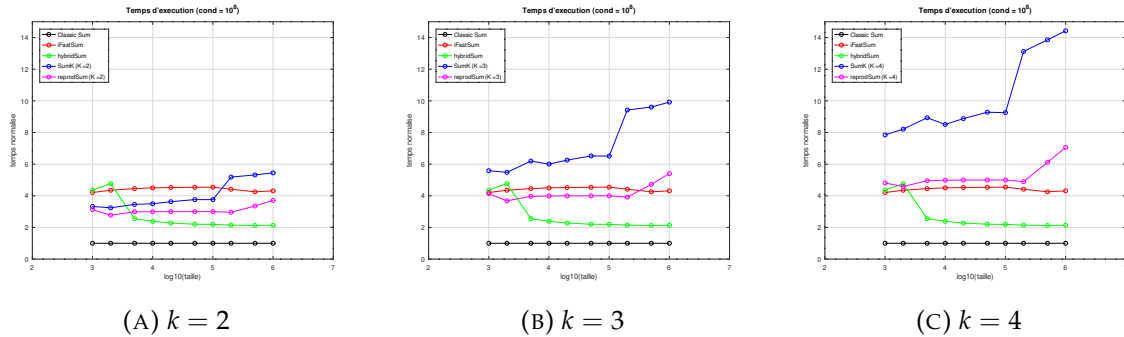


FIGURE 4.7 – L'évolution du temps d'exécution normalisé par rapport à la taille du vecteur et pour un $conditionnement = 8$ et pour des différentes valeurs de k

Le graphe 4.7a illustre le phénomène pour un $k = 2$ et un $conditionnement = 10^8$. Tout d'abord, nous avons fixé la courbe de ClassicSum comme un repère lorsque nous avons normalisé le temps d'exécution des algorithmes. Nous remarquons pour Sumk une progression linéaire lente du temps d'exécution pour les vecteurs de taille inférieure à 10^5 , puis et pour les tailles supérieures nous remarquons une accélération dans sa progression. Cette accélération est due à la nature itérative de l'algorithme, puisqu'il aura besoin de recharger le contenu de la mémoire cache à partir de la mémoire centrale pour les larges vecteurs, ce qui ajoute un temps supplémentaire sur le temps d'exécution. En observant les courbes 4.7b, 4.7c illustrant le phénomène pour $k = 3$, $k = 4$ respectivement, nous remarquons un comportement similaire, mais avec des coûts plus élevés à chaque fois la valeur de k augmente. Pour l'algorithme ReprodSum, il affiche un comportement progressif linéaire lent similaire à celui de Sumk, avec un sur-coût aussi pour les vecteurs de grandes tailles. La seule différence est que ReprodSum est caractérisé par une complexité $((4(k - 1) + 3)n$ opérations) inférieure à celle de SumK $((6(k - 1) + 1)n$ opérations), ce qui est traduit par un coût moins élevé en terme du temps d'exécution. Les performances de Sumk, et ReprodSum ne sont pas affectées par le conditionnement du problème, nous pouvons remarquer dans la figure 4.8 que les deux algorithmes montrent le même sur-coût indépendamment du conditionnement.

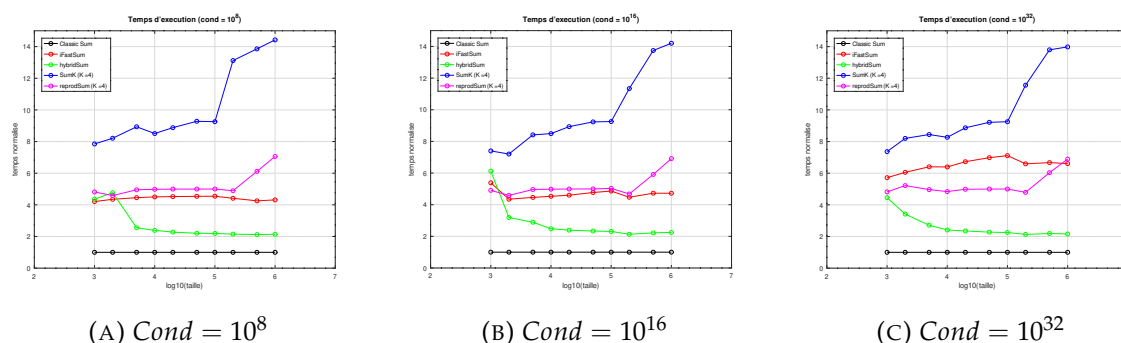


FIGURE 4.8 – L'évolution du temps d'exécution normalisé par rapport à la taille du vecteur et pour un $k = 4$ et pour des conditionnements fixes

De la figure 4.8, nous pouvons analyser le comportement de iFastSum, qui affiche une progression linéaire aussi, mais cette progression n'est pas remarquable pour les petits conditionnements (voir le graphe 4.8a). La vitesse de sa progression augmente avec l'augmentation de la taille et le conditionnement du problème (Dans le graphe 4.8c la progression est remarquable cette fois-ci). Finalement, La courbe de HybridSum affiche une tendance baissière, à partir du coût le plus élevé pour l'algorithme, qui est enregistré pour le vecteur avec la plus petite taille. La tendance baissière de la courbe de HybridSum est similaire à la celle de la courbe de la fonction inverse $f(x) = \frac{1}{x}$, avec la seule différence que la courbe de la fonction inverse rapproche de la droite $Y = 0$, alors que la courbe de HybridSum rapproche de la droite $Y = 2$, qui représente le double du temps d'exécution de ClassicSum (et aussi le temps minimal pour HybridSum ici). Ce comportement qui peut paraître bizarre est à cause du coût fixe de l'opération finale de l'accumulation (ici en utilisant iFastSum), qui est plus important que le coût de l'opération de transformation des petits vecteurs. En augmentant la taille des vecteurs, ce coût fixe devient de plus en plus négligeable devant le coût de transformation des grands vecteurs.

Nous pouvons conclure qu'en terme de performances, HybridSum est le meilleur choix derrière ClassicSum surtout pour les problèmes de grandes tailles et mal-conditionnés. iFastSum le suit, mais avec des performances plus modestes par rapport à HybridSum pour les problèmes de grandes tailles et mal-conditionnés. ReprodSum est très proche

de *iFastSum*. Alors que *SumK* reste la solution la plus coûteuse parmi les cinq (pour $k = 4$). Il est possible pour améliorer les performances d'utiliser *Sumk* avec des petites valeurs pour le paramètre k , ce qui permet d'effectuer moins d'itérations sur le vecteur. Mais les sections précédentes ont montré que ça aura un impact important sur la qualité numérique des résultats. Donc c'est un choix qui n'est pas vraiment à recommander.

4.5 Conclusion

Dans ce chapitre, nous avons comparé les cinq algorithmes en terme de la précision et de la reproductibilité de leurs résultats, et de leurs performances. À la fin de chaque comparaison, nous avons essayé de sélectionner le meilleur algorithme. Maintenant, nous allons essayer de récapituler nos conclusions. *ClassicSum* est la solution naïve du problème de la sommation, il est le plus rapide à retourner un résultat. Malheureusement, ses résultats ne sont ni précis ni reproductibles, même pour les problèmes à petites tailles et bien conditionnés. *SumK* un algorithme que son rendement dépend directement de son paramètre k , il peut fournir des résultats corrects et reproductibles, mais au prix de sa performance. Dans certains cas des problèmes testés, *Sumk* écoule un temps équivalent à 14 fois le temps nécessaire pour l'exécution de *ClassicSum*. *ReprodSum* un algorithme reproductible, avec de moyennes performances, mais qui perd la propriété importante de la précision. *iFastSum*, qui est originalement une version améliorée de *Sumk*, retourne toujours des résultats correctement arrondis, qui sont implicitement reproductibles. Ses performances sont par contre moyennes. *HybridSum* un autre algorithme précis, et qui assurent aussi la reproductibilité de ses résultats, mais avec de meilleures performances.

CONCLUSION GÉNÉRALE

La sommation flottante est une opération essentielle pour le calcul scientifique. Cette opération qui semble très simple à comprendre et à implémenter, pourrait en pratique poser plusieurs problèmes liés d'un côté à la précision des résultats, et d'un autre à sa reproductibilité numérique. Ces deux problèmes surgissent à cause de l'opération d'arrondi effectuée après chaque addition de deux nombres flottants. En effet, parmi les conséquences de ces arrondis, nous pouvons observer que plusieurs propriétés mathématiques sont perdues sur l'arithmétique flottante. Par exemple la distributivité de la multiplication sur l'addition n'est plus valide. Ce qui nous intéresse plus dans le cadre de ce travail est la non-associativité de l'addition flottante, ce qui engendre le problème de la non-reproductibilité numérique lors de l'accumulation de n nombre. Pour répondre à ces problèmes, plusieurs chercheurs ont contribué avec leurs travaux dans les efforts de l'amélioration de cette opération. Dans ce mémoire nous avons étudié certains de ces travaux, afin de pouvoir répondre à la question importante suivante "Quel le meilleur algorithme pour la sommation flottante en terme de précision, reproductibilité, et performances?". Maintenant, après une étude comparative basée sur une démarche scientifique, nous pouvons dire qu'aucun algorithme n'est toujours parfait! Seulement l'utilisateur de la sommation peut décider l'algorithme qu'il peut utiliser, selon ses besoins et objectifs. Pour un utilisateur qui cible la précision iFastSum et HybridSum sont ses meilleures choix. Pour un autre qui cible la

reproductibilité `ReprodSum` peut s'avérer suffisant. Ces choix sont faits en ignorant l'aspect des performances de chaque algorithme. Généralement, nous nous intéressons à plusieurs aspects à la fois. Ces trois aspects ne sont pas les seuls à prendre en considération, la taille et le conditionnement du problème à traiter doivent être aussi prise en compte avant faire le choix. Donc au lieu de donner une recommandation d'un algorithme, nous estimons qu'il est plus pratique de citer les avantages et les inconvénients de chacune des solutions étudiées, et aussi donner notre point de vue sur le contexte idéal d'utilisation de chaque solution.

ClassicSum : c'est l'algorithme le plus simple à comprendre et implémenter, son avantage est qu'il est aussi le plus rapide parmi les solutions étudiées, mais malheureusement il n'offre aucune garantie sur la qualité numérique des résultats fournies. Il peut être utilisé dans des applications qui manipulent des données dont les sommes sont bien conditionnées, et qui acceptent d'avoir un nombre réduit de chiffres significatifs corrects sans chercher à reproduire les résultats jusqu'au dernier bit en cas de portabilité du code ou l'introduction de parallélisme.

SumK : un algorithme avec une précision paramétrable, basé sur le principe de la distillation. L'algorithme `SumK` fait plusieurs itérations sur le vecteur d'entrée pour améliorer le résultat. Il est très pratique à utiliser sur des sommes ayant un conditionnement modéré [$10^3 - 10^{16}$]. Sur des somme ayant un conditionnement inférieur à 10^{16} , `SumK` avec $k = 2$ peut assurer des résultats numériques confiantes, mais la reproductibilité numérique n'est pas toujours assurée jusqu'au dernier bit. L'un des avantages de cet algorithme est qu'il est possible d'augmenter sa précision si nécessaire tout simplement en augmentant la valeur de son paramètre k . Mais selon nos expériences au lieu d'utiliser `SumK` avec $k > 2$, il est plus intéressant d'utiliser un algorithme comme `HybridSum` ou `iFastSum` à cause du sur-coût introduit en terme de temps de calcul due à l'augmentation de la valeur de k .

ReprodSum : un algorithme intéressant dans des applications où il est nécessaire de passer par une étape de validation ou certification qui exige la reproductibilité des résultats numériques sans nécessité d'avoir des résultats corrects jusqu'au dernier bit. ReprodSum est aussi paramétrable avec un argument k qui permet d'augmenter la précision contre un sur-coût en terme de temps de calcul. Mais comme pour SumK, et pour les mêmes raisons, nous recommandons d'utiliser iFastSum, ou bien HybridSum au lieu d'augmenter la valeur de k (au delà de $k = 2$) pour ReprodSum.

HybridSum et iFastSum : ces deux algorithmes sont numériquement équivalents. Ils assurent tout les deux des résultats correctement arrondis, et par conséquent reproductibles. Par contre, à cause de la différence dans leurs principes de fonctionnement, ils montrent des sur-coûts différents en terme de temps d'exécution. iFastSum est plus efficace pour les petites sommes ($n < 5000$), alors que HybridSum est plus intéressant pour les sommes des grands vecteurs. Nous recommandons l'utilisation de ces algorithmes pour des problèmes qui sont mal conditionnés ou dans le cas où nous cherchons l'arrondi correct sans avoir une idée sur le conditionnement.

Ce sujet de recherche reste une zone très vaste pour les scientifiques et les experts. L'optimisation des solutions existantes est un perspective important. Nous pouvons améliorer soit une solution complètement, ou même une partie importante de la solution par exemple le processus de la distillation dans Sumk et iFastSum. Les solutions que nous avons abordé sont des solutions séquentielles, qui s'exécutent sur une seule machine. Pourtant des solutions parallèles seront plus intéressantes, surtout que la sommation flottante est utilisée dans des systèmes distribués, tel que les systèmes météorologiques. Plusieurs horizons de recherches sont disponibles, et il y aura toujours d'autres qui émergeront des problèmes rencontrés par les scientifiques.

BIBLIOGRAPHIE

- [1] Theodorus Jozef DEKKER. « A floating-point technique for extending the available precision ». In : *Numerische Mathematik* 18.3 (1971), p. 224-242.
- [2] James DEMMEL et Hong Diep NGUYEN. « Fast reproducible floating-point summation ». In : *2013 IEEE 21st Symposium on Computer Arithmetic*. IEEE. 2013, p. 163-172.
- [3] *Floating-Point Exceptions (GCC)*. https://www.gnu.org/software/libc/manual/html_node/FP-Exceptions.html. Accessed : 2020-09-05.
- [4] *GNU Make*. <https://www.gnu.org/software/make/manual/make.html>. Accessed : 2020-10-05.
- [5] Yun HE et Chris HQ DING. « Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications ». In : *The Journal of Supercomputing* 18.3 (2001), p. 259-277.
- [6] « IEEE Standard for Floating-Point Arithmetic ». In : *IEEE Std 754-2008* (2008), p. 1-70.
- [7] *Infinity and NaN (GCC)*. https://www.gnu.org/software/libc/manual/html_node/Infinity-and-NaN.html. Accessed : 2020-09-05.
- [8] Brian W KERNIGHAN, Dennis M RITCHIE et al. *The C programming language*. T. 2. prentice-Hall Englewood Cliffs, NJ, 1988.

-
- [9] Donald E KNUTH. *Art of computer programming, volume 2 : Seminumerical algorithms*. Addison-Wesley Professional, 2014.
- [10] Philippe LANGLOIS, Rafife NHEILI et Christophe DENIS. « Recovering numerical reproducibility in hydrodynamic simulations ». In : *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*. IEEE. 2016, p. 63-70.
- [11] Jean-Michel MULLER et al. *Handbook of floating-point arithmetic*. T. 1. Springer, 2018.
- [12] Takeshi OGITA, Siegfried M RUMP et Shin'ichi OISHI. « Accurate sum and dot product ». In : *SIAM Journal on Scientific Computing* 26.6 (2005), p. 1955-1988.
- [13] *Optimize Options (Using the GNU Compiler Collection (GCC))*. <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>. Accessed : 2020-10-05.
- [14] Michael L OVERTON. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.
- [15] Katsuhisa OZAKI et al. « Generalization of error-free transformation for matrix multiplication and its application ». In : *Nonlinear Theory and Its Applications, IEICE* 4.1 (2013), p. 2-11.
- [16] Eric S. ROBERTS. *The art and science of C - a library-based introduction to computer science*. Addison-Wesley, 1995. ISBN : 978-0-201-54322-3.
- [17] Siegfried M RUMP, Takeshi OGITA et Shin'ichi OISHI. « Accurate floating-point summation part I : Faithful rounding ». In : *SIAM Journal on Scientific Computing* 31.1 (2008), p. 189-224.
- [18] James E. SMITH. « A Study of Branch Prediction Strategies ». In : *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ISCA '98. Barcelona, Spain : Association for Computing Machinery, 1998, 202-215. ISBN : 1581130589. DOI : 10.1145/285930.285980. URL : <https://doi.org/10.1145/285930.285980>.
- [19] Pat H STERBENZ. « Floating-point computation ». In : (1974).

-
- [20] Michela TAUFER et al. « Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs ». In : *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, p. 1-9.
- [21] *The GNU MPFR Library*. <https://www.mpfr.org/>. Accessed : 2020-09-23.
- [22] Yong-Kang ZHU et Wayne B HAYES. « Correct rounding and a hybrid approach to exact floating-point summation ». In : *SIAM Journal on Scientific Computing* 31.4 (2009), p. 2981-3001.