Contribution to error analysis of algorithms in floating-point arithmetic

Antoine Plet

Numéro National de Thèse : 2017LYSEN038

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée par
l'Ecole Normale Supérieure de Lyon

Ecole Doctorale 512
Ecole Doctorale en Informatique et Mathématiques de Lyon

Spécialité de doctorat :
Informatique

Soutenue publiquement le 07/07/2017, par :
Antoine PLET

Contribution to error analysis of algorithms in floating-point arithmetic

Contribution à l'analyse d'algorithmes en arithmétique à virgule flottante

---

Thèse soutenue après avis de :

BOLDO Sylvie,	Chargée de recherche Inria,	Rapporteuse
TANG Ping Tak Peter	PhD, Principal Engineer, Intel Corporation,	Rapporteur

Devant le jury composé de :

BOLDO Sylvie,	Chargée de recherche Inria,	Rapporteuse
ERHEL Jocelyne,	Directrice de recherche Inria,	Examinatrice
MARTEL Matthieu,	Professeur, Université de Perpignan Via Dominitia,	Examinateur
LOUVET Nicolas,	MCF, Université Claude Bernard Lyon 1,	Co-encadrant
MULLER Jean-Michel,	Directeur de recherche CNRS,	Directeur

Contents

Sommaire

Résumé en français par chapitre

Introduction générale

Contexte et motivations

L'arithmétique à virgule flottante est utilisée pour approcher les calculs sur les nombres réels, à partir d'un nombre de chiffres significatifs fini et fixe. Elle est disponible sur la plupart des ordinateurs actuels et fournit de bons compromis entre la vitesse d'exécution et la précision des calculs. De plus, cette arithmétique est rigoureusement spécifiée par le standard IEEE 754 [17] qui détermine le comportement de chaque opération élémentaire en virgule flottante (telles que $+$, $-$, $\times$ et $\div$). Cependant, une conséquence bien connue de la précision finie est que chaque opération élémentaire peut engendrer une erreur d'arrondi. Ainsi, même si chaque opération de base est aussi précise que possible, le résultat d'une succession d'opérations en arithmétique flottante peut être très différent du résultat exact. De ce point de vue, deux situations peuvent intervenir :

- dans le cas le plus favorable, les erreurs d'arrondi générées par chaque opération se compensent (voir par exemple [16, §1.14]), ce qui conduit à un résultat très précis ;
- elles peuvent aussi s'accumuler ou être amplifiées au cours du calcul, entraînant une forte perte de précision. Dans le pire cas, le résultat calculé peut être complètement faux : son signe peut être incorrect, par exemple, à cause de cancellations catastrophiques.

Dans ce contexte, il est très appréciable de pouvoir estimer l'erreur entachant le résultat calculé. Par exemple, une erreur relative strictement inférieure à 1 suffit pour garantir le signe du résultat, ce qui peut être suffisant pour certaines applications (voir par exemple [33]). Rappelons que deux types d'erreur peuvent être considérés : si on note respectivement x et $\widehat{x}$ les résultats exact et calculé, l'erreur absolue s'exprime par $|x-\widehat{x}|$ et l'erreur relative par $|x-\widehat{x}|/|x|$ (en supposant que x ne s'annule pas). La plupart du temps, l'erreur relative est privilégiée car elle donne une indication sur le nombre de chiffres significatifs corrects (voir [16, §1.2]), ce qui n'est pas le cas de l'erreur absolue. Prouver une borne supérieure sur l'erreur relative constitue déjà une bonne source d'information sur la précision d'un calcul. De plus, des exemples pour lesquels les erreurs générées sont proches de la borne peuvent être utilisés pour garantir la finesse de cette borne.

De nombreux travaux ont déjà été publiés sur le sujet de borner les effets des erreurs d'arrondi. Dans cette thèse, nous traiterons le sujet de la réduction de bornes d'erreur

existantes ainsi que la problématique de l'automatisation des preuves d'optimalité de bornes d'erreur connues.

Borne supérieure sur l'erreur générée par un algorithme

Un objectif de l'analyse d'erreur est de trouver une borne supérieure sur l'erreur que génère un algorithme. Les bornes sur l'erreur relative s'expriment généralement en fonction de l'unité d'arrondi u, qui représente une borne presque optimale sur l'erreur d'arrondi produite par une opération élémentaire avec un arrondi au plus proche ; en base β et précision p, on a $u = \beta^{1-p}/2$. De manière générale, u est représentatif de la meilleure borne d'erreur que l'on peut attendre pour un calcul en arithmétique à virgule flottante.

Idéalement, la borne sur l'erreur relative dépend uniquement de la base β et de la précision p, et est de l'ordre de grandeur de l'unité d'arrondi, auquel cas on dit que l'algorithme admet une "petite" borne d'erreur uniforme. Cependant, dans certains cas, il n'est pas possible d'obtenir une borne d'erreur uniforme qui soit informative ; la borne d'erreur doit alors dépendre des données du problème pour caractériser les entrées pour lesquelles l'erreur peut être grande devant u. Lorsqu'il s'agit d'un algorithme "inverse-stable", la dépendance aux données est classiquement exprimée en fonction du nombre de conditionnement du problème (voir par exemple [16]).

Dans cette thèse, nous nous concentrons sur des algorithmes pour lesquels on peut obtenir une borne sur l'erreur relative de la forme $\alpha u + o(u)$ lorsque u tend vers 0, avec $\alpha \in \mathbb{R}_{>0}$. Étant donné un tel algorithme, nous appliquons les techniques d'analyse d'erreur pour réduire le coefficient du premier ordre α autant que possible. En particulier, nous analysons un algorithme qui calcule une approximation de l'inverse d'un nombre complexe non nul, représenté par deux nombres à virgule flottante pour sa partie réelle et sa partie imaginaire.

Borne inférieure sur la plus grande erreur générée par un algorithme

Lorsque l'on mène une analyse d'erreur pour une brique de base du calcul numérique (tel que l'évaluation de $ab + cd$ par exemple), on souhaite obtenir la meilleure borne possible sur l'erreur relative car cela a une influence directe sur les analyses des algorithmes qui reposent sur cette brique. De plus, des bornes d'erreurs qui ne sont pas optimales ne permettent pas toujours de comparer des algorithmes en termes de précision du résultat. Considérons par exemple deux algorithmes A et B pour évaluer une même fonction. Même si A admet une borne d'erreur qui est strictement inférieure à celle connue pour B, on ne peut pas conclure que A est plus précis que B. En revanche, s'il existe un jeu d'entrées pour lequel l'erreur générée par B est supérieure à la borne d'erreur pour A, alors on sait que, en pire cas, A est plus précis que B. Pour cette raison, on peut déduire plus d'information sur la précision du résultat calculé lorsque les bornes d'erreur optimales sont connues pour les deux algorithmes.

De plus, lorsqu'une borne d'erreur de la forme $\alpha u + o(u)$ est connue, si on peut déterminer un exemple paramétré par le précision p pour lequel l'erreur générée est de la forme $\alpha u + o(u)$, alors on dira que la borne d'erreur est asymptotiquement optimale. Cela signifie que le coefficient du premier ordre α de la borne d'erreur est optimal, et que la borne

d'erreur ne peut pas être significativement améliorée.

Comme nous venons juste de le voir, deux types d'exemples peuvent être considérés. Des exemples numériques, avec β et p fixés (par exemple $\beta = 2$ et $p = 24$ dans le format IEEE binary32 [17]), fournissent des bornes inférieures sur la plus grande erreur que peut générer un algorithme, et peuvent être utilisés pour comparer plusieurs algorithmes implantés dans un format spécifique. D'un autre côté, il est aussi parfois possible de trouver des exemples paramétrés par le précision, comme mentionné plus haut. De tels exemples sont évidemment préférables car ils prouvent l'optimalité asymptotique de la borne d'erreur ; on peut noter aussi qu'ils fournissent en général de bons exemples pour les formats IEEE.

Comme nous allons le voir par la suite, la vérification à la main d'exemples paramétrés par la précision s'avère délicate et contient de nombreuses sources d'erreurs. Une partie de notre travail était donc de développer une bibliothèque pour calculer automatiquement avec des nombres à virgule flottante paramétrés par la précision p, que l'on appellera nombres à virgule flottante symboliques. Cette bibliothèque est fondée sur un formalisme rigoureux et est implantée dans le logiciel de calcul formel Maple.

Plan du manuscrit

Le manuscrit est divisé en trois parties ; dans cette section, nous décrivons succinctement le contenu de chacune.

Analyse d'erreur en arithmétique à virgule flottante (Partie I)

La première partie de cette thèse est une introduction succincte à l'analyse d'erreur directe en arithmétique à virgule flottante. Le Chapitre 1 est une introduction à l'arithmétique flottante, inspirée du standard IEEE 754 [17], du Handbook of Floating-Point Arithmetic [32] et de [16]. Nous définissons dans ce chapitre les nombres à virgule flottante et les différentes fonctions d'arrondi, sous l'hypothèse d'exposants non bornés. Nous illustrons aussi avec un exemple la perte de précision possible résultant d'une cancellation catastrophique dans un algorithme contenant seulement trois opérations en virgule flottante, pour évaluer un déterminant 2x2. Nous présentons quelques algorithmes alternatifs qui existent pour cette évaluation et qui sont plus précis, en particulier l'algorithme de Kahan, qui sera utilisé à nouveau dans la Partie II. Cet exemple est aussi l'occasion d'introduire la possibilité de calculer sur des nombres à virgule flottante symboliques, qui sera le sujet de la Partie III.

Dans le Chapitre 2, nous décrivons les outils et modèles disponibles dans la littérature pour calculer une borne supérieure sur l'erreur générée par un algorithme. Plus précisément, nous introduisons le *modèle standard* et ses raffinements, et nous illustrons leur utilisation sur un algorithme de division complexe qui est analysé dans [21]. L'objectif est de montrer que ces modèles sont simples à utiliser et déjà très efficaces pour l'analyse d'erreur de nombreux algorithmes. Cela montre aussi comment toute amélioration du modèle peut mener à des bornes d'erreur plus petites et plus simples. Nous concluons ce chapitre sur la notion de finesse d'une borne d'erreur, en introduisant différents niveaux de finesse que nous sommes capables d'atteindre pour la plupart dans le cas de l'inversion

complexe dans la Partie II. Parmi eux, l'optimalité asymptotique d'une borne repose sur du calcul en arithmétique à virgule flottante symbolique, ce qui motive la Partie III.

Analyse d'un algorithme d'inversion complexe (Partie II)

La seconde partie de cette thèse est dédiée à l'analyse d'erreur d'un algorithme en arithmétique à virgule flottante pour l'inversion complexe, lorsque les nombres complexes sont représentés par des nombres à virgule flottante pour leur parties réelle et imaginaire. Dans ce contexte d'arithmétique complexe, on peut mener une analyse d'erreur par composante, en fournissant une borne d'erreur à la fois sur la partie réelle et sur la partie imaginaire, ou une analyse d'erreur au sens de la norme, en fournissant une borne sur l'erreur exprimée en fonction du module complexe. Les analyses de l'addition et de la soustraction complexes sont directes et de nombreux algorithmes pour la multiplication complexe ont déjà été analysés, à la fois par composante et au sens de la norme, conduisant dans la plupart des cas à des bornes d'erreur asymptotiquement optimales (voir [9, 16, 20, 8, 24]). La dernière opération à considérer est la division complexe, pour laquelle l'algorithme naïf a déjà été analysé, soit par composante dans [21], soit au sens de la norme dans [9, 16, 4]. Cependant, nous n'avons pas d'information concernant la finesse des bornes sur l'erreur au sens de la norme pour la division complexe.

Dans le Chapitre 3, nous menons l'analyse par composante d'un algorithme d'inversion complexe, en tant que cas particulier de la division. Une borne d'erreur de la forme $3u + \mathcal{O}(u^2)$ était déjà connue dans [9] ; notre contribution est la preuve de la borne d'erreur plus simple $3u$, accompagnée d'exemples qui illustrent la finesse de cette borne. Cette amélioration vient de la considération de l'opération spécifique de mise au carré d'un nombre à virgule flottante, pour laquelle nous avons été capables d'obtenir une meilleure borne que celles fournies par les modèles standard ou raffiné. Le reste de l'analyse est très similaire à l'exemple détaillé dans le Chapitre 2, dans les Sections 2.2 et 2.3. Nous prouvons ensuite que cette borne d'erreur est asymptotiquement optimale pour les précisions paires, avec un certificat constitué de nombres à virgule flottante symboliques comme entrées, et fine, dans le cas des précisions impaires, avec des exemples numériques pour diverses précisions, parmi lesquelles les précisions standards ($p = 53$ et $p = 113$).

Le Chapitre 4 est ensuite dédié à l'analyse d'erreur de l'inversion complexe au sens de la norme, pour laquelle la borne $3u+\mathcal{O}(u^2)$ était déjà connue. Notre contribution consiste à réduire cette borne pour obtenir $\gamma u + 20u^2$ où $\gamma = 2.70712\cdots$ et à fournir des exemples numériques pour les précisions standards $p = 24$, 53 et 113 pour lesquels l'erreur finale est proche de cette borne. Cette analyse est fondée sur le modèle qui borne l'erreur absolue générée par une opération élémentaire et sur une analyse détaillée de la fonction obtenue comme borne. Elle est divisée en trois étapes principales : premièrement une réduction du domaine et l'analyse de quelques cas limites simples, ensuite une disjonction de cas, découpant ainsi le domaine en plusieurs sous-domaines qui sont analysés un par un, en utilisant de l'analyse réelle classique, enfin le dernier sous-domaine est analysé séparément car il est le plus délicat.

Ces résultats sont publiés dans Numerical Algorithms [22].

Une arithmétique à virgule flottante symbolique (Partie III)

Comme mentionné plus haut, l'utilisation de calculs sur des nombres à virgule flottante symboliques permet de prouver l'existence de comportements problématiques dans certains algorithmes ainsi que de prouver la finesse de bornes d'erreur. En particulier, de tels calculs conduisent à des propriétés qui sont valables pour toutes les précisions standards actuellement disponibles, mais aussi pour les possibles nouvelles précisions qui pourraient apparaître dans le futur. Cependant, ces calculs sont faits pour l'instant à la main ce qui demande beaucoup de temps et favorise les erreurs. En conséquence, on ne peut pas s'attendre à tester de nombreuses valeurs lorsque l'on cherche un jeu de nombres à virgule flottante symboliques satisfaisant une propriété donnée. Ainsi, la problématique est d'automatiser de tels calculs, et d'établir des définitions rigoureuses pour assurer que l'on calcule effectivement ce que l'on attend, c'est-à-dire que les calculs symboliques coïncident avec les résultats numériques lorsque l'on fixe une précision. Dans les exemples mentionnés plus tôt, les expressions sont paramétrées par la base β et la précision p ; dans cette thèse, nous nous concentrerons sur la paramétrisation par la précision seulement, pour une base fixée.

Le Chapitre 5 est dédié à la définition rigoureuse d'une arithmétique à virgule flottante symbolique qui imite les définitions standards de l'arithmétique à virgule flottante classique. Ces définitions commencent avec l'ensemble ordonné $\mathbb{SQ} = \mathbb{Q}(\beta^k)$ des rationnels symboliques, où β est la base fixée et k la variable symbolique. Nous expliquons que les notions liées aux nombres à virgule flottante peuvent être transférées à cet ensemble et nous définissons les entiers symboliques $\mathbb{SZ}$ et les fonctions d'arrondi de $\mathbb{SQ}$ vers $\mathbb{SZ}$. Finalement, pour une précision p — c'est-à-dire une fonction affine en k asymptotiquement supérieure à 2, avec des coefficients entiers — nous définissons l'ensemble de nombres à virgule flottante symboliques $\mathbb{SF}_p$ et les fonctions d'arrondi standards, en prouvant qu'elles peuvent être calculées à partir de fonctions d'arrondi vers les entiers symboliques. Le résultat final assure que cette arithmétique à virgule flottante symbolique satisfait nos besoins : quand on évalue le résultat d'un calcul symbolique pour une précision donnée (i.e. une valeur donnée de k), on obtient le même résultat que si on avait calculé directement dans l' arithmétique à virgule flottante classique correspondante (quand la précision est suffisamment grande).

Nous avons implanté cette arithmétique symbolique en Maple pour obtenir un outil automatique pour calculer avec des nombres à virgule flottante symboliques. Cette implantation est décrite dans le Chapitre 6 avec quelques détails concernant nos choix de conception. L'utilisation de la bibliothèque est ensuite illustrée sur quatre exemples de la littérature pour montrer comment elle permet de vérifier facilement les certificats existants d'optimalité asymptotique fondés sur des nombres à virgule flottante symboliques.

Partie I : Analyse d'erreur en arithmétique à virgule flottante

L'arithmétique à virgule flottante est utilisée pour simuler l'arithmétique réelle (sur un ensemble infini) au sein d'un format en précision finie (et donc un ensemble fini de valeurs).

Elle vise l'efficacité, à la fois en terme de précision des calculs et de vitesse d'exécution, et la simplicité d'utilisation. Cette arithmétique était initialement légèrement différente d'une architecture à l'autre, en particulier en ce qui concerne les comportements exceptionnels, ce qui rendait son utilisation plus délicate. En 1985, un premier standard a émergé : le standard IEEE 754-1985 [1], qui définit l'arithmétique à virgule flottante binaire. Un second standard, le standard IEEE 854-1987 [2], voit le jour en 1987, définissant l'arithmétique à virgule flottante décimale. En 2008, ces deux standards sont révisés et unifiés pour former le standard IEEE 745-2008 [17]. Ce dernier met en place un formalisme rigoureux pour les nombres à virgule flottante, les opérations élémentaires, et les comportements exceptionnels. Sans entrer dans les détails, étant donnée une base β et une précision p, un nombre à virgule flottante est un nombre rationnel qui tient sur p chiffres (entiers dans $\{0, 1, \ldots, \beta - 1\}$), multiplié par une puissance entière de la base ; cette puissance entière appartient à un domaine borné. Cinq fonctions d'arrondi sont définies pour spécifier quel nombre à virgule flottante est la représentation d'un nombre réel donné. Le standard impose aussi l'arrondi correct pour les opération élémentaires ($+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$ et FMA : une opération à trois paramètres qui calcule $ab + c$) : cela signifie que ces opérations doivent être aussi précises que possible en fonction de la fonction d'arrondi choisie. Toutes ces notions sont détaillées dans le Chapitre 1.

L'arithmétique à virgule flottante est restreinte par la précision finie du format. En conséquence, chaque opération peut générer une erreur et ces erreurs réduisent la précision du résultat final. Plus précisément, une erreur peut être amplifiée par les opérations suivantes et les cancellations catastrophiques. Ainsi, même si chaque opération est aussi précise que possible, la succession d'opérations en virgule flottante peut générer une grande erreur sur le résultat final, ne contenant parfois aucune information sur le résultat exact. Pour cette raison, tout algorithme utilisant l'arithmétique à virgule flottante devrait être fourni avec une analyse d'erreur, c'est-à-dire une borne sur l'erreur générée par l'algorithme, valide pour toutes les entrées. Dans cette thèse, nous nous concentrons sur de petits algorithmes, pour lesquels les bornes d'erreur sont généralement indépendantes des entrées : idéalement, elles correspondent aux erreurs générées dans le pire cas. Le formalisme décrit par le standard IEEE 754 permet de mener des analyses d'erreur rigoureuses, et d'exprimer et de prouver des propriétés sur les calculs en virgule flottante. Plusieurs modèles sont disponibles pour mener une analyse d'erreur systématique d'un algorithme et nous décrirons dans le Chapitre 2 le *modèle standard* et ses raffinements qui seront utilisés dans la Partie II.

Nous introduisons dans le Chapitre 1 les principales définitions du standard IEEE 754-2008 pour l'arithmétique à virgule flottante sous l'hypothèse d'exposants non bornés. Pour finir, un exemple est détaillé pour montrer que l'arithmétique à virgule flottante n'est pas infaillible et nécessite beaucoup de soin lors de son utilisation. En particulier, cet exemple illustre la nécessité d'une analyse rigoureuse pour chaque algorithme.

Dans le Chapitre 2, nous présentons les outils qui seront utilisés pour mener les analyses d'erreur d'algorithmes utilisant la virgule flottante. Plusieurs modèles, avec différents degrés de précision, qui sont utilisés en analyse d'erreur sont ensuite décrits, et illustrés sur un exemple de la littérature. Enfin, nous introduisons la notion de finesse d'une borne d'erreur et définissons trois différentes manières de décrire cette finesse, qui seront utilisées dans la Partie II.

Chapitre 1 : Arithmétique à virgule flottante

Ce chapitre expose les principales définitions introduites par le standard IEEE 754. Il est inspiré du standard IEEE 754 [17], du Handbook of Floating-Point Arithmetic [32] et de [16], dans lesquels plus de détails sont disponibles. Nous décrivons le formalisme moins contraint qui sera considéré dans cette thèse, qui est valide du moment qu'aucun dépassement de capacité n'intervient au cours des calculs. Ensuite, un exemple illustre une problématique qui apparaît avec cette arithmétique et est utilisé comme une introduction à une arithmétique à virgule flottante symbolique qui sera définie dans la Partie III.

Nous nous intéressons aux nombres à virgule flottante avec des exposants non bornés qui sont, en base β et précision p, soit zéro, soit les nombres rationnels que l'ont peut écrire sous la forme :

$$x = \pm M \cdot \beta^{e-p+1},$$

avec $M \in \mathbb{N}$, $\beta^{p-1} \leqslant M < \beta^p$ et $e \in \mathbb{Z}$ (non borné). Cette description est valable en l'absence de dépassement de capacité et donc tous les résultats seront valides sous cette hypothèse.

On définit ensuite les fonctions d'arrondi standards qui sont

- l'arrondi vers zéro RZ,
- l'arrondi vers le bas RD,
- l'arrondi vers le haut RU,
- l'arrondi au plus proche RN, associé à une règle de décision pour les cas d'ambiguïté qui sera par défaut tiesToEven ou quand explicitement mentionné, tiesToAway.

L'ensemble des nombres à virgule flottante n'est pas stable par les opérations élémentaires que sont $+$, $-$, $\times$, $/$, $\sqrt{}$ et FMA : en arithmétique à virgule flottante chaque opération s'accompagne d'une erreur d'arrondi. Le standard IEEE requiert l'arrondi correct pour les opérations élémentaires, c'est-à-dire que le résultat calculé doit être l'arrondi (au sens de la fonction d'arrondi sélectionnée) du résultat exact, de sorte à réduire l'erreur d'arrondi. Pourtant, dès que les opérations s'enchaînent, l'erreur finale peut être très grande comme le montre l'exemple suivant : si on évalue en base 2 et précision p un déterminant 2x2 avec la formule

$$\mathrm{RN}(\mathrm{RN}(ad) - \mathrm{RN}(bc)), \tag{1}$$

alors les entrées

$$a = d = 2^{p-1} + 2^{p-2} - 1, \quad b = a + 1 \quad \text{et} \quad c = a - 1$$

génèrent une erreur relative de l'ordre de 2^p en précision p (voir [21]) et donc le résultat calculé ne contient à priori aucune information sur le résultat exact. L'utilisation de l'instruction FMA ne résout pas le problème et nous verrons deux algorithmes (proposés par Kahan et par Cornea, Harrison et Tang), qui résolvent cette instabilité, avec une erreur relative toujours inférieure à 2^{1-p} et $2^{1-p} + \mathcal{O}(2^{-2p})$.

Cet exemple fait ressortir l'importance de l'analyse d'erreur en arithmétique à virgule flottante ainsi que le rôle que peuvent jouer les nombres à virgule flottante paramétrés par la précision.

Chapitre 2 : Modèles pour l'analyse d'erreur

Dans le chapitre précédent, nous avons vu que, bien qu'étant optimalement précise pour chaque opération élémentaire, l'arithmétique IEEE 754 peut conduire à un résultat final entaché d'une erreur très importante. En conséquence, il est nécessaire de fournir des bornes supérieures sur les erreurs générées par les algorithmes de calcul numérique. Si x et $\widehat{x}$ sont respectivement les résultats exact et approché, alors on souhaite borner l'erreur absolue $|\widehat{x} - x|$, ou relative $|\widehat{x} - x|/|x|$.

Dans ce chapitre, nous commençons par introduire quelques outils qui seront utilisés pour l'analyse d'erreur d'algorithmes en virgule flottante : l'exposant, les fonctions ulp (**unit in the last place**) et ufp (**unit in the first place**) et l'unité d'arrondi $u = \beta^{1-p}/2$.

Nous verrons ensuite différents modèles qui permettent de borner les erreurs d'arrondi :

- $|x - \widehat{x}| < \mathrm{ulp}(x)$, pour borner l'erreur absolue,
- $\widehat{x} = x(1+\epsilon)$ avec $|\epsilon| < u$ (**modèle standard**),
- $\widehat{x} = x(1+\epsilon)$ avec $|\epsilon| < u/(1+u)$, un raffinement du modèle standard.

L'utilisation du modèle standard et de son raffinement est illustrée avec un exemple d'analyse d'erreur d'un algorithme de division complexe de [21]. Nous mentionnons aussi la possibilité d'analyser plus en détail chaque opération élémentaire pour chercher les bornes d'erreur optimales, comme c'est le cas dans [24]. C'est une approche qui sera utilisée dans la Partie II pour analyser un algorithme d'inversion complexe.

Enfin, nous définissons trois niveaux de finesse d'une borne d'erreur :

- une borne est **optimale** si on a des exemples pour lesquels l'erreur générée atteint cette borne,
- elle est **asymptotiquement optimale** si on a des exemples, en toute précision, pour lesquels l'erreur générée est équivalente à la borne lorsque $p \to \infty$,
- elle est **fine** si on a des exemples pour lesquels l'erreur générée est proche de la borne.

Partie II : Analyse d'un algorithme d'inversion complexe

Dans cette seconde partie, nous considérons les nombres complexes en virgule flottante, représentés par $z = x + iy$ où x et y sont des nombres à virgule flottante. Nous supposons que l'arithmétique à virgule flottante sous-jacente est en base 2 et précision $p \geqslant 2$; comme expliqué plus haut, nous supposons des exposants non bornés, ce qui signifie que nos résultats s'appliqueront à des calculs en virgule flottante telle que décrite dans le standard IEEE 754 [17] tant qu'aucun dépassement de capacité n'intervient. La précision du résultat complexe qui est calculé en utilisant l'arithmétique à virgule flottante classique peut être attestée de deux manières : si $z = x + iy$ est le résultat exact et $\widehat{z} = \widehat{x} = i\widehat{y}$ le résultat calculé, on peut mener

- une analyse d'erreur (relative) **par composante** : on cherche une fonction positive $B_C(p)$ telle que pour toute entrée dans F_p,

$$|\widehat{x} - x| \leqslant B_C(p) \cdot |x| \quad \text{et} \quad |\widehat{y} - y| \leqslant B_C(p) \cdot |y|. \tag{2}$$

  Quand x et y ne sont pas nuls, $B_C(p)$ est une borne valide sur l'erreur relative par composante $E_C = \max(|\widehat{x} - x|/|x|, |\widehat{y} - y|/|y|)$.

- une analyse d'erreur (relative) **au sens de la norme** : on cherche une fonction positive $B_N(p)$ telle que pour toute entrée dans F_p,

$$|\widehat{z} - z| \leqslant B_N(p) \cdot |z|. \tag{3}$$

  Quand z ne s'annule pas, $B_C(p)$ est une borne valide sur l'erreur relative au sens de la norme $E_N = |\widehat{z} - z|/|z|$.

Nous décrivons seulement ici le cas de bornes uniformes (bornes qui dépendent uniquement de β et p, et non des entrées) car dans ce chapitre, nous considérons des opérations élémentaires sur les nombres complexes en virgule flottante : dans la plupart des cas, on est capable de trouver des bornes d'erreur uniformes satisfaisantes, qui sont plus faciles à intégrer dans des analyses d'erreur plus grandes.

On peut remarquer que toute borne d'erreur relative par composante est aussi une borne d'erreur relative valide au sens de la norme. Si $B(p)$ satisfait (2), on a $|\widehat{z} - z|^2 = (\widehat{x} - x)^2 + (\widehat{y} - y)^2 \leqslant B(p)^2 \cdot (x^2 + y^2)$, de sorte que $B(p)$ satisfait aussi (3). En conséquence, mener l'analyse d'erreur par composante d'un algorithme est une méthode pour obtenir une borne d'erreur au sens de la norme. Nous verrons des exemples où cette méthode conduit à des bornes d'erreur au sens de la norme asymptotiquement optimales. Toutefois, ce n'est pas le cas pour l'inversion complexe en virgule flottante : l'analyse directe au sens de la norme présentée dans le Chapitre 4 améliore la borne d'erreur par composante (asymptotiquement optimale) qui est prouvée dans le Chapitre 3. Dans le reste de l'introduction à la Partie II, nous résumons les résultats connus sur la précision des opérations élémentaires sur les complexes en virgule flottante.

Addition et soustraction complexes. Avec la représentation ci-dessus, l'addition (et la soustraction) de deux nombres complexes en virgule flottante est immédiate : le résultat de l'addition ou de la soustraction de $a + ib$ et $c + id$ est donné par la formule

$$\mathrm{RN}(a \pm c) + i\mathrm{RN}(b \pm d).$$

En conséquence, la borne $u/(1+u)$ donnée par le modèle raffiné est aussi une borne sur les erreurs relatives par composante et au sens de la norme pour l'addition (et la soustraction) complexe. De plus, les exemples donnés dans [24] pour prouver l'optimalité du modèle raffiné pour l'addition et la soustraction en virgule flottante sont aussi valides pour les opérations complexes, car tout nombre à virgule flottante peut être vu comme un nombre complexe en virgule flottante dont la partie imaginaire est nulle.

Multiplication complexe. La formule exacte

$$(a + ib) \cdot (c + id) = (ac - bd) + i(ad + bc)$$

correspond au calcul de deux déterminants 2x2. Une première méthode pour calculer cette multiplication est d'utiliser l'algorithme naïf décrit par la formule (1), introduit dans le Chapitre 1, qui contient six opérations en virgule flottante. Du point de vue par composante, nous avons expliqué que de possibles cancellations catastrophiques conduisent à une erreur relative de l'ordre de $1/u$ en arithmétique à virgule flottante binaire. Cependant, Brent, Percival et Zimmerman ont prouvé dans [8] que l'erreur relative au sens de la norme est toujours bornée par $\sqrt{5}u \approx 2.24u$, et que cette borne est asymptotiquement optimale pour l'arithmétique à virgule flottante binaire.

L'utilisation de l'opération de FMA évite une opération par composante, si on calcule le résultat comme $\mathrm{RN}(ac - \mathrm{RN}(bd)) + i\mathrm{RN}(ad + \mathrm{RN}(bc))$. Toutefois, cela ne résout pas le problème des cancellations catastrophiques et l'exemple donné dans le Chapitre 1 génère encore ce comportement. Dans [19], les auteurs ont prouvé que l'erreur relative au sens de la norme est toujours bornée par $2u$ et que cette borne est asymptotiquement optimale. Cet algorithme utilise seulement quatre opérations en virgule flottante pour une meilleure précision en pire cas au sens de la norme comparé à l'algorithme précédent.

Un troisième algorithme, reposant sur l'algorithme de Kahan pour évaluer un déterminant 2x2, est plus précis du point de vue par composante. L'erreur relative par composante est bornée par $2u$ [20] et un certificat d'optimalité asymptotique est aussi fourni pour assurer cette finesse de la borne. En conséquence, l'erreur relative au sens de la norme est aussi bornée par $2u$. Cette borne a été prouvée asymptotiquement optimale dans [19] : l'utilisation de l'algorithme de Kahan conduit à une meilleure précision par composante mais n'améliore pas la précision au sens de la norme comparé à l'algorithme précédent, tout en étant plus coûteux en termes de nombre d'opérations en virgule flottante (dix opérations sont utilisées).

Similairement, Cornea, Harrison et Tang ont proposé un autre algorithme pour calculer un déterminant 2x2 dans [10]. Muller (pour la base 2) puis Jeannerod (en base quelconque β) ont prouvé dans [31] et [18], respectivement, que cet algorithme génère une erreur relative par composante bornée par $2u + \mathcal{O}(u^2)$ qui est asymptotiquement optimale. La même borne $2u + \mathcal{O}(u^2)$ a été prouvée asymptotiquement optimale aussi au sens de la norme dans [19]. L'utilisation de cet algorithme pour calculer une multiplication complexe requiert 14 opérations en virgule flottante.

Division complexe. La précision de la division complexe n'est pas aussi bien comprise que celle de la multiplication complexe. Le résultat exact est donné par la formule

$$(a + ib)/(c + id) = (ac + bd)/(c^2 + d^2) - i(ad - bc)/(c^2 + d^2).$$

Des bornes d'erreur relative au sens de la norme pour l'algorithme naïf sont données par Champagne dans [9, p.29] ($5.45u + \mathcal{O}(u^2)$) et par Higham dans [16] ($5\sqrt{2}u + \mathcal{O}(u^2) \approx 7.07u + \mathcal{O}(u^2)$). De plus, Baudin a remarqué dans [4] que la formule consiste à multiplier le numérateur $(a + ib)$ par le conjugué du dénominateur $(c - id)$ avant de diviser par le nombre à virgule flottante calculé pour $c^2 + d^2$: il en a déduit que tout algorithme de

division construit sur ce schéma admet une borne d'erreur relative au sens de la norme de la forme $(3+\alpha)u+\mathcal{O}(u^2)$, où $\alpha\cdot u$ est le terme de premier ordre de la borne d'erreur associée à l'algorithme de multiplication. Par exemple, calculer une multiplication complexe avec l'algorithme naïf génère une erreur relative au sens de la norme bornée par $\sqrt{5}u$ de sorte que l'algorithme de division complexe naïf génère une erreur relative au sens de la norme bornée par $(3+\sqrt{5})u+\mathcal{O}(u^2) \approx 5.24u+\mathcal{O}(u^2)$. Dans le cas de l'algorithme de Kahan, cela conduit à la borne $5u+\mathcal{O}(u^2)$; la borne d'erreur relative par composante $5u+\mathcal{O}(u^2)$ a été prouvée correcte et asymptotiquement optimale pour les précisions paires dans [21]. Concernant les bornes d'erreur au sens de la norme, on a peu d'information sur leur finesse. En raison de l'espace des paramètres de la division complexe qui est de dimension quatre, il est difficile de trouver des exemples qui statuent sur la finesse de ces bornes d'erreur.

Inversion complexe. Étant donné que trouver une borne fine pour la division complexe est difficile, nous avons décidé d'analyser en premier lieu un cas particulier de la division : l'inversion. Cette partie traite de la précision de l'inversion d'un nombre complexe non nul donné par ses parties réelle et imaginaire qui sont des nombres à virgule flottante. Pour un nombre complexe non nul $x+a+ib$, son inverse $z=x+iy$ satisfait

$$x = \frac{a}{a^2+b^2}, \qquad y = -\frac{b}{a^2+b^2}.$$

En supposant que a et b sont des nombres à virgule flottante, nous nous sommes concentré sur l'approximation $\widehat{z}=\widehat{x}+i\,\widehat{y}$, qui peut être calculée classiquement en virgule flottante suivant la formule

$$\widehat{x} = \mathrm{RN}\left(\frac{a}{\mathrm{RN}(\mathrm{RN}(a^2)+\mathrm{RN}(b^2))}\right) \tag{4}$$

pour la partie réelle, et avec une expression similaire pour la partie imaginaire $\widehat{y}$.

Nous fournissons une analyse précise de cet algorithme, en base $\beta=2$, pour l'erreur relative par composante $\mathrm{E_C} = \max\bigl(|x-\widehat{x}|/|x|, |y-\widehat{y}|/|y|\bigr)$ et au sens de la norme $\mathrm{E_N} = |z-\widehat{z}|/|z|$. Pour cet algorithme, on peut déjà observer qu'aucune cancellation catastrophique ne peut intervenir. Dans chaque cas, nous bornons l'erreur *maximale* par une fonction $B(p)$ qui dépend uniquement de la précision p, et nous étudions la finesse de cette borne comme décrit dans le Chapitre 2. Nous rappelons que, dans ce contexte, nous distinguons trois niveaux de finesse : une borne peut être optimale, asymptotiquement optimale ou fine.

L'erreur relative par composante générée par la formule (4) satisfait $\mathrm{E_C} \leqslant 3u+\mathcal{O}(u^2)$, où $u=2^{-p}$ est l'unité d'arrondi. Notre première contribution est de montrer que le terme $\mathcal{O}(u^2)$ peut en fait être supprimé, ce qui conduit à la borne plus simple $\mathrm{E_C} \leqslant 3u$ (en supposant $p \geqslant 4$). De plus, lorsque la précision est paire, nous montrons que cette borne est asymptotiquement optimale en fournissant des nombres à virgule flottante a et b paramétrés par p, pour lesquels $\mathrm{E_C} \geqslant 3u-\frac{31}{2}u^{\frac{3}{2}}+\mathcal{O}(u^2)$; pour les précisions impaires, nous montrons que la borne $3u$ est fine, en particulier pour les formats binaires IEEE 754 correspondant ($p=53, 113$). Cela est expliqué dans le Chapitre 3.

La borne d'erreur relative au sens de la norme $\mathrm{E_N} \leqslant 3u+\mathcal{O}(u^2)$ peut être trouvée dans [9, p. 30], et une application directe de notre analyse d'erreur par composante conduit

à $\mathrm{E_N} \leqslant 3u$. Notre seconde contribution principale est de montrer que si $p \geqslant 10$, alors la borne suivante, plus petite, est valide : $\mathrm{E_N} < \gamma u + 9u^2$, où γ est une constante dans $(2.70712, 2.70713)$. Dans le format IEEE 754 binary32 ($p = 24$), cela implique $\mathrm{E_N} < 2.707131u$. Les techniques et la disjonction de cas que l'on a utilisés pour prouver cette borne sont inspirés de [39], mais nous utilisons aussi intensivement l'analyse réelle pour le traitement de chaque cas. Nous fournissons des exemples numériques qui montrent que cette borne est fine pour les formats IEEE 754 standards ($p = 24, 53, 133$). Cela est expliqué dans le Chapitre 4.

Nous concluons cette partie avec quelques remarques sur les implications de cette analyse d'erreur pour la division complexe. Les parties techniques des preuves qui peuvent être laissées de côté en première lecture sont rassemblées en Annexe A.

Plusieurs auteurs [37, 38, 35, 5] ont suggéré des méthodes pour éviter les dépassements de capacités liés aux opérations intermédiaires uniquement, et certaines parmi elles peuvent être utilisées avec la formule (4). Bien sûr, si le calcul introduit de nouvelles erreurs d'arrondi, ce qui est le cas par exemple dans la méthode de Smith [37], alors notre borne d'erreur peut ne plus être valide. Cependant, en suivant la technique développée par Priest dans [35], il est possible de multiplier a et b par une puissance de deux pour éviter de tels dépassements de capacité, sans introduire de nouvelles erreurs d'arrondi : dans ce cas, notre analyse est valide tant qu'aucun dépassement de capacité n'intervient dans le calcul. Néanmoins, nous n'allons pas détailler plus les techniques de mise à l'échelle ici et nous concentrer seulement sur l'erreur maximale sous l'hypothèse d'exposants non bornés.

Chapitre 3 : Analyse d'erreur par composante de l'inversion complexe

Dans ce chapitre, nous nous concentrons sur l'erreur relative par composante de l'algorithme décrit par la formule (4) qui calcule une approximation de l'inverse d'un nombre complexe non nul en virgule flottante $a + ib$, a et b étant des nombres à virgule flottante.

Nous remarquons tout d'abord que, comme $a + ib$ n'est pas nul, $x = a/(a^2 + b^2)$ et $y = -b/(a^2 + b^2)$ ne peuvent pas s'annuler simultanément, et que si l'une de ces deux quantités s'annule, alors le résultat calculé est très précis. Supposons par exemple que $x = 0$ (le cas $y = 0$ est similaire), en utilisant le modèle raffiné, on vérifie facilement que les valeurs $\widehat{x}$ et $\widehat{y}$ retournées par l'algorithme vérifient :

- $\widehat{x} = 0$, ce qui signifie que la partie réelle est calculée exactement ;
- $\widehat{y} = -\mathrm{RN}(b/\mathrm{RN}(b^2))$ et l'erreur relative sur la partie imaginaire est bornée par $2u$ (et donc inférieure à la borne que nous donnerons pour le cas général). En effet, on a $\widehat{y} = -1/b \cdot (1 + \epsilon_2)/(1 + \epsilon_1)$, avec $|\epsilon_i| \leqslant u/(1 + u)$ d'après le modèle raffiné.

Ainsi, le reste de ce chapitre est dévoué à l'analyse de $\mathrm{E_C} = \max\left(|x - \widehat{x}|/|x|, |y - \widehat{y}|/|y|\right)$ pour x et y non nuls. Des applications répétées du modèle raffiné donnent immédiatement $\mathrm{E_C} \leqslant 3u + \mathcal{O}(u^2)$. Nous prouvons ci-dessous que, si $p \geqslant 4$, alors le terme $\mathcal{O}(u^2)$ peut être supprimé, conduisant à la borne plus simple $3u$.

Pour cela, nous montrons que si $p \neq 3$, alors la borne sur l'erreur relative $u/(1+u)$ dans le modèle raffiné peut être remplacée par $u/(1+3u)$ lors de l'évaluation d'un carré $\mathrm{RN}(a^2)$ au lieu du cas général du produit (quand $p = 3$, il est facile de vérifier que la borne $u/(1+u)$ est atteinte en élevant au carré le nombre à virgule flottante $3/2 \cdot 2^e$, $e \in \mathbb{Z}$). Cette légère amélioration va s'avérer suffisante pour montrer que $\mathrm{E_C} \leqslant 3u$.

Pour finir, nous montrons que cette borne d'erreur est asymptotiquement optimale pour les précisions paires et fine pour les précisions impaires. Plus précisément, en précision paire $p \geqslant 12$, on considère les nombres à virgule flottante symboliques suivants :

$$\begin{aligned} a &= 2^{\frac{p}{2}-1} + 5 \cdot 2^{-2} + 2^{-\frac{p}{2}+2}, \\ b &= 2^{p-1} + 2^{\frac{p}{2}-1} + 1. \end{aligned}$$

Le résultat de l'algorithme d'inversion est $\widehat{x} = 2^{-\frac{3p}{2}+1} + 2^{-2p} - 2^{-\frac{5p}{2}+2}$. Comme $x = a/(a^2+b^2)$, on déduit que

$$\frac{x - \widehat{x}}{x} = 3u - \tfrac{31}{2}u^{\frac{3}{2}} + \mathcal{O}(u^2),$$

ce qui prouve l'optimalité asymptotique de la borne $3u$ pour les précisions paires. Pour les précisions impaires, nous fournissons des exemples numériques, en particulier pour les formats standards ($p = 53$ et $p = 113$), pour lesquels l'erreur est proche de la borne.

p	Entrées a et b	E_C/u
15	$a = 16732$ $b = 23252 \cdot 2^3$	2.93047...
17	$a = 66078$ $b = 93014 \cdot 2^8$	2.96359...
19	$a = 131435$ $b = 370969 \cdot 2^8$	2.98509...
53	$a = 4508053433127332$ $b = 6369149602646415 \cdot 2^{16}$	2.97894...
113	$a = 5192393427440123027423416459819356$ $b = 7343016638055329519853569740503421 \cdot 2^{16}$	2.97647...

Table 1: Exemples en précision impaire.

Chapitre 4 : Analyse d'erreur au sens de la norme de l'inversion complexe

Dans ce chapitre, nous étudions l'erreur relative au sens de la norme de l'Algorithme décrit par la formule 4, c'est-à-dire,

$$\mathrm{E_N} = \sqrt{a^2+b^2}\sqrt{(x-\widehat{x})^2 + (y-\widehat{y})^2}.$$

Comme expliqué dans l'introduction de la Partie II, la borne d'erreur par composante $3u$, obtenue dans le Chapitre 3, est aussi une borne valide au sens de la norme. Dans cette section, nous établissons le résultat suivant, qui contient une borne plus petite grâce à la prise en compte des corrélations entre les différentes erreurs d'arrondi générées par l'algorithme.

Théorème 4.1 *Si $p \geqslant 10$, alors l'erreur relative au sens de la norme pour la formule (4) vérifie $\mathrm{E_N} \leqslant \gamma u + 9u^2$, où γ est défini par*

$$\gamma = \frac{\sqrt{8778980525057 + 16793600(8\sqrt{2} - \sqrt{127}) - 550842155008\sqrt{254}}}{8192\,(16 - \sqrt{254})},$$

et satisfait $\gamma \in (2.70712, 2.70713)$.

Si $p \geqslant 10$, $\mathrm{E_N} < 2.70713u + 9u^2$ est donc une borne rigoureuse sur l'erreur relative au sens de la norme générée. On peut aussi noter que le terme du second ordre dans la borne d'erreur peut être absorbé par le terme du premier ordre, au coup d'une légère surestimation : par exemple, pour $p \geqslant 24$, on a $9u = 9 \cdot 2^{-24} < 10^{-6}$ de sorte que $\mathrm{E_N} < 2.707131u$. Les exemples numériques listés dans la Table 2 montrent que la borne d'erreur du Théorème 4.1 est fine pour les formats du standard IEEE 754 ($p = 24, 53, 113$).

p	Entrées a et b	E_N/u
24	$a = 11863283$ $b = 11865457 \cdot 2^{12}$	$2.69090\ldots$
53	$a = 4503599709991314$ $b = 6369051770002436 \cdot 2^{26}$	$2.70679\ldots$
113	$a = 2^{112}$ $b = 7343016637207171132572330391109909 \cdot 2^{56}$	$2.70559\ldots$

Table 2: Exemples avec une erreur relative au sens de la norme proche de γu.

Pour commencer, nous prouvons une borne supérieure sur l'erreur relative au sens de la norme de la forme :

$$E_N \leqslant \sqrt{f_2(a,b) \cdot u^2 + f_3(a,b) \cdot u^3}.$$

Nous réalisons aussi une première réduction de domaine qui conduit à la borne supérieure uniforme $f_3(a,b) < 25$, pour $p \geqslant 10$. Ensuite, nous considérons quelques cas particuliers pour lesquels f_2 est facilement bornée par $(2 + \sqrt{2}/2 + 3u)^2$ de sorte que la borne donnée dans le Théorème 4.1 en découle directement. L'analyse d'erreur sur le domaine restant est détaillée par la suite, au travers de sept cas qui sont résumés dans le tableau ci-dessous.

$\mathrm{ufp}(s_a + s_b)$	$\mathrm{ufp}(b^2)$	e	$\mathrm{ufp}\left(\frac{a}{s}\right)$	f_2	$\mathrm{E_N}$
1	1	$\geqslant 2$	$\leqslant 2^{-\frac{e}{2}}$	6.565	$2.6u$
4	2	$= -1$	$\leqslant \frac{1}{4}$	$\left(2+\frac{\sqrt{2}}{2}\right)^2$	$\left(2+\frac{\sqrt{2}}{2}\right)u + 5u^2$
		$\geqslant 0$	$\leqslant 2^{-2-\frac{e}{2}}$	$\left(\frac{7}{4}+\frac{\sqrt{2}}{2}\right)^2$	$2.5u$
2	1	$\geqslant 1$	$\leqslant 2^{-1-\frac{e}{2}}$	$\left(\frac{7}{4}+\frac{\sqrt{3}}{2}\right)^2$	$2.65u$
		$= 0$	$\leqslant \frac{1}{4}$	$\left(\frac{5}{2}\right)^2$	$\frac{5}{2}u + 5u^2$
	2	$\geqslant 1$	$\leqslant 2^{-\frac{3+e}{2}}$	$\left(2+\frac{\sqrt{2}}{2}\right)^2$	$\left(2+\frac{\sqrt{2}}{2}\right)u + 5u^2$
		$\geqslant 2$, pair	$= 2^{-1-\frac{e}{2}}$	$\gamma^2 + 20u$	$\gamma u + 9u^2$

L'analyse du premier cas est détaillée pour permettre au lecteur de comprendre comment chaque cas est traité ; le dernier cas fait l'objet d'une analyse plus approfondie qui conduit à la borne finale. Les détails des cas intermédiaires sont présentés dans l'Annexe A.

Conclusion de la Partie II

Concluons cette partie avec quelques remarques sur la division complexe. Comme mentionné dans l'introduction de cette partie, Baudin a remarqué dans [4] que tout algorithme calculant une multiplication complexe peut être utilisé pour calculer une division complexe suivant l'égalité

$$\frac{a+ib}{c+id} = \frac{1}{c^2+d^2}\big((a+ib)\cdot(c-id)\big).$$

Cette égalité signifie que l'on multiplie d'abord par le conjugué du dénominateur, pour diviser ensuite chaque composante par un nombre à virgule flottante approchant $c^2 + d^2$. Le fait intéressant est qu'une borne sur l'erreur relative au sens de la norme de la forme $\alpha u + \mathcal{O}(u^2)$ pour l'algorithme de multiplication implique directement la borne $(3+\alpha)u + \mathcal{O}(u^2)$ sur l'erreur relative au sens de la norme générée par l'algorithme de division correspondant.

Nous pouvons mettre en avant une second manière de faire correspondre un algorithme de multiplication complexe et un algorithme de division complexe, en utilisant simplement une multiplication par l'inverse du dénominateur :

$$\frac{a+ib}{c+id} = \frac{1}{c+id}\cdot(a+ib).$$

D'après notre analyse d'erreur de l'algorithme décrit par la formule (4) pour l'inversion complexe, toute borne $\alpha u + \mathcal{O}(u^2)$ pour la multiplication complexe implique la borne $(\gamma+\alpha)u + \mathcal{O}(u^2)$ pour l'algorithme de division correspondant, où γ est défini dans le Théorème 4.1. Comme $\gamma < 3$, ce second schéma pour la division complexe conduit à des algorithmes pour lesquels les meilleures bornes d'erreur connues sont meilleures (i.e. plus petites) qu'avec le premier schéma.

De plus, le nombre d'opérations en virgule flottante de chaque sorte (addition ou soustraction, multiplication, inverse et FMA) ne dépend pas du schéma : pour le même

coût en nombre d'opérations, la garantie sur la précision du résultat est plus forte en calculant une division complexe comme une multiplication par l'inverse.

Pour finir, nous mentionnons les plus grandes erreurs relatives au sens de la norme que nous ayons obtenues pour l'algorithme reposant sur la multiplication complexe naïve suivie d'une division (noté mul/div) et pour l'algorithme qui calcule d'abord une inversion complexe naïve puis utilise la multiplication complexe naïve (noté inv/mul) :

Arithmétique	Algorithme mul/div	
	Entrée	Erreur
binary32 ($p = 24$)	$\frac{5935365+i\cdot 11910483/2}{11863437+i\cdot 11864709}$	$5.07951u$
binary64 ($p = 53$)	$\frac{5935365+i\cdot 11910483/2}{11863437+i\cdot 11864709}$	$5.04208u$
binary128 ($p = 113$)	$\frac{736070367558372747372516958272345 9/4+i\cdot 183909524503601985250136536112733 1}{735009507599575839659580201503840 1+i\cdot 734368822629130634496405664399866 5}$	$5.01830u$

Arithmétique	Algorithme inv/mul	
	Entrée	Erreur
binary32 ($p = 24$)	$\frac{11898033+i\cdot 11894677}{2972123/4+i\cdot 742117}$	$4.72945u$
binary64 ($p = 53$)	$\frac{6379358682446203+i\cdot 6400634450993511}{3194317788255377+i\cdot 6369097858326577/2}$	$4.71008u$

On peut conclure de ces deux tables que les bornes sur l'erreur relative au sens de la norme que nous connaissons pour ces algorithmes — environ $5.24u$ pour mul/div et $4.94u$ pour inv/mul — ne sont pas très éloignées des bornes optimales. De plus, ces exemples numériques suffisent pour prouver que l'algorithme inv/mul est plus précis en pire cas au sens de la norme que l'algorithme mul/inv pour la division complexe en virgule flottante.

Partie III : Une arithmétique à virgule flottante symbolique

Nous avons vu plus précédemment l'utilisation d'exemples paramétrés par la précision pour prouver l'existence d'un comportement instable ou l'optimalité asymptotique d'une borne d'erreur. De tels exemples peuvent être construits en calculant d'abord les plus grandes erreurs générées par l'algorithme pour des petites valeurs de la précision, puis en devinant des entrées "génériques" paramétrées par β et/ou p, et finalement en effectuant les calculs à la main — calculs qui sont en général délicats et favorisent les erreurs — pour vérifier si les entrées devinées correspondent effectivement aux cas que l'on cherchait. Notre objectif dans cette partie est de montrer comment automatiser cette dernière étape pour gagner du temps dans la recherche, éviter les erreurs, et être capable d'essayer beaucoup plus de candidats.

Nous appelons les fonctions qui représentent les nombres à virgules flottant paramétrés par la précision p les nombres à virgule flottante symboliques (avec une base $\beta \geqslant 2$ fixé et paire). Nous proposons de manipuler ces nombres dans un logiciel de calcul formel tel que Maple, presque comme les nombres réels ou les polynômes peuvent être manipulés dans ce système.

L'arithmétique à virgule flottante a déjà été formalisée dans différents logiciels de preuve interactifs : voir par exemple [15, 11], et plus récemment la description de la bibliothèque Flocq dans [7], qui a été pensée pour l'assistant de preuve Coq. Une telle formalisation peut être utilisée pour générer des preuves de propriétés sur des algorithmes et programmes utilisant la virgule flottante. De plus, des calculs en arithmétique multi-précision peuvent être exécutés en utilisant Flocq pour des vérifications (voir [29]), mais dans ce cas, la précision de chaque opération est fixée : un format particulier avec une précision numérique $p \in \mathbb{N}$ est défini pour les opérations intermédiaires. A contrario, lorsque l'on calcule avec des nombres à virgule flottante symbolique comme nous allons le définir, la précision p est une fonction affine de la variable entière k, ce qui permet de traiter des exemples "génériques".

Nous rappelons brièvement comment l'arithmétique à virgule flottante peut être formalisée. L'ensemble $\mathbb{F}_p$ des nombres à virgule flottante (avec exposants non bornés), en base β et précision p, est le sous-ensemble de $\mathbb{Q}$ contenant les nombres de la forme $M \cdot \beta^e$, avec $|M|$ un entier strictement inférieur à β^p, et $e \in \mathbb{Z}$. Le résultat exact d'une opération élémentaire ($+$, $-$, $\times$ ou $/$) entre deux éléments de $\mathbb{F}_p$ est un nombre rationnel. Il est ensuite arrondi vers $\mathbb{F}_p$ en respectant une fonction d'arrondi donnée pour obtenir le résultat calculé. Cette opération d'arrondi peut être réalisée en trois étapes : d'abord l'argument est normalisé avec une division par une puissance de la base β ; ensuite, on arrondi vers un entier ; enfin, on ramène le résultat à l'échelle initiale. Pour un nombre rationnel (positif), en base β et précision p, ce schéma est décrit par le diagramme ci-dessous, où $\circ$ dénote la fonction d'arrondi vers les entiers :

$$\begin{array}{ccccccc} \mathbb{Q}_{>0} & \longrightarrow & \mathbb{Q} \cap [\beta^{p-1}, \beta^p) & \longrightarrow & \mathbb{Z} \cap [\beta^{p-1}, \beta^p] & \longrightarrow & \mathbb{F}_p \\ r & \longmapsto & \mu := r/\operatorname{ulp}_p(r) & \longmapsto & M := \circ(\mu) & \longmapsto & M \operatorname{ulp}_p(r) \end{array}$$

Par exemple, en base $\beta = 10$ et précision $p = 3$, l'arrondi au plus proche de $r = 0.17446$ peut être calculé ainsi :

$$0.17446 \longrightarrow 174.46 \longrightarrow 174 \longrightarrow 0.174 \quad \text{car } \operatorname{ulp}_3(r) = 10^{-3}.$$

Ce schéma peut aussi être représenté (pour l'arrondi au plus proche) par la formule suivante :

$$\operatorname{RN}_p(r) = \operatorname{ulp}_p(r) \lfloor r/\operatorname{ulp}_p(r) \rceil.$$

Nous définissons dans un premier temps l'ensemble $\mathbb{L}$ des fonctions affines à coefficients entiers qui sera utilisé pour représenter à la fois la précision et les exposants des nombres à virgule flottante symboliques. Notre analogue de $\mathbb{Q}$ est l'ensemble $\mathbb{SQ}$ des rationnels symboliques ; on considère les fractions rationnelles en β^k :

$$\mathbb{SQ} = \mathbb{Q}(\beta^k).$$

Nous introduisons ensuite l'ensemble $\mathbb{SZ}$ des entier symboliques, et nous montrons comment les éléments de $\mathbb{SQ}$ peuvent être arrondis vers $\mathbb{SZ}$. En utilisant la notion d'entiers symboliques, on définit l'ensemble $\mathbb{SF}_p$ des nombres à virgule flottante symboliques en base β et précision $p \in \mathbb{L}$. Finalement, on définit les fonctions d'arrondi standards et prouvons qu'elles peuvent être calculées en utilisant une version adaptée de l'équation (2).

Le tableau suivant récapitule la correspondance entre les données numériques et leurs contreparties symboliques :

Données numériques	Données symboliques
$\mathbb{Z}$	$\mathbb{L}$
$\mathbb{Q}$	$\mathbb{SQ}$
$\mathbb{Z}$	$\mathbb{SZ}$
$\mathbb{F}_p$	$\mathbb{SF}_p$

Les résultats exacts des opérations élémentaires sur les éléments de $\mathbb{SF}_p$ peuvent toujours être représentés dans $\mathbb{SQ}$. Ils sont alors arrondis vers $\mathbb{SF}_p$, parfois au coût d'une hypothèse supplémentaire telle que la parité de la variable k.

Dans le Chapitre 5, nous introduisons notre formalisation d'une arithmétique à virgule flottante symbolique, paramétrée par la précision. Nous définissons successivement les ensemble $\mathbb{SQ}$ des rationnels symboliques, $\mathbb{SZ}$ des entiers symboliques et $\mathbb{SF}_p$ des nombres à virgule flottante symboliques. Ces définitions sont complétées par les fonctions d'arrondi standards, de $\mathbb{SQ}$ ver $\mathbb{SZ}$ dans un premier temps, puis de $\mathbb{SQ}$ vers $\mathbb{SF}_p$. Nous concluons ce chapitre en prouvant que les calculs symboliques correspondent bien à l'arithmétique à virgule flottante classique lorsqu'une précision numérique est choisie. Par la suite, dans le Chapitre 6, nous présentons une implantation de ce formalisme en Maple et illustrons son utilisation sur plusieurs exemples de la littérature.

Chapitre 5 : Définition d'une arithmétique à virgule flottante symbolique

Ce chapitre est dédié à la formalisation de notre arithmétique à virgule flottante symbolique dans l'objectif de vérifier rapidement et en confiance des certificats d'optimalité asymptotique de bornes d'erreur connues, tels que celui de l'inversion complexe en virgule flottante (voir Chapitre 3). Dans un premier temps, nous introduisons l'ensemble ordonné $\mathbb{SQ} = \mathbb{Z}[\beta^k]$ des nombres rationnels symboliques et leur principaux attributs (exposant, fonctions ulp et ufp). Ces fonctions sur les rationnels symboliques reflètent le comportement asymptotique des éléments de $\mathbb{SQ}$: par exemple, si $f \in \mathbb{SQ}$, alors pour k suffisamment grand, l'exposant (symbolique) de f évalué en k est égal à l'exposant (numérique) de $f(k)$.

On décrit ensuite les entiers symboliques $\mathbb{SZ}$ comme les rationnels symboliques qui s'évaluent asymptotiquement sur des entiers. On montre alors que les éléments de $\mathbb{SZ}$ peuvent toujours s'écrire sous la forme :

$$f(k) = \frac{T(\beta^k)}{M \cdot \beta^d}, \tag{5}$$

avec $T \in \mathbb{Z}[X]$, $M \in \mathbb{Z}$ et $d \in \mathbb{N}$ premiers entre eux. Les entiers symboliques sont complètement caractérisés par la propriété suivante : étant donné $f \in \mathbb{Q}[\beta^k]$, soient T, M, et d comme dans (5), alors $f \in \mathbb{SZ}$ si et seulement si $T(0) \equiv 0 \pmod{\beta^d}$ et $T(\beta^m) \equiv 0$

(mod M) pour tout $m \in \{0, 1, \cdots, \mathrm{ord}_M(\beta) - 1\}$. De plus, si $f \in \mathbb{SZ}$ alors $f(k) \in \mathbb{Z}$ pour tout $k \geqslant d$.

On définit aussi les fonctions d'arrondi vers les entiers avec par exemple l'arrondi au plus proche qui doit satisfaire la contrainte $\lfloor f \rceil \in \arg\min_{g \in \mathbb{SZ}} |g - f|$. Pour gérer les cas d'ambiguïté, on utilisera les deux règles de décision standard : tiesToEven qui choisit l'entier pair et tiesToAway qui choisit l'entier le plus grand en valeur absolue. Nous verrons que les fonctions d'arrondi définies sont en fait des fonctions partielles mais que l'on est capable, au prix d'une hypothèse supplémentaire sur la variable k de toujours calculer de l'information et retourner un résultat éventuellement partiel. En particulier, les algorithmes de calculs de ces fonctions d'arrondi se déduisent de la propriété présentée ci-dessus et de la définition de $\mathrm{ord}_M(\beta)$.

Pour finir, on introduit l'ensemble $\mathbb{SF}_p$ des nombres à virgule flottante symboliques, en précision $p \in \mathbb{L}$ asymptotiquement supérieure à 2, via la définition :

$$\mathbb{SF}_p = \{0\} \cup \{M\beta^e \ : \ M \in \mathbb{SZ},\ e \in \mathbb{L},\ \beta^{p-1} \leqslant |M| < \beta^p\}.$$

On définit les fonctions d'arrondi standards vers $\mathbb{SF}_p$ similairement au cas des entiers symboliques et on prouve que ces fonctions peuvent être calculées à partir des fonctions d'arrondi vers les entiers symboliques, par exemple :

$$\mathrm{RD}_p(f) = \mathrm{ulp}_p(f) \cdot \lfloor f / \mathrm{ulp}_p(f) \rfloor \text{ si } f \text{ est non nul.}$$

Nous concluons cette dernière section en prouvant que les calculs symboliques que nous décrivons ainsi correspondent bien à l'arithmétique à virgule flottante classique dès que l'on évalue les résultats pour k suffisamment grand : cela montre que notre arithmétique à virgule flottante symbolique est adaptée pour tester des certificats d'optimalité et d'autres calculs en virgule flottante paramétrés par la précision.

Chapitre 6 : Une bibliothèque Maple pour l'arithmétique à virgule flottante symbolique

Dans le chapitre précédent, nous avons défini une arithmétique sur des données symboliques dans $\mathbb{SQ}$ et $\mathbb{SF}_p$. Nous décrivons dans ce chapitre une bibliothèque qui implante cette arithmétique et illustrons son utilisation sur des exemples de la littérature. La formalisation de cette arithmétique décrit les comportements asymptotiques, qui sont valides lorsque $k \to \infty$; en pratique, on souhaite calculer une borne inférieure k_0 à partir de laquelle le comportement asymptotique est atteint. Nous décrivons donc l'heuristique de calcul de ce k_0 implantée dans la bibliothèque. Ensuite, nous détaillons l'interface de cette bibliothèque Maple et nous présentons quelques exemples d'utilisation de la bibliothèque pour illustrer son utilisation simple et intuitive.

Pour résumé ce chapitre, nous allons considérer un des exemples présentés, qui traite de la division complexe. Cet exemple concerne l'algorithme CompDivS décrit dans [21] pour calculer le quotient de deux nombres complexes en virgule flottante, c'est-à-dire une approximation $\widehat{x} + i\,\widehat{y}$ de $(a + ib)/(c + id)$, où toutes les variables sont des nombres à virgule flottante. Les auteurs de [21] sont intéressés par l'erreur relative par composante ;

l'Exemple 8 de ce papier fournit des entrées paramétrées par la précisions pour lesquelles le calcul de la partie réelle conduit à une erreur relative équivalente à la borne $5u$.

Avant tout calcul, le package doit être créé et chargé, en fixant la base (ici, $\beta = 2$) et la variable symbolique (on utilisera k) :

```
> read("libsfp0.6.mpl"):
> beta := 2:
> SF2 := SF(beta, k):
> with(SF2):
```

Après avoir lu le fichier et fixé la base à 2, la troisième ligne construit le package pour une arithmétique à virgule flottante en base 2, avec le paramètre k. Ce package est ensuite chargé pour une utilisation plus confortable.

Ensuite, l'algorithme CompDivS s'appuie sur l'algorithme de Kahan qui s'implante intuitivement de la manière suivante en arithmétique symbolique, où `rn` désigne la fonction d'arrondi vers le nombre à virgule le plus proche avec la règle de décision tiesToEven :

```
> Kahan := proc(a, b, c, d, p)
>   wh := rn(bc, p);
>   e  := rn(wh - bc, p);
>   fh := rn(ad - wh, p);
>   rn(fh + e, p)
> end proc:
```

On notera la surcharge des opérations exactes qui permet d'écrire du code simple.

Comme l'exemple présenté est valable pour les précisions paires, on choisit $p = 2k$, et on définit les variables d'entrées :

```
> p := 2*k:
> a := SQ(beta^p-5*beta^(p/2-1)):
> b := SQ(-beta^(p/2) + 5/beta - 3*beta^(-p/2)):
> c := SQ(beta^p - beta):
> d := SQ(beta^(3*p/2)+beta^p):
```

On calcule ensuite la partie réelle `r` du résultat exact et celle de son approximation par l'algorithme CompDivS `rh` :

```
> r := (a*c+b*d)/(c^2+d^2):
> Dh := rn(c^2 + rn(d^2, p), p):
> Gh := Kahan(a, -b, d, c, p):
> rh := rn(Gh/Dh, p);
    rh := -8^(-k)-(1/2)*16^(-k)
> get_omega(rh);
    1
> get_k0(rh);
    4
```

La variable `rh` contient alors le résultat approché, mais aussi deux informations supplémentaires : une valeur k_0 et une valeur ω. Ces valeurs signifient que le résultat retourné est valable pour toute valeur de k qui est à la fois supérieure à k_0 et un multiple de ω. Dans l'exemple, on sait donc que le résultat est valable pour toute valeur de $k \geqslant 4$.

Puis on calcule l'erreur relative `err` que l'on peut exprimer en fonction de l'unité d'arrondi $u = 2^{-2k}$ et on calcule son développement asymptotique lorsque $u \to 0$.

```
> err := (rh - r)/r:
> series(expr_ur(err, p, u), u=0, 3) assuming u > 0:
```

Cela donne le résultat `err` $= 5u - \frac{23}{2}u^{3/2} + \mathcal{O}(u^2)$ lorsque $u \to 0$, ce qui confirme le résultat donné dans [21].

Conclusion de la Partie III

Nous rapportons quelques mesures de temps d'exécution pour montrer que notre bibliothèque n'est pas seulement d'un intérêt théorique, mais peut aussi être utilisée efficacement pour vérifier des exemples de la littérature, pris parmi [8, 20, 21, 31].

Les mesures des temps de calcul pour chaque exemple (moyenne sur 100 exécutions) sont donnés dans la Table 3 : les mesures ont été faites sous Maple 18, avec la commande `time[real]()`, sur un ordinateur portable équipé d'un processeur Intel Cire i5 (4310U, 2GHz) sous Linux 3.16. Tous les exemples listés ici utilisent seulement l'arrondi au plus proche, avec tiesToEven : la colonne "#`rn`" renseigne le nombre d'appels à la fonction `rn()` de la bibliothèque requis par l'exemple, ce qui donne une idée de sa "taille".

Bien que le code ne soit pas structuré avec comme principal objectif la performance, chaque exemple est vérifié en quelques dizaines de millisecondes, même l'Exemple 8 de [21] qui met en jeu une division symbolique et 12 appels à la fonction d'arrondi. Au final, la bibliothèque vérifie 29 exemples (25 en base 2 et 4 en base 10) en moins de 0,5 secondes de temps CPU sur un ordinateur portable récent, ce que nous considérons comme suffisamment rapide étant donné nos objectifs principaux.

Les expériences confirme qu'il est possible de vérifier efficacement des exemples de la littérature qui mettent en jeu des nombres à virgule flottante symbolique dans un logiciel de calcul formel. Nous espérons aussi que ce travail rende plus facile l'analyse de briques de base, petites mais importantes, du calcul numérique.

Perspectives

Nous terminons ce document en présentant quelques unes de nos perspectives de travail sur les problèmes traités dans cette thèse.

Division complexe en virgule flottante

La borne sur erreur relative au sens de la norme $\gamma u + 20u^2$ présentée au Chapitre 4 pour l'algorithme décrit par la formule 4 pour l'inversion complexe suggère une méthode alternative pour calculer le quotient de deux nombres complexes en arithmétique à virgule flottante : à la place de la formule

$$z_1/z_2 = (z_1 \cdot \bar{z_2})/|z_2|^2, \tag{6}$$

qui correspond à multiplier par le conjugué puis diviser par la norme au carré, on peut utiliser la formule suivante

$$z_1/z_2 = z_1 \cdot (1/z_2), \tag{7}$$

Table 3: Temps de vérification des exemples avec notre bibliothèque d'arithmétique symbolique.

Exemple (base 2)	#rn	Temps
• De [20]:		
Exemple 3.7 (p pair)	4	17 ms
Exemple 3.7 (p impair)	4	16 ms
Exemple 4.4	4	12 ms
Exemple 4.6	4	13 ms
Exemple 6.2	4	15 ms
Exemple 6.3 (p pair)	4	15 ms
Exemple 6.3 (p impair)	4	15 ms
Exemple 6.4 (p pair)	4	16 ms
Exemple 6.6 (partie 1)	4	12 ms
Exemple 6.6 (partie 2)	4	12 ms
Exemple 6.7 (partie 1)	4	13 ms
Exemple 6.7 (partie 2)	4	13 ms
• De [31]:		
Exemple	6	21 ms
• De [8]:		
Exemple (p pair)	6	18 ms
Exemple (p impair)	6	19 ms

Exemple (base 2)	#rn	Temps
• De [21]:		
Exemple 1	3	10 ms
Exemple 2	3	12 ms
Exemple 3 (p pair)	3	10 ms
Exemple 4	2	9 ms
Exemple 5 (p pair)	6	21 ms
Exemple 5 (p impair)	6	20 ms
Exemple 6 (p pair)	2	10 ms
Exemple 6 (p impair)	2	10 ms
Exemple 7 (p pair)	2	11 ms
Exemple 8 (p pair)	12	43 ms

Exemple (base 10)	#rn	Temps
• De [20]:		
Exemple 3.7 (p pair)	4	16 ms
Exemple 3.7 (p impair)	4	16 ms
Exemple 4.4	4	13 ms
Exemple 4.6	4	12 ms

qui correspond à multiplier par l'inverse. Nous avons vu dans la conclusion de la Partie II que cette seconde approche conduit à des bornes d'erreur au sens de la norme plus petites, quelque soit l'algorithme de multiplication considéré : soit l'algorithme naïf, avec ou sans FMA, soit ceux qui reposent sur l'algorithme de Kahan ou de Cornea, Harrison et Tang [10]. Cela fournit de nouveaux algorithmes à considérer lorsque l'on implante la division complexe, en fonction de l'algorithme de multiplication sélectionné.

Nous avons réussi à trouver des exemples numériques pour conclure qu'implanter l'équation (7) est plus précis en pire cas que l'équation (6) quand on utilise la multiplication naïve sans FMA, mais il nous manque des exemples similaires pour les autres algorithmes. Trouver des exemples numériques ou symboliques (à l'aide de notre bibliothèque d'arithmétique à virgule flottante symbolique), pour lesquels les erreurs sont proches de la borne, nous permettrait de classifier ces algorithmes par rapport à leur précision en pire cas. Des mesures de temps d'exécution d'implantations pratiques sont aussi nécessaires pour clarifier les compromis possibles entre la rapidité et la précision parmi ces algorithmes.

Arithmétique à virgule flottante symbolique

Notre bibliothèque Maple pour l'arithmétique à virgule flottante symbolique nous a déjà permis de vérifier environ 30 certificats d'optimalité asymptotique présentés dans la littérature. Cependant, nous ne l'avons pas encore utilisée pour la recherche de nouveaux certificats ou exemples symboliques. La prochaine étape serait d'automatiser, au moins

en partie, cette recherche de certificats. Plus précisément, on pourrait essayer d'utiliser les algorithmes qui reposent sur la disjonction d'intervalles (voir par exemple [30, §11.2]), avec des intervalles dont les bornes seraient des nombres à virgules flottante symboliques. La stratégie habituelle consiste à diviser les intervalles en sous-intervalles jusqu'à obtenir une information satisfaisante sur chaque sous-intervalle. Dans notre cas, nous cherchons des intervalles sur lesquels l'erreur relative prend une valeur proche de la borne ; si ce n'est pas le cas, alors l'intervalle est abandonné. Une suggestion pour accélérer les calculs serait d'analyser les erreurs qui entachent les résultats intermédiaires pour espérer en déduire qu'un sous-intervalle ne peut plus conduire à une erreur proche de la borne avant la fin de l'algorithme. D'autre part, une optimisation de la bibliothèque permettrait d'accélérer encore les calculs.

Une seconde piste de recherche serait d'attaquer le problème de l'opération de racine carrée avec les nombres à virgule flottante symboliques. Dans le Chapitre 5, pour le cas de la division, nous utilisons la périodicité de la décomposition des rationnels en base β pour calculer le résultat d'une opération d'arrondi. Cette régularité conduit à plusieurs résultats, par exemples, en fonction de la parité de la variable symbolique k, et nous avons décidé de sélectionner un de ces résultats, en l'associant à la contrainte correspondante sur k. Dans le cas de la racine carrée, nous pouvons identifier deux niveaux de difficulté au moment de l'arrondi :

- des cas simples, tels que $\sqrt{2^{2k}-1}$ en précision $2k$, pour lequel le développement asymptotique $\sqrt{2^{2k}-1} = 2^k(1-2^{-2k-1}-2^{-4k-3}-\ldots)$ conduit au résultat 2^k-2^{-k} ;
- des cas difficiles, tels que $\sqrt{2}$ en précision k, pour lesquels aucune forme close n'est connue, et le développement en base β ne fait pas apparaître de périodicité.

Notons que lorsque l'on veut arrondir $\sqrt{2^k-1}$ en précision k, les deux situations décrites ci-dessus coexistent : "k pair" correspond au cas simple et "k impair" correspond à la partie difficile. Comme suggéré par les termes, les cas simples sont faciles à inclure à la bibliothèque en calculant directement le résultat arrondi pour la racine carrée. D'un autre côté, nous avons traité dans [23] avec des cas plus difficiles. Nous avons prouvé l'optimalité asymptotique des bornes d'erreur des algorithmes classiques pour évaluer $\sqrt{a^2+b^2}$ et $c/\sqrt{a^2+b^2}$ en utilisant des exemples contenant une expression semblable à $\mathrm{RU}_p(\sqrt{2})$. Pour calculer avec ces exemples, nous avons utilisé le modèle standard pour les fonctions d'arrondi dirigées pour obtenir un encadrement "suffisamment fin" de $\mathrm{RD}_p(\sqrt{G})$ où G est une expression symbolique paramétrée par p. Ensuite, nous avons essentiellement utilisé la monotonie des fonctions exactes et d'arrondi utilisées dans le calcul pour prouver qu'une erreur relative donnée est atteinte. La plupart de ces calculs pourraient être fait avec une arithmétique d'intervalle sur les nombres symboliques mais il faut aussi utiliser quelques propriétés supplémentaires de la virgule flottante. Il serait intéressant de tenter de formaliser une telle arithmétique d'intervalle pour traiter automatiquement cet exemple (et d'autres évidemment).

Une dernière perspective serait de transférer notre formalisme et bibliothèque vers un assistant de preuve tel que Coq [6] pour atteindre une confiance plus grande dans les calculs. En pratique, l'objectif principal serait de vérifier les certificats d'optimalité asymptotique trouvés après l'exploration de larges domaines avec une implantation rapide.

Deux types d'implantation pourraient être faits : soit la théorie et les algorithmes sont transférés en Coq pour refaire les calculs au sein de Coq, soit la théorie seule est développée dans l'assistant de preuve et la génération d'une preuve vérifiable par Coq est ajoutée à la bibliothèque Maple.

Introduction

1 Context and motivations

Floating-point arithmetic is used to approximate calculations on real numbers using a fixed and finite number of significant digits. It is available on most current computers and usually provides good compromises between speed and accuracy. Moreover, this arithmetic is rigorously specified by the IEEE 754 standard [17] that clearly describes the behavior of each elementary floating-point operation (such as $+$, $-$, $\times$ and $\div$). However, it is well known that a consequence of the finite precision is that every elementary operation may suffer from a rounding error. So, even if each basic operation is as accurate as possible, the result of a sequence of floating-point operations may differ largely from the exact result. With this respect, two situations may occur:

- in the most favorable case, the rounding errors generated by each operation cancel each other (see for example [16, §1.14]), leading to a highly accurate result;
- they may also accumulate or be magnified through the computation, leading to a large loss of accuracy. In the worst case, the computed result may be completely wrong: its sign may be incorrect, for example, due to catastrophic cancellations.

In this context, it is extremely valuable to be able to estimate the error contained in the computed result. For example, a relative error smaller than 1 suffices to guarantee the sign of the result, which may already be enough for some applications (see for example [33]). Let us recall that two kinds of errors can be considered: if x and $\widehat{x}$ are the exact and computed results respectively, the absolute error is $|x - \widehat{x}|$, while the relative error is $|x - \widehat{x}|/|x|$ (assuming x does not vanish). Most of the time we are interested in the relative error, since it gives an indication about the number of correct significant digits (see [16, §1.2]), while the absolute error does not. Clearly, proving an upper bound on the relative error already gives a good way to assess the accuracy of the computation. Moreover, examples that generate an error close to this upper bound can be used to guarantee its tightness.

Much work have already been published on the topic of upper bounding the effect of rounding errors. In this thesis, we address both the theme of reducing existing error bounds, and the problematic of providing automated tools to prove the optimality of known error bounds.

1.1 Upper bound on the error generated by an algorithm

A purpose of rounding error analysis is to find an upper bound on the error generated by an algorithm. Relative error bounds are usually expressed in terms of the unit roundoff u, which is a sharp bound on the rounding error generated by an elementary operation when rounding to the nearest; in base β and precision p, we have $u = \beta^{1-p}/2$. Roughly speaking, u is representative of the best relative error bound that can be expected from a computation in floating-point arithmetic.

Ideally, the relative rounding error bound only depends on the base β and the precision p, and is of the order of magnitude of the unit roundoff, in which case we say that the algorithm admits a "small" uniform error bound. However, in some cases, it is not possible to derive a meaningful uniform error bound; the error bound must then depend on the problem data to characterize the inputs for which the error might be large compared to u. When dealing with a backward-stable algorithm, the dependence on the input data is then classically expressed in terms of the condition number of the problem (see for instance [16]).

In this thesis, we focus on algorithms for which we can obtain relative error bounds of the form $\alpha u + o(u)$ as u goes to zero, where $\alpha \in \mathbb{R}_{>0}$. Given such an algorithm, we apply error analysis techniques to reduce the first order coefficient α as much as possible. In particular, we analyze an algorithm that computes an approximation of the inverse of a complex number represented by its real and imaginary parts as floating-point numbers.

1.2 Lower bound on the largest error generated by an algorithm

When working out an error analysis for a basic building block of numerical computing (such as $ab + cd$ for instance), we would like to get the best possible relative error bound because it has a direct influence on the analyses of the algorithms relying on this block. Moreover, non-optimal bounds are not sufficient to compare algorithms in terms of accuracy. Let us consider for example two algorithms A and B for evaluating the same function. Even if A admits an error bound that is smaller than the one of B, we cannot conclude that A is more accurate than B. If in addition, there exists a set of input data for which the error generated by B is larger than the error bound of A, then we know that, in the worst case, B generates a relative error larger than the one of A. For this reason, more information can be deduced about the accuracy of the computed result when the optimal error bounds are known for both algorithms.

Additionally, when a relative error bound of the form $\alpha u + o(u)$ is known, if we can determine an example parametrized by the precision p for which the generated error is of the form $\alpha u + o(u)$, then we will say that the error bound is asymptotically optimal. It means that the first order coefficient α in the error bound is optimal, and that the error bound cannot be significantly improved.

As we have just seen, two kinds of examples can be considered. Numerical ones, with β and p fixed (for instance $\beta = 2$ and $p = 24$ in IEEE binary32 format [17]), provide lower bounds on largest error generated by a given algorithm, and can be used to compare algorithms implemented within a specific format. On the other hand, it is also sometimes possible to find examples parametrized by the precision, as already mentioned above.

Such examples are of course preferable since they prove the asymptotic optimality of the error bound; note that they usually provide good examples for the IEEE 754 formats.

As we will see, verification of examples parametrized by the precision can be tedious and error prone when performed with bare hands. A part of our work was therefore to develop a library for dealing automatically with floating-point numbers parametrized by p, referred to as symbolic floating-point numbers. This library is based on a rigorous formalism for the arithmetic of symbolic floating-point numbers and is implemented in the Maple computer algebra system.

2 Outline of the manuscript

The manuscript is divided into three parts; in this section, we summarize the content of each one of them.

2.1 Error analysis in floating-point arithmetic (Part I)

The first part of this thesis briefly introduces the reader to forward error analysis in floating-point arithmetic. Chapter 1 is an introduction to floating-point arithmetic, inspired from the IEEE 754 standard [17], the Handbook of floating-point arithmetic [32] and [16]. We define in this chapter the floating-point numbers and the various rounding functions, under the assumption of an unbounded exponent range. We also illustrate with an example the possible loss of accuracy due to a catastrophic cancellation in an algorithm containing only three floating-point operations to evaluate a two-by-two determinant. We present some existing alternatives for this evaluation that are more accurate, in particular Kahan's algorithm, which will be reused later in Part II. This example also introduces the possible calculations on floating-point numbers that are parametrized by the precision (referred as symbolic floating-point numbers), which are the topic of Part III.

In Chapter 2 we describe the tools and models available in the literature to compute an upper bound on the error generated by an algorithm. More precisely, we introduce the so-called *standard model* and its possible refinements, and illustrate their use on an algorithm for complex division that is analyzed in [21]. The aim is to show that these models are simple to use and are already very efficient for the error analysis of many algorithms. It also shows that any refinement of the model can lead to better and simpler error bounds. We conclude this chapter discussing the sharpness of an error bound, introducing different levels of sharpness that we are able to get in the specific case of complex inversion in Part II. Among them, the asymptotic optimality of a bound relies on symbolic floating-point computations as certificates, that motivate Part III.

2.2 Analysis of a complex inversion algorithm (Part II)

The second part of this thesis is dedicated to the error analysis of a floating-point algorithm for complex inversion when complex numbers are represented by two floating-point numbers for their real and imaginary parts. In the context of complex arithmetic, we can perform a componentwise error analysis, providing a bound on both the errors on the

real and imaginary parts, or a normwise error analysis, providing a bound on the error expressed using the complex modulus. The analyses of complex addition and subtraction are straightforward and many complex multiplication algorithms have been analyzed already, from both the componentwise and normwise points of view, leading in most cases to asymptotically optimal error bounds (e.g. see [9, 16, 20, 8, 24]). The last operation to consider is complex division, for which the naive algorithm was already analyzed from a componentwise algorithm in [21] and from a norwmise point of view in [9, 16, 4]. However we have no information about the sharpness of the bounds for the norwmise accuracy of complex division. As a first step, we decided to address the simpler problem of complex inversion.

In Chapter 3, we perform the componenwise analysis of this complex inversion algorithm. An error bound of the form $3u + \mathcal{O}(u^2)$ was already known in [9]; our contribution is the proof of the simpler error bound $3u$, and examples that illustrate the sharpness of this bound. The improvement comes from the consideration of the specific operation of squaring a floating-point number for which we were able to get a better bound than the ones provided by the standard and refined models. The remaining part of the analysis is then very similar to the example detailed in Chapter 2, in Sections 2.2 and 2.3. We then proved that this error bound is asymptotically optimal in the case of even precisions, with a certificate of symbolic floating-point numbers as input, and sharp, in the case of odds precisions, with numerical examples for various odd precisions, among which the standard ones ($p = 53$ and $p = 113$).

Chapter 4 is then dedicated to the normwise error analysis of complex inversion, for which the bound $3u + \mathcal{O}(u^2)$ was also known. Our contribution is to reduce this bound to $\gamma u + 20u^2$ where $\gamma = 2.70712\cdots$ and provide numerical examples for the standard precisions $p = 24$, 53 and 113 for which the final error is close to this bound. This analysis is based on the model that bounds the absolute error of one basic operation and a detailed analysis of the resulting bounding function. It is divided into three main steps: first a domain reduction and the analysis of some simple corner cases, then a disjunction of cases, splitting the domain into several subdomains that are analyzed one by one, using classic real analysis; the last subdomain is addressed separately because it is the most intricate one.

These results were published in Numerical Algorithms [22].

2.3 A symbolic floating-point arithmetic (Part III)

As mentioned earlier, the use of symbolic floating-point computations allows one to prove the existence of problematic behaviors in floating-point algorithms and also to prove the sharpness of error bounds. In particular, such computations lead to properties that are valid for all the standard precisions actually available, but also for possible new ones that might appear in the future. However, they are for now performed by hand which is time consuming and error prone. As a consequence, we cannot expect to test a large amount of values when looking for a set of symbolic floating-point numbers satisfying a given property. Therefore, the problematic is how to automate such computations, and how can we specify rigorous definitions in order to ensure that we are computing what we expect, that is symbolic computations that contain the results of numerical computations when

choosing a precision. In the examples mentioned above, the expressions were parametrized both by the base β and the precision p; in this thesis, we will focus on a parametrization by the precision only, for a fixed base.

Chapter 5 is dedicated to the rigorous definition of a symbolic floating-point arithmetic that mimics the standard definitions of classic floating-point arithmetic. These definitions start with the ordered set $\mathbb{SQ} = \mathbb{Q}(\beta^k)$ of symbolic rationals, where β is the fixed base and k the symbolic variable. We explain that the notions related to floating-point numbers can be transfered to this set and we define the symbolic integers $\mathbb{SZ}$ and the rounding functions from $\mathbb{SQ}$ to $\mathbb{SZ}$. Finally, for a precision p — that is an affine function of k being ultimately larger than 1, with integer coefficients — we define the set of symbolic floating-point numbers $\mathbb{SF}_p$ and the standard rounding functions, proving that they can be computed using the rounding-to-integer functions. The final result is that this symbolic floating-point arithmetic satisfies our need: when evaluating the result of a symbolic computation for a specific precision (that is a specific value of k), we get the same result as if we computed directly in the classical floating-point arithmetic with this specific precision (when this precision is large enough).

We implemented this formalization in Maple to get an automatic tool for computing with symbolic floating-point numbers. This implementation is described in Chapter 6 with some details about our design choices. The use of this library is then illustrated on four examples from the literature to show how it allows one to easily check the existing certificates of asymptotic optimality based on symbolic floating-point numbers.

Part I

Error analysis in floating-point arithmetic

The floating-point arithmetics are used to simulate the real arithmetic (on an infinite set) within a finite precision format (and thus, a finite set of numbers). They aim to be efficient, both in terms of accuracy and speed, and simple to use. These arithmetics were initially slightly different from one architecture to the other, especially for the exceptional behaviors, making their use more intricate. In 1985, a first standard was built: the IEEE 754-1985 standard [1], defining binary floating-point arithmetic. A second standard, the IEEE 854-1987 standard [2], was released in 1987, generalizing to decimal floating-point arithmetic. In 2008, these two standards were revised and unified into the IEEE 754-2008 standard [17]. This standard sets a rigorous formalism for the floating-point numbers, the basic operations, and the exceptional behaviors. Roughly speaking, given a radix β and a precision p, a floating-point number is a rational number that fits into p digits (integers in $\{0, 1, \ldots, \beta-1\}$) scaled by an integer power of the radix; this integer belongs to a bounded domain. Five rounding functions are defined to specify which floating-point number is the representation of a given real number. The standard also requires correct rounding for the basic operations ($+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$ and FMA: a three parameters operation calculating $ab + c$): it means that these operations must be as accurate as possible with respect to the chosen rounding function. All these notions are detailed in Chapter 1.

Floating-point arithmetic is restricted by the finite precision of the format. As a consequence, every operation may generate an error and these errors reduce the accuracy of the final result. More precisely, an error can be magnified by the next operations and catastrophic cancellations. Hence, even if each operation is as accurate as possible, performing successive floating-point operations may generate a large error in the final result. This is illustrated in Section 1.3 with only three operations after which the computed result contains no information about the exact result. For this reason, any algorithm using floating-point arithmetic should be provided with an error analysis, that is a bound on the error generated by the algorithm, valid for any input. In this thesis, we focus on small algorithms, for which the error bounds are usually independent of the input: ideally, they match the error generated in the worst case. The formalism given by the IEEE 754 standard allows to perform rigorous error analyses and to express and prove properties about floating-point computations. Various models are available to perform a systematic error analysis of an algorithm and we will described in Chapter 2 the so-called *standard model* and its refinement that will be used in Part II.

We introduce in Chapter 1 the main definitions of the IEEE 754-2008 standard for floating-point arithmetic under the hypothesis of an unbounded exponent range. Finally,

an example is detailed to show that floating-point arithmetic is not blindly reliable and requires much care when being used. In particular, this example illustrate the need for a rigorous analysis of each algorithm.

In Chapter 2, we introduce the tools that will be used to perform the error analysis of a floating-point algorithm. Various models, with different degrees of accuracy, that are used in error analyses are then described, and illustrated on an example from the literature. Finally, we discuss the notion of tightness of an error bound, defining three different ways of assessing the tightness for an error bound, that will be used in Part II.

Chapter 1

Floating-point arithmetic

This chapter exposes the main definitions introduced in the IEEE 754 standard, starting from the floating-point numbers, to the rounding-direction attributes. It is inspired from the IEEE 754 standard [17], the Handbook of floating-point arithmetic [32] and [16], in which more details can be found. It also describes a less constrained formalism that will be considered in this thesis, which is valid when no underflow of overflow occur in the computation. Then, an example illustrates the problematic that arises in such arithmetics and is used as an introduction to a symbolic floating-point arithmetic that will be defined in Part III.

1.1 Floating-point numbers

A **floating-point format** is essentially defined by four parameters:

- a **radix** β (integer larger than 1),
- a **precision** p (integer larger than 1),
- and two integers e_{min} and e_{max} satisfying $e_{min} < e_{max}$ and $e_{min} = 1 - e_{max}$.

The complete definition of a format also specifies the representation of a floating-point data as a bit string, a topic that will not be addressed in this thesis. In such a format, the **floating-point numbers** are the rationals x that can be written as

$$x = \pm M \cdot \beta^{e-p+1}, \tag{1.1}$$

where M is a nonnegative integer satisfying $M < \beta^p$, and e is an integer such that $e_{min} \leqslant e \leqslant e_{max}$. Therefore, the set of floating-point numbers, denoted $\mathbb{F}_p$ is a finite subset of the rationals which is symmetric with respect to zero. As an example, the repartition on the real axis of the nonnegative floating-point numbers in radix 2 and precision 3 is represented in Figure 1.1. The smallest positive floating-point number is $\beta^{e_{min}-p+1}$ and the largest one is $(\beta^p - 1) \cdot \beta^{e_{max}-p+1}$. Between two consecutive powers of the radix, the floating-point are uniformly distributed. The IEEE 754-2008 standard defines five basic floating-point formats whose parameters are summarized in Figure 1.2.

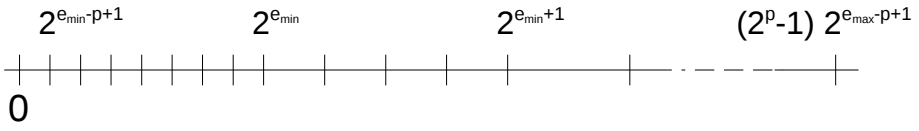

Figure 1.1: Floating-point numbers on the real axis ($\beta = 2$, $p = 3$)

	Binary format ($\beta = 2$)			Decimal format ($\beta = 10$)	
Parameter	**binary32**	**binary64**	**binary128**	**decimal64**	**decimal128**
p	24	53	113	16	34
e_{max}	127	1023	16383	384	6144

Figure 1.2: Basic floating-point formats in the IEEE 754-2008 standard

Since the set of floating-point numbers is not closed by most operations such as addition or multiplication, the exact result of each operation is then rounded as described in Section 1.2. Three exceptions, defined by the IEEE 754 standard, can occur when performing an operation:

- an **overflow** occurs when rounding the exact result with no constraint on the exponent range leads to a number larger in magnitude than the largest floating-point number. For instance, adding $(\beta^p - 1) \cdot \beta^{e_{max}-p+1}$ to itself would raise an overflow;
- an **underflow after rounding** occurs when rounding the exact result with no constraint on the exponent range leads to a nonzero number, smaller in magnitude than $\beta^{e_{min}}$. For instance, dividing $\beta^{e_{min}-p+1}$ by β would lead to an underflow;
- an **underflow before rounding** occurs when the exact result is nonzero and smaller in magnitude than $\beta^{e_{min}}$.

In the following, "underflow" will refer to "underflow after rounding". These exceptions are strongly related to the range of exponents allowed by the definition of floating-point numbers. In the remainder of this thesis, we assume an **unbounded exponent range** so that any nonzero floating-point number can be represented as

$$x = \pm M \cdot \beta^{e-p+1}, \tag{1.2}$$

where $M \in \mathbb{N}$, $\beta^{p-1} \leqslant M < \beta^p$ and $e \in \mathbb{Z}$ (unbounded). This assumption simplifies the analysis of algorithms since the exceptional behaviors are ignored. However, the results are valid as long as underflow and overflow do not occur in the computation.

1.2 Rounding functions and correct rounding

As we already mentioned, the set of floating-point numbers is not closed under the elementary arithmetic operations ($+$, $-$, $\times$ and $\div$) and most functions. Thus, rounding functions are used to choose which floating-point number will represent the exact result

of an operation. The IEEE 754-2008 standard defines five rounding functions that specify the way to round any real x into a floating-point format. These rounding functions are presented here under the assumption of an unbounded exponent range, as we explained in Section 1.1. Three of them are "directed" rounding functions:

- **roundTowardPositive**: $\mathrm{RU}(x)$ is the smallest floating-point number not smaller than x;
- **roundTowardNegative**: $\mathrm{RD}(x)$ is the largest floating-point number not larger than x;
- **roundTowardZero**: $\mathrm{RZ}(x)$ is the closest floating-point number to x, not larger in magnitude than $|x|$.

The two remaining rounding functions both consist in rounding to the nearest floating-point number; they only differ in case of a **midpoint**, that is when the real x is exactly halfway between two consecutive floating-point numbers. The standard defines two tie-breaking rules to deal with the midpoints: in case of a tie, **roundToAway** selects the largest floating-point number in magnitude whereas **roundToEven** selects the one with an even integral significand. The rounding function roundToEven is the default function and is referred to as RN; when the same notation is used for roundToAway, then the tie-breaking rule is explicitly mentioned. These definitions are illustrated on Figure 1.3.

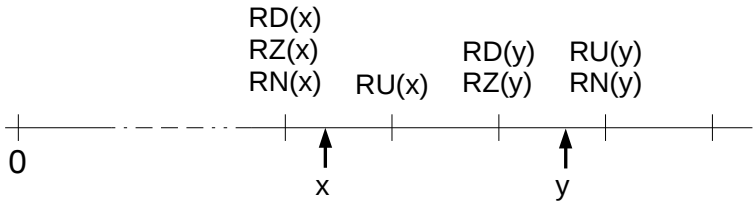

Figure 1.3: Four rounding modes, x and y are positive

An operation is said to be **correctly rounded** when the computed result is exactly the result of the rounding function applied to the exact result. For example, the addition is correctly rounded if the computed result is $\circ(x+y)$ where $\circ$ is the selected rounding function. The IEEE 754-2008 standard requires the correct rounding for the basic arithmetic operations: addition, subtraction, multiplication, division, FMA (**fused multiply-add**, computes an expression of the form $xy+z$ 'with only one rounding') and square root.

These rounding functions satisfy the following regularity properties (assuming as always assumed that overflow and underflow do not occur): for any integer e, any real x and any rounding function $\circ$, we have

$$\circ(\beta^e \cdot x) = \beta^e \cdot \circ(x). \tag{1.3}$$

Moreover, RN and RZ are symmetric with respect to zero:

$$\mathrm{RN}(-x) = -\mathrm{RN}(x) \quad \text{and} \quad \mathrm{RZ}(-x) = -\mathrm{RZ}(x), \tag{1.4}$$

whereas

$$\mathrm{RD}(-x) = -\mathrm{RU}(x). \tag{1.5}$$

1.3 An example leading to a large relative error

With the guarantee of correct rounding for the basic operations, each operation is as accurate as possible. However, when successive operations are performed, the computed result may drastically differ from the exact result. In this section, we consider a naive algorithm (Algorithm 1) to evaluate an expression of the form $ad - bc$ (two-by-two determinant) where a, b, c and d are floating-point numbers and RN is roundToEven.

This example illustrates the need for rigorous analyses of floating-point algorithms and also one possible use of floating-point numbers that are parametrized by the precision, that we will refer to as symbolic floating-point numbers. We will see in Part III how to define a symbolic floating-point arithmetic for dealing automatically with this example.

Algorithm 1 – NAIVE 2-BY-2 DETERMINANT
(Naive evaluation of a two-by-two determinant)
Input: a, b, c and d are binary floating-point numbers.
Output: $\widehat{x}$ is a floating-point number approximating $x = ad - bc$.

1. $\widehat{v} = \mathrm{RN}(ad)$;
2. $\widehat{w} = \mathrm{RN}(bc)$;
3. $\widehat{x} = \mathrm{RN}(\widehat{v} - \widehat{w})$;
4. **return** $\widehat{x}$;

Algorithm 1 performs only three operations but leads for some inputs to a very inaccurate result as mentioned in [21]. We consider the set of binary floating-point inputs

$$a = d = 2^{p-1} + 2^{p-2} - 1, \quad b = a + 1 \quad \text{and} \quad c = a - 1.$$

We will show that for any precision larger than 1, the relative error $|\widehat{x} - x|/x$ is close to 2^p. But first, let us mention that this set of input values is composed of symbolic floating-point numbers in precision p. Indeed, these inputs are integers that belongs to the interval $[2^{p-1}, 2^p - 1)$ so that the form (1.2) is satisfied with $e = 0$.

The algorithm produces the following intermediate results:

- The product $ad = 2^{2p-1} + 2^{2p-4} - 2^p - 2^{p-1} + 1$ is surrounded by the consecutive floating-point numbers $2^{2p-1} + 2^{2p-4} - 2^{p+1}$ and $2^{2p-1} + 2^{2p-4} - 2^p$. Since ad is strictly above the midpoint $2^{2p-1} + 2^{2p-4} - 2^p - 2^{p-1}$, it is rounded to $\widehat{v} = 2^{2p-1} + 2^{2p-4} - 2^p$.
- The exact product $bc = ad - 1 = 2^{2p-1} + 2^{2p-4} - 2^p - 2^{p-1}$ is surrounded by the same two consecutive floating-point numbers as ad, but bc is a midpoint. We use here the tie-breaking rule which selects the even significand and round ad to $\widehat{w} = 2^{2p-1} + 2^{2p-4} - 2^{p+1}$.
- Then, $\widehat{v} - \widehat{w} = 2^p$ is a floating-point number so that $\widehat{x} = 2^p$.
- On the other hand, $x = ad - bc = 1$.

Therefore, the relative error generated by Algorithm 1 on this set of inputs is $|\widehat{x}-x|/|x| = 2^p - 1$. In particular, for any of the standard precisions, it means that the relative error is larger than one: the computed result contains absolutely no information about the exact result, even the sign might be wrong (in fact, the sign is still correct because the rounding function is nondecreasing). This phenomenon is called a catastrophic **cancellation** and illustrates the need for rigorous analysis of floating-point algorithms.

The naive use of the FMA instruction does not solve the problem directly. Indeed, it corresponds to the formula $\widehat{x} = \mathrm{RN}(\mathrm{RN}(ad) - bc))$ and the set of input described above leads to a relative error $2^{p-1}-1$, which is also fully inaccurate. Two alternative algorithms have been proposed:

- Algorithm 2 is attributed to Kahan by Higham [16, Exercice 2.27]; Higham proved it to be highly accurate under a certain hypothesis, giving an error bound that depends on the inputs. The general case was addressed in [20] where the relative error is proved to be always bounded by β^{1-p} where p is the precision. Algorithm 2 will be reused in Chapter 2 to illustrate the standard model of error analysis.

- a second algorithm was proposed by Cornea, Harrison and Tang in [10] with a first error analysis in binary floating-point arithmetic, which was refined later in [31] and further in [18] for any radix β: the relative error generated by this algorithm is bounded by $\beta^{1-p} + \mathcal{O}(\beta^{-2p})$.

In these algorithms, the FMA instruction is also used to compute the errors generated in the products. More precisely, the error generated by a floating-point multiplication is a floating-point number; as a consequence, it can be computed exactly if a FMA instruction is available (see for example [26]). This is the property that is used in both algorithms.

In this section, we manipulated for the first time symbolic floating-point numbers; we will see in Part III that this calculation can be formalized and automated.

Algorithm 2 – KAHAN
(Kahan's algorithm to evaluate a two-by-two determinant)
Input: a, b, c and d are floating-point numbers.
Output: $\widehat{x}$ is a floating-point number approximating $x = ad - bc$.

1. $\widehat{v} = \mathrm{RN}(ad)$;
2. $\widehat{w} = \mathrm{RN}(bc)$;
3. $e = \mathrm{RN}(\widehat{w} - bc)$; // FMA instruction (exact, no rounding occurs)
4. $\widehat{f} = \mathrm{RN}(ad - \widehat{w})$; // FMA instruction
5. $\widehat{x} = \mathrm{RN}\left(\widehat{f} + e\right)$;
6. **return** $\widehat{x}$;

Chapter 2

Models for error analysis

In the previous chapter, we have seen that, while being optimally accurate for each elementary operation (i.e. each operation returns the best possible result), the IEEE 754 arithmetic can lead to very large error in the final result. As a consequence, we need to provide upper bounds for the error generated by an algorithm when evaluating an expression. If x and $\widehat{x}$ are the exact and computed results respectively, then we want to bound either the absolute error $|\widehat{x} - x|$, or the relative error $|\widehat{x} - x|/|x|$.

In this chapter, we first introduce some tools that will be used for the error analysis of floating-point algorithms in the next chapters. We then introduce different models, for the absolute or the relative error and explain how these models can be refined to get simpler and/or better error bounds. The uses of the so-called standard model and its refined version are illustrated with an algorithm for complex division analyzed in [21]. This analysis is related to a recent work by Jeannerod and Rump [24] about optimal error bounds for the basic operations. Finally, we discuss the different cases that we will distinguish when studying the tightness of an error bound.

2.1 Classical tools for error analysis

We first extend the notion of exponent to any real number: the exponent of zero is $-\infty$ and the base β **exponent** of a nonzero real number x is the unique integer e_x such that

$$\beta^{e_x} \leqslant |x| < \beta^{e_x+1}. \tag{2.1}$$

We then introduce the notion of **unit in the last place**, introduced by Kahan in [25] as the distance between the two floating-point numbers surrounding a real number. It was then extended by Goldberg [14] to the real numbers: given a floating-point format, the ulp function is defined by $\mathrm{ulp}(0) = 0$ and, for $x \neq 0$, $\mathrm{ulp}(x)$ is the unique integral power of β such that

$$\beta^{p-1} \leqslant \frac{|x|}{\mathrm{ulp}(x)} < \beta^{p}. \tag{2.2}$$

When x is a nonzero real number, this function gives the distance between the two consecutive floating-point numbers surrounding x (between x and its successor if x is a positive

floating-point number). When x is a nonzero floating-point number, $\mathrm{ulp}(x)$ also indicates the weight of the least significant digit of x.

We similarly define the notion of **unit in the first place**, introduced in [36]: given a floating-point format, the ufp function is defined by $\mathrm{ufp}(0) = 0$ and, for $x \neq 0$, $\mathrm{ufp}(x)$ is the unique integral power of β such that

$$1 \leqslant \frac{|x|}{\mathrm{ufp}(x)} < \beta. \tag{2.3}$$

This function indicates the weight of the most significant digit in the (possibly infinite) decomposition of x in base β. By definition, we have the following ordering for any real x:

$$\mathrm{ufp}(x) \leqslant |x|. \tag{2.4}$$

Finally, we define the **unit roundoff** associated with rounding to the nearest:

$$u = \frac{1}{2} \cdot \beta^{1-p}. \tag{2.5}$$

The ulp and ufp functions are closely related one to each other and are both related to the exponent e_x of x:

- $\mathrm{ulp}(x) = \beta^{e_x - p + 1}$;
- $\mathrm{ufp}(x) = 2u \cdot \mathrm{ulp}(x) = \beta^{e_x}$.

We now recall two properties about RN that make use of these notions: the first one gives an upper bound on the absolute error generated by one rounding. The second one gives a characterization for the result of the rounding function RN.

Property 2.1. *For any real number x, we have*

$$|\mathrm{RN}(x) - x| \leqslant \tfrac{1}{2} \cdot \mathrm{ulp}(x) = u \cdot \mathrm{ufp}(x). \tag{2.6}$$

The converse is false: if x is a midpoint, then the two closest floating-point numbers to x satisfies (2.6). However, we have the following characterization of the result of RN:

Property 2.2. *For any real number x and any floating-point number $\widehat{x}$, if $\widehat{x}$ is not a power of β and $|\widehat{x} - x| < 1/2 \cdot \mathrm{ulp}(\widehat{x})$, then $\widehat{x} = \mathrm{RN}(x)$.*

In this property, the inequality has to be strict to exclude the case of a midpoint. The hypothesis that $\widehat{x}$ is not a power of β excludes for example the case $x = \beta^{e+1} - \beta^{e-p} - \beta^{e-p-1}$ and $\widehat{x} = \beta^{e+1}$, with $e \in \mathbb{Z}$: we have $\widehat{x} - x = \beta^{e-p} + \beta^{e-p-1}$, so that $\mathrm{ulp}(\widehat{x}) = \beta^{e-p+2}$. Hence, $|\widehat{x} - x| < 1/2 \cdot \mathrm{ulp}(\widehat{x})$ is satisfied, while $\mathrm{RN}(x) = \beta^{e+1} - \beta^{e-p+1}$.

Finally, one can observe that, by definition, $\mathrm{ufp}(x) \leqslant |x|$, so that Property 2.1 implies that

$$|\mathrm{RN}(x) - x| \leqslant u \cdot |x|. \tag{2.7}$$

In particular, it means that the relative error generated by a correctly rounded operation is bounded by u, which is the commonly used unit for expressing relative error bounds (see [16]).

2.2 Standard model for error analysis

In order to prove rigorous error bounds, some models arise to answer the need for an easy manipulation of the rounding functions. The 'standard model' (see §2.2 in [16]), used to perform the error analysis of an algorithm, consists in replacing any occurrence of a rounding function by the exact result perturbed by a relative error that is bounded by the unit roundoff. As a direct consequence of (2.7), for any real x,

$$\mathrm{RN}(x) = x \cdot (1+\epsilon) \quad \text{with } |\epsilon| \leqslant u. \tag{2.8}$$

For the directed rounding modes, the bound on $|\epsilon|$ becomes $2u$. This model is simple but already powerful when analyzing small algorithms. It also allows for a systematic analysis of larger algorithms (see [16]), even if the bound is not always optimal. Let us illustrate this model by an example from [21] about an algorithm for complex floating-point division. For this algorithm, we consider complex floating-point numbers $a+ib$ and $c+id$, represented by their real and imaginary parts as floating-point numbers, and KAHAN denotes Algorithm 2, introduced in Section 1.3. Algorithm 3 implements the formula:

$$\frac{a+ib}{c+id} = \frac{ac+bd}{\delta} + i\frac{bc-ad}{\delta}, \quad \text{where } \delta = c^2+d^2.$$

The computed result $\widehat{R}+i\widehat{I}$ is a complex floating-point number approximating the exact result $R+iI = (a+ib)/(c+id)$.

Algorithm 3 – COMPDIVS
(Complex floating-point division)

Input: $a+ib$ and $c+id$ are complex floating-point numbers.
Output: $\widehat{R}+i\widehat{I}$ approximates $R+iI$.

1. $\widehat{\delta} = \mathrm{RN}(c^2 + \mathrm{RN}(d^2))$;
2. $\widehat{g}_{re} = \mathrm{KAHAN}(a, b, -d, -c)$;
3. $\widehat{g}_{im} = \mathrm{KAHAN}(b, a, d, c)$;
4. $\widehat{R} = \mathrm{RN}\left(\widehat{g}_{re}/\widehat{\delta}\right)$;
5. $\widehat{I} = \mathrm{RN}\left(\widehat{g}_{im}/\widehat{\delta}\right)$;
6. **return** $\widehat{R}+i\widehat{I}$;

The authors proved that the relative error generated when computing $\widehat{R}$ is bounded by $5u+13u^2$ for any precision $p \geqslant 5$. Because of the symmetries in the computation of $\widehat{R}$ and $\widehat{I}$, this error analysis is also valid for the imaginary part. The proof is fully detailed in the article but will be reproduced here with some slight changes to illustrate the use of the standard model. The proof is split into two parts: we first bound the relative error generated when computing the denominator $\widehat{\delta}$; we then use this bound and the error bound for Kahan's algorithm given in Section 1.3 to conclude the analysis.

Lemma 2.3. *The relative error generated when computing $\widehat{\delta}$ is bounded by $2u + u^2$.*

Proof. The proof is a list of successive inequalities. The standard model gives the following inequalities when approximating d^2:

$$d^2(1-u) \leqslant \mathrm{RN}\big(d^2\big) \leqslant d^2(1+u).$$

Then, we deduce that

$$\delta(1-u) \leqslant c^2 + \mathrm{RN}\big(d^2\big) \leqslant \delta(1+u).$$

Using once again the standard model gives an interval for the computed value $\widehat{\delta}$:

$$\delta(1-u)^2 \leqslant \widehat{\delta} \leqslant \delta(1+u)^2,$$

which leads to the result. □

In the article, the proof uses the inequality (2.6) with the ulp function to bound this relative error by $2u$. We will see in the next section how a slight refinement of the standard model leads to this simpler bound with the same reasoning. With the previous lemma, we prove the following theorem with a slightly larger bound: the additional term u^2 in Lemma 2.3 directly impacts the term in u^2 in the final result.

Theorem 2.4. *For any precision $p \geqslant 6$, the relative error when computing $\widehat{R}$ is bounded by $5u + 14u^2$.*

Proof. We know at this point that $\widehat{\delta} = \delta(1+\epsilon_1)$ with $|\epsilon_1| \leqslant 2u + u^2$. We also know that Kahan's algorithm generates a relative error bounded by $2u$ (cf. Section 1.3) so that $\widehat{g}_{re} = (ac+bd)(1+\epsilon_2)$ with $|\epsilon_2| \leqslant 2u$. Finally, we have from (2.8) that $\widehat{R} = \widehat{g}_{re}/\widehat{\delta}(1+\epsilon_3)$ with $|\epsilon_3| \leqslant u$. All together, these equalities give

$$\widehat{R} = R\frac{(1+\epsilon_2)(1+\epsilon_3)}{1+\epsilon_1}.$$

We conclude by verifying the inequalities $1 - 5u - 14u^2 \leqslant (1+\epsilon_2)(1+\epsilon_3)/(1+\epsilon_1) \leqslant 1 + 5u + 14u^2$ when $p \geqslant 6$. □

2.3 Refined standard model for error analysis

With the standard model, we have a straightforward method to perform the error analysis of an algorithm. However, we sometimes get bounds that contain a term in $\mathcal{O}(u^2)$ which is in fact avoidable. For instance, in the previous section, the bound in Lemma 2.3 is $2u+u^2$, instead of the bound of $2u$ proved in the original article. More generally, we often get bounds of the form $\alpha \cdot u + \mathcal{O}(u^2)$ and we expect from a more accurate model to allow us to get rid of this term in $\mathcal{O}(u^2)$ when possible. There are cases were this $\mathcal{O}(u^2)$ term is not a computational outfit and corresponds to actually attained error (see for example [18]). The refined standard model, attributed to Dekker [12] and Knuth [27, p.232] in [3] is a more accurate model for this purpose. With this model, the same scheme of proof as in the previous section is used: we first get a tighter error bound for the computation of the denominator $\widehat{\delta}$, and report this improvement in the rest of the proof with an additional use of the refined model for the division operation.

Property 2.5. *For any real x, we have*

$$\mathrm{RN}(x) = x \cdot (1+\epsilon) \quad \textit{with } |\epsilon| \leqslant \frac{u}{1+u}. \tag{2.9}$$

A simple proof is to consider first the case $x \in [1, \beta)$, for which $\mathrm{ufp}(x) = 1$. Either $x \leqslant 1+u$ and we deduce from Property 2.1 that $|\mathrm{RN}(x) - x|/|x| \leqslant u/|x| \leqslant u/(1+u)$; or $1 \leqslant x < 1+u$ and $\mathrm{RN}(x) = 1$ which leads to $|\mathrm{RN}(x) - x|/|x| = (x-1)/x \leqslant u/(1+u)$ over the interval $[1, 1+u]$. The result is then extended to the general case using (1.3)

With this new model, we can perform the same analysis as in the previous section to remove the term u^2 from the error bound when computing $\widehat{\delta} = \mathrm{RN}(c^2 + \mathrm{RN}(d^2))$.

Lemma 2.6. *The relative error generated when computing $\widehat{\delta}$ is bounded by $2u$.*

Proof. The proof is similar to the one of Lemma 2.3. From (2.9), we deduce that

$$d^2(1 - \tfrac{u}{1+u}) \leqslant \mathrm{RN}(d^2) \leqslant d^2(1 + \tfrac{u}{1+u}).$$

Then, we deduce that $\delta(1 - u/(1+u)) \leqslant c^2 + \mathrm{RN}(d^2) \leqslant \delta(1 + u/(1+u))$ so that (2.9) implies

$$\delta(1 - \tfrac{u}{1+u})^2 \leqslant \widehat{\delta} \leqslant \delta(1 + \tfrac{u}{1+u})^2,$$

which concludes the proof. □

The use of the refined model led us to a simpler and smaller bound on the relative error generated when computing $\widehat{\delta}$, with no additional calculations. This improved intermediate result and the refined model can be applied to the proof of Theorem 2.4. In this case, the term in u^2 cannot be removed but it is slightly lowered (compared to Theorem 2.4 and to the result given in [21]).

Theorem 2.7. *For any precision $p \geqslant 5$, the relative error when computing $\widehat{R}$ is bounded by $5u + 12u^2$.*

Proof. Applying Lemma 2.6 and the same reasoning as for the proof of Theorem 2.4, we get

$$\widehat{R} = R\frac{(1+\epsilon_2)(1+\epsilon_3)}{1+\epsilon_1} \quad \text{with } |\epsilon_1|, |\epsilon_2| \leqslant 2u \text{ and } |\epsilon_3| \leqslant u/(1+u).$$

We conclude by checking the inequalities $(1+2u)(1+u/(1+u))/(1-2u) \leqslant 5u + 12u^2$ when $p \geqslant 5$. □

The standard model can be refined further by computing a specific error bound for each basic operation: for instance, the floating-point addition is a special case of rounding, when the input to be rounded is the sum of two floating-point numbers. This is addressed by Jeannerod and Rump in [24] and we will also focus on the squaring operation in Chapter 3.

2.4 Tightness of an error bound

At this point, it is important to know if the bounds that we proved are tight: if the known bound is tight, then it is not worth spending more time to get a non significant improvement, unless the purpose is to get a simpler bound as we did for Lemma 2.6. Another reason is that the error bounds are to be used to compare different algorithms approximating the same quantity. We describe various algorithms to compute a two-by-two determinant in Section 1.3; in order to reliably compare these algorithms in terms of accuracy, we need to get a tight error bound for each algorithm. Finally, we consider here basic building blocks for larger algorithms: getting a small improvement on some error bounds may lead to a significant improvement for a larger algorithm.

If we consider the naive algorithm for evaluating a two-by-two determinant described as Algorithm 1 in Chapter 1, a quick analysis leads to the bound $(2u+u^2)(|ad|+|bc|)/(ad-bc)$ on the relative error. This bound alone can be very large in case of a cancellation but it does not imply that the result is inaccurate: to show this inaccuracy, a set of inputs for which the algorithm is inaccurate had to be exhibited. In this specific case, a set of inputs parametrized by the radix and the precision is provided in [21, §I.(3)]. Similarly, for the analysis of the complex floating-point division described in Sections 2.2 and 2.3, the authors provided two kinds of information about the sharpness of the bound:

- a set of inputs parametrized by the precision p, (p even), for which the generated relative error is equivalent to the error bound $2u$, as the precision goes to infinity;
- some numerical examples for the standard odd precisions (53, 113), for which the relative error generated is close to the bound.

Finally, in the case of the refined standard model, the inequality in (2.9) is attained for $x = 1 + u$.

These examples describe different situations that can occur in practice, in the case of bounds that are independent on the inputs: we define these situations more rigorously below. For a bound $B(p)$ on an error function $E(p, a, \ldots)$, three main situations can occur:

- the bound is **optimal** if there exist inputs $a(p), \ldots$, for which $B(p) = E(p, a(p), \ldots)$, for any precision. This is the case for the refined model (2.9). It means that the bound cannot be improved without taking the inputs into account.
- the bound is **asymptotically optimal** if there exist inputs $a(p), \ldots$, for any precision, such that $B(p)$ and $E(p, a(p), \ldots)$ are equivalent as the precision goes to infinity. This is the case for the even precision with the complex floating-point division. It means that the dominating term in the bound cannot be improved; the bound can still be improved and/or simplified as we did from Section 2.2 to Section 2.3.
- the bound is **sharp** if there exist inputs $a(p), \ldots$, for different precisions (at least the standard ones), such that, for these precisions, $B(p)$ and $E(p, a(p), \ldots)$ are close (see [39] for a similar use of the word "sharp"). In this case, we do not have much hope of improving the dominating term of the bound, but this is not proved.

In practice, we are looking for examples to indicate the tightness of our bounds, either examples that are parametrized by the precision to prove the optimality or asymptotic optimality of a bound, of numerical ones to prove the sharpness of a bound.

Part II

Analysis of a complex inversion algorithm

In this second part, we consider complex floating-point numbers represented as $z = x + iy$ where x and y are floating-point numbers. We assume that the underlying floating-point arithmetic has radix 2 and precision $p \geqslant 2$; as explained at the end of Section 1.1, we also assume an unbounded exponent range, which means that our results apply to practical floating-point calculations according to the IEEE 754 standard [17] as long as underflow and overflow do not occur. The accuracy of a complex result that is computed using classical floating-point arithmetic can be assessed in two ways: if $z = x + iy$ is the exact result and $\widehat{z} = \widehat{x} + i\widehat{y}$ is the computed result, we can perform

- a **componentwise** (relative) error analysis: we are looking for a nonnegative function $B_C(p)$ such that for any inputs in $\mathbb{F}_p$,

$$|\widehat{x} - x| \leqslant B_C(p) \cdot |x| \quad \text{and} \quad |\widehat{y} - y| \leqslant B_C(p) \cdot |y|. \tag{2.10}$$

  When x and y do not vanish, $B_C(p)$ is a valid bound on the componentwise relative error $E_C = \max(|\widehat{x} - x|/|x|, |\widehat{y} - y|/|y|)$.

- a **normwise** (relative) error analysis: we are looking for a nonnegative function $B_N(p)$ such that for any inputs in $\mathbb{F}_p$,

$$|\widehat{z} - z| \leqslant B_N(p) \cdot |z|. \tag{2.11}$$

  When z does not vanish, $B_N(p)$ is a valid bound on the normwise relative error $E_N = |\widehat{z} - z|/|z|$.

We only described here the case of uniform bounds (bounds that depend only on β and p, not on the inputs) because in this chapter, we only consider basic operations on complex floating-point numbers: in most cases, we are able to find satisfactory uniform error bounds, which are easier to integrate into larger error analyses.

One should notice that any componentwise relative error bound is also a valid normwise relative error bound. If $B(p)$ satisfies (2.10), we have $|\widehat{z}-z|^2 = (\widehat{x}-x)^2+(\widehat{y}-y)^2 \leqslant B(p)^2 \cdot (x^2+y^2)$, so that $B(p)$ also satisfies (2.11). As a consequence, performing a componentwise relative error analysis of an algorithm is a method to get a normwise error bound. We will see some examples for which this method leads to an asymptotically optimal normwise error bound. However, it is not the case for complex floating-point inversion: the direct normwise analysis presented in Chapter 4 improves the (nearly optimal) componentwise relative error bound that is proven in Chapter 3. In the remainder of this introduction, we summarize known results about the accuracy of the basic complex operations using floating-point arithmetic.

Complex addition and subtraction. With the representation above, the addition (and subtraction) of two complex floating-point numbers is straightforward: the result of the addition or subtraction of $a+ib$ and $c+id$ is given by the formula

$$\mathrm{RN}(a \pm c) + i\mathrm{RN}(b \pm d).$$

As a consequence, the bound $u/(1+u)$ given by the refined model (2.9) is also a bound on both the componentwise and normwise relative errors of complex addition (and subtraction). Moreover, the examples given in [24] to prove the optimality of the refined model for the floating-point addition and subtraction are also valid for the complex operations since any floating-point number can be seen as a complex floating-point number whose imaginary part is zero.

Complex multiplication. The exact formula

$$(a+ib)\cdot(c+id) = (ac-bd) + i(ad+bc)$$

corresponds to the calculation of two two-by-two determinants. A first method to perform this multiplication is to use the classical Algorithm 1, introduced in Section 1.3, which involves six floating-point operations. From a componentwise point of view, we explained that catastrophic cancellations can occur leading to a relative error of the order of magnitude of $1/u$ in binary floating-point arithmetic. However, Brent, Percival and Zimmerman proved in [8] that the normwise relative error is always bounded by $\sqrt{5}u \approx 2.24u$, and that this bound is asymptotically optimal for binary floating-point arithmetic.

Using the FMA operation saves one operation per determinant, if we compute the result as $\mathrm{RN}(ac-\mathrm{RN}(bd))+i\mathrm{RN}(ad+\mathrm{RN}(bc))$. However, it does not solve the possibility of catastrophic cancellation and the example given in Section 1.3 still generates such a behavior. In [19], the authors proved that the normwise relative error is always bounded by $2u$ and that this bound is asymptotically optimal. This algorithm uses only four floating-point operations for a better worst-case normwise accuracy than Algorithm 1.

A third algorithm, based on Kahan's algorithm for the evaluation of a two-by-two determinant (cf. Algorithm 2 in Section 1.3), is more accurate from the componentwise point of view. The componentwise relative error was proved to be bounded by $2u$ in [20] and a certificate for asymptotic optimality is also provided to ensure that the bound $2u$ is asymptotically optimal. As a consequence, the normwise relative error is also bounded by $2u$. This bound was proved to be asymptotically optimal in [19]. The use of Kahan's algorithm leads to a better componentwise accuracy but does not improve the normwise accuracy compared to the previous algorithm, while being much more costly in terms of number of floating-point operations (ten are needed).

Similarly, Cornea, Harrison and Tang proposed Algorithm 4 to compute a two-by-two determinant in [10]. Muller (for the radix 2) then Jeannerod (for any radix β) proved in [31] and [18], respectively, that this algorithm generates a componentwise relative error bounded by $2u+\mathcal{O}(u^2)$ which is asymptotically optimal. The same bound $2u+\mathcal{O}(u^2)$ was proved to be asymptotically optimal also from a normwise point of view in [19]. Using this algorithm to perform a complex multiplication requires 14 floating-point operations.

Algorithm 4 – CHT
(Cornea Harrison and Tang's algorithm to evaluate a two-by-two determinant)

Input: a, b, c and d are floating-point numbers.
Output: $\widehat{x}$ is a floating-point number approximating $x = ad - bc$.

1. $\widehat{v} = \mathrm{RN}(ad)$;
2. $e_v = \mathrm{RN}(ad - \widehat{v})$; // FMA instruction (exact, no rounding occurs)
3. $\widehat{w} = \mathrm{RN}(bc)$;
4. $e_w = \mathrm{RN}(bc - \widehat{w})$; // FMA instruction (exact, no rounding occurs)
5. $\widehat{f} = \mathrm{RN}(\widehat{v} - \widehat{w})$;
6. $\widehat{e} = \mathrm{RN}(e_v + e_w)$;
7. $\widehat{x} = \mathrm{RN}\left(\widehat{f} + \widehat{e}\right)$;
8. **return** $\widehat{x}$;

Complex division. The accuracy of complex division is not as well understood as the one of complex multiplication is. The exact result is given by the formula

$$(a + ib)/(c + id) = (ac + bd)/(c^2 + d^2) - i(ad - bc)/(c^2 + d^2).$$

Normwise relative error bounds for the naive algorithm are given by Champagne in [9, p.29] ($5.45u + \mathcal{O}(u^2)$) and by Higham in [16] ($5\sqrt{2}u + \mathcal{O}(u^2) \approx 7.07u + \mathcal{O}(u^2)$). Moreover, Baudin remarked in [4] that the formula consists in multiplying the numerator $(a + ib)$ by the conjugate of the denominator $(c - id)$ before dividing by the floating-point

number computed for $c^2 + d^2$: he deduced that every division algorithm based on this scheme admits a normwise relative error bound of the form $(3 + \alpha)u + \mathcal{O}(u^2)$, where $\alpha \cdot u$ is the first order term of the error bound associated to the multiplication algorithm. For instance, computing a complex multiplication with the naive algorithm generates a normwise relative error bounded by $\sqrt{5}u$ so that the naive division algorithm generates a normwise relative error bounded by $(3 + \sqrt{5})u + \mathcal{O}(u^2) \approx 5.24u + \mathcal{O}(u^2)$. In the case of Kahan's algorithm (Algorithm 2), it leads to the bound $5u + \mathcal{O}(u^2)$; as mentioned in Chapter 2, the componentwise relative error bound $5u + \mathcal{O}(u^2)$ was derived and proved asymptotically optimal for the even precisions in [21]. For the normwise error bounds, we do not have much information about theirs tightness. Because the parameter space of complex division is four dimensional, it is difficult to find examples to assess the tightness of the previous normwise error bounds.

Complex inversion. Since finding a tight bound for complex division is difficult, we decided to analyze as an intermediate step a particular case of division: complex inversion. This part deals with the accuracy of the inversion of a nonzero complex number given by its real and imaginary parts as floating-point numbers. Given a nonzero complex number $a + ib$, its inverse $z = x + iy$ satisfies

$$x = \frac{a}{a^2 + b^2}, \qquad y = -\frac{b}{a^2 + b^2}. \tag{2.12}$$

Assuming a and b are floating-point numbers, we focus on the approximation $\widehat{z} = \widehat{x} + i\,\widehat{y}$ that can be computed classically in floating-point arithmetic according to

$$\widehat{x} = \mathrm{RN}\left(\frac{a}{\mathrm{RN}(\mathrm{RN}(a^2) + \mathrm{RN}(b^2))}\right) \tag{2.13}$$

for the real part, and with a similar expression for the imaginary part $\widehat{y}$. This scheme corresponds to Algorithm 5 below.

Algorithm 5 – Inversion
(Inversion of a nonzero complex floating-point number)
Input: $a + ib$ is a nonzero complex floating-point number.
Output: $\widehat{z} = \widehat{x} + i\widehat{y}$ approximates the inverse of $a + ib$.

1. $s_a = \mathrm{RN}(a^2)$;
2. $s_b = \mathrm{RN}(b^2)$;
3. $s = \mathrm{RN}(s_a + s_b)$;
4. $\widehat{x} = \mathrm{RN}(a/s)$;
5. $\widehat{y} = \mathrm{RN}(-b/s)$;
6. **return** $\widehat{x} + i\widehat{y}$;

We provide an accuracy analysis of Algorithm 5, with $\beta = 2$, for both the componentwise relative error $\mathrm{E_C} = \max\bigl(|x - \widehat{x}|/|x|, |y - \widehat{y}|/|y|\bigr)$ and the normwise relative error $\mathrm{E_N} = |z - \widehat{z}|/|z|$. For this algorithm, one can already notice that no cancellation can occur (compared to the case of complex division based on Algorithm 1 for the numerators). In each case, we bound the *largest* error by a function $B(p)$ depending only on the precision p, and study the tightness of that bound as described in Section 2.4. We recall that in this context, we typically distinguish between three levels of quality: optimal, asymptotically optimal or sharp error bounds.

The componentwise relative error generated by Algorithm 5 can easily be bounded according to $\mathrm{E_C} \leqslant 3u + \mathcal{O}(u^2)$, where $u = 2^{-p}$ is the unit roundoff. Our first contribution is to show that the term $\mathcal{O}(u^2)$ can in fact be removed, which leads to the simpler bound $\mathrm{E_C} \leqslant 3u$ (assuming $p \geqslant 4$). Furthermore, when p is even, we show that this bound is asymptotically optimal by providing floating-point numbers a and b parametrized by p and for which $\mathrm{E_C} \geqslant 3u - \frac{31}{2}u^{\frac{3}{2}} + \mathcal{O}(u^2)$; when p is odd, we show that the bound $3u$ is sharp, especially for the corresponding basic IEEE 754 binary formats ($p = 53, 113$). This is explained in Chapter 3.

The normwise relative error bound $\mathrm{E_N} \leqslant 3u + \mathcal{O}(u^2)$ can be found in [9, p. 30], and a direct application of our componentwise error analysis leads further to $\mathrm{E_N} \leqslant 3u$. Our second main contribution is to show that if $p \geqslant 10$ then the following smaller bound holds: $\mathrm{E_N} < \gamma u + 9u^2$, where γ is an explicit constant in $(2.70712, 2.70713)$. When using for example the IEEE 754 binary32 format ($p = 24$), this implies $\mathrm{E_N} < 2.707131u$. The techniques and the case distinction we use to prove this bound are inspired from [39], but we also extensively use real analysis and differentiation for the treatment of each case. We provide numerical examples to show that the bound we obtain is sharp for the basic IEEE 754 formats ($p = 24, 53, 113$). This is explained in Chapter 4.

We conclude this part with some remarks on the implications of these error analyses for complex floating-point division. The technical parts of the proofs that can be skipped at first reading are gathered in Appendix A.

Several authors [37, 38, 35, 5] have suggested ways of avoiding spurious overflows and underflows in complex division, and some of them may be used in Algorithm 5. Of course, if the computation introduces further rounding errors, which is the case for example in Smith's method [37], then our error bounds may not hold anymore. However, following the technique developed by Priest in [35], it is possible to scale a and b by a power of two in order to avoid spurious overflows and underflows without introducing new rounding errors: in that case, our analyses are valid if neither overflow nor underflow occurs during the computation. Nonetheless, we do not deal with scaling techniques here, and focus only on the largest error assuming an unbounded exponent range.

Chapter 3

Componentwise error analysis of complex inversion

In this chapter, we focus on the componentwise relative error of Algorithm 5 which computes an approximation of the inverse of a nonzero complex floating-point number $a+ib$, a and b being floating-point numbers.

We note first that since $a+ib$ is nonzero, $x = a/(a^2+b^2)$ and $y = -b/(a^2+b^2)$ cannot be both zero, and that if one of them is zero then the returned result is very accurate. Assume for example that $x = 0$ (the case $y = 0$ is similar). In that case, $a = 0$ and $y = -1/b$. Using the refined model (2.9), it is then easily checked that the values $\widehat{x}$ and $\widehat{y}$ returned by the algorithm are as follows:

- $\widehat{x} = 0$, which means that the real part is computed exactly;
- $\widehat{y} = -\mathrm{RN}(b/\mathrm{RN}(b^2))$ and the relative error on the imaginary part is bounded by $2u$ (and thus smaller than the bound we are going to give in the general case). Indeed, we have $\widehat{y} = -1/b \cdot (1+\epsilon_2)/(1+\epsilon_1)$, with $|\epsilon_i| \leqslant u/(1+u)$ using (2.9).

Therefore, the remainder of this chapter is devoted to analyzing $\mathrm{E_C} = \max\left(|x-\widehat{x}|/|x|, |y-\widehat{y}|/|y|\right)$ for x and y nonzero. Repeated applications of the bound in (2.9) give immediately $\mathrm{E_C} \leqslant 3u + \mathcal{O}(u^2)$. We show below that if $p \geqslant 4$, then the $\mathcal{O}(u^2)$ term can in fact be removed, leading to the simpler bound $3u$.

To do this, we prove that if $p \neq 3$, then the relative error bound $u/(1+u)$ in the refined standard model (2.9) can be replaced by $u/(1+3u)$ when evaluating a square $\mathrm{RN}(a^2)$ instead of a general product (when $p = 3$, it is easily checked that the bound $u/(1+u)$ is attained when squaring the floating-point numbers $3/2 \cdot 2^e$, $e \in \mathbb{Z}$). This slight refinement will turn out to be enough to show that Algorithm 5 satisfies $\mathrm{E_C} \leqslant 3u$. Finally, we prove that this error bound is asymptotically optimal for the even precisions and sharp for the odd ones.

3.1 Error analysis of the squaring operation

As mentioned earlier, a specific error analysis is performed for the squaring operation (approximating the square of a binary floating-point number). For this purpose, we first

prove that the square of a binary floating-point number cannot be too close to $2+2u$, as soon as the precision is not equal to 3.

Lemma 3.1. *Let a be a floating-point number. If $p \neq 3$ then $|a^2 - (2+2u)| \geqslant 4u^2$.*

Proof. If $|a| < 1$ then $|a^2 - (2+2u)| > 1 + 2u$, and the result follows from the fact that $1 + 2u > 4u^2$ for any $p > 0$. Assume now that $|a| \geqslant 1$. To handle this case, we show first that

$$a^2 = 2 + 2u \quad \Rightarrow \quad p = 3. \tag{3.1}$$

Since $|a|$ is a floating-point number not smaller than 1, there exists a positive integer A such that $|a| = A \cdot 2^{1-p} = A \cdot 2u$. The equality $a^2 = 2 + 2u$ is thus equivalent to $A^2 = (2^p + 1) \cdot 2^{p-1}$ and, using the (unique) decomposition $A = (2B + 1) \cdot 2^C$ with $B, C \in \mathbb{N}$, it can also be rewritten $(2B + 1)^2 \cdot 2^{2C} = (2^p + 1) \cdot 2^{p-1}$. Now, $p > 0$ implies that $2^p + 1$ is odd and at least 3, so $B \neq 0$ and $(2B + 1)^2 = 2^p + 1$. The latter equality can be rewritten as $4B(B + 1) = 2^p$ and its unique solution over $\mathbb{N}^2_{>0}$ is $(B, p) = (1, 3)$, so (3.1) follows.

If $p \neq 3$ then, by (3.1) we have $a^2 \neq 2 + 2u$, that is, $A^2 \neq (2^p + 1) \cdot 2^{p-1}$. Since the latter inequality involves only integers, it is equivalent to $|A^2 - (2^p + 1) \cdot 2^{p-1}| \geqslant 1$ and thus to $|a^2 - (2 + 2u)| \geqslant 4u^2$. □

Then we prove that the relative error generated when approximating the square of binary floating-point number in precision $p \neq 3$ is bounded by $u/(1 + 3u)$, thus refining further the standard model. The analysis is split into four cases: three cases for which the result is close to $2 + 2u$ (yet not too close according to Lemma 3.1), the last case covers the remaining domain.

Lemma 3.2. *Let a be a floating-point number. If $p \neq 3$ then $\mathrm{RN}(a^2) = a^2(1 + \epsilon)$ with $|\epsilon| \leqslant u/(1 + 3u)$.*

Proof. Since scaling a by a power of the 2 does not modify the generated error, we can assume that $1 \leqslant a < 2$. If $a = 1$ then $\mathrm{RN}(a^2) = a^2$ and the result is clear. If $1 < a < \sqrt{2}$ then it follows from a being a floating-point number that $p \geqslant 4$ and that a belongs to the non-empty interval $[1 + 2u, \sqrt{2})$. Consequently, $1 + 4u < a^2 < 2$ and thus $|\epsilon| \leqslant u \,\mathrm{ufp}(a^2)/a^2 = u/a^2 < u/(1 + 4u)$. Finally, if $\sqrt{2} < a < 2$ then $2 < a^2 < 4$ and, by Lemma 3.1, it suffices to consider the following four subcases:

- If $2 < a^2 \leqslant 2 + 2u - 4u^2$ then $\mathrm{RN}(a^2) = 2$ and, therefore,

$$|\epsilon| = 1 - \frac{2}{a^2} \leqslant 1 - \frac{2}{2 + 2u - 4u^2} \leqslant \frac{u}{1 + 3u}.$$

- If $2 + 2u + 4u^2 \leqslant a^2 < 2 + 4u$ then $\mathrm{RN}(a^2) = 2 + 4u$ and, therefore,

$$|\epsilon| = \frac{2 + 4u}{a^2} - 1 \leqslant \frac{2 + 4u}{2 + 2u + 4u^2} - 1 \leqslant \frac{u}{1 + 3u}.$$

- If $2+4u \leqslant a^2 < 2+6u$ then $\mathrm{RN}(a^2) = 2+4u$ and, therefore,

$$|\epsilon| = 1 - \frac{2+4u}{a^2} \leqslant 1 - \frac{2+4u}{2+6u} = \frac{u}{1+3u}.$$

- If $2+6u \leqslant a^2 < 4$ then $\mathrm{ufp}(a^2) = 2$ and $|\epsilon| \leqslant 2u/a^2 \leqslant 2u/(2+6u) = u/(1+3u)$.

□

3.2 Componentwise error bound for complex inversion

In this section, we prove that the term $\mathcal{O}(u^2)$ can be removed from the bound $3u+\mathcal{O}(u^2)$ in the componentwise relative error analysis of Algorithm 5 for the complex inversion in binary floating-point arithmetic. The proof follows the classical scheme described in Sections 2.2 and 2.3, and then relies on the refined bound given by Lemma 3.2 to conclude.

Theorem 3.3. *If $p \geqslant 4$ then the componentwise relative error for Algorithm 5 satisfies $\mathrm{E_C} \leqslant 3u$.*

Proof. Due to the symmetry of Algorithm 5, it suffices to show that $|x-\widehat{x}| \leqslant 3u|x|$. From the refined model (2.9) and Lemma 3.2 we have

$$s_a = a^2(1+\epsilon_a), \qquad s_b = b^2(1+\epsilon_b), \qquad s = (s_a+s_b)(1+\epsilon_s), \qquad \widehat{x} = \frac{a}{s}(1+\epsilon_x)$$

with $|\epsilon_a|, |\epsilon_b| \leqslant u/(1+3u)$ and $|\epsilon_s|, |\epsilon_x| \leqslant u/(1+u)$. Hence

$$\widehat{x} = \frac{a}{a^2(1+\epsilon_a)+b^2(1+\epsilon_b)} \cdot \frac{1+\epsilon_x}{1+\epsilon_s}$$

and, using $x = a/(a^2+b^2)$, we deduce that $\varphi x \leqslant \widehat{x} \leqslant \varphi' x$ with

$$\varphi := \frac{1-\frac{u}{1+u}}{(1+\frac{u}{1+3u})(1+\frac{u}{1+u})} \qquad \text{and} \qquad \varphi' := \frac{1+\frac{u}{1+u}}{(1-\frac{u}{1+3u})(1-\frac{u}{1+u})}.$$

It is easily checked that $\varphi > 1-3u$ and $\varphi' = 1+3u$, which completes the proof. □

We conclude this section by showing that the componentwise bound $\mathrm{E_C} \leqslant 3u$ is asymptotically optimal or sharp for the even and odd precisions respectively. More precisely, when the precision p is even, the following example shows that the componentwise error bound $3u$ is asymptotically optimal as $p \to \infty$. Assuming an even $p \geqslant 12$, let us consider the following symbolic binary floating-point numbers in precision p:

$$\begin{aligned} a &= 2^{\frac{p}{2}-1} + 5 \cdot 2^{-2} + 2^{-\frac{p}{2}+2}, \\ b &= 2^{p-1} + 2^{\frac{p}{2}-1} + 1. \end{aligned}$$

Observing that $a = (2^{p-3} + 5 \cdot 2^{p/2-4} + 1) \cdot 2^{-p/2+2}$, it can be checked that both a and b are indeed binary floating-point numbers in precision p, according to the definition with an

unbounded exponent range given by (1.2). With these values as inputs of Algorithm 5, let us prove, assuming that $p \geqslant 12$ is even, that the intermediate and final results are:

$$\begin{aligned} s_a &= 2^{p-2} + 5 \cdot 2^{\frac{p}{2}-2} + 11 \cdot 2^{-1}, \\ s_b &= 2^{2p-2} + 2^{\frac{3p}{2}-1} + 3 \cdot 2^{p-1}, \\ s &= 2^{2p-2} + 2^{\frac{3p}{2}-1} + 2^{p+1}, \\ \widehat{x} &= 2^{-\frac{3p}{2}+1} + 2^{-2p} - 2^{-\frac{5p}{2}+2}. \end{aligned}$$

We first check that these expressions are floating-point numbers, observing that they can be written as in (1.2):

$$\begin{aligned} s_a &= (2^{p-1} + 5 \cdot 2^{\frac{p}{2}-1} + 11) \cdot 2^{-1}, \\ s_b &= (2^{p-1} + 2^{\frac{p}{2}} + 3) \cdot 2^{p-1}, \\ s &= (2^{p-1} + 2^{\frac{p}{2}} + 4) \cdot 2^{p-1}, \\ \widehat{x} &= (2^{p-1} + 2^{\frac{p}{2}-2} - 1)2^{-\frac{5p}{2}+2}. \end{aligned}$$

Then we prove that $s_a = \mathrm{RN}(a^2)$, $s_b = \mathrm{RN}(b^2)$, $s = \mathrm{RN}(s_a + s_b)$ and $\widehat{x} = \mathrm{RN}(a/s)$ using Property 2.2 of Section 2.1. For each equality, we compute the exact result and compare the difference between this exact result and the announced computed result to one half of the unit in the last place of the exact result.

Computation of $\mathrm{RN}(a^2)$**:** The exact result is $a^2 = 2^{p-2} + 5 \cdot 2^{\frac{p}{2}-2} + 11 \cdot 2^{-1} + 2^{-4} + 10 \cdot 2^{-\frac{p}{2}} + 2^{-p+4}$ and $\mathrm{ulp}(s_a) = 2^{-1}$. Hence, we have

$$|a^2 - s_a| = 2^{-4} + 10 \cdot 2^{-\frac{p}{2}} + 2^{4-p} < 2^{-2} = \tfrac{1}{2}\mathrm{ulp}(s_a)$$

and $s_a = \mathrm{RN}(a^2)$.

Computation of $\mathrm{RN}(b^2)$**:** The exact result is $b^2 = 2^{2p-2} + 2^{\frac{3p}{2}-1} + 2^p + 2^{p-2} + 2^{\frac{p}{2}} + 1$ and $\mathrm{ulp}(s_b) = 2^{p-1}$. Hence, we have

$$|b^2 - s_b| = 2^{p-2} - 2^{\frac{p}{2}} - 1 < 2^{p-2} = \tfrac{1}{2}\mathrm{ulp}(s_b)$$

and $s_b = \mathrm{RN}(b^2)$.

Computation of $\mathrm{RN}(s_a + s_b)$**:** The exact result is $s_a + s_b = 2^{2p-2} + 2^{\frac{3p}{2}-1} + 3 \cdot 2^{p-1} + 2^{p-2} + 5 \cdot 2^{\frac{p}{2}-2} + 11 \cdot 2^{-1}$ and $\mathrm{ulp}(s) = 2^{p-1}$. Hence, we have

$$|s_a + s_b - s| = 2^{p-2} - 5 \cdot 2^{\frac{p}{2}-2} - 11 \cdot 2^{-1} < 2^{p-2} = \tfrac{1}{2}\mathrm{ulp}(s)$$

and $s = \mathrm{RN}(s_a + s_b)$.

Computation of $\mathrm{RN}(a/s)$**:** The exact result is

$$\frac{a}{s} = 2^{-\frac{3p}{2}+1} + 2^{-2p} - 2^{-\frac{5p}{2}+1} - 2^{-3p+2} + \mathcal{O}\big(2^{-\frac{7p}{2}}\big).$$

Moreover, $\mathrm{ulp}(\widehat{x}) = 2^{-\frac{5p}{2}+2}$. Hence, we have

$$\left|\frac{a}{s} - \widehat{x}\right| = \frac{2^{-\frac{5p}{2}+1} + 2^{-\frac{7p}{2}+5}}{1 + 2^{-\frac{p}{2}+1} + 2^{-p+3}} < 2^{-\frac{5p}{2}+1} = \tfrac{1}{2}\mathrm{ulp}(\widehat{x})$$

and $\widehat{x} = \mathrm{RN}(a/s)$.

Since $x = a/(a^2+b^2)$, we deduce that

$$\frac{x - \widehat{x}}{x} = 3u - \tfrac{31}{2}u^{\frac{3}{2}} + \mathcal{O}(u^2).$$

As a consequence, in this example the componentwise relative error in the computed $\widehat{x}$ is at least $3u - \frac{31}{2}u^{\frac{3}{2}} + \mathcal{O}(u^2)$, which shows the asymptotic optimality (as $p \to \infty$) of the bound when p is even.

When p is odd, we have not found an input set parametrized by the precision to prove the asymptotic optimality of the error bound $3u$. However, we illustrate the sharpness of the bound by numerical examples in Table 3.1. To find such examples, we first run the algorithm on a very large set of inputs for low precisions, keeping track of the worst generated errors. Thus, we find examples for the lowest precisions and we also guess some possible locations for examples in higher precisions. We also take into account the details of the analysis to refine these locations (e.g. most exact operations should return a result slightly above a power of 2 so that the relative error generated by the rounding may be close to the bound given by the refined model (2.9)) and then run a large bunch of tests.

p	Inputs a and b	E_C/u
15	$a = 16732$ $b = 23252{\cdot}2^3$	2.93047…
17	$a = 66078$ $b = 93014{\cdot}2^8$	2.96359…
19	$a = 131435$ $b = 370969{\cdot}2^8$	2.98509…
53	$a = 4508053433127332$ $b = 6369149602646415{\cdot}2^{16}$	2.97894…
113	$a = 5192393427440123027423416459819356$ $b = 7343016638055329519853569740503421{\cdot}2^{16}$	2.97647…

Table 3.1: Examples with p odd and a componentwise relative error close to $3u$.

Chapter 4

Normwise error analysis of complex inversion

In this chapter, we focus on the normwise relative error of Algorithm 5, that is,

$$\mathrm{E_N} = \sqrt{a^2+b^2}\sqrt{(x-\widehat{x})^2+(y-\widehat{y})^2}.$$

As explained in the introduction to Part II, the componentwise bound $3u$, obtained in Chapter 3, is also a valid normwise error bound. In this section, we establish the following result, which achieves a smaller bound by keeping track of the correlations between the various rounding errors committed in the algorithm.

Theorem 4.1. *If $p \geqslant 10$ then the normwise relative error for Algorithm 5 satisfies $\mathrm{E_N} \leqslant \gamma u + 9u^2$, where γ is defined by*

$$\gamma = \frac{\sqrt{8778980525057 + 16793600(8\sqrt{2}-\sqrt{127}) - 550842155008\sqrt{254}}}{8192\,(16-\sqrt{254})}, \tag{4.1}$$

and is such that $\gamma \in (2.70712, 2.70713)$.

If $p \geqslant 10$, $\mathrm{E_N} < 2.70713u + 9u^2$ is therefore a rigorous bound for the normwise error of Algorithm 5. It should also be noticed that the second order term in the error bound can be absorbed by the first order term, at the cost of a slight overestimation: for example, for $p \geqslant 24$, we have $9u = 9\cdot 2^{-24} < 10^{-6}$ so that $\mathrm{E_N} < 2.707131u$. The numerical examples listed in Table 4.1 show that the error bound of Theorem 4.1 is sharp for the basic IEEE 754 formats ($p = 24, 53, 113$).

In Section 4.1, we derive, from the inequalities (2.6) and (2.4), an upper bound on the normwise relative error of the form:

$$E_N \leqslant \sqrt{f_2(a,b)\cdot u^2 + f_3(a,b)\cdot u^3}.$$

We also perform a first domain reduction that leads to the uniform upper bound $f_3(a,b) < 25$, for $p \geqslant 10$. Then, we address some corner cases for which f_2 is easily bounded by $(2+\sqrt{2}/2+3u)^2$ so that the bound given in Theorem 4.1 follows. The error analysis on the remaining domain is then detailed in Section 4.2, through seven cases summarized at the beginning of the section.

p	Inputs a and b	E_N/u
24	$a = 11863283$ $b = 11865457 \cdot 2^{12}$	2.69090...
53	$a = 4503599709991314$ $b = 6369051770002436 \cdot 2^{26}$	2.70679...
113	$a = 2^{112}$ $b = 7343016637207171132572330391109909 \cdot 2^{56}$	2.70559...

Table 4.1: Examples with a normwise relative error close to γu.

4.1 Preliminaries

4.1.1 A first nonuniform bound

The first step in the error analysis of Algorithm 5 is to reduce the input domain. Since the function RN is symmetric with respect to zero, the signs of a and b are not relevant and we can assume that both a and b are nonnegative. Swapping the inputs a and b does not affect the relative error; moreover, if $a = 0$, then a simple analysis (already detailed at the beginning of Chapter 3), based on the refined model (2.9), leads to the upper bound $2u$ for $\mathrm{E_N}$, so we can assume that $0 < a \leqslant b$. Finally, multiplying or dividing by two both a and b does not affect either the relative error, and we can restrict the analysis to the case $1 \leqslant b < 2$.

From the definition of the ufp function (2.3) and the input range reduction above, we know that $\mathrm{ufp}(b^2) \in \{1, 2\}$. Moreover, b is a floating-point number, so that $1 \leqslant b \leqslant 2 - 2u$ and thus $1 \leqslant b^2 \leqslant 4 - 4u$. Since $4 - 4u$ is a floating-point number, and using the monotonicity of the rounding function RN, we deduce that $1 \leqslant s_b < 4$. Using again the monotonicity of RN, we also deduce that $0 < s_a < 4$. Hence $1 < s_a + s_b < 8$, which implies $\mathrm{ufp}(s_a + s_b) \in \{1, 2, 4\}$.

We now define δ_a, δ_b, δ_s, δ_x, and δ_y as follows:

$$\begin{aligned}
s_a &= a^2 + \delta_a u, & |\delta_a| &\leqslant \mathrm{ufp}\big(a^2\big),\\
s_b &= b^2 + \delta_b u, & |\delta_b| &\leqslant \mathrm{ufp}\big(b^2\big),\\
s &= s_a + s_b + \delta_s u, & |\delta_s| &\leqslant \mathrm{ufp}(s_a + s_b),\\
\widehat{x} &= \tfrac{a}{s} + \delta_x u, & |\delta_x| &\leqslant \mathrm{ufp}\big(\tfrac{a}{s}\big),\\
\widehat{y} &= -\left(\tfrac{b}{s} + \delta_y u\right), & |\delta_y| &\leqslant \mathrm{ufp}\big(\tfrac{b}{s}\big).
\end{aligned}$$

Let us also define, for later use,

$$\delta = \delta_a + \delta_b + \delta_s \quad \text{and} \quad \epsilon = \frac{|\delta|}{a^2 + b^2},$$

so that $|\delta|u$ and ϵu are the absolute and relative errors, respectively, in the evaluation of $a^2 + b^2$. Since $0 < a \leqslant b$, $\mathrm{ufp}(b^2) \leqslant 2$ and $\mathrm{ufp}(s_a + s_b) \leqslant 4$, we deduce that $|\delta| \leqslant 8$.

Moreover, it can be deduced from (2.9) that $\epsilon \leqslant 2$. (This bound on ϵ already appeared in [8, p. 1471].)

With these notations, let us also define

$$f_2(a,b) = (a^2+b^2)\Big(\operatorname{ufp}\big(\tfrac{a}{s}\big)^2+\operatorname{ufp}\big(\tfrac{b}{s}\big)^2\Big)+2\frac{|\delta|\Big(\operatorname{ufp}\big(\tfrac{a}{s}\big)a+\operatorname{ufp}\big(\tfrac{b}{s}\big)b\Big)}{a^2+b^2}+\left(\frac{\delta}{a^2+b^2}\right)^2. \quad (4.2)$$

In the remainder of this subsection, we will prove that, for $\kappa \in \mathbb{R}^*$,

$$f_2(a,b) \leqslant \kappa \quad \Rightarrow \quad \mathrm{E_N} \leqslant \sqrt{\kappa}u + \frac{25}{2\sqrt{\kappa}}u^2. \quad (4.3)$$

The inequality in (4.3) will be used repeatedly through the chapter to bound the normwise relative error E_N from a bound on f_2, and finally obtain Theorem 4.1.

To prove (4.3), we start from the equality

$$x - \widehat{x} = \frac{a}{s\,(a^2+b^2)}\delta u - \delta_x u,$$

and since a similar equality holds for $y - \widehat{y}$. We deduce that

$$\frac{\mathrm{E_N^2}}{u^2} = \left(a^2+b^2\right)\left(\delta_x^2+\delta_y^2\right) - 2\frac{\delta(a\delta_x+b\delta_y)}{a^2+b^2+\delta u}+\left(\frac{\delta}{a^2+b^2+\delta u}\right)^2.$$

Then, using the triangular inequality, we obtain

$$\frac{\mathrm{E_N^2}}{u^2} \leqslant (a^2+b^2)\Big(\operatorname{ufp}\big(\tfrac{a}{s}\big)^2+\operatorname{ufp}\big(\tfrac{b}{s}\big)^2\Big)+2\frac{|\delta|\Big(\operatorname{ufp}\big(\tfrac{a}{s}\big)a+\operatorname{ufp}\big(\tfrac{b}{s}\big)b\Big)}{a^2+b^2-|\delta|u}+\left(\frac{\delta}{a^2+b^2-|\delta|u}\right)^2.$$

For $p \geqslant 2$, the inequality $\epsilon u < 1$ holds and we use the equality $\frac{1}{a^2+b^2-|\delta|u} = \frac{1}{a^2+b^2}\left(1+\frac{\epsilon}{1-\epsilon u}u\right)$ and the inequality $\left(1+\frac{\epsilon}{1-\epsilon u}u\right)^2 \leqslant 1+\frac{2\epsilon}{(1-\epsilon u)^2}u$ to get

$$\mathrm{E_N^2} \leqslant f_2(a,b)u^2 + f_3(a,b)u^3, \quad (4.4)$$

with f_2 defined by (4.2) and

$$f_3(a,b) = 2\Big(\operatorname{ufp}\big(\tfrac{a}{s}\big)a+\operatorname{ufp}\big(\tfrac{b}{s}\big)b\Big)\frac{\epsilon^2}{1-\epsilon u}+\frac{2\epsilon^3}{(1-\epsilon u)^2}.$$

As explained in the introduction of the chapter, we first roughly bound f_3 to focus then on f_2 which is the dominant term in (4.4). From (2.4), we have

$$\operatorname{ufp}\big(\tfrac{a}{s}\big)a+\operatorname{ufp}\big(\tfrac{b}{s}\big)b \leqslant \frac{a^2+b^2}{s} \leqslant \frac{a^2+b^2}{a^2+b^2-|\delta|u} = \frac{1}{1-\epsilon u},$$

and since $0 \leqslant \epsilon \leqslant 2$, it follows that $f_3(a,b) \leqslant \frac{2\epsilon^2(1+\epsilon)}{(1-\epsilon u)^2} < 25$ for $p \geqslant 10$. As a consequence, assuming that f_2 is bounded by $\kappa \in \mathbb{R}^*$, we have

$$\mathrm{E_N} \leqslant \sqrt{\kappa u^2+25u^3} = \sqrt{\kappa}u\sqrt{1+\frac{25}{\kappa}u} \leqslant \sqrt{\kappa}u+\frac{25}{2\sqrt{\kappa}}u^2.$$

The next subsection is devoted to some corner cases for which f_2 can be easily bounded so that (4.3) directly implies the bound in Theorem 4.1.

4.1.2 Some corner cases

We can first roughly bound f_2 using the inequality (2.4), which will allow us to conclude in some particular cases and to further reduce the remaining domain to analyze. From (4.2) we have

$$\begin{aligned} f_2(a,b) &\leqslant \left(\frac{a^2+b^2}{a^2+b^2-|\delta|u}\right)^2 + 2\frac{|\delta|\,(a^2+b^2)}{(a^2+b^2)\,(a^2+b^2-|\delta|u)} + \left(\frac{\delta}{a^2+b^2}\right)^2 \\ &= \left(1+\epsilon+\frac{\epsilon}{1-\epsilon u}u\right)^2. \end{aligned}$$

This last bound is increasing with respect to ϵ and u (*i.e.*, decreasing with respect to the precision p). Therefore, if $\epsilon \leqslant 1+\frac{\sqrt{2}}{2}+u$, and as soon as $p \geqslant 5$, we have $f_2(a,b) \leqslant \left(2+\frac{\sqrt{2}}{2}+3u\right)^2$ and, from (4.3),

$$\mathrm{E_N} \leqslant \left(2+\tfrac{\sqrt{2}}{2}\right)u + 8u^2. \tag{4.5}$$

Below are five cases that lead to the inequality $\epsilon \leqslant 1+\frac{\sqrt{2}}{2}+u$, so they will not be addressed in Section 4.2.

• If $a=b$, then $s_a = s_b$ and $s = s_a + s_b$ so that $\delta_s = 0$ and one can check that $\epsilon \leqslant 1$. In this case, the previous bound (4.5) holds and we can continue the analysis assuming that

$$a < b. \tag{4.6}$$

• If $b=1$, then $s_b = b^2 = 1$ and $\delta_b = 0$. Moreover, from (4.6) we have $a<1$, so that $s_a < 1$, which implies $\mathrm{ufp}(1+s_a) = 1$ and $\epsilon \leqslant 1$. Again, the bound (4.5) holds and we can continue the analysis assuming that $1 < b$. In fact, since b is a floating-point number, we can assume that

$$1+2u \leqslant b. \tag{4.7}$$

• If $a=1$, then $\delta_a = 0$ and we can distinguish three cases. If $\mathrm{ufp}(b^2) = 1$ then $\mathrm{ufp}(1+s_b) = 2$ and $\epsilon \leqslant \frac{3}{2}$. If $\mathrm{ufp}(b^2) = 2$ then either $\mathrm{ufp}(1+s_b) = 2$ which implies $\epsilon \leqslant \frac{4}{3}$, or $\mathrm{ufp}(1+s_b) = 4$ and then $\epsilon \leqslant \frac{3}{2}+u$. In all these cases, (4.5) holds, hence we can assume now that

$$a \neq 1. \tag{4.8}$$

• If $a^2+b^2 < \mathrm{ufp}(s_a+s_b)$, then we have $(s_a+s_b) - \mathrm{ufp}(s_a+s_b) < (\delta_a+\delta_b)u \leqslant (a^2+b^2)u < \mathrm{ufp}(s_a+s_b)u = \frac{1}{2}\mathrm{ulp}(s_a+s_b)$. Since $\mathrm{ufp}(s_a+s_b)$ is a floating-point number, we can deduce that $s = \mathrm{RN}(s_a+s_b) = \mathrm{ufp}(s_a+s_b)$ hence $\epsilon \leqslant 1$ and (4.5) holds. In the following, we can then assume that

$$\mathrm{ufp}(s_a+s_b) \leqslant a^2+b^2. \tag{4.9}$$

• One last case is when $s_a+s_b \geqslant \sqrt{2}\,\mathrm{ufp}(s_a+s_b)$. In this case, $\epsilon \leqslant 1+\frac{\sqrt{2}}{2}+u$ and the previous bound (4.5) holds. Therefore, we now assume that

$$s_a+s_b < \sqrt{2}\,\mathrm{ufp}(s_a+s_b). \tag{4.10}$$

4.2 Case analysis

4.2.1 Overview of the case analysis

The analysis goes through the possible values of $\mathrm{ufp}(s_a + s_b)$, which are 1, 2, and 4. In each case, we first deduce upper bounds for $\mathrm{ufp}(b^2)$, $\mathrm{ufp}\left(\frac{a}{s}\right)$, and $\mathrm{ufp}\left(\frac{b}{s}\right)$. This leads to a new function g, which is greater than or equal to f_2, and which depends on a and b as well as on a third parameter, e, defined as the unique integer such that

$$\mathrm{ufp}\left(a^2\right) = 2^{-e}.$$

The function g does not involve floating-point operations anymore and can be seen as a continuous and differentiable function over real inputs. We then look for an upper bound κ on this function over a restricted domain D containing all the floating-point numbers we are interested in, so that $f_2 \leqslant \kappa$ and we then use (4.3). For this step, we mainly use real analysis, especially partial derivatives. In some cases, we can maximize with respect to a and b at the same time. The last step is always to maximize with respect to e, using the change of variable $t = 2^{-e}$ and considering t as a continuous variable.

The analysis is split into seven cases depending on the values of some ufp functions involved in the definition (4.2) of f_2. Note that, since $a^2 < 4$, we have $e \geqslant -1$. In each case but the last one, we end up with a bound smaller than or equal to $\left(2 + \frac{\sqrt{2}}{2}\right)^2$ for f_2, from which we conclude using (4.3) that $\mathrm{E_N} \leqslant \left(2 + \frac{\sqrt{2}}{2}\right)u + 5u^2$. The last case is similar although we have a slightly larger bound $\gamma^2 + 20u$ for f_2 (we have $2 + \frac{\sqrt{2}}{2} = 2.70710\ldots$, while $\gamma = 2.70712\ldots$), which leads to $\mathrm{E_N} \leqslant \gamma u + 9u^2$. The table below summarizes the bounds in each case, under the assumptions (4.6) to (4.10).

$\mathrm{ufp}(s_a + s_b)$	$\mathrm{ufp}(b^2)$	e	$\mathrm{ufp}\left(\frac{a}{s}\right)$	f_2	$\mathrm{E_N}$
1	1	$\geqslant 2$	$\leqslant 2^{-\frac{e}{2}}$	6.565	$2.6u$
4	2	$= -1$	$\leqslant \frac{1}{4}$	$\left(2 + \frac{\sqrt{2}}{2}\right)^2$	$\left(2 + \frac{\sqrt{2}}{2}\right)u + 5u^2$
		$\geqslant 0$	$\leqslant 2^{-2-\frac{e}{2}}$	$\left(\frac{7}{4} + \frac{\sqrt{2}}{2}\right)^2$	$2.5u$
2	1	$\geqslant 1$	$\leqslant 2^{-1-\frac{e}{2}}$	$\left(\frac{7}{4} + \frac{\sqrt{3}}{2}\right)^2$	$2.65u$
		$= 0$	$\leqslant \frac{1}{4}$	$\left(\frac{5}{2}\right)^2$	$\frac{5}{2}u + 5u^2$
	2	$\geqslant 1$	$\leqslant 2^{-\frac{3+e}{2}}$	$\left(2 + \frac{\sqrt{2}}{2}\right)^2$	$\left(2 + \frac{\sqrt{2}}{2}\right)u + 5u^2$
		$\geqslant 2$, even	$= 2^{-1-\frac{e}{2}}$	$\gamma^2 + 20u$	$\gamma u + 9u^2$

We give all the details of the analysis of the first case. For the other cases, we only give a sketch of the analysis, while deferring the details to Appendix A, in Sections A.1 to A.5.

4.2.2 Case $\mathrm{ufp}(s_a + s_b) = 1$

In this case, we can deduce from (4.10) that $1 \leqslant s_a + s_b < \sqrt{2}$. As a consequence, we must have $b < \sqrt{2}$ (otherwise we would have $s_a + s_b > 2$), hence

$$\mathrm{ufp}\big(b^2\big) = 1.$$

Since $s_a < \sqrt{2} - 1 < \frac{1}{2}$ and $s_a = \mathrm{RN}(a^2)$, we have $a^2 < \frac{1}{2}$, and

$$e \geqslant 2.$$

Moreover, we know from (4.7) that $b \geqslant 1 + 2u$ so we have $b^2 \geqslant b(1 + 2u) \geqslant b + 2u$, which is a floating-point number because $\mathrm{ufp}(b) = 1$. Consequently $s_b \geqslant b + 2u$ and $s \geqslant s_a + s_b - u \geqslant s_a + b + u > b$, hence $\frac{b}{s} < 1$, which implies

$$\mathrm{ufp}\big(\tfrac{b}{s}\big) \leqslant \tfrac{1}{2}.$$

Finally, $s = \mathrm{RN}(s_a + s_b) \geqslant 1$ so $\frac{a}{s} \leqslant a < 2^{\frac{1-e}{2}}$ and

$$\mathrm{ufp}\big(\tfrac{a}{s}\big) \leqslant 2^{-\frac{e}{2}}.$$

Therefore, using (4.2) we deduce in this case that $f_2(a, b) \leqslant g_1(a, b, e)$, with

$$g_1(a, b, e) := \left(a^2 + b^2\right)\left(2^{-e} + \tfrac{1}{4}\right) + 2\frac{\left(2 + 2^{-e}\right)\left(2^{-\frac{e}{2}}a + \frac{b}{2}\right)}{a^2 + b^2} + \left(\frac{2 + 2^{-e}}{a^2 + b^2}\right)^2.$$

Let us now characterize explicitly the domain over which we will bound $g_1(a, b, e)$. First, we know that $2^{-\frac{e}{2}} \leqslant a < 2^{\frac{1-e}{2}}$. Next, since $s_a + s_b < \sqrt{2}$ and $s_a > 0$, we have $s_b < \sqrt{2}$, so that $b^2 < \sqrt{2} + u$ and $1 < b < \sqrt{\sqrt{2} + u}$. Finally, we have $a^2 + b^2 \leqslant s_a + \mathrm{ufp}(a^2)u + s_b + \mathrm{ufp}(b^2)u < \sqrt{2} + \frac{5}{4}u$, which concludes the domain analysis: it suffices to look for an upper bound for g_1 over the domain

$$D_1 := \left\{(a, b, e) \mid 2^{-\frac{e}{2}} \leqslant a < 2^{\frac{1-e}{2}}, 1 \leqslant b < \sqrt{\sqrt{2} + u}, a^2 + b^2 < \sqrt{2} + \tfrac{5}{4}u, e \geqslant 2\right\}.$$

We now compute the partial derivatives of g_1 with respect to a and b,

$$\frac{\partial g_1}{\partial a} = 2a\left(2^{-e} + \tfrac{1}{4}\right) + \frac{2 + 2^{-e}}{a^2 + b^2}2^{1-\frac{e}{2}} - 4a\frac{\left(2 + 2^{-e}\right)\left(2^{-\frac{e}{2}}a + \frac{b}{2}\right)}{\left(a^2 + b^2\right)^2} - 4a\frac{\left(2 + 2^{-e}\right)^2}{\left(a^2 + b^2\right)^3},$$

$$\frac{\partial g_1}{\partial b} = 2b\left(2^{-e} + \tfrac{1}{4}\right) + \frac{2 + 2^{-e}}{a^2 + b^2} - 4b\frac{\left(2 + 2^{-e}\right)\left(2^{-\frac{e}{2}}a + \frac{b}{2}\right)}{\left(a^2 + b^2\right)^2} - 4b\frac{\left(2 + 2^{-e}\right)^2}{\left(a^2 + b^2\right)^3},$$

and the next step is to prove that they are both negative over the domain D_1. Since $\frac{1}{b}\frac{\partial}{\partial b}g_1 - \frac{1}{a}\frac{\partial}{\partial a}g_1 = \frac{2+2^{-e}}{a^2+b^2}\left(\frac{1}{b} - \frac{1}{a}2^{1-\frac{e}{2}}\right) < 0$ over D_1, it is sufficient to prove that $\frac{\partial}{\partial a}g_1 < 0$. Since $2a\frac{2+2^{-e}}{a^2+b^2} > 0$, we can rewrite this inequality as

$$\frac{\left(2^{-e} + \frac{1}{4}\right)\left(a^2 + b^2\right)}{2 + 2^{-e}} + \frac{2^{-\frac{e}{2}}}{a} < 2\frac{2^{-\frac{e}{2}}a + \frac{b}{2}}{a^2 + b^2} + 2\frac{2 + 2^{-e}}{\left(a^2 + b^2\right)^2}.$$

This inequality follows from the following three relations:

$$\begin{aligned}
\frac{\left(2^{-e}+\frac{1}{4}\right)(a^2+b^2)}{2+2^{-e}}+\frac{2^{-\frac{e}{2}}}{a} &< \frac{\sqrt{2}+\frac{5}{4}u}{4}+1 \quad \text{for } (a,b,e)\in D_1,\\
\frac{\sqrt{2}+\frac{5}{4}u}{4}+1 &< \frac{1}{\sqrt{2}+\frac{5}{4}u}+\frac{4}{\left(\sqrt{2}+\frac{5}{4}u\right)^2} \quad \text{for } p\geqslant 3,\\
\frac{1}{\sqrt{2}+\frac{5}{4}u}+\frac{4}{\left(\sqrt{2}+\frac{5}{4}u\right)^2} &< 2\frac{2^{-\frac{e}{2}}a+\frac{b}{2}}{a^2+b^2}+2\frac{2+2^{-e}}{(a^2+b^2)^2} \quad \text{for } (a,b,e)\in D_1.
\end{aligned}$$

Since both $\frac{\partial g_1}{\partial a}$ and $\frac{\partial g_1}{\partial b}$ are negative over D_1, since $(a,b,e)\in D_1$ implies $a\geqslant 2^{-\frac{e}{2}}$ and $b\geqslant 1$, and since $(2^{-\frac{e}{2}},1,e)\in D_1$, we deduce that $g_1(a,b,e)\leqslant g_1(2^{-\frac{e}{2}},1,e)=:h_1(t)$, with $t=2^{-e}$ and

$$h_1(t)=(t+1)\left(t+\tfrac{1}{4}\right)+\frac{(t+2)(2t+1)}{t+1}+\left(\frac{t+2}{t+1}\right)^2.$$

Since $e\geqslant 2$, we have $0<t\leqslant\frac{1}{4}$ and

$$h_1'(t)=\frac{8t^4+37t^3+63t^2+43t+1}{4(t+1)^3}>0.$$

Overall, we have $f_2(a,b)\leqslant g_1(a,b,e)\leqslant h_1(t)\leqslant h_1\left(\frac{1}{4}\right)=6.565$. Therefore, we conclude using (4.3) that $E_N\leqslant\sqrt{6.565}u+25/(2\sqrt{6.565})u^2\leqslant 2.6u$ for any $p\geqslant 8$.

4.2.3 Case $\mathrm{ufp}(s_a+s_b)=4$

From (4.10) and (4.6), we know that $4\leqslant s_a+s_b<4\sqrt{2}$ and $s_a<s_b$. As a consequence, we have $2<s_b$ which implies $2<b^2$, so that

$$\mathrm{ufp}\left(b^2\right)=2 \quad \text{and} \quad \sqrt{2}<b\leqslant 2-2u.$$

Since 4 is a floating-point number, we have $s=\mathrm{RN}(s_a+s_b)\geqslant 4$ and $\frac{b}{s}\leqslant\frac{b}{4}<\frac{1}{2}$ hence

$$\mathrm{ufp}\left(\tfrac{b}{s}\right)\leqslant\tfrac{1}{4}.$$

In the same way, $\frac{a}{s}\leqslant\frac{a}{4}<2^{-\frac{3+e}{2}}$ so that

$$\mathrm{ufp}\left(\tfrac{a}{s}\right)\leqslant 2^{-2-\frac{e}{2}}.$$

We now distinguish two subcases, namely $e=-1$ and $e\geqslant 0$.

▶ **Subcase $e=-1$**

We have $\mathrm{ufp}\left(\frac{a}{s}\right)\leqslant 2^{-\frac{3}{2}}$, hence $\mathrm{ufp}\left(\frac{a}{s}\right)\leqslant\frac{1}{4}$, thus we deduce from (4.2) that $f_2(a,b)\leqslant g_2(a,b)$ with

$$g_2(a,b):=\frac{a^2+b^2}{8}+\frac{4(a+b)}{a^2+b^2}+\left(\frac{8}{a^2+b^2}\right)^2.$$

From (4.10), we know that $s_a + s_b < 4\sqrt{2}$, which implies $a^2 + b^2 < 4\sqrt{2} + 4u$. The domain of interest is then given by

$$D_2 := \{(a, b) \mid \sqrt{2} \leqslant a \leqslant b < 2,\ a^2 + b^2 < 4\sqrt{2} + 4u\}.$$

Computing the partial derivatives of g_2 with respect to a and b, and proving that they are both negative over the domain D_2 (detailed computations are in Appendix A, §A.1), we end up with $f_2(a, b) \leqslant g_2(\sqrt{2}, \sqrt{2}) = \left(2 + \frac{\sqrt{2}}{2}\right)^2$.

▶ **Subcase** $e \geqslant 0$

Since $|\delta| \leqslant \mathrm{ufp}(a^2) + \mathrm{ufp}(b^2) + \mathrm{ufp}(s_a + s_b) = 6 + 2^{-e}$ and $\mathrm{ufp}\left(\frac{a}{s}\right) \leqslant 2^{-2-\frac{e}{2}}$, from (4.2) we get $f_2(a, b) \leqslant g_3(a, b, e)$ with

$$g_3(a, b, e) := \frac{(a^2 + b^2)\,(2^{-e} + 1)}{16} + \frac{(6 + 2^{-e})(2^{-\frac{e}{2}}a + b)}{2\,(a^2 + b^2)} + \left(\frac{6 + 2^{-e}}{a^2 + b^2}\right)^2.$$

From (4.9), $a^2 + b^2$ is lower bounded by 4, and we restrict the analysis of g_3 to the domain

$$D_3 := \{(a, b, e) \mid 2^{-\frac{e}{2}} \leqslant a \leqslant 2^{\frac{1-e}{2}}, \sqrt{2} \leqslant b < 2, 4 \leqslant a^2 + b^2 < 4\sqrt{2} + 4u, e \geqslant 0\}.$$

First, it can be checked that the partial derivative of g_3 with respect to b is negative over D_3 (details are in Appendix A, §A.2). Since $b \geqslant \sqrt{4 - a^2}$, and $(a, b, e) \in D_3$ implies $(a, \sqrt{4 - a^2}, e) \in D_3$, we deduce that $g_3(a, b, e) \leqslant g_3(a, \sqrt{4 - a^2}, e)$, where

$$g_3(a, \sqrt{4 - a^2}, e) = \frac{2^{-e} + 1}{4} + \frac{(6 + 2^{-e})(2^{-\frac{e}{2}}a + \sqrt{4 - a^2})}{8} + \frac{(6 + 2^{-e})^2}{16}.$$

We then compute $\frac{\partial}{\partial a} g_3(a, \sqrt{4 - a^2}, e) = \frac{6+2^{-e}}{8}\left(2^{-\frac{e}{2}} - \frac{a}{\sqrt{4-a^2}}\right)$, which is nonnegative because $a^2 \leqslant \frac{2a^2}{1+2^{-e}} \leqslant \frac{4 \cdot 2^{-e}}{1+2^{-e}}$. Since $(2^{\frac{1-e}{2}}, \sqrt{4 - 2^{1-e}}, e) \in D_3$, we have $g_3(a, b, e) \leqslant g_3(2^{\frac{1-e}{2}}, \sqrt{4 - 2^{1-e}}, e) =: h_3(t)$, with $t = 2^{-e}$ and

$$h_3(t) = \frac{t + 1}{4} + \frac{(6 + t)(\sqrt{2}t + \sqrt{4 - 2t})}{8} + \frac{(6 + t)^2}{16}.$$

Since

$$h_3'(t) = 1 + \frac{t}{8}\left(1 + \sqrt{2}\right) + \frac{\sqrt{4 - 2t}}{8} + \frac{t + 6}{8}\left(\sqrt{2} - \frac{1}{\sqrt{4 - 2t}}\right)$$

is positive for $0 < t \leqslant 1$, we deduce $f_2(a, b) \leqslant h_3(1) = \left(\frac{7}{4} + \frac{\sqrt{2}}{2}\right)^2 = 6.037\ldots$.

4.2.4 Case $\mathrm{ufp}(s_a + s_b) = 2$

From (4.9) we have $2 \leqslant a^2 + b^2$, and from (4.10) we have $2 \leqslant s_a + s_b < 2\sqrt{2}$ hence

$$e \geqslant 0.$$

Since 2 is a floating-point number, we know that $s \geqslant 2$. Therefore $\frac{a}{s} < 2^{-\frac{1+e}{2}}$, hence

$$\mathrm{ufp}\left(\tfrac{a}{s}\right) \leqslant 2^{-1-\frac{e}{2}}, \tag{4.11}$$

and $\frac{b}{s} < 1$ so that

$$\mathrm{ufp}\left(\tfrac{b}{s}\right) \leqslant \tfrac{1}{2}.$$

We handle separately the two possible values, 1 and 2, for $\mathrm{ufp}(b^2)$.

▶ **Subcase** $\mathrm{ufp}(b^2) = 1$

We distinguish the cases $e \geqslant 1$ and $e = 0$.

• **Subsubcase** $e \geqslant 1$**:** From (4.2) we have $f_2(a,b) \leqslant g_4(a,b,e)$ with

$$g_4(a,b,e) := \frac{(a^2+b^2)\,(2^{-e}+1)}{4} + \frac{(3+2^{-e})\left(2^{-\frac{e}{2}}a+b\right)}{a^2+b^2} + \left(\frac{3+2^{-e}}{a^2+b^2}\right)^2.$$

From (4.9), we know that a^2+b^2 is lower bounded by 2. On the other hand, we have $a^2+b^2 \leqslant s_a+s_b+(\mathrm{ufp}(a^2)+\mathrm{ufp}(b^2))\,u < 2\sqrt{2}+2u$ and $1 < b < \sqrt{2}$, hence we can restrict the analysis to the domain

$$D_4 := \{(a,b,e) \mid 2^{-\frac{e}{2}} \leqslant a < 2^{\frac{1-e}{2}}, 1 < b < \sqrt{2}, 2 \leqslant a^2+b^2 < 2\sqrt{2}+2u, e \geqslant 1\}.$$

We can first compute the partial derivative of g_4 with respect to b and prove it is negative over D_4 for $p \geqslant 4$ (see the details in Appendix A, §A.3). Since $(a, \sqrt{2-a^2}, e)$ is in D_4, we deduce that $g_4(a,b,e) \leqslant g_4(a,\sqrt{2-a^2},e)$, and we have

$$g_4(a,\sqrt{2-a^2},e) = \frac{2^{-e}+1}{2} + \frac{(3+2^{-e})\left(2^{-\frac{e}{2}}a+\sqrt{2-a^2}\right)}{2} + \frac{(3+2^{-e})^2}{4}.$$

Next, we can compute the derivative of $g_4(a,\sqrt{2-a^2},e)$ with respect to a (see §A.3) and check that the maximum is attained at $a_0 = 2^{-\frac{e}{2}}\sqrt{\frac{2}{1+2^{-e}}}$, so that $g_4(a,b,e) \leqslant g_4(a_0,\sqrt{2-a_0^2},e) =: h_4(t)$ with $t = 2^{-e}$ and

$$h_4(t) = \frac{t+1}{2} + \frac{3+t}{2}\left(t\sqrt{\frac{2}{1+t}} + \sqrt{2-\frac{2t}{1+t}}\right) + \frac{(3+t)^2}{4}.$$

Since $h_4'(t) > 0$ for $0 < t \leqslant \frac{1}{2}$, we conclude that $f_2(a,b) \leqslant g_4(a,b,e) \leqslant h_4(\frac{1}{2}) = \left(\frac{7}{4}+\frac{\sqrt{3}}{2}\right)^2$.

• **Subsubcase** $e = 0$**:** According to (4.8), we assume that $1 < a$, so that $\mathrm{ufp}(b^2) = \mathrm{ufp}(a^2) = 1$. It follows that $s \geqslant s_a + s_b - 2u \geqslant a^2+b^2-4u$, hence $\frac{a}{s} \leqslant \frac{a}{a^2+b^2-4u}$. Since a and b are both floating-point numbers, and from (4.6), we know that $b \geqslant a+2u$ so that $b^2-4u > a^2$. By computing its partial derivative, it can then be checked that $\frac{a}{a^2+b^2-4u}$ is increasing with respect to a, which implies $\frac{a}{s} \leqslant \frac{b-2u}{(b-2u)^2+b^2-4u}$. This last expression is decreasing with respect to b, and since $b \geqslant 1+2u$ we deduce $\frac{a}{s} \leqslant \frac{1}{2(1+2u^2)} < \frac{1}{2}$. Thus,

$$\mathrm{ufp}\left(\tfrac{a}{s}\right) \leqslant \tfrac{1}{4}.$$

In the same way, it can be derived from $\frac{b}{s} \leqslant \frac{b}{a^2+b^2-4u}$ that

$$\mathrm{ufp}\left(\tfrac{b}{s}\right) \leqslant \tfrac{1}{4}.$$

Combining these bounds on $\mathrm{ufp}\left(\frac{a}{s}\right)$ and $\mathrm{ufp}\left(\frac{b}{s}\right)$ with (4.2) gives $f_2(a,b) \leqslant g_5(a,b)$, where

$$g_5(a,b) := \frac{a^2+b^2}{8} + \frac{2\,(a+b)}{a^2+b^2} + \frac{16}{\left(a^2+b^2\right)^2}.$$

Hence it remains to bound $g_5(a,b)$ over the domain D_5 defined by

$$D_5 := \{(a,b) \mid 1 \leqslant a \leqslant b < \sqrt{2} \text{ and } a^2+b^2 < 2\sqrt{2}+2u\}.$$

In this domain, we have $\frac{\partial}{\partial b} g_5(a,b) < 0$ (details are in Appendix A, §A.4), so that $g_5(a,b) \leqslant g_5(a,a) = \frac{a^2}{4} + \frac{4}{a^4} + \frac{2}{a}$, which is maximal for $a = 1$. Therefore, we deduce that $f_2(a,b) \leqslant g_5(a,b) \leqslant g_5(1,1) = \left(\frac{5}{2}\right)^2$.

▶ **Subcase** $\mathrm{ufp}(b^2) = 2$

In this paragraph, $a^2 < 1$ (otherwise we would have $s_a + s_b \geqslant 2+1$ while from (4.10) we have $s_a + s_b < 2\sqrt{2}$), hence $e \geqslant 1$. We split the inequality (4.11) into two possible cases. Either $\mathrm{ufp}\left(\frac{a}{s}\right) < 2^{-1-\frac{e}{2}}$ which implies $\mathrm{ufp}\left(\frac{a}{s}\right) \leqslant 2^{-\frac{3+e}{2}}$, or $\mathrm{ufp}\left(\frac{a}{s}\right) = 2^{-1-\frac{e}{2}}$ in which case e is even.

• **Subsubcase** $\mathrm{ufp}\left(\frac{a}{s}\right) < 2^{-1-\frac{e}{2}}$**:** We deduce from (4.2) and $|\delta| \leqslant 4 + 2^{-e}$ that $f_2(a,b) \leqslant g_6(a,b,e)$ with

$$g_6(a,b,e) := \frac{\left(a^2+b^2\right)\left(2^{-1-e}+1\right)}{4} + \frac{\left(4+2^{-e}\right)\left(2^{-\frac{1+e}{2}}a+b\right)}{a^2+b^2} + \left(\frac{4+2^{-e}}{a^2+b^2}\right)^2.$$

We can compute the derivatives of g_6 (details are provided in Appendix A, §A.5) with respect to a and b and prove that they are negative over the domain

$$D_6 := \{(a,b,e) \mid 2^{-\frac{e}{2}} \leqslant a < 2^{\frac{1-e}{2}}, \sqrt{2} \leqslant b < 2, \\ 2 \leqslant a^2+b^2 < 2\sqrt{2} + \left(2+2^{-e}\right)u, e \geqslant 1\}.$$

For $(a,b,e) \in D_6$, we deduce that $g_6(a,b,e) \leqslant g_6(2^{-\frac{e}{2}}, \sqrt{2}, e) =: h_6(t)$ with

$$h_6(t) = \frac{(t+2)\left(\frac{t}{2}+1\right)}{4} + \frac{\sqrt{2}\,(4+t)\left(\frac{t}{2}+1\right)}{t+2} + \left(\frac{4+t}{t+2}\right)^2, \quad t = 2^{-e}.$$

We can maximize $h_6(t)$ for $0 < t \leqslant \frac{1}{2}$, which leads to $f_2(a,b) \leqslant h_6(0) = \left(2+\frac{\sqrt{2}}{2}\right)^2$.

• **Subsubcase** $\mathrm{ufp}\left(\frac{a}{s}\right) = 2^{-1-\frac{e}{2}}$**:** In this case, e is even, hence $e \geqslant 2$. We have $f_2(a,b) \leqslant g_7(a,b,e)$ with

$$g_7(a,b,e) := \frac{\left(a^2+b^2\right)\left(2^{-e}+1\right)}{4} + \frac{\left(4+2^{-e}\right)\left(2^{-\frac{e}{2}}a+b\right)}{a^2+b^2} + \left(\frac{4+2^{-e}}{a^2+b^2}\right)^2.$$

The lower bound $2^{-\frac{e}{2}}$ for a does not lead to a sufficiently tight bound for f_2 in this case: to get a better bound, we exploit further the hypothesis $\mathrm{ufp}\left(\frac{a}{s}\right) = 2^{-1-\frac{e}{2}}$. This gives $s2^{-1-\frac{e}{2}} \leqslant a$, which implies $a^2 - 2^{1+\frac{e}{2}}a + b^2 + \delta u \leqslant 0$, hence

$$a \geqslant 2^{\frac{e}{2}} - \sqrt{2^e - 2 + (4 + 2^{-e})u} = a_0 + \eta(u)$$

with

$$a_0 = 2^{\frac{e}{2}} - \sqrt{2^e - 2}, \quad \eta(u) < 0, \quad |\eta(u)| \in \mathcal{O}(u).$$

Therefore, we analyze g_7 over the domain

$$D_7 := \{(a, b, e) \mid a_0 + \eta(u) \leqslant a < 2^{\frac{1-e}{2}}, \sqrt{2} \leqslant b < 2, \\ 2 \leqslant a^2 + b^2 < 2\sqrt{2} + \left(2 + 2^{-e}\right) u, e \geqslant 2, e \text{ even}\}.$$

The remainder of this analysis is described in the next subsection, we give here the main steps. We first maximize g_7 with respect to b, computing the derivative with respect to b and proving that it is negative. We then maximize $g_7(a, \sqrt{2}, e)$ with respect to a; again we compute the derivative and prove that it is negative so that $g_7(a, b, e) \leqslant g_7(a_0 + \eta(u), \sqrt{2}, e)$. Unlike the previous cases, we introduced here a dependence on u to get a more precise analysis; we now need to isolate this dependence in order to get an upper bound that is an affine function over u. We finally maximize the resulting expression with respect to e (using the fact that e is an even integer): $g_7(a_0+\eta(u), \sqrt{2}, e) \leqslant g_7(a_0, \sqrt{2}, e) + 20u \leqslant \gamma^2 + 20u$. The final upper bound is therefore $f_2 \leqslant \gamma^2 + 20u$ over D_7.

4.2.5 Case $\mathrm{ufp}(s_a + s_b) = 2$, $\mathrm{ufp}(b^2) = 2$ and $\mathrm{ufp}(a/s) = 2^{-1-e/2}$

In this section, we detail the case where $\mathrm{ufp}(s_a + s_b) = 2$, $\mathrm{ufp}(b^2) = 2$ and $\mathrm{ufp}(a/s) = 2^{-1-e/2}$, that leads to maximizing the function

$$g_7(a, b, e) := \frac{(a^2 + b^2)\left(2^{-e} + 1\right)}{4} + \frac{\left(4 + 2^{-e}\right)\left(2^{-\frac{e}{2}}a + b\right)}{a^2 + b^2} + \left(\frac{4 + 2^{-e}}{a^2 + b^2}\right)^2$$

over the domain

$$D_7 := \{(a, b, e) \mid a_0 + \eta(u) \leqslant a < 2^{\frac{1-e}{2}}, \sqrt{2} \leqslant b < 2, \\ 2 \leqslant a^2 + b^2 < 2\sqrt{2} + \left(2 + 2^{-e}\right) u, e \geqslant 2, e \text{ even}\}.$$

We first maximize g_7 with respect to b. We have

$$\frac{\partial g_7}{\partial b} = \frac{b}{2}\left(2^{-e} + 1\right) + \frac{4 + 2^{-e}}{a^2 + b^2} - 2b\frac{\left(2^{-\frac{e}{2}}a + b\right)\left(4 + 2^{-e}\right)}{\left(a^2 + b^2\right)^2} - 4b\frac{\left(4 + 2^{-e}\right)^2}{\left(a^2 + b^2\right)^3},$$

and we want to prove that $\frac{\partial}{\partial b}g_7 < 0$ over D_7. Multiplying by $\frac{(a^2+b^2)^2}{(4+2^{-e})b}$ and using the inequality $\frac{1}{b} < 1$, we only need to prove that

$$\frac{\left(a^2 + b^2\right)^2\left(2^{-e} + 1\right)}{2\left(4 + 2^{-e}\right)} + a^2 + b^2 < 2\left(2^{-\frac{e}{2}}a + b\right) + 4\frac{4 + 2^{-e}}{a^2 + b^2}.$$

Since $e \geqslant 2$, we can derive this inequality for $p \geqslant 2$ from the three following ones using the definition of D_7:

$$\frac{\left(a^2+b^2\right)^2\left(2^{-e}+1\right)}{2\left(4+2^{-e}\right)} + a^2 + b^2 < \frac{\left(2\sqrt{2}+\left(2+\frac{1}{4}\right)u\right)^2\left(\frac{1}{4}+1\right)}{8} + 2\sqrt{2} + \left(2+\frac{1}{4}\right)u,$$

$$\frac{\left(2\sqrt{2}+\left(2+\frac{1}{4}\right)u\right)^2\left(\frac{1}{4}+1\right)}{8} + 2\sqrt{2} + \left(2+\frac{1}{4}\right)u < 2\sqrt{2} + \frac{16}{2\sqrt{2}+\left(2+\frac{1}{4}\right)u},$$

and

$$2\sqrt{2} + \frac{16}{2\sqrt{2}+\left(2+\frac{1}{4}\right)u} < 2\left(2^{-\frac{e}{2}}a+b\right) + 4\frac{4+2^{-e}}{a^2+b^2}.$$

Therefore, g_7 is decreasing with respect to b, and for all (a,b,e) in D_7, we have

$$g_7(a,b,e) \leqslant g_7(a,\sqrt{2},e). \tag{4.12}$$

We now maximize $g_7(a,\sqrt{2},e)$ with respect to a. Let us recall that in D_7,

$$a \geqslant a_0+\eta(u) = 2^{\frac{e}{2}} - \sqrt{2^e-2+(4+2^{-e})u};$$

and prove that

$$\frac{\partial}{\partial a}g_7(a,\sqrt{2},e) < 0.$$

Using the notation $t = 2^{-e}$, we prove the inequality $a_0+\eta(u) \geqslant 2^{-1-\frac{e}{2}}$: it is equivalent to $(\frac{1}{4}-u)t \geqslant -1+4u$ which holds for $p \geqslant 2$ since $\frac{1}{4}-u \geqslant 0 \geqslant -1+4u$. Moreover, we have

$$\frac{(a^2+2)^2}{a(4+2^{-e})}\frac{\partial}{\partial a}g_7(a,\sqrt{2},e) = \frac{(2^{-e}+1)(a^2+2)^2}{2(4+2^{-e})} + \frac{2^{-\frac{e}{2}}}{a}(a^2+2) - 2\left(2^{-\frac{e}{2}}a+\sqrt{2}\right) - 4\frac{4+2^{-e}}{a^2+2},$$

with $\frac{(a^2+2)^2}{a(4+2^{-e})} > 0$ for $a \in I := \left[2^{-1-\frac{e}{2}}, 2^{\frac{1-e}{2}}\right]$. For $e \geqslant 2$ and $a \in I$, we have

$$\frac{(a^2+2)^2}{a(4+2^{-e})}\frac{\partial}{\partial a}g_7(a,\sqrt{2},e) < \frac{125}{128} + 5 - 2\sqrt{2} - \frac{32}{5} < 0.$$

As a consequence, $g_7(a,\sqrt{2},e)$ is decreasing with respect to a over I, and since $a_0+\eta(u) \in I$, the maximum of $g_7(a,\sqrt{2},e)$ for $a \in \left[a_0+\eta(u), 2^{\frac{1-e}{2}}\right]$ is $g_7(a_0+\eta(u),\sqrt{2},e)$. Thus, for (a,b,e) in D_7, we have

$$g_7(a,\sqrt{2},e) \leqslant g_7(a_0+\eta(u),\sqrt{2},e). \tag{4.13}$$

In order to continue maximizing, we isolate the dependence on u. Let us prove that

$$g_7(a_0+\eta(u),\sqrt{2},e) \leqslant g_7(a_0,\sqrt{2},e)+20u. \tag{4.14}$$

For this purpose, we first prove that

$$-2u<\eta(u)<0 \quad \text{and} \quad a_0<\tfrac{\sqrt{2}}{2}. \tag{4.15}$$

For the first inequality, we have

$$\begin{aligned}
|\eta(u)| &= \sqrt{2^e-2+(4+2^{-e})u}-\sqrt{2^e-2}\\
&= \frac{2^{\frac{e}{2}}}{\sqrt{1-2^{-e}(2-(4+2^{-e})u)}+\sqrt{1-2^{1-e}}}\left(2^{1-e}-2^{-e}(2-(4+2^{-e})u\right)\\
&= \frac{2^{-\frac{e}{2}}}{\sqrt{1-2^{-e}(2-(4+2^{-e})u)}+\sqrt{1-2^{1-e}}}(4+2^{-e})u\\
&\leqslant \frac{2^{-1}(4+2^{-2})}{\sqrt{2}}u \text{ since } e\geqslant 2\\
&< 2u.
\end{aligned}$$

We recall that $a_0=2^{e/2}-\sqrt{2^e-2}$. The second inequality is equivalent to $0<\sqrt{2}\cdot 2^{e/2}-2.5$ which holds since $e\geqslant 2$. Let us now consider the quantity $\lambda_0(u)$ that appears in $g_7(a_0+\eta(u),\sqrt{2},2)$, defined by

$$\lambda_0(u)=\frac{1}{(a_0+\eta(u))^2+2}.$$

We prove that

$$\lambda_0(u)<\frac{1}{a_0^2+2}+a_0u \quad \text{and} \quad \lambda_0(u)^2<\left(\frac{1}{a_0^2+2}\right)^2+a_0u. \tag{4.16}$$

We have

$$\lambda_0(u)=\frac{1}{a_0^2+2}-\frac{2a_0+\eta(u)}{(a_0^2+2)((a_0+\eta(u))^2+2)}\eta(u),$$

and the first inequality is a consequence of the first part of (4.15). Moreover, we have

$$\begin{aligned}
\lambda_0(u)^2=\left(\frac{1}{a_0^2+2}\right)^2 &- \frac{4a_0}{(a_0^2+2)^2((a_0+\eta(u))^2+2)}\eta(u)\\
&+\frac{(2a_0+\eta(u))^2-2((a_0+\eta(u))^2+2)}{(a_0^2+2)^2((a_0+\eta(u))^2+2)^2}\eta(u)^2,
\end{aligned}$$

and using again (4.15), we deduce the second inequality. From the definition of g_7, using $0<a_0+\eta(u)<a_0$ and the previous upper bounds (4.16) on $\lambda_0(u)$ and $\lambda_0(u)^2$, we obtain

$$\begin{aligned}
g_7(a_0+\eta(u),\sqrt{2},e)<\frac{(a_0^2+2)\,(2^{-e}+1)}{4} &+\left(4+2^{-e}\right)\left(2^{-\frac{e}{2}}a_0+\sqrt{2}\right)\left(\frac{1}{a_0^2+2}+a_0u\right)\\
&+(4+2^{-e})^2\left(\frac{1}{(a_0^2+2)^2}+a_0u\right),
\end{aligned}$$

so that $g_7(a_0+\eta(u),\sqrt{2},e) < g_7(a_0,\sqrt{2},e) + (4+2^{-e})\left(2^{-\frac{e}{2}}a_0 + \sqrt{2} + 4 + 2^{-e}\right)a_0 u$. The inequality $g_7(a_0+\eta(u),\sqrt{2},e) < g_7(a_0,\sqrt{2},e) + 20u$ then follows from $e \geqslant 2$ and $a_0 < \frac{\sqrt{2}}{2}$.

The last step is to prove that

$$g_7(a_0,\sqrt{2},e) \leqslant \gamma^2 \quad \text{for } e \text{ an even positive integer.} \tag{4.17}$$

With $\tau = \sqrt{1-2^{1-e}}$, we have $g_7(a_0,\sqrt{2},e) =: h_7(\tau)$ with $h_7(\tau)$ a rational function in τ. The variable τ belongs to $\left[\sqrt{2}/2, 1\right]$, and $h_7'(\tau) = \frac{P(\tau)}{32\,(\tau+1)^2}$ with

$$P(\tau) = 3\tau^7 + 11\tau^6 - 5\tau^5 - (12\sqrt{2}+85)\tau^4 - (32\sqrt{2}+143)\tau^3 \\ - (23-8\sqrt{2})\tau^2 + (64\sqrt{2}+113)\tau + 36\sqrt{2} + 33.$$

Using Descartes' rule of signs, we check that P has exactly one root in the interval $(0,1]$: the sequence of coefficients of $P(\tau)$ has two changes of sign so that P has 0 or 2 positive roots while the sequence of coefficients of $P(\tau+1)$ has one change of sign so that P has one root larger than 1. Moreover, since $P(\sqrt{1-2^{-5}}) > 0$ and $P(\sqrt{1-2^{-7}}) < 0$, we deduce that this root belongs to $\left(\sqrt{1-2^{-5}}, \sqrt{1-2^{-7}}\right)$, and that h_7 is increasing over $\left[\sqrt{2}/2, \sqrt{1-2^{-5}}\right]$ and decreasing over $\left[\sqrt{1-2^{-7}}, 1\right]$. Finally, observing that $h_7(\sqrt{1-2^{-5}}) < h_7(\sqrt{1-2^{-7}})$, we conclude that $h_7(\sqrt{1-2^{-7}})$ is an upper bound for h_7, for e an even positive integer.

We define γ as the positive square root of $h_7(\sqrt{1-2^{-7}})$ so that gathering the inequalities (4.12), (4.13), (4.14) and (4.17), we get $f_2(a,b) \leqslant \gamma^2 + 20u$. From (4.3), we derive the final upper bound $\gamma u + 9u^2$ for $\mathrm{E_N}$, which concludes the proof of Theorem 4.1.

Conclusion of Part II

Let us conclude with some remarks about complex division. As mentioned in the introduction to this part, Baudin remarked in [4] that any algorithm computing a complex multiplication can be used to compute a complex division according to the equality

$$\frac{a+ib}{c+id} = \frac{1}{c^2+d^2}\big((a+ib)\cdot(c-id)\big).$$

This equality means that we first multiply by the conjugate of the denominator, and then divide each component by a floating-point number. The interesting fact is that a normwise relative error bound of the form $\alpha u + \mathcal{O}(u^2)$ for the multiplication algorithm directly implies the bound $(3+\alpha)u + \mathcal{O}(u^2)$ on the normwise relative error generated by the corresponding division algorithm.

We can highlight another way to match a complex multiplication algorithm to a complex division algorithm, basically using a multiplication by the inverse of the denominator:

$$\frac{a+ib}{c+id} = \frac{1}{c+id}\cdot(a+ib).$$

According to our error analysis of the inversion algorithm described by Algorithm 5, any bound $\alpha u + \mathcal{O}(u^2)$ for the complex multiplication implies the bound $(\gamma+\alpha)u + \mathcal{O}(u^2)$ for the division algorithm, where γ is defined in Theorem 4.1. Since $\gamma < 3$, this second scheme for complex division leads to algorithms for which the best known normwise error bounds are better (i.e. smaller) than for the first scheme.

Moreover, if the expression c^2+d^2 is computed in the same way for both schemes, the number of floating-point operations of each kind (addition/subtraction, multiplication, inverse and FMA) does no depend on the scheme: for the same cost, we have a better accuracy guarantee when computing a complex division as an inversion followed by a multiplication.

Finally, we mention the highest normwise relative error we obtained for the algorithm based on the naive complex multiplication followed by a division (referred to as mul/div) and the algorithm based on the naive complex inversion followed by the naive complex multiplication (referred to as inv/mul):

Arithmetic	Algorithm mul/div	
	Input	Error
binary32 ($p=24$)	$\frac{5935365+i\cdot 11910483/2}{11863437+i\cdot 11864709}$	$5.07951u$
binary64 ($p=53$)	$\frac{5935365+i\cdot 11910483/2}{11863437+i\cdot 11864709}$	$5.04208u$
binary128 ($p=113$)	$\frac{736070367558372747372516958272345 9/4+i\cdot 183909524503601985250136536112733 1}{735009507599575839659580201503840 1+i\cdot 734368822629130634496405664399866 5}$	$5.01830u$

Arithmetic	Algorithm inv/mul	
	Input	Error
binary32 ($p = 24$)	$\frac{11898033+i\cdot 11894677}{2972123/4+i\cdot 742117}$	$4.72945u$
binary64 ($p = 53$)	$\frac{6379358682446203+i\cdot 6400634450993511}{3194317788255377+i\cdot 6369097858326577/2}$	$4.71008u$

We can conclude from these two tables that the normwise relative error bounds we know for these algorithm — about $5.24u$ for mul/div and $4.94u$ for inv/mul — are not too far from the optimal bounds. Additionally, these numerical examples are sufficient to prove that the algorithm inv/mul is more accurate than the algorithm mul/div for the complex floating-point division, in terms of worst case normwise relative error.

Part III

A symbolic floating-point arithmetic

We have seen in Sections 1.3, 2.4 and 3.2 the use of examples parametrized by the precision to prove the existence of an unstable behavior or the asymptotic optimality of an error bound. Such examples can be built by first computing the error generated by the algorithm with small values of the precision p, then guessing some "generic" inputs parametrized by β and p, and finally carrying paper-and-pencil calculations, that are in general tedious and error prone, to check if those guessed inputs effectively correspond to cases we are looking for. Our goal in this part is to show how to automate this last step in order to save time, to avoid possible errors, and to be able to try many more candidates.

We refer to functions representing floating-point numbers parametrized by the precision p (with $\beta \geqslant 2$ fixed and even) as *symbolic floating-point numbers*. We propose to manipulate such symbolic floating-point numbers, in a computer algebra system such as Maple, very much as real numbers or polynomials can be manipulated within such a system.

Floating-point arithmetic has already been formalized in various interactive theorem provers: see for example [15, 11], and more recently the description of the Flocq library in [7], that was designed for the Coq proof assistant. Such a formalization can be used to generate formal proofs of properties of floating-point algorithms and programs. Also, multiple-precision floating-point computations can be performed using Flocq for verification purposes (see [29] for details), but in that case the precision for each operation is fixed: a particular format with a fixed precision $p \in \mathbb{N}$ is defined for each intermediate operation. In contrast, when computing with symbolic floating-point numbers as we are going to define them below, the precision p is a linear function of an integer variable k, which allows us to deal with "generic" examples.

Let us briefly recall how conventional floating-point arithmetic can be formalized. The set $\mathbb{F}_p$ of floating-points numbers (with an unbounded exponent range), in base β and precision p, is a subset of $\mathbb{Q}$ containing all the numbers of the form $M \cdot \beta^e$, with $|M| \in \mathbb{Z}$ less than β^p, and $e \in \mathbb{Z}$. The exact result of an elementary operation ($+$, $-$, $\times$ or $/$) between two elements of $\mathbb{F}_p$ is a rational number in $\mathbb{Q}$. It is then rounded to an element in $\mathbb{F}_p$ according to a specified rounding function to get the computed result. This rounding operation can be performed in three steps: first, the argument is normalized with a division by a power of the base β; then, we round to an integer; finally, we scale back the result. Let us recall that the ulp function has been defined in Section 2.1; here we had the precision as a subscript to remove any ambiguity. For a (positive) rational, in

base β and precision p, this scheme is described by the diagram below, where $\circ$ denotes the rounding-to-integer function:

$$\begin{array}{ccccccc} \mathbb{Q}_{>0} & \longrightarrow & \mathbb{Q} \cap [\beta^{p-1}, \beta^p) & \longrightarrow & \mathbb{Z} \cap [\beta^{p-1}, \beta^p] & \longrightarrow & \mathbb{F}_p \\ r & \longmapsto & \mu := r/\operatorname{ulp}_p(r) & \longmapsto & M := \circ(\mu) & \longmapsto & M \operatorname{ulp}_p(r) \end{array} \tag{1}$$

For instance, in base $\beta = 10$ and precision $p = 3$, the rounded to nearest value of $r = 0.17446$ is obtained through:

$$0.17446 \longrightarrow 174.46 \longrightarrow 174 \longrightarrow 0.174 \quad \text{since } \operatorname{ulp}_3(r) = 10^{-3}.$$

This scheme can also be summarized (for rounding to the nearest) by the following formula:

$$\operatorname{RN}_p(r) = \operatorname{ulp}_p(r) \lfloor r/\operatorname{ulp}_p(r) \rceil. \tag{2}$$

We will first define the set $\mathbb{L}$ of affine functions with integer coefficients that will be used to represent both the precision and the exponent of symbolic floating-point numbers. Our analogue of $\mathbb{Q}$ will be the set $\mathbb{SQ}$ of symbolic rationals; we consider rational functions in β^k:

$$\mathbb{SQ} = \mathbb{Q}(\beta^k).$$

Then, we introduce the set $\mathbb{SZ}$ of symbolic integers, and we show how elements in $\mathbb{SQ}$ can be rounded to elements in $\mathbb{SZ}$. Using the notion of symbolic integer, we define the set $\mathbb{SF}_p$ of symbolic floating-point numbers in base β and precision $p \in \mathbb{L}$. Finally, we define the standard rounding functions and prove that they can be computed using an adapted version of equation (2). The following table summarizes the matching between numerical data and their symbolic counterparts:

Numerical data	Symbolic data
$\mathbb{Z}$	$\mathbb{L}$
$\mathbb{Q}$	$\mathbb{SQ}$
$\mathbb{Z}$	$\mathbb{SZ}$
$\mathbb{F}_p$	$\mathbb{SF}_p$

The exact results of elementary operations on elements of $\mathbb{SF}_p$ can always be represented in $\mathbb{SQ}$. It is then rounded to $\mathbb{SF}_p$, sometimes at the cost of an additional assumption such as the parity of the variable k.

In Chapter 5, we introduce our formalization for a symbolic floating-point arithmetic, parametrized by the precision. We successively define the sets $\mathbb{SQ}$ of symbolic rationals, $\mathbb{SZ}$ of symbolic integers and $\mathbb{SF}_p$ of symbolic floating-point numbers. These definitions are completed with the standard rounding functions, from $\mathbb{SQ}$ to $\mathbb{SZ}$ first and then from $\mathbb{SQ}$ to $\mathbb{SF}_p$. We conclude this chapter proving that the symbolic computations transfer to the classical floating-point arithmetic when a numerical precision is chosen. Then, in Chapter 6, we present an implementation of this formalism in Maple and illustrate its use on various examples from the literature.

Chapter 5

Definition of a symbolic floating-point arithmetic

This chapter is dedicated to the formalization of our symbolic floating-point arithmetic in order to quickly and reliably check optimality certificates of known eror bounds, such as the one for complex floating-point inversion (see Section 3.2). In Section 5.1, we introduce the ordered set $\mathbb{SQ}$ of symbolic rational and the main attributes (exponent, ulp and ufp functions) of the elements of this set. We then describe the symbolic integers $\mathbb{SZ}$ in Section 5.2 and the classic rounding functions from $\mathbb{SQ}$ to $\mathbb{SZ}$. In particular, we explain that we can only define partial rounding functions and how we deal with the symbolic rationals that cannot be rounded, with an additional assumption on the variable $k \in \mathbb{N}$. We also give algorithms to compute the rounding functions. Finally, in Section 5.3, we introduce the set $\mathbb{SF}_p$ of symbolic floating-point numbers in precision p where p is an affine function of k eventually larger than 1, with integer coefficients, and define the standard rounding functions from $\mathbb{SQ}$ to $\mathbb{SF}_p$. We show how these (partial) rounding functions can be computed based on the previous algorithms to round from $\mathbb{SQ}$ to $\mathbb{SZ}$. We conclude this last section by proving that the symbolic computations we described match the classical floating-point arithmetic when we evaluate the results for k large enough: this shows that our symbolic floating-point arithmetic is suitable for testing optimality certificates and other floating-point computations parametrized by the precision.

5.1 Symbolic rationals

Inspired from the examples from the literature, we could consider the polynomials over β^k with rational coefficients $\mathbb{Q}[\beta^k]$ as basic objects since the computed results of floating-point operations belongs to this set. However, we decided to split the floating-point operations into the exact operation and the rounding one. As a consequence, we need our set to be closed under division and thus consider the set of rational functions in β^k with rational coefficients. We call such functions *symbolic rationals* and denote their set by $\mathbb{SQ}$:

$$\mathbb{SQ} = \mathbb{Q}(\beta^k).$$

Every element f of $\mathbb{SQ}$ thus has the form

$$f(k) = \frac{P(\beta^k)}{Q(\beta^k)}, \qquad P, Q \in \mathbb{Z}[X], \qquad Q \neq 0$$

and evaluates to a rational as k is large enough (so that the denominator does not vanish). Furthermore, since $\beta > 1$ and k is a symbolic variable, the set $\mathbb{SQ}$ is isomorphic to $\mathbb{Q}(X)$ and can be made an *ordered field* in the same way; see for example [28, p. 450]: for $f \in \mathbb{SQ}$ nonzero, we write $P(X) = P_m X^m + \cdots + P_0$ and $Q(X) = Q_n X^n + \cdots + Q_0$ with $P_m, Q_n \neq 0$, and we say that f is *positive* if the ratio of the leading coefficients of P and Q is positive:

$$f > 0 \qquad \text{if} \qquad \frac{P_m}{Q_n} > 0.$$

5.1.1 Inequalities, absolute value, and sign

This ordering for $\mathbb{SQ}$ allows us to write and use inequalities and enclosures about its elements in exactly the same way as for the rationals. For example, if $-f$ is positive then we write $f < 0$ and say that f is *negative*. Also, for $f, g, h \in \mathbb{SQ}$, we write $f \leqslant g < h$ or, equivalently, $g \in [f, h)$ to indicate that $g - f$ is positive or zero, and that $g - h$ is negative. Finally, the *absolute value* and *sign* functions are defined as usual:

$$|f| = \begin{cases} f & \text{if } f \geqslant 0, \\ -f & \text{if } f < 0; \end{cases} \qquad \operatorname{sign}(f) = \begin{cases} -1 & \text{if } f < 0, \\ 0 & \text{if } f = 0, \\ 1 & \text{if } f > 0. \end{cases}$$

These definitions matches the usual ones for rationals when the asymptotic behaviors are reached. For instance, for $f \in \mathbb{SQ}$, we have $|f(k)| = |f|(k)$ for all k large enough.

5.1.2 Exponent, ufp, and ulp

If $e \in \mathbb{L}$ then β^e is a positive element of $\mathbb{SQ}$, so that any $f \in \mathbb{SQ}$ satisfies either $|f| \geqslant \beta^e$ or $|f| < \beta^e$. The property below shows further that if f is nonzero, then there is a single interval of the form $[\beta^e, \beta^{e+1})$ containing $|f|$, and we shall call the corresponding $e \in \mathbb{L}$ the *exponent* of f.

Property 5.1. *For $f \in \mathbb{SQ}$ nonzero, let P, Q, m, n be as above and let e_0 be the unique integer such that $\beta^{e_0} \leqslant |P_m/Q_n| < \beta^{e_0+1}$. Then there exists a unique linear function $e \in \mathbb{L}$ such that $\beta^e \leqslant |f| < \beta^{e+1}$. Furthermore,*

$$e(k) = \begin{cases} e_0 + k(m-n) - 1 & \text{if } |f| < \beta^{e_0+k(m-n)}, \\ e_0 + k(m-n) & \text{otherwise.} \end{cases}$$

Proof. By setting $X = \beta^k$, we have $f(k) = P(X)/Q(X)$, which can be rewritten $f(k) = X^{m-n} \sum_{i \geqslant 0} c_i X^{-i}$ with $c_0 = P_m/Q_n$ and $c_i \in \mathbb{Q}$ for all $i \geqslant 1$. By definition of the absolute value on $\mathbb{SQ}$, $|f(k)| = X^{m-n}(|c_0| + \sum_{i \geqslant 1} \widetilde{c}_i X^{-i})$ with $\widetilde{c}_i = \operatorname{sign}(c_0) c_i$ for all $i \geqslant 1$. Since

$|c_0|$ is strictly larger than β^{e_0-1} for $\beta > 1$, we deduce that $|f(k)| - \beta^{e_0-1}X^{m-n}$ is positive in $\mathbb{SQ}$. Similarly, since $|c_0|$ is strictly smaller than β^{e_0+1}, $|f(k)| - \beta^{e_0+1}X^{m-n}$ is negative in $\mathbb{SQ}$. The existence of e then follows from comparing $|f(k)|$ to $\beta^{e_0}X^{m-n}$. To establish uniqueness, let $e' \in \mathbb{L}$ be such that $\beta^{e'} \leqslant |f| < \beta^{e'+1}$. This implies $\beta^{-1} < \beta^{e'-e} < \beta$ and, since $e' - e$ is an integer-valued function of k, we must have $e' - e = 0$. □

The classic notions of *unit in the first place* (ufp) and *unit in the last place* (ulp) can then be extended to symbolic rationals as follows: if $f \in \mathbb{SQ}$ is nonzero and has exponent e, then

$$\operatorname{ufp}(f) = \beta^e \quad \text{and} \quad \operatorname{ulp}_p(f) = \beta^{e-p+1} \quad \text{for } p \in \mathbb{L};$$

if $f = 0$ then we take $\operatorname{ufp}(0) = \operatorname{ulp}_p(0) = 0$,

At this stage, we can already perform step 1 (scaling) of (1) in our symbolic context: for $p \in \mathbb{L}$,

$$\begin{array}{ccc} \mathbb{SQ}_{>0} & \longrightarrow & \mathbb{SQ} \cap [\beta^{p-1}, \beta^p) \\ f & \longmapsto & f/\operatorname{ulp}_p(f) \end{array}$$

5.2 Symbolic integers

Our symbolic analog of $\mathbb{Z}$ will consist of those elements of $\mathbb{SQ}$ that are ultimately integer valued, that is, $f \in \mathbb{Q}(\beta^k)$ such that $f(k) \in \mathbb{Z}$ as $k \to \infty$. We call such elements *symbolic integers* and denote their set by $\mathbb{SZ}$. Any symbolic integer is an element of $\mathbb{Q}[\beta^k]$, that is a polynomial in β^k with rational coefficient, that is $\mathbb{SZ} \subseteq \mathbb{Q}[\beta^k]$ (see Exercise 93 in [34, p. 130]). Hence, $\mathbb{SZ}$ can be defined by

$$\mathbb{SZ} = \Big\{ f \in \mathbb{Q}[\beta^k] : f(k) \in \mathbb{Z} \text{ as } k \to \infty \Big\}.$$

In this section, we explain how to perform the second step in (1), that is, rounding from $\mathbb{SQ}$ to $\mathbb{SZ}$. For this purpose, we first compute an approximation by $\mathbb{SZ}$, which is then corrected according to a specific rounding mode. Before detailing these two steps, we give a characterization of $\mathbb{SZ}$ that will be used when computing an approximation.

Any $f \in \mathbb{Q}[\beta^k]$ can be expressed as

$$f(k) = \frac{T(\beta^k)}{M \cdot \beta^d}, \tag{5.1}$$

where $T \in \mathbb{Z}[X]$, $d \in \mathbb{N}$ and M and β are coprime integers. Let us briefly explain how to obtain this expression. We define the sequence $(M_i)_{i\in\mathbb{N}}$, where M_0 is the least common multiple of the denominators of the coefficients in f and $M_{i+1} = M_i/\gcd(M_i, \beta)$. Since this sequence of positive integers is non increasing, it is eventually constant. Let d be the first index such that $M_{d+1} = M_d$, so that $\gcd(M_d, \beta) = 1$. Defining $T_0 \in \mathbb{Z}[X]$ such that $f(k) = T_0(\beta^k)/M_0$, it can be checked that $T = \beta^d(M_d/M_0)T_0$ belongs to $\mathbb{Z}[X]$, which gives (5.1) by denoting M_d by M.

Let $\operatorname{ord}_M(\beta)$ be the order of β modulo M, as in [13, p. 506]: $\operatorname{ord}_M(\beta)$ is the smallest $\alpha \in \mathbb{N}_{>0}$ such that $\beta^\alpha \equiv 1 \pmod{M}$. The fact that a symbolic rational belongs to $\mathbb{SZ}$ can then be characterized using the following property.

Property 5.2. *For $f \in \mathbb{Q}[\beta^k]$, let T, M, and d be as in (5.1). Then, $f \in \mathbb{SZ}$ if and only if $T(0) \equiv 0 \pmod{\beta^d}$ and $T(\beta^k) \equiv 0 \pmod{M}$ for all $k \in \{0, 1, \cdots, \mathrm{ord}_M(\beta) - 1\}$. Moreover, if $f \in \mathbb{SZ}$ then $f(k) \in \mathbb{Z}$ for all $k \geqslant d$.*

Proof. Since β^d and M are coprime, $f \in \mathbb{SZ}$ if and only if $T(\beta^k)/M \in \mathbb{SZ}$ and $T(\beta^k)/\beta^d \in \mathbb{SZ}$. The condition $T(\beta^k)/M \in \mathbb{SZ}$ means that there exists $k_0 \in \mathbb{N}$ such that, for $k \geqslant k_0$, $T(\beta^k) \equiv 0 \pmod{M}$, or equivalently, that T vanishes modulo M over the set $\{\beta^k \pmod{M} : 0 \leqslant k < \mathrm{ord}_M(\beta)\}$. On the other hand, for all $k \geqslant d$, we have $\beta^k \equiv 0 \pmod{\beta^d}$ so that $T(\beta^k) \equiv P(0) \pmod{\beta^d}$. Thus, the condition $T(\beta^k)/\beta^d \in \mathbb{SZ}$ is equivalent to $T(0) \equiv 0 \pmod{\beta^d}$. □

5.2.1 Approximation

For $f \in \mathbb{SQ}$ and $g \in \mathbb{SZ}$, we say that g is an *approximation* of f by an element of $\mathbb{SZ}$ if $f - g$ is bounded, that is, if there exists $C \in \mathbb{N}$ such that $|f - g| < C$ for the ordering in $\mathbb{SQ}$.

We will see in this subsection how to determine whether a symbolic rational can be approximated by an element in $\mathbb{SZ}$ or not, and how to compute such an approximation when it is possible. Property 5.3 reduces the problem of approximating elements of $\mathbb{SQ}$ to the one of approximating elements of $\mathbb{Q}[\beta^k]$. Then, Property 5.4 focuses on the approximation of elements of $\mathbb{Q}[\beta^k]$, and Property 5.6 shows that it is always possible to specialize a symbolic rational (e.g. to the case of an even variable k) to enforce the existence of an approximation.

Property 5.3. *For $f \in \mathbb{SQ}$, let $f(k) = P(\beta^k)/Q(\beta^k)$ with $P, Q \in \mathbb{Z}[X]$ and S be the quotient of the Euclidean division of P by Q in $\mathbb{Q}[X]$. Then, $g \in \mathbb{SZ}$ approximates f if and only if g approximates $S(\beta^k)$.*

Proof. If $P = S \cdot Q + R$, with $S, R \in \mathbb{Q}[X]$ and $\deg(R) < \deg(Q)$, then $f(k) = S(\beta^k) + f_R(k)$ where $f_R(k) = R(\beta^k)/Q(\beta^k)$ is strictly bounded by 1. Hence, using the triangular inequality, for any $g \in \mathbb{SZ}$, $|S(\beta^k) - g| - 1 < |f - g| < |S(\beta^k) - g| + 1$ and the result follows. □

We now only address the problem of approximating a symbolic rational in $\mathbb{Q}[\beta^k]$. The following property gives a necessary and sufficient condition for the existence of an approximation and how to compute one when possible.

Property 5.4. *For $f \in \mathbb{Q}[\beta^k]$, let T, M and d be as in (5.1). Then, f admits an approximation in $\mathbb{SZ}$ if and only if $T(\beta^k) \equiv T(1) \pmod{M}$ for $k \in \{1, \cdots, \mathrm{ord}_M(\beta) - 1\}$. When it exists, such an approximation is given by*

$$g(k) = \frac{T(\beta^k) - c}{M \cdot \beta^d},$$

where $c \in \mathbb{Z}$ is such that $c \equiv T(1) \pmod{M}$ and $c \equiv T(0) \pmod{\beta^d}$.

Proof. Given $f \in \mathbb{Q}[\beta^k]$ and an approximation $g \in \mathbb{SZ}$, $f - g$ belongs to $\mathbb{Q}[\beta^k]$ and is bounded, so that $f-g = \lambda \in \mathbb{Q}$. In particular, we have $M\cdot\beta^d\cdot\lambda = T(\beta^k)-M\cdot\beta^d\cdot g(k) \in \mathbb{SZ}$ so that $\lambda = c/(M \cdot \beta^d)$ with $c \in \mathbb{Z}$. We deduce that f admits an approximation in $\mathbb{SZ}$ if and only if there exists $c \in \mathbb{Z}$ such that $(T(\beta^k) - c)/(M \cdot \beta^d) \in \mathbb{SZ}$. According to Property 5.2, this is equivalent to $c \equiv T(\beta^k) \pmod M$ for $k \in \{0, \dots, \mathrm{ord}_M(\beta) - 1\}$ and $c \equiv T(0) \pmod{\beta^d}$ and the result follows. □

The following example shows that some elements in $\mathbb{SQ}$ do not admit an approximation in $\mathbb{SZ}$.

Example 5.5. Let $\beta = 2$, we define

$$\xi_1(k) = \frac{2 \cdot 2^k + 22}{3}, \tag{5.2}$$

then we have (5.1) with $P(X) = 2X + 22$, $M = 3$ and $d = 0$. Since $P(1) \equiv 0 \pmod 3$ and $P(2) \equiv 2 \pmod 3$, we deduce from Property 5.4 that ξ_1 cannot be approximated in $\mathbb{SZ}$.

However, the following property shows that it is always possible to provide a partial result, at the cost of an additional assumption about k.

Property 5.6. *For $f \in \mathbb{Q}[\beta^k]$, let T, M, d be as in (5.1) and $\omega = \mathrm{ord}_M(\beta)$. Then, for any $\varphi \in \mathbb{Z}$, $f(\omega k + \varphi)$ admits an approximation in $\mathbb{SZ}$.*

Proof. Denoting by m the degree of $T \in \mathbb{Z}[X]$, we write $T(X) = \sum_{i=0}^m T_i X^i$. We define $\widetilde{T} \in \mathbb{Z}[X]$ by $\widetilde{T}(X) = \sum_{i=0}^m \beta^{\varphi i} T_i X^{\omega i}$, so that $T(\beta^{\omega k+\varphi}) = \widetilde{T}(\beta^k)$ and $f(\omega k + \varphi) = \widetilde{T}(\beta^k)/(M \cdot \beta^d)$. Moreover, for $k \in \{1, \cdots, \omega - 1\}$, we have $\beta^{\omega k+\varphi} \equiv \beta^\varphi \pmod M$, hence $\widetilde{T}(\beta^k) \equiv T(\beta^\varphi) \equiv \widetilde{T}(1) \pmod M$. We then conclude using Property 5.4. □

Going back to Example 5.5, we have $\omega = \mathrm{ord}_3(2) = 2$ so that $\xi_1(2k)$ and $\xi_1(2k + 1)$ admit approximations in $\mathbb{SZ}$. More precisely, we give one approximation for each: $\xi_1(2k) \in \mathbb{SZ}$ and $\xi_1(2k+1)-2/3 \in \mathbb{SZ}$. They can be computed using Algorithm 6 described below.

Combining Properties 5.3, 5.4 and 5.6 leads to Algorithm 6, that computes a minimal $\omega \in \mathbb{N}_{>0}$ and $g \in \mathbb{SQ}$ such that $g(\omega k) \in \mathbb{SZ}$ is an approximation of $f(\omega k)$, for any $f \in \mathbb{SQ}$.

5.2.2 Rounding

Let $\arg\min_{x\in\mathcal{X}} N(x) = \{y \in \mathcal{X} : N(y) \leqslant N(x), \text{for all } x \in \mathcal{X}\}$. We want to define the standard rounding functions from $\mathbb{SQ}$ to $\mathbb{SZ}$ (truncation, floor, ceiling and round to nearest), so that rounding $f \in \mathbb{SQ}$ to $\mathbb{SZ}$ amounts to compute a particular approximation of f. However, we have seen in the previous subsection that there exist elements in $\mathbb{SQ}$ that cannot be approximated in $\mathbb{SZ}$. As a consequence, we can only define partial rounding functions, and we define them using the following constraints:

- truncation: $\mathrm{trunc}(f) \in \arg\min_{\substack{g\in\mathbb{SZ}\\ |g|\leqslant|f|}} |g - f|$;

Algorithm 6 – Approximate
(Compute an approximation in $\mathbb{SZ}$ of a symbolic rational)
Input: f a symbolic rational.
Output: $g \in \mathbb{SQ}$ and $\omega \in \mathbb{N}_{>0}$ minimal such that $g(\omega k)$ is a symbolic integer approximating $f(\omega k)$.

1. Given $P, Q \in \mathbb{Z}[X]$ such that $f(\beta^k) = P(\beta^k)/Q(\beta^k)$, let S be the quotient in the Euclidean division of P by Q in $\mathbb{Z}[X]$;
2. Let $T \in \mathbb{Z}[x]$, $M, d \in \mathbb{N}$ such that $S(\beta^k) = T(\beta^k)/(M \cdot \beta^d)$ as in (5.1);
3. $c_M = T(1) \pmod M$;
4. $c_\beta = T(0) \pmod{\beta^d}$;
5. Let $c \in \mathbb{Z}$ be such that $c \equiv c_M \pmod M$ and $c \equiv c_\beta \pmod{\beta^d}$;
6. $g = (T - c)/(M \cdot \beta^d)$;
7. Let ω be the smallest positive integer such that $T(\beta^{\omega k}) \equiv c_M \pmod M$ for $k = 1, \ldots, \mathrm{ord}_M(\beta) - 1$;
8. **return** g, ω;

- floor: $\lfloor f \rfloor \in \arg\min_{\substack{g \in \mathbb{SZ} \\ g \leqslant f}} |g - f|$;
- ceiling: $\lceil f \rceil \in \arg\min_{\substack{g \in \mathbb{SZ} \\ g \geqslant f}} |g - f|$;
- round to nearest: $\lfloor f \rceil \in \arg\min_{g \in \mathbb{SZ}} |g - f|$.

Three situations can occur. When the set in the right hand side of the constraint is empty, the function is undefined. When the set contains exactly one element, this is the rounded result. When rounding to the nearest, and only in this case, we can have two elements in the set: for this reason, a tie-breaking function s must be used, that can be either tiesToAway which chooses the element with the largest absolute value, or tiesToEven, which selects the "even" one. In the latter case, we mimic the notion of parity for integers using the following definition.

Definition 5.7. *For $f \in \mathbb{SZ}$, f is even if $f/2 \in \mathbb{SZ}$ and odd if $f - 1$ is even.*

This definition is chosen so that an even (resp. odd) symbolic integer evaluates to even (resp. odd) integers as $k \to \infty$. The following property states that any symbolic integer is either even or odd, as expected for a notion of parity.

Property 5.8. *For $f \in \mathbb{SZ}$, let T, M, d as in (5.1). The symbolic integer f is even if and only if $T(0)/\beta^d$ is even and f is odd if and only if $T(0)/\beta^d$ is odd.*

Proof. We recall that β is even. From $f(k)/2 = (\beta/2) \cdot T(\beta^k)/(M \cdot \beta^{d+1})$ and Property 5.2, f is even if and only if $(\beta/2) \cdot T(0) \equiv 0 \pmod{\beta^{d+1}}$, which is equivalent to $T(0)/\beta^d$ being even. Similarly, f is odd if and only if $T(0)/\beta^d - M$ is even, which is equivalent to $T(0)/\beta^d$ being odd because M is odd. □

As already mentioned, the result of any rounding function applied to $f \in \mathbb{SQ}$, when it exists, is an approximation of f by $\mathbb{SZ}$. More precisely, we have the following result.

Theorem 5.9. *For $f \in \mathbb{SQ}$ and $\circ \in \{trunc, \lfloor\cdot\rfloor, \lceil\cdot\rceil, \lfloor\cdot\rceil\}$, $\circ(f)$ is defined if and only if f admits an approximation in $\mathbb{SZ}$.*

Proof. We assume that $g \in \mathbb{SZ}$ is an approximation of f and we will prove the result for $\lfloor\cdot\rceil$. Let $C \in \mathbb{N}$ be such that $|g-f| < C$. For any $g_1 \in \mathbb{SZ}$, we can distinguish two possible cases: either $|g_1 - f| \geqslant C > |g - f|$ and we can ignore g_1 when looking for the arg min; or $|g_1 - f| < C$ and we deduce that $g_1 - g \in \mathbb{Z} \cap (-2C, 2C)$. Consequently, the domain of minimization is reduced to $\{g + n : n \in \mathbb{Z} \cap (-2C, 2C)\}$ which is nonempty and finite and the result follows. The same reasoning holds, with an additional constraint, for the other rounding functions. □

As we have seen in the previous subsection, it is always possible to specialize $f \in \mathbb{SQ}$ in $f(\omega k) \in \mathbb{SQ}$ with $\omega \in \mathbb{N}_{>0}$, so that $f(\omega k)$ admits an approximation in $\mathbb{SZ}$. Theorem 5.9 shows that we can do the same with the rounding partial functions: for $f \in \mathbb{SQ}$ and a rounding function $\circ \in \{trunc, \lfloor\cdot\rfloor, \lceil\cdot\rceil, \lfloor\cdot\rceil\}$, we compute $\omega \in \mathbb{N}_{>0}$ and $g \in \mathbb{SZ}$ such that $g(\omega k) = \circ(f(\omega k))$, with ω as small as possible. Algorithm 7 below describes how to compute ω and g, when rounding f to the nearest using a tie breaking function s.

Algorithm 7 – Rounding to the nearest integer
(Compute the nearest symbolic integer to a symbolic rational)
Input: f a symbolic rational.
Output: $g \in \mathbb{SQ}$ and $\omega \in \mathbb{N}_{>0}$ such that $g(\omega k) = \lfloor f(\omega k)\rceil$.

1. $g_1, \omega =$ Approximate(f);
2. $C = \lim_{k\to\infty} f(k) - g_1(k)$;
3. $g_2 = g_1 + \lfloor C\rceil$;
4. **if** $|g_2 - f| < 1/2$ **then** $g = g_2$;
5. **else if** $|g_2 - f| = 1/2$ **then** $g = s(\{f - 1/2, f + 1/2\})$;
6. **else** $g = g_2 + \text{sign}(f - g_2)$;
7. **return** g, ω;

In Algorithm 7, since $g(\omega k)$ is an approximation of $f(\omega k)$ in $\mathbb{SZ}$, we know that C belongs to $\mathbb{Q}$ and we can write $f = g_1 + C + \epsilon$, where $\epsilon \in \mathbb{SQ}$ is such that $\epsilon(k) \to 0$ as $k \to \infty$. Then $|g_2(k) - f(k)| \leqslant |\lfloor C\rceil - C| + |\epsilon(k)|$, hence $|g_2 - f| < 1$: the correct rounding g is then deduced from g_2. Note that when $|g_2 - f| > 1/2$, g_2 is one of the two integers surrounding f, but not the nearest one. For example, if $f = 1/2 + 2^{-k}$ (with tiesToEven), then f is first approximated in $\mathbb{SZ}$ by $g_1 = 0$ and $C = 1/2$; the algorithm then computes $g_2 = 0$ while $\lfloor f\rceil = 1 = g_2 + 1$.

Algorithm 7 can easily be modified to deal with the other rounding functions: only the correction step needs to be adapted, as in Algorithm 8.

Algorithm 8 – FLOOR
(Compute the floor of a symbolic rational)
Input: f a symbolic rational.
Output: $g \in \mathbb{SQ}$ and $\omega \in \mathbb{N}_{>0}$ such that $g(\omega k)$ is the floor of $f(\omega k)$.

1. $g_1, \omega =$ Approximate(f);
2. $C = \lim_{k\to\infty} f(k) - g_1(k)$;
3. $g_2 = g_1 + \lfloor C \rceil$;
4. **if** $g_2 > f$ **then** $g = g_2 - 1$;
5. **else** $g = g_2$;
6. **return** g, ω;

5.3 Symbolic floating-point arithmetic

In this section, we deal with the last step of (1): we first define symbolic floating-point numbers as particular elements of $\mathbb{SQ}$, and then we define the rounding functions from $\mathbb{SQ}$ to $\mathbb{SF}_p$. Finally, we show that these rounding functions match asymptotically the classical ones from $\mathbb{Q}$ to $\mathbb{F}$.

5.3.1 Symbolic floating-point numbers

In conventional base-β floating-point arithmetic (and assuming an unlimited exponent range), a nonzero precision-p floating-point number is a number that can be written $M\beta^e$, where M is an integer in the range $[\beta^{p-1}, \beta^p)$ and e is an integer. The formalism introduced earlier allows us to define a set of symbolic floating-point numbers, with M a symbolic integer in $\mathbb{SZ}$, and e a linear exponent in $\mathbb{L}$. More precisely,

Definition 5.10. *Given a function $p \in \mathbb{L}_{>1}$, let us define*

$$\mathbb{SF}_p = \{0\} \cup \{M\beta^e \,:\, M \in \mathbb{SZ},\, e \in \mathbb{L},\, \beta^{p-1} \leqslant |M| < \beta^p\}.$$

We have $\mathbb{SF}_p \subset \mathbb{SQ}$, and every $h \in \mathbb{SF}_p$ satisfies the following: there exists $k_0 \in \mathbb{N}$ such that for all $k \geqslant k_0$, $h(k)$ is a floating-point number in base β and precision $p(k) \in \mathbb{N}_{\geqslant 2}$.

The set $\mathbb{SF}_p$ can thus be considered as a set of "symbolic floating-point numbers" whose precision p is parametrized by the variable k. In particular, $0 \in \mathbb{SF}_p$ and $\mathbb{SF}_p = -\mathbb{SF}_p$. Moreover, $h \in \mathbb{SF}_p - \{0\}$ can be written as

$$h = H \cdot \mathrm{ulp}_p(h), \tag{5.3}$$

with $H \in \mathbb{SZ}$ and $\beta^{p-1} \leqslant |H| < \beta^p$: we call $|H|$ the significand of h. Hence, $f \in \mathbb{SQ} - \{0\}$ is a symbolic floating-point number in precision p if and only if $f/\mathrm{ulp}_p(f) \in \mathbb{SZ}$, as the definition of ulp_p already implies that $\beta^{p-1} \leqslant |f/\mathrm{ulp}_p(f)| < \beta^p$.

5.3.2 Rounding to symbolic floating-point numbers

Given a precision $p \in \mathbb{L}_{>1}$, as we did in Section 5.2.2, we define rounding functions as partial functions from $\mathbb{SQ}$ to $\mathbb{SF}_p$ satisfying the following constraints:

- rounding toward zero: $\mathrm{RZ}_p(f) \in \arg\min_{\substack{h \in \mathbb{SF}_p \\ |h| \leqslant |f|}} |h - f|$;
- rounding toward negative: $\mathrm{RD}_p(f) \in \arg\min_{\substack{h \in \mathbb{SF}_p \\ h \leqslant f}} |h - f|$;
- rounding toward positive: $\mathrm{RU}_p(f) \in \arg\min_{\substack{h \in \mathbb{SF}_p \\ h \geqslant f}} |h - f|$;
- rounding to the nearest: $\mathrm{RN}_p(f) \in \arg\min_{h \in \mathbb{SF}_p} |h - f|$.

When rounding to the nearest, a tie breaking function s'_p is needed when $f \in \mathbb{SQ}$ is exactly halfway between two consecutive symbolic floating-point numbers $\{h_1, h_2\}$, whereas the other rounding functions are already fully determined.

We assume that s'_p has the following regularities; for $e \in \mathbb{L}$,

$$s'_p(\{\beta^e \cdot h_1, \beta^e \cdot h_2\}) = \beta^e \cdot s'_p(\{h_1, h_2\}) \quad \text{and} \quad s'_p(\{-h_1, -h_2\}) = -s'_p(\{h_1, h_2\}). \tag{5.4}$$

Moreover, noticing that the pairs of consecutive integers in $[\beta^{p-1}, \beta^p]$ match the pairs of consecutive floating-point numbers in precision p in the same interval, we make the additional assumption that, for any precision $p \in \mathbb{L}_{>1}$ and any $h \in \mathbb{SZ} \cap [\beta^{p-1}, \beta^p)$,

$$s'_p(\{h, h+1\}) = s(\{h, h+1\}), \tag{5.5}$$

where s is the tie-breaking function for rounding to the nearest symbolic integer, introduced in Algorithm 7. For instance, tiesToEven, that selects the symbolic floating-point number whose significand is even, and tiesToAway, that selects the one with the largest absolute value, satisfy the hypothesis above with their integer counterparts.

It can be checked that these rounding functions are nondecreasing: if $\circ_p$ denotes either RZ_p, RD_p, RU_p, or RN_p, and if $f_1, f_2 \in \mathbb{SQ}$ are such that $\circ_p(f_1)$ and $\circ_p(f_2)$ are defined, then we have

$$f_1 \leqslant f_2 \quad \Rightarrow \quad \circ_p(f_1) \leqslant \circ_p(f_2).$$

Moreover, the regularity assumptions (5.4) about the tie-breaking function s'_p transfer to RN_p: given $f \in \mathbb{SQ}$, we have

$$\mathrm{RN}_p(\beta^e \cdot f) = \beta^e \cdot \mathrm{RN}_p(f) \quad \text{and} \quad \mathrm{RN}_p(-f) = -\mathrm{RN}_p(f).$$

The following theorem provides a practical way of evaluating the rounding functions: it shows that a relation similar to (2) holds for all the rounding functions defined above, which means that we can use the algorithms introduced earlier (such as Algorithm 7) to round symbolic rationals to symbolic floating-point numbers. The proof is only given for rounding to the nearest, but the same reasoning applies to the other rounding functions (without having to take into account the case of midpoints).

Theorem 5.11. *Let $p \in \mathbb{L}_{\geqslant 2}$ and $(\circ_p, \circ)$ be either $(\mathrm{RZ}_p, \mathrm{trunc})$, $(\mathrm{RD}_p, \lfloor\cdot\rfloor)$, $(\mathrm{RU}_p, \lceil\cdot\rceil)$, or $(\mathrm{RN}_p, \lfloor\cdot\rceil)$. In the case of $(\mathrm{RN}_p, \lfloor\cdot\rceil)$, let s and s'_p be two tie-breaking functions defined over $\mathbb{SZ}$ and $\mathbb{SF}_p$ respectively, and satisfying (5.5). Then, for any $f \in \mathbb{SQ} - \{0\}$, we have*

$$\circ_p(f) = \mathrm{ulp}_p(f) \cdot \circ(f/\mathrm{ulp}_p(f))$$

Proof. As already mentioned, we give the proof for $(\circ_p, \circ) = (\mathrm{RN}_p, \lfloor\cdot\rceil)$. Defining $F = f/\mathrm{ulp}_p(f)$, $\sigma = \arg\min_{g\in\mathbb{SZ}} |g - F|$ and $\sigma' = \arg\min_{h\in\mathbb{SF}_p} |h - f|$, let us first prove

$$\sigma' = \mathrm{ulp}_p(f) \cdot \sigma. \tag{5.6}$$

Since changing f to $-f$ modifies both σ and σ' to $-\sigma$ and $-\sigma'$, respectively, we can assume $f > 0$ without loss of generality. Let $e \in \mathbb{L}$ be such that $\beta^e \leqslant f < \beta^{e+1}$.

For any $h^* \in \sigma'$, let us consider $g^* = h^*/\mathrm{ulp}_p(f)$. Since β^e and β^{e+1} are two elements in $\mathbb{SF}_p$ surrounding f, we deduce $\beta^e \leqslant h^* \leqslant \beta^{e+1}$, which implies that $g^* \in \mathbb{SZ}$. Let g be any element in $\mathbb{SZ}$. If $\beta^{p-1} \leqslant g \leqslant \beta^p$, then $\mathrm{ulp}_p(f) \cdot g \in \mathbb{SF}_p$, so that $|\mathrm{ulp}_p(f) \cdot g^* - f| \leqslant |\mathrm{ulp}_p(f) \cdot g - f|$, and $|g^* - F| \leqslant |g - F|$. Since $\beta^{p-1} \leqslant F \leqslant \beta^p$, if $g < \beta^{p-1}$ then $|\beta^{p-1} - F| < |g - F|$, and if $\beta^p < g$ then $|\beta^p - F| < |g - F|$. Since both β^p and β^{p-1} belong to $\mathbb{SZ}$, in any case we have $|g^* - F| \leqslant |g - F|$, which implies $g^* \in \sigma$, and $h^* \in \mathrm{ulp}_p(f) \cdot \sigma$.

Conversely, for any $g^* \in \sigma$, we consider $h^* = \mathrm{ulp}_p(f) \cdot g^*$. Since β^{p-1} and β^p belong to $\mathbb{SZ}$ and are surrounding F, we have $\beta^{p-1} \leqslant g^* \leqslant \beta^p$, which implies that $h^* \in \mathbb{SF}_p$. Moreover, since $\beta^e \leqslant f < \beta^{e+1}$, any $h \in \mathbb{SF}_p$ minimizing $|h - f|$ must satisfy $\beta^e \leqslant h \leqslant \beta^{e+1}$, which implies $h/\mathrm{ulp}_p(f) \in \mathbb{SZ}$. Then, we have $|g^* - F| \leqslant |h/\mathrm{ulp}_p(f) - F|$, so that $|h^* - f| \leqslant |h - f|$, which shows that $h^* \in \sigma'$, and concludes the proof of (5.6).

If σ'_p is empty or is a singleton, then the result follows directly from (5.6). If σ'_p is a pair, then we can use (5.6), (5.4) and (5.5) successively to get the result. □

5.3.3 Relationship with the classic floating-point arithmetic

In this subsection, we prove that the symbolic rounding of an element f in $\mathbb{SQ}$ evaluates, for k large enough, to the classic rounding of $f(k) \in \mathbb{Q}$ to an element in $\mathbb{F}_{p(k)}$, where k (and thus p(k)) is a fixed integer.

In the case of rounding to the nearest, we assume that we are given a family of tie-breaking functions $(s''_p)_{p\in\mathbb{N}_{\geqslant 2}}$ that choose one element among pairs of consecutive floating-point numbers in $\mathbb{F}_{p(k)}$. If $\{h_1, h_2\}$ is a pair of consecutive elements in $\mathbb{SF}_p$, then for k large enough, $\{h_1(k), h_2(k)\}$ is a pair of consecutive floating-point numbers in $\mathbb{F}_{p(k)}$. As a consequence, we can assume that ties are broken using the same strategy when rounding from $\mathbb{SQ}$ to $\mathbb{SF}_p$ and when (classically) rounding from $\mathbb{Q}$ to $\mathbb{F}_{p(k)}$: for all $p \in \mathbb{L}_{\geqslant 2}$, and any pair of consecutive elements $\{h_1, h_2\} \subset \mathbb{SF}_p$,

$$s'_p(\{h_1, h_2\}) = s''_{p(k)}(\{h_1(k), h_2(k)\}) \quad \text{as } k \to \infty. \tag{5.7}$$

Hence, assuming that the previous equality holds, we can safely use the same notation for the tie-breaking functions over $\mathbb{SF}_p$ and $\mathbb{F}_{p(k)}$, and we denote by $\mathrm{RN}_{p(k)}$ the classic function that rounds any real to $\mathbb{F}_{p(k)}$ using the tie-breaking function $s'_{p(k)}$. Note that it can be checked using (5.3) that (5.7) is satisfied for both tiesToEven and tiesToAway.

Theorem 5.12. *Given $p \in \mathbb{L}_{\geqslant 2}$, let $\circ_p$ denote either RZ_p, RD_p, RU_p, or RN_p, and assume that (5.7) holds when using RN_p. If $f, h \in \mathbb{SQ}$ are such that $h = \circ_p(f)$, then*

$$h(k) = \circ_{p(k)}(f(k)) \quad \text{as } k \to \infty.$$

Proof. If $f = 0$, then $h = 0$ and the result is clear. We consider $f \in \mathbb{SQ} - \{0\}$ and define $e \in \mathbb{L}$ such that $\beta^e \leqslant |f| \leqslant \beta^{e+1}$. For k large enough, the definitions of the symbolic exponent and ulp functions match the numerical ones, that is, $e(k)$ is the numerical exponent of $f(k)$, and $(\mathrm{ulp}_p(f))(k) = \mathrm{ulp}_{p(k)}(f(k))$. Since both $\pm\beta^e$ and $\pm\beta^{e+1}$ are symbolic floating-point numbers in precision p, by definition of h we have $\beta^e \leqslant |h| \leqslant \beta^{e+1}$. Moreover, we deduce from Theorem 5.11 that

$$|h - f| \leqslant \tfrac{1}{2}\,\mathrm{ulp}_p(f). \tag{5.8}$$

If the inequality is strict, then, for k large enough, we have $\beta^{e(k)} \leqslant |h(k)| \leqslant \beta^{e(k)+1}$ and $|h(k) - f(k)| < \frac{1}{2}\,\mathrm{ulp}_{p(k)}(f(k))$, and the result follows. In case of equality in (5.8), then h results from applying the tie-breaking function s'_p to a pair of consecutive symbolic floating-point numbers: in this case, the result follows from (5.7). □

The consequence of this theorem is that the symbolic floating-point arithmetic we defined can be used to compute the result of algorithms using floating-point number on inputs that are parametrized by the precision. In particular, in the case of asymptotically optimal error bounds, the certificates of optimality can be checked with this arithmetic.

Chapter 6

A Maple library for symbolic floating-point arithmetic

In the previous chapter, we defined the arithmetic over the symbolic data in $\mathbb{SQ}$ and $\mathbb{SF}_p$. We describe in this chapter a Maple library that implements this symbolic floating-point arithmetic and illustrate its use on examples from the literature. The formalization above describes asymptotic behaviors, that are valid as $k \to \infty$; in practice, we want to compute a lower bound k_0 above which the asymptotic behavior is reached. For example, if $\widehat{f} \in \mathbb{SF}_p$ is the nearest symbolic floating-point number to $f \in \mathbb{SQ}$ in precision $p \in \mathbb{L}_{\geqslant 2}$, it is given with $k_0 \in \mathbb{N}$ such that for all $k \geqslant k_0$, $\widehat{f}(k) = \mathrm{RN}_{p(k)}(f(k))$. We describe in Section 6.1 how such a k_0 is computed inside the library. Then, we detail the interface of our Maple library in Section 6.2. Finally, we give in Section 6.3 some examples of the use of this library to illustrate its simple and intuitive use on practical cases.

6.1 Practical handling of the asymptotic behavior

To compute a lower bound from which the asymptotic behavior is reached, every element of $\mathbb{SQ}$ is given with an additional $k_0 \in \mathbb{N}$ that ensures that the denominator does not vanish. The heuristic to compute such a k_0 is to find the minimal value of k realizing a strict ordering of the exponents appearing in the denominator. More formally, if the denominator is $\sum_{i=1}^{n} c_i \beta^{e_i}$ with $|c_i| \in \{1, 2, \ldots, \beta - 1\}$ and $e_i \in \mathbb{L}$, then we compute the smallest integer k_0 such that:

$$e_1(k) > e_2(k) > \cdots > e_n(k) \quad \text{for all } k \geqslant k_0. \tag{6.1}$$

Moreover, for any operation on elements of $\mathbb{SQ}$, given with their own k_0, the result is also given with a new k_0 such that evaluating the symbolic result for $k \geqslant k_0$ gives the correct numerical result.

We also chose in our implementation to force the new value of k_0, coming with the result, to be greater than or equal to any k_0 given as input. For example, if f and k_0 are given by (6.1) with $n \geqslant 1$, the sign function computes a pair $(s, k_0') \in \{0, \pm 1\} \times \mathbb{N}$ such that $k_0' \geqslant k_0$ and $\mathrm{sign}(f(k)) = s$ for all $k \geqslant k_0'$. Hence, we ensure a nondecreasing

behavior for the k_0's, and the last k_0 obtained after a sequence of operations guarantees that the asymptotic behavior is reached for every operation in this sequence.

When comparing two exponents $a_1k + b_1$ and $a_2k + b_2$ in $\mathbb{L}$, either $a_1 = a_2$ and the ordering is valid for every $k \in \mathbb{N}$, or $a_1 \neq a_2$ and the comparison is valid as soon as $k \geqslant k_0 = \lfloor (b_2 - b_1)/(a_1 - a_2) \rfloor + 1$. All the updates of k_0 come from the algorithm comparing two exponents in $\mathbb{L}$. Together with the nondecreasing behavior of k_0 we enforce, this is the building block for the computation of a valid k_0.

6.2 Description of the library

Our Maple library defines a procedure that, given an even fixed base β and a symbolic variable k, builds a Maple package for a symbolic floating-point arithmetic in base β. This package defines the class of symbolic numbers $\mathbb{SQ}$, overloading the basic operations (`+`, `-`, `*`, `/` and `^`). It also defines a set of functions over objects in $\mathbb{SQ}$ for a symbolic integer arithmetic and a symbolic floating-point one. Elements of $\mathbb{SZ}$ and $\mathbb{SF}_p$ are just objects of the same class $\mathbb{SQ}$.

More precisely, an object f of the class $\mathbb{SQ}$ contains 3 attributes:

- one representing f as a rational function over a local variable according to the change of variable $X = \beta^k$;
- two integers k_0 and ω such that the calculations that led to f are correct for any k larger that k_0 and multiple of ω.

This class implements the following methods:

- a constructor `SQ`, that takes as a parameter an element in $\mathbb{SQ}$, parametrized by k. It computes an initial k_0 that ensures that the denominator does not vanish, and sets $\omega = 1$;
- the overloaded exact operations `+`, `-`, `*`, `/` and `^` on $\mathbb{SQ}$;
- an accessor `get_fun`, that returns the rational function representing f;
- two accessors `get_k0` and `get_omega`, that return k_0 and ω respectively;
- `update_k0` that replaces the previous k_0 of $f \in \mathbb{SQ}$ by the maximum between the previous and the new k_0, and returns f.

The package defines 3 elements of $\mathbb{SQ}$ (`zero`, `oneHalf` and `one`), and the following functions over $\mathbb{SQ}$:

- `sign`: returns $(s, k_0) \in \{-1, 0, 1\} \times \mathbb{N}$ such that s is the sign of $f(k)$ for $k \geqslant k_0$;
- `cmp`(f_1, f_2) and `abs`(f): return respectively `sign`$(f_1 - f_2)$ and $|f|$;
- `exponent`(f): returns $(e, k_0) \in \mathbb{L} \times \mathbb{N}$ such that $\beta^{e(k)} \leqslant |f|(k) < \beta^{e(k)+1}$ for $k \geqslant k_0$;

- `tiesToEven` and `tiesToAway`: two tie-breaking functions for rounding to the nearest integer, whose parameters are two consecutive symbolic integers; `tiesToEven` selects the even symbolic integer and `tiesToAway` the largest one comparing the absolute values;
- `trunc`(f,p), `floor`(f,p), `ceil`(f,p), and `round`(f,p): rounding functions from $\mathbb{SQ}$ to $\mathbb{SZ}$, overloading the classic ones; the tie-breaking function for `round` can be given as a third parameter and is set by default to `tiesToEven`;
- `ulp`(f,p): returns $g \in \mathbb{SQ}$ such that $g(k) = \mathrm{ulp}_{p(k)}(f(k))$ for $k \geqslant$ `get_k0`(g);
- `rz`(f,p), `rd`(f,p), `ru`(f,p) and `rn`(f,p): rounding functions from $\mathbb{SQ}$ to $\mathbb{SF}_p$, where $p \in \mathbb{L}$ is the second parameter; they rely on the rounding-to-integer functions so that the tie-breaking function can be chosen similarly to `round`, using an optional third parameter;
- `expr_ur`(f,p,u), where u is a symbolic variable: expresses f as a rational function with respect to the unit roundoff $u = \beta^{1-p}/2$. If a and b are such that $p = ak + b$, since $X = \beta^k$, the function performs the change of variable $X = (\beta^{1-b}/u)^{1/a}$ in the internal representation of f as a rational function.

6.3 Examples

Below, we give four examples that illustrate the use of our Maple library. The first example (Example 4.6 in [20]) is devoted to the computation of a two-by-two determinant with Kahan's algorithm (see Algorithm 2); we run this algorithm on the set of inputs parametrized by the precision p given in the article, for which the relative error is equivalent to $2u$ as $u \to 0$ (or $p \to \infty$ while β is fixed). It provides a certificate of asymptotic optimality for the relative error bound $2u$.

The second example concerns the algorithm CompDivS given in [21] to compute a complex floating-point quotient, that is an approximation $\widehat{x} + i\,\widehat{y}$ of $(a + ib)/(c + id)$, where all variables are taking floating-point values. The authors of [21] are interested in the componentwise relative error. Example 8 in that paper provides inputs parametrized by the precision for which the computation of the real part leads to a relative error equivalent to the a priori bound of $5u$. In this example, we use the division operation and an even precision.

In the third example of this section (from [8]), we illustrate the use of `get_omega()` to figure out if an additional condition on the divisibility of p is needed, when such a condition is not known in advance.

The final example deals with the complex floating-point inversion algorithm (Algorithm 5) analyzed in Part II. We proved that the componentwise relative error is always bounded by $3u$ and gave an example that shows the asymptotic optimality of this bound when the precision is even. The library is used to check the computations presented in Chapter 3 about this example.

Before any computation, the package has to be built and loaded, for a fixed base. One possible way to do so in base 10 is with the following command lines:

```
> read("libsfp0.6.mpl"):
> beta := 10:
> SF10 := SF(beta, k):
> with(SF10):
```

After reading the files and setting the base to 10, the third line creates a package for a symbolic floating-point arithmetic in base 10, with the parameter k. This package can then be loaded for a simpler use since it avoids the necessity of recalling the package for each function call.

6.3.1 Kahan's algorithm

The algorithm is implemented as a Maple procedure as follows, using the exact operations in $\mathbb{SQ}$, and the RN function from our Maple library:

```
> Kahan := proc(a, b, c, d, p)
>   wh := rn(bc, p);
>   e  := rn(wh - bc, p);
>   fh := rn(ad - wh, p);
>   rn(fh + e, p)
> end proc:
```

Then we set the input variables as Maple expressions parametrized by k as in [20], and we compute an expression for the exact result x:

```
> p := k:
> a := SQ(beta^(p-1) + 1):
> b := a:
> c := SQ(beta^(p-1) + (beta/2)*beta^(p-2)):
> d := SQ(2*beta^(p-1) + (beta/2)*beta^(p-2)):
> x := ad - bc;
  x := 10^(2k-2) + 10^(k-1)
```

Next, we apply the symbolic version of Kahan's algorithm to get the approximate result xh, in precision $p = k$:

```
> xh := Kahan(a, b, c, d, p);
    xh := 10^(2k-2)
> get_omega(xh);
    1
> get_k0(xh);
    3
```

The paper-and-pencil computation of xh is rather tedious and occupies half a page in the article; it can be performed now with a few lines of Maple code. Since get_omega(xh) returns 1 and get_k0(xh) returns 3, we know that the computed result is valid for any $k \geqslant 3$, that is for $p \geqslant 3$, and using a numerical evaluation, it can be checked that for $p = 2$ the algorithm returns 120 while $\mathtt{xh}(2) = 100$.

We now compute the error and express it with respect to $u = 1/2 \cdot 10^{1-p}$ using the representation as a rational function over $X = 10^k = 5/u$.

```
> err := abs((x - xh)/x);
    err := 2/(4^k + 2)
> simplify(get_fun(err)(5/u));
    2*u/(1+2*u)
```

These computations double-check the certificate of optimality for the relative error bound of Kahan's algorithm for $\beta = 10$. Note that the last instruction could be replaced by `simplify(expr_ur(err, p, u))`, so that the change of variable is computed automatically.

6.3.2 Complex floating-point division

This example deals with floating-point division, and illustrates how to handle the case of an even precision, in base 2. The paper-and-pencil computation of the result is more involved than in the previous case, especially for rounding the quotient, while it is of course much more efficient with our library. We first build the appropriate package:

```
> read("libsfp0.6.mpl"):
> beta := 2:
> SF2 := SF(beta, k):
> with(SF2):
```

Since p is even, we set $p = 2k$:

```
> p := 2*k:
> a := SQ(beta^p-5*beta^(p/2-1)):
> b := SQ(-beta^(p/2) + 5/beta - 3*beta^(-p/2)):
> c := SQ(beta^p - beta):
> d := SQ(beta^(3*p/2)+beta^p):
```

As previously mentioned, a, b, c, d are built into the data structure. We compute the exact real part of the result `r` and the approximate one `rh` using the algorithm provided in [21]:

```
> r := (a*c+b*d)/(c^2+d^2):
> Dh := rn(c^2 + rn(d^2, p), p):
> Gh := Kahan(a, -b, d, c, p):
> rh := rn(Gh/Dh, p);
    rh := -8^(-k)-(1/2)*16^(-k)
> get_omega(rh);
    1
> get_k0(rh);
    4
```

Since `get_omega(xh)` and `get_k0(xh)` return 1 and 4 respectively, we know that the computed result is valid for any $k \geqslant 4$. In fact, it can be checked by numerical evaluation for $k = 3$ that the algorithm and the expression `rh` produce the same value. Hence, `rh` is valid for all $k \geqslant 3$.

Next, we compute the relative error, express it with respect to the unit roundoff $u = 2^{-2k}$ and compute an equivalent as $u \to 0$.

```
> err := (rh - r)/r:
> series(expr_ur(err, p, u), u=0, 3) assuming u > 0:
```

This gives `err` $= 5u - \frac{23}{2}u^{3/2} + \mathcal{O}(u^2)$ when $u \to 0$, which confirms the result given in [21].

6.3.3 Complex floating-point multiplication

We illustrate the use of `get_omega()` when an additional condition on the divisibility of k is needed. The example we give is taken from Corollary 4 in [8]: the authors provide a set of parametrized inputs to prove that, in base 2 and assuming that the precision is even, the relative error bound $\sqrt{5}$u for the complex floating-point multiplication is asymptotically optimal. Let us consider only one of the inputs proposed, namely $f = 2/3 \cdot (1 + 11 \cdot 2^{-p})$, and see if it is possible to use this input for any precision. For this purpose, we ignore the assumption on the parity of p, and we round f using the `rn` function from the library:

```
> beta := 2: p := k:
> f := SQ(2/3*(1+11*beta^(-p))):
> fh := rn(f, p);
    fh := 2/3+(22/3)*2^(-k)
> get_k0(fh);
    6
> get_omega(fh);
    2
```

Note that `get_omega(fh)` returns 2, as the attribute ω of the computed result `fh`. From the integer returned by `get_k0(fh)`, we know that '$k \geqslant 6$ even' is a sufficient condition to ensure that the computed rounding is correct. Under this condition, since `rn`$(f, p) = f$, we known that f is a symbolic floating-point number in precision $p = k$.

If we want to determine the rounding of f when p is odd, it is of course possible to redo the computation in precision $p = 2k + 1$:

```
> beta := 2: p := 2*k+1:
> f := SQ(2/3*(1+11*beta^(-p))):
> fh := rn(f, p);
    fh := 2/3+(23/6)*4^(-k)
> get_k0(fh);
    3
> get_omega(fh);
    1
```

As indicated by `get_k0(fh)` and `get_omega(fh)`, the computed rounding is then valid for any $k \geqslant 3$.

6.3.4 Complex floating-point inversion

Finally, we detail how to check the certificate of optimality given in Chapter 3 for the componentwise analysis of complex floating-point inversion with Algorithm 5. We assume that the base has been set to 2 and that the symbolic variable is k. The algorithm to compute the real part can be implemented in the following way:

```
> CompInvR := proc(a, b, p)
>   sa := rn(a^2, p);
>   sb := rn(b^2, p);
>   s := rn(sa + sb, p);
>   rn(a/s, p)
> end proc:
```

Then we set the input variables as Maple expressions parametrized by k, with an even precision:

```
> p := 2k:
> a := SQ(2^(p/2-1) + 5*2^(-2) + 2^(-p/2+2)):
> b := SQ(2^(p-1) + 2^(p/2-1) + 1):
```

Next, we compute the exact result `x` and the computed result `xh`, in precision $p = 2k$:

```
> x := a/(a^2 + b^2):
> xh := CompInvR(a, b, p):
    xh := 10^(2k-2)
> get_omega(xh);
    1
> get_k0(xh);
    8
```

Since `get_omega(xh)` returns 1 and `get_k0(xh)` returns 8, we know that the computed result is valid for any $k \geqslant 12$, that is for any even precision larger that or equal to 16.

We now compute the error and express it with respect to the unit roundoff u. Since we are interested in the first orders in the asymptotic expansion as $u \to 0$, we only compute the following:

```
> err := abs((x - xh)/x):
> series(expr_ur(err, p, u), u=0) assuming u>0;
    3*u - 31/2 *u^(3/2) + O(u^2)
```

These computations double-check the certificate of optimality described in Section 3.2 for the componentwise relative error bound of complex inversion using Algorithm 5.

Conclusion of Part III

We finally report some practical computing times to show that our library is not only of theoretical interest, but that it can be used to efficiently check examples from the literature, taken from [8, 20, 21, 31].

The individual measured computing times for each example (average on 100 runs) are reported in Table 1: the measurements were performed under Maple 18, with the command `time[real]()`, on a laptop equipped with an Intel Core i5 (4310U, 2 GHz) running Linux 3.16. All the examples listed here use rounding to the nearest only, with tiesToEven: we report in the column "#`rn`" the number of calls to the library function `rn()` required by the examples, as a rough indication of their "size".

Although the code is not designed with high performance as the primary target, each example is checked within a few tens of milliseconds, even Example 8 from [21] involving a symbolic division and 12 calls to the rounding function. Overall, the library checks 29 examples (25 in base 2, and 4 in base 10) similar to the ones given in Section 6.3, in less than 0.5 second of CPU time on a recent laptop, which we consider as sufficiently fast for our purposes.

These experiments confirm that it is possible to efficiently check examples from the literature involving symbolic floating-point numbers in a computer algebra system. We also hope that this work will make easier the analysis of small yet important building blocks of numerical computing.

Table 1: Timings for checking examples with our symbolic floating-point arithmetic library.

Example (base 2)	#`rn`	Timing
• From [20]:		
Example 3.7 (p even)	4	17 ms
Example 3.7 (p odd)	4	16 ms
Example 4.4	4	12 ms
Example 4.6	4	13 ms
Example 6.2	4	15 ms
Example 6.3 (p even)	4	15 ms
Example 6.3 (p odd)	4	15 ms
Example 6.4 (p even)	4	16 ms
Example 6.6 (1st part)	4	12 ms
Example 6.6 (2d part)	4	12 ms
Example 6.7 (1st part)	4	13 ms
Example 6.7 (2d part)	4	13 ms
• From [31]:		
Example	6	21 ms
• From [8]:		
Example (p even)	6	18 ms
Example (p odd)	6	19 ms

Example (base 2)	#`rn`	Timing
• From [21]:		
Example 1	3	10 ms
Example 2	3	12 ms
Example 3 (p even)	3	10 ms
Example 4	2	9 ms
Example 5 (p even)	6	21 ms
Example 5 (p odd)	6	20 ms
Example 6 (p even)	2	10 ms
Example 6 (p odd)	2	10 ms
Example 7 (p even)	2	11 ms
Example 8 (p even)	12	43 ms

Example (base 10)	#`rn`	Timing
• From [20]:		
Example 3.7 (p even)	4	16 ms
Example 3.7 (p odd)	4	16 ms
Example 4.4	4	13 ms
Example 4.6	4	12 ms

Perspectives

We conclude this document by presenting some of our perspectives for future work on the problems addressed in this thesis.

1 Complex floating-point division

The normwise relative error bound $\gamma u + 20u^2$ presented in Chapter 4 for Algorithm 5 for complex floating-point inversion suggested an alternative method to compute the quotient of two complex floating-point numbers: instead of using the formula

$$z_1/z_2 = (z_1 \cdot \bar{z_2})/|z_2|^2, \tag{1}$$

which corresponds to multiplying by the conjugate and dividing by the square of the norm, we can use

$$z_1/z_2 = z_1 \cdot (1/z_2), \tag{2}$$

which corresponds to multiplying by the inverse. We saw in the conclusion of Part II that this second scheme leads to smaller normwise error bounds, for all the multiplication algorithms considered: either the naive one, with or without FMA, the one based on Kahan's algorithm, or the one proposed by Cornea, Harrison and Tang in [10]. This gives a few more algorithms to consider when implementing complex division, depending on the complex multiplication algorithm used.

We were able to find numerical examples to conclude that implementing equation (2) is more accurate in the worst case than equation (1) when considering the naive multiplication without FMA, but we lack such examples for the other algorithms. Finding numerical or symbolic examples (with the help of our symbolic floating-point library), for which the errors are close to the bound, would allow us to classify these algorithms with respect to their worst-case accuracy. Time measurements of practical implementations are also needed to clarify the possible compromises between speed and accuracy among these algorithms.

2 Symbolic floating-point arithmetic

Our Maple library for symbolic floating-point arithmetic already allowed us to check around 30 certificates of asymptotic optimality from the literature. However, we did not yet use it to search for new certificates or symbolic examples. The next step would be to

automate, at least partially, this search for certificates. More precisely, we may try to use algorithms based on interval subdivision (see for example [30, §11.2]), with intervals whose bounds are symbolic floating-point numbers. The usual strategy consists in splitting the intervals into subintervals until a satisfying conclusion is reached for each subinterval. In our case, we are looking for intervals over which the relative error reaches a value close to the bound; if it is not the case, then the interval is dismissed. One suggestion to speed up the computation would be to analyze the errors generated on the intermediate results to possibly deduce that a subinterval cannot lead to an error close to the bound, before the end of the algorithm. Additionally, optimizing the library would accelerate further the computations.

A second path of research would be to address the problem of the square root operation with the symbolic floating-point numbers. In Chapter 5, we saw that in the case of division, we need to use the periodicity of the expansions of rationals in base β to compute the result of a rounding operation. This regularity leads to multiple results depending, for example, on the parity of the symbolic variable k, and we decided to select one of these results, associated with the corresponding constraint on k. In the case of the square root operation, we can identify two difficulty levels when rounding:

- simple cases, such as $\sqrt{2^{2k}-1}$ in precision $2k$, for which the asymptotic expansion $\sqrt{2^{2k}-1} = 2^k(1 - 2^{-2k-1} - 2^{-4k-3} - \ldots)$ leads to the result $2^k - 2^{-k}$;
- difficult cases, such as $\sqrt{2}$ in precision k, for which no closed form is known, and no periodicity appears in the development in base β.

Note that when rounding $\sqrt{2^k-1}$ in precision k, both situations above can occur: "k even" corresponds to the simple case above and "k odd" to the difficult one. As suggested by their names, simple cases are easy to deal with by computing directly the rounded result.

On the other hand, we coped in [23] with a more difficult case. We proved the asymptotic optimality of error bounds of classic algorithms for evaluating $\sqrt{a^2+b^2}$ and $c/\sqrt{a^2+b^2}$ using examples based on an expression similar to $\mathrm{RU}_p(\sqrt{2})$. To check these examples, we used the standard model for directed rounding functions to get a "sufficiently tight" enclosure of $\mathrm{RD}_p(\sqrt{G})$ where G is a symbolic expression parametrized by p. Then, we essentially use the monotonicity of the exact functions and the rounding ones involved in the computation to prove that a certain relative error is reached. Most of these computations could be performed with an interval arithmetic based on symbolic numbers but also requires the use of additional floating-point properties. It may be interesting to see if it is possible to formalize such an interval arithmetic to deal with this example (and of course some others) in an automated way.

A last perspective would be to transfer our formalism and library into a proof assistant such as Coq [6] to get a higher confidence in the computations. In practice, the main objective would be to verify the certificates of asymptotic optimality found by performing a large search with a fast implementation. Two kinds of implementations could be done: either the theory and the algorithms are transfered to the Coq system so that the computations are done within Coq, or the theory alone is developed in the proof system, and we add to the Maple library the generation of a proof that can be checked by Coq.

Bibliography

[1] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754–1985, 1985.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Radix Independent Floating-Point Arithmetic*. ANSI/IEEE Standard 854–1987, 1987.

[3] N. Aubrun, E. Duchêne, J. Duparc, C.-P. Jeannerod, A. Miquel, A. Parreau, G. Theyssier, and N. Revol. *Informatique Mathématique, Une photographie en 2017*. Bruno Salvy (éd), CNRS Éditions, 2017.

[4] M. Baudin. Error bounds of complex arithmetic, June 2011. available at http://forge.scilab.org/upload/compdiv/files/complexerrorbounds_v0.2.pdf.

[5] M. Baudin and R. L. Smith. A robust complex division in Scilab, October 2012. available at http://arxiv.org/abs/1210.4539.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[7] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *20th IEEE Symposium on Computer Arithmetic (ARITH), Tübingen, Germany*, pages 243–252, Los Alamitos, CA, USA, 2011. IEEE Computer Society Press.

[8] R. Brent, C. Percival, and P. Zimmermann. Error bounds on complex floating-point multiplication. *Mathematics of Computation*, 76:1469–1481, 2007.

[9] W. P. Champagne. On finding roots of polynomials by hook or by crook. Master's thesis, University of Texas, Austin, Texas, 1964.

[10] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium®-based Systems*. Intel Press, Hillsboro, OR, USA, 2002.

[11] M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In *14th International Conference on Theorem*

Proving in Higher Order Logics (TPHOLs'01), Edinburgh, Scotland, UK, pages 169–184, Berlin Heidelberg, 2001. Springer-Verlag.

[12] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[13] J. V. Z. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

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

[15] J. Harrison. A machine-checked theory of floating point arithmetic. In *12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99), Nice, France*, pages 113–130, Berlin Heidelberg, 1999. Springer-Verlag.

[16] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, USA, second edition, 2002.

[17] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society, New York, Aug. 2008. available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[18] C.-P. Jeannerod. A radix-independent error analysis of the Cornea-Harrison-Tang method. *ACM Transactions on Mathematical Software*, 2016.

[19] C.-P. Jeannerod, P. Kornerup, N. Louvet, and J.-M. Muller. Error bounds on complex floating-point multiplication with an FMA. *Mathematics of Computation*, 86(304):pp. 881–898, 2017.

[20] C.-P. Jeannerod, N. Louvet, and J.-M. Muller. Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants. *Mathematics of Computation*, 82:2245–2264, 2013.

[21] C.-P. Jeannerod, N. Louvet, and J.-M. Muller. On the componentwise accuracy of complex floating-point division with an FMA. In *21st IEEE Symposium on Computer Arithmetic (ARITH), Austin, TX, USA*, pages 83–90, Los Alamitos, CA, USA, 2013. IEEE Computer Society Press.

[22] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Plet. Sharp error bounds for complex floating-point inversion. *Numerical Algorithms*, 73(3):735–760, Nov. 2016.

[23] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Plet. The classical relative error bounds for computing $\sqrt{a^2 + b^2}$ and $c/\sqrt{a^2 + b^2}$ in binary floating-point arithmetic are asymptotically optimal. In *24th IEEE Symposium on Computer Arithmetic (ARITH), London, England, UK*, Los Alamitos, CA, USA, 2017. IEEE Computer Society Press.

[24] C.-P. Jeannerod and S. M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Mathematics of Computation*, 2016.

[25] W. Kahan. A logarithm too clever by half. Available at http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT, 2004.

[26] A. H. Karp and P. Markstein. High-precision division and square root. *ACM Trans. Math. Softw.*, 23(4):561–589, Dec. 1997.

[27] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 1998.

[28] S. Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics*. Springer, New York, revised third edition, 2002.

[29] G. Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012.

[30] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[31] J.-M. Muller. On the error of computing $ab + cd$ using Cornea, Harrison and Tang's method. *ACM Transactions on Mathematical Software*, 41(2):7:1–7:8, 2015.

[32] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[33] V. Y. Pan and Y. Yu. Certification of numerical computation of the sign of the determinant of a matrix. *Algorithmica*, 30(4):708–724, 2001.

[34] G. Pólya and G. Szegö. *Problems and Theorems in Analysis*, volume 2 of *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen*. Springer-Verlag Berlin Heidelberg New York, 1976.

[35] D. M. Priest. Efficient scaling for complex division. *ACM Transactions on Mathematical Software*, 30(4), Dec. 2004.

[36] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation, Part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[37] R. L. Smith. Algorithm 116: Complex division. *Communications of the ACM*, 5(8):435, 1962.

[38] G. W. Stewart. A note on complex division. *ACM Transactions on Mathematical Software*, 11(3):238–241, Sept. 1985.

[39] A. Ziv. Sharp ULP rounding error bound for the hypotenuse function. *Mathematics of Computation*, 68(227):1143–1148, 1999.

Appendix A

Details of the error analysis of complex inversion

A.1 Partial derivatives of g_2

Computing the partial derivatives of g_2 with respect to a and b gives

$$\frac{\partial g_2}{\partial a} = \frac{a}{4} + \frac{4}{a^2+b^2} - \frac{8a(a+b)}{\left(a^2+b^2\right)^2} - \frac{256a}{\left(a^2+b^2\right)^3},$$
$$\frac{\partial g_2}{\partial b} = \frac{b}{4} + \frac{4}{a^2+b^2} - \frac{8b(a+b)}{\left(a^2+b^2\right)^2} - \frac{256b}{\left(a^2+b^2\right)^3}.$$

First, we know that $b > a$ so $\frac{1}{b}\frac{\partial}{\partial b}g_2(a,b) < \frac{1}{a}\frac{\partial}{\partial a}g_2(a,b)$. We just have to prove that $\frac{\partial}{\partial a}g_2(a,b) < 0$, that is,

$$\frac{\left(a^2+b^2\right)^2}{4} + \frac{4\left(a^2+b^2\right)}{a} < 8(a+b) + \frac{256}{a^2+b^2}.$$

Since for $(a,b) \in D_2$ we have $\sqrt{2} < a, b$, and $a^2+b^2 < 4\sqrt{2}+4u$, it is enough to check that

$$\frac{\left(4\sqrt{2}+4u\right)^2}{4} + \frac{4\left(4\sqrt{2}+4u\right)}{\sqrt{2}} < 16\sqrt{2} + \frac{256}{4\sqrt{2}+4u},$$

which holds for $p \geqslant 2$.

A.2 Partial derivative of g_3

We compute the partial derivative of g_3 with respect to b, and check that this derivative is negative over the domain D_3. We have

$$\frac{\partial g_3}{\partial b} = \frac{b\left(2^{-e}+1\right)}{8} + \frac{6+2^{-e}}{2\left(a^2+b^2\right)} - b\frac{\left(6+2^{-e}\right)\left(2^{-\frac{e}{2}}a+b\right)}{\left(a^2+b^2\right)^2} - 4b\frac{\left(6+2^{-e}\right)^2}{\left(a^2+b^2\right)^3},$$

and we check that

$$\frac{b\left(2^{-e}+1\right)}{8}+\frac{6+2^{-e}}{2\left(a^2+b^2\right)}<b\frac{\left(6+2^{-e}\right)\left(2^{-\frac{e}{2}}a+b\right)}{\left(a^2+b^2\right)^2}+4b\frac{\left(6+2^{-e}\right)^2}{\left(a^2+b^2\right)^3}.$$

Multiplying both sides by $\frac{(a^2+b^2)^2}{b(6+2^{-e})}$ and since $1 \leqslant b$, it is enough to prove that

$$\frac{\left(2^{-e}+1\right)\left(a^2+b^2\right)^2}{8\left(6+2^{-e}\right)}+\frac{a^2+b^2}{2}<2^{-\frac{e}{2}}a+b+4\frac{6+2^{-e}}{a^2+b^2}.$$

This follows from the following sequence of three inequalities

$$\begin{aligned}\frac{\left(2^{-e}+1\right)\left(a^2+b^2\right)^2}{8\left(6+2^{-e}\right)}+\frac{a^2+b^2}{2} &< \frac{2\left(4\sqrt{2}+4u\right)^2}{48}+\frac{4\sqrt{2}+4u}{2},\\ \frac{2\left(4\sqrt{2}+4u\right)^2}{48}+\frac{4\sqrt{2}+4u}{2} &< 4\frac{6}{4\sqrt{2}+4u}+1 \quad \text{for } p \geqslant 3,\\ 4\frac{6}{4\sqrt{2}+4u}+1 &< 4\frac{6+2^{-e}}{a^2+b^2}+2^{-\frac{e}{2}}a+b.\end{aligned}$$

A.3 Partial derivatives of g_4

The partial derivative of g_4 with respect to b is given by

$$\frac{\partial g_4}{\partial b}=\frac{b\left(2^{-e}+1\right)}{2}+\frac{3+2^{-e}}{a^2+b^2}-2b\frac{\left(2^{-\frac{e}{2}}a+b\right)\left(3+2^{-e}\right)}{\left(a^2+b^2\right)^2}-4b\frac{\left(3+2^{-e}\right)^2}{\left(a^2+b^2\right)^3}.$$

We want to prove that $\frac{\partial}{\partial b}g_4(a,b,e)<0$ or, equivalently, that

$$\frac{\left(a^2+b^2\right)^2\left(2^{-e}+1\right)}{2\left(3+2^{-e}\right)}+\frac{a^2+b^2}{b}<2\left(2^{-\frac{e}{2}}a+b\right)+4\frac{3+2^{-e}}{a^2+b^2}.$$

This inequality can be derived from the following ones:

$$\begin{aligned}\frac{\left(a^2+b^2\right)^2\left(2^{-e}+1\right)}{2\left(3+2^{-e}\right)}+\frac{a^2+b^2}{b} &< \frac{2\left(2\sqrt{2}+2u\right)^2}{6}+2\sqrt{2}+2u,\\ \frac{\left(2\sqrt{2}+2u\right)^2}{3}+2\sqrt{2}+2u &< 2+\frac{12}{2\sqrt{2}+2u} \quad \text{for } p \geqslant 4,\\ 2+\frac{12}{2\sqrt{2}+2u} &< 2\left(2^{-\frac{e}{2}}a+b\right)+4\frac{3+2^{-e}}{a^2+b^2}.\end{aligned}$$

The partial derivative of $g_4(a,\sqrt{2-a^2},e)$ with respect to a is

$$\frac{\partial}{\partial a}g_4(a,\sqrt{2-a^2},e)=\frac{3+2^{-e}}{2}\left(2^{-\frac{e}{2}}-\frac{a}{\sqrt{2-a^2}}\right),$$

which is zero if $a=a_0$ with $a_0=2^{-\frac{e}{2}}\sqrt{\frac{2}{1+2^{-e}}}$, positive if $a<a_0$, and negative if $a>a_0$.

A.4 Partial derivative of g_5

We have

$$\frac{\partial g_5}{\partial b} = \frac{b}{4} + \frac{2}{a^2+b^2} - \frac{4\,(a+b)}{\left(a^2+b^2\right)^2}b - \frac{64}{\left(a^2+b^2\right)^3}b,$$

and it can be checked that this partial derivative is negative using the following inequalities:

$$\begin{aligned}
\frac{\left(a^2+b^2\right)^2}{4} + \frac{2}{b}\left(a^2+b^2\right) &< \frac{\left(2\sqrt{2}+2u\right)^2}{4} + 2\left(2\sqrt{2}+2u\right), \\
\frac{\left(2\sqrt{2}+2u\right)^2}{4} + 2\left(2\sqrt{2}+2u\right) &< 8 + \frac{64}{2\sqrt{2}+2u} \quad \text{for } p \geqslant 2, \\
8 + \frac{64}{2\sqrt{2}+2u} &< 4\,(a+b) + \frac{64}{a^2+b^2}.
\end{aligned}$$

A.5 Partial derivatives of g_6

The partial derivatives of g_6 with respect to a and b are

$$\frac{\partial g_6}{\partial a} = \frac{a}{4}\left(2^{-e}+2\right) + \frac{4+2^{-e}}{a^2+b^2}2^{-\frac{1+e}{2}} - 2a\frac{\left(2^{-\frac{1+e}{2}}a+b\right)\left(4+2^{-e}\right)}{\left(a^2+b^2\right)^2} - 4a\frac{\left(4+2^{-e}\right)^2}{\left(a^2+b^2\right)^3}$$

and

$$\frac{\partial g_6}{\partial b} = \frac{b}{4}\left(2^{-e}+2\right) + \frac{4+2^{-e}}{a^2+b^2} - 2b\frac{\left(2^{-\frac{1+e}{2}}a+b\right)\left(4+2^{-e}\right)}{\left(a^2+b^2\right)^2} - 4b\frac{\left(4+2^{-e}\right)^2}{\left(a^2+b^2\right)^3}.$$

For $(a,b,e) \in D_6$, it can be checked that $\frac{\partial g_6}{\partial a}(a,b,e) < 0$ and $\frac{\partial g_6}{\partial b}(a,b,e) < 0$ as follows. Note first that $2^{-\frac{e}{2}} \leqslant a$ implies

$$\frac{4+2^{-e}}{a^2+b^2}2^{-\frac{1+e}{2}} \leqslant \frac{4+2^{-e}}{\sqrt{2}\left(a^2+b^2\right)}a,$$

and that $\sqrt{2} \leqslant b$ implies

$$\frac{4+2^{-e}}{a^2+b^2} \leqslant \frac{4+2^{-e}}{\sqrt{2}\left(a^2+b^2\right)}b.$$

Thus, the same expression can be used as an upper bound for both $\frac{1}{a}\frac{\partial g_6}{\partial a}$ and $\frac{1}{b}\frac{\partial g_6}{\partial b}$. Then, multiplying it by $\frac{(a^2+b^2)^2}{4+2^{-e}}$, it is enough to prove that

$$\frac{\left(a^2+b^2\right)^2\left(2^{-1-e}+1\right)}{2\left(4+2^{-e}\right)} + \frac{a^2+b^2}{\sqrt{2}} < 2\left(2^{-\frac{1+e}{2}}a+b\right) + 4\frac{4+2^{-e}}{a^2+b^2}.$$

This last inequality follows from the following three ones:

$$\frac{\left(a^2+b^2\right)^2\left(2^{-1-e}+1\right)}{2\left(4+2^{-e}\right)} + \frac{a^2+b^2}{\sqrt{2}} < \frac{\left(2\sqrt{2}+\left(2+\frac{1}{2}\right)u\right)^2\left(\frac{1}{4}+1\right)}{8} + 2 + \frac{2+\frac{1}{2}}{\sqrt{2}}u,$$

$$\frac{\left(2\sqrt{2}+\left(2+\frac{1}{2}\right)u\right)^2\left(\frac{1}{4}+1\right)}{8}+2+\frac{2+\frac{1}{2}}{\sqrt{2}}u<2\sqrt{2}+\frac{16}{2\sqrt{2}+\left(2+\frac{1}{2}\right)u}\qquad\text{for } p\geqslant 2,$$

and

$$2\sqrt{2}+\frac{16}{2\sqrt{2}+\left(2+\frac{1}{2}\right)u}<2\left(2^{-\frac{1+e}{2}}a+b\right)+4\frac{4+2^{-e}}{a^2+b^2}.$$